Project no.	034084
Project acronym:	SELFMAN
Project title:	*Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components*

European Sixth Framework Programme Priority 2, Information Society Technologies

The Adventures of Selfman - Year Three

Due date of deliverable:	July 15, 2009
Actual submission date:	July 15, 2009
Start date of project:	June 1, 2006
Duration:	36 months
Dissemination level:	PU

Contents

1 Introduction **14**

2 D3.1c: Final report on formal models for transactions over structured overlay networks **16**
2.1 Executive Summary 16
2.2 Partners Contributing to the Deliverable 17
2.3 Introduction 18
2.3.1 Transaction algorithm described using components . . 18
2.3.2 Evaluation 18
2.3.3 Visualization of the Transaction Algorithm 20
2.3.4 Eager locking for synchronous collaboration 20
2.4 Papers and publications 22

3 D3.2b: Report on replicated storage service over a structured overlay network **25**
3.1 Executive summary 25
3.2 Contractors contributing to the Deliverable 26
3.3 Results 27
3.4 Papers and publications 29
3.4.1 Scalaris: Users and Developers Guide 29
3.4.2 Scalaris: Reliable Transactional P2P Key/Value Store . 29
3.4.3 A Scalable, Transactional Data Store for Future Internet Services 30
3.4.4 Visualizing Transactional Algorithms for DHTs 30
3.4.5 DeTransDraw: Decentralized transactional collaborative drawing 31

4 D3.3b: Report on simple database query layer for replicated storage service **32**
4.1 Executive summary 32
4.2 Contractors contributing to the Deliverable 33

4.3 Results . . . 34
4.3.1 Scalaris . . . 34
4.3.2 Beernet . . . 35
4.4 Papers and publications . . . 38
4.4.1 Scalaris: Users and Developers Guide . . . 38
4.4.2 Scalaris Java Interface . . . 38

5 D4.1b: Second report on self-configuration support 39
5.1 Executive summary . . . 39
5.2 Contractors contributing to the Deliverable . . . 40
5.3 Results . . . 41
5.3.1 A formal specification of the Fractal deployment model 41
5.3.2 Self-configurable dynamic architectures in Oz . . . 43
5.3.3 Self-configuration mechanisms in the Kompics component model . . . 46
5.3.4 Self-configuration mechanisms in PeerTV . . . 48
5.4 Papers and publications . . . 50

6 D4.1c: Self-configuration support (software) 51
6.1 Executive summary . . . 51
6.2 Contractors contributing to the Deliverable . . . 52
6.3 The WorkflOz library . . . 53
6.3.1 Design . . . 53
6.3.2 Patterns . . . 53

7 D4.2b: Second report on self-healing support 62
7.1 Executive summary . . . 62
7.2 Contractors contributing to the Deliverable . . . 63
7.3 Results . . . 64
7.3.1 Self-healing in structured overlay networks . . . 64
7.3.2 Self-healing mechanisms in the Kompics component model . . . 65
7.3.3 Self-healing mechanisms in cluster systems using Fractal 67
7.4 Papers and publications . . . 70

8 D4.2c: Self-healing support (software) 71
8.1 Executive summary . . . 71
8.2 Contractors contributing to the Deliverable . . . 72
8.3 The SicSim Architecture . . . 73

9 D4.3b: Second report on self-tuning support 75
9.1 Executive summary . . . 75
9.2 Contractors contributing to the Deliverable . . . 76
9.3 Results . . . 77
9.3.1 Load Balancing in a Distributed Key/Value store . . . 77
9.3.2 Network Size Estimation . . . 80
9.4 Papers and publications . . . 83

10 D4.3c: Self-tuning support (software) 84
10.1 Executive summary . . . 84
10.2 Contractors contributing to the Deliverable . . . 85
10.3 Results . . . 86
10.3.1 Algorithm . . . 86

11 D4.4b: Self-protection support 89
11.1 Executive summary . . . 89
11.2 Contractors contributing to the deliverable . . . 90
11.3 Software for small world network and social network experiments 91
11.4 Initial SWN simulator . . . 92
11.5 Social network crawler . . . 97
11.6 A more efficient SWN simulator for social networks . . . 99

12 D5.2b: Demonstrator application for J2EE (software) 102
12.1 Executive summary . . . 102
12.2 Contractors contributing to the Deliverable . . . 103
12.3 Results . . . 104
12.4 Papers and publications . . . 106
12.4.1 Scalaris: Reliable Transactional P2P Key/Value Store . 106
12.4.2 A Scalable, Transactional Data Store for Future Internet Services . . . 107

13 D5.3: Demonstrator application for Mozart (software) 108
13.1 Executive summary . . . 108
13.2 Contractors contributing to the Deliverable . . . 109
13.3 Sindaca recommendation system . . . 110
13.3.1 Introduction . . . 110
13.3.2 After sign-in and voting . . . 111
13.3.3 Making a recommendation . . . 112
13.3.4 Data storage structure . . . 113
13.3.5 Configuration . . . 115

14 D5.4a: Qualitative evaluation of autonomic features of Selfman applications 117
14.1 Executive summary . . . 117
14.2 Contractors contributing to the Deliverable . . . 119
14.3 Methodology and process . . . 120
14.3.1 Background under discussion . . . 120
14.3.2 Approach specificities . . . 124
14.4 Assessment process . . . 128
14.4.1 Qualitative assessment of elementary AB . . . 129
14.4.2 Qualitative assessment of global ABs . . . 131
14.5 Experimental assessment of local ABs . . . 133
14.5.1 PeerTV (Peerialism(P6)) . . . 133
14.5.2 Scalaris (ZIB(P5)) . . . 142
14.5.3 The gPhone application (UCL(P1)) . . . 155
14.6 Experimental assessment of autonomics in Selfman applications 171
14.6.1 PeerTV (Peerialism(P6)) . . . 171
14.6.2 Scalaris (ZIB(P5)) . . . 173
14.6.3 The gPhone application (UCL(P1)) . . . 175
14.7 Discussion . . . 176
14.7.1 Experimental results . . . 176
14.7.2 Evaluation methodology . . . 178

15 D5.4b: Quantitative evaluation of autonomic features of Selfman applications 179
15.1 Executive summary . . . 179
15.2 Contractors contributing to the Deliverable . . . 181
15.3 Methodology and process . . . 182
15.3.1 Background under discussion . . . 182
15.3.2 Approach specificities . . . 184
15.4 Assessment process . . . 185
15.4.1 Quantitative assessment of elementary AB . . . 185
15.4.2 Quantitative assessment of global ABs . . . 187
15.5 Experimental assessment of local ABs . . . 188
15.5.1 PeerTV (Peerialism(P6)) . . . 188
15.5.2 Scalaris (ZIB(P5)) . . . 193
15.5.3 The gPhone application (UCL(P1)) . . . 196
15.6 Discussion . . . 202
15.6.1 Experimental results . . . 202
15.6.2 Evaluation methodology . . . 203

16 D5.6: Evaluation of security mechanisms 205
16.1 Executive summary . . . 205
16.2 Contractors contributing to the deliverable . . . 206
16.3 Self-protection support & mechanisms . . . 207
16.3.1 Small world networks (SWN) as a kind of SON . . . 207
16.3.2 Software component security . . . 221
16.4 Application level security . . . 223
16.4.1 Security issues for Wikipedia . . . 224
16.5 Papers and publications . . . 229

17 D5.7: Guidelines for building self-managing applications 232
17.1 Executive summary . . . 232
17.2 Contractors contributing to the Deliverable . . . 234
17.3 Introduction . . . 235
17.3.1 Context of this report . . . 236
17.3.2 General guidelines . . . 236
17.3.3 Phase transitions . . . 237
17.3.4 Interdisciplinary nature . . . 238
17.3.5 Structure of this report . . . 239
17.4 The general architecture . . . 240
17.4.1 Three-layered architecture . . . 241
17.4.2 Combining structured overlay networks and components 242
17.4.3 Failure detection . . . 243
17.5 Examples of the general architecture . . . 244
17.5.1 A self-management architecture built with the Kompics component model . . . 245
17.5.2 Using self management to provide availability and scalability: the Scalaris example . . . 248
17.5.3 Using a relaxed ring to simplify overlay maintenance: the Beernet example . . . 250
17.6 Design rules for feedback structures . . . 251
17.6.1 Stigmergy should be used with care . . . 252
17.6.2 Loop management corresponds to data abstraction . . . 252
17.6.3 Loop management should control a natural parameter . 252
17.6.4 Take advantage of different time scales . . . 253
17.6.5 Complex components should be sandboxed . . . 253
17.6.6 Use push-pull to improve regulation . . . 254
17.6.7 Handle failures with reversible phase transitions . . . 254
17.7 Overall design of a self-managing system . . . 254
17.7.1 Decomposition: defining the management tasks . . . 255
17.7.2 Orchestration: handling the interactions . . . 255

17.7.3 Forms of interaction . . . 256
17.7.4 Examples of interaction . . . 257
17.8 Design rules for the self-management axes . . . 258
17.8.1 Making it self-tuning . . . 258
17.8.2 Making it self-protecting . . . 259
17.8.3 Making it self-healing . . . 263
17.8.4 Making it self-configuring . . . 263
17.9 Conclusions . . . 264
17.9.1 Future work . . . 264

18 D5.8: Self-managing distributed collaborative drawing tool on mobile devices 266
18.1 Executive summary . . . 266
18.2 Contractors contributing to the Deliverable . . . 267
18.3 Introduction . . . 268
18.4 Specification . . . 269
18.5 Architecture . . . 271
18.5.1 DeTransDraw for computers . . . 271
18.5.2 DeTransDrawid . . . 274
18.6 Implementation . . . 276
18.6.1 DeTransDraw . . . 276
18.6.2 DeTransDrawid . . . 276
18.7 Future work . . . 279
18.8 Papers and publications . . . 280

19 D6.5c: Final progress and assessment report with lessons learned 281
19.1 Executive summary . . . 281
19.2 Contractors contributing to the Deliverable . . . 282
19.3 Results . . . 283
19.3.1 Lessons learned . . . 283
19.3.2 SELFMAN and the problem of parallel research . . . 283

A Publications 285
A.1 Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions . . . 286
A.2 Dealing with Network Partitions and Mergers . . . 298
A.3 Network Size Estimation for Structured Overlays . . . 313
A.4 The Relaxed-Ring: a Fault-Tolerant Topology for Structured Overlay Networks . . . 327
A.5 Visualizing Transactional Algorithms for DHTs . . . 350

A.6 Transactional DHT Algorithms 353
A.7 The Design of a Transactional Key-Value Store Using the Kompics Component Model 394
A.8 Scalaris: Users and Developers Guide 406
A.9 Scalaris: Reliable Transactional P2P Key/Value Store 446
A.10 A Scalable, Transactional Data Store for Future Internet Services 454
A.11 Scalaris Java Interface 467
A.12 Developing, Simulating, and Deploying Peer-to-Peer Systems using the Kompics Component Model 498
A.13 Practical Protocol Composition, Encapsulation and Sharing in Kompics 507
A.14 Kompics Programming Manual 514
A.15 A Design Methodology for Self-Management in Distributed Environments 531
A.16 Using Global Information for Load Balancing in DHTs 539
A.17 Node Placement in a Distributed Key/Value-store 546
A.18 Security issues in small world network routing 559
A.19 Small world networks as (semi)-structured overlay networks . . 562
A.20 Wiki credibility enhancement 568
A.21 A Toolkit for Peer-to-Peer Distributed User Interfaces: Concepts, Implementation, and Applications 573
A.22 Decentralized Transactional Collaborative Drawing 584
A.23 Decentralized Transactional Collaborative Drawing *(demo)* . . 590

List of Figures

2.1 Performance of Scalaris: (a) Read operation, (b) Modify operation for different numbers of local threads and cluster sizes. 19
2.2 Adapted Paxos consensus algorithm with eager locking, no read-phase, and eager notification to readers. 21

5.1 A composition of task components. 45
5.2 Kompics "HelloWorld" example. 47
5.3 Examples component architectures. 48

6.1 An example of cycle in task control flow 61

7.1 Kompics "HelloWorld" example. 65
7.2 Kompics control port. 66
7.3 Examples component architectures. 67
7.4 A managed system and its replicated management subsystem. 69

8.1 SicSim main modules 74

9.1 Impact of global information on a randomized load balancing algorithm. 78
9.2 The effect of different access workloads and key distributions. . 79
9.3 Imbalance of the system using difference balancing strategies and increasing the system size. 80

10.1 A node n_i with successor and predecessor and the ranges of responsibilities 87
10.2 Pseudo code of Karger's algorithm. 88

11.1 Simulator Interface 93
11.2 Ring Structure - Perfect SWN node positions 94
11.3 Ring Structure - Shuffled node positions 94
11.4 Ring Structure - Recovered node positions 95
11.5 An example of a Social Network Database 97

12.1 Wikipedia on Scalaris. 104
12.2 Screenshot of the Bavarian Wikipedia on Scalaris. Images are not included in the dump. 106

13.1 Sindaca's welcome page with sign in form 111
13.2 After sign in, users can vote for suggested recommendations. . 112
13.3 Adding a new recommendation. 113
13.4 State of recommendation proposed by the user. 114
13.5 Sindaca's relational model 115

14.1 Organization of autonomic computing characteristics based on ISO/IEC 9126 standard quality factors (inspired from [71] and [98]) 123
14.2 Hierarchy between autonomic computing characteristics (extracted from [49]) 123
14.3 Definition of adaptation time, reaction time and stabilization time 124
14.4 Adaptation of ISO/IEC 9126 to autonomic computing field . . 126
14.5 Mapping between MAPE-loop stages and their duration . . . 127
14.6 PeerTV System Architecture 134
14.7 Sequence diagram of a join and stabilization. 142
14.8 Sequence diagram of the join AB 155
14.9 Ring maintenance as a feedback loop. New peer join as new predecessor of the current responsible of its key 156
14.10 Ring maintenance as a feedback loop. New peer is accepted to join between p and r, and becomes the new successor of peer p 156
14.11 Ring maintenance as a feedback loop. A peer is notified about its new successor 157
14.12 Failure recovery sequence diagram 163
14.13 Failure recovery as a feedback loop 163
14.14 Finger maintenance with failure detection and correction-on-use 166

15.1 Effect of Self-Optimization Level on Saving Percentage 191
15.2 Effect of Self-Optimization Level on Total Performance Score . 191
15.3 K-ary routing in Chord and Chord# 193
15.4 Performance of Scalaris: (a) Read operation, (b) Modify operation for different numbers of local threads and cluster sizes. 195
15.5 Average amount of branches depending on the size of the network and the quality of the connectivity. 197
15.6 Average size of branches depending on the quality of connections. 199

15.7 Bandwidth consumption of ring maintenance in Chord and the Relaxed-Ring. 199
15.8 Bootstrapping bandwidth usage on different networks. 200
15.9 Average number of hops vs number of peers. 204

16.1 Network topology of Chord, SWN, and Random Network . . . 208
16.2 Routing Success and Node Failure 210
16.3 SWN vs. Chord on Greediness 211
16.4 Average and Deviation in Routing Length vs. Greediness . . . 212
16.5 Consecutive range ID attack 213
16.6 Increase in the average number of hops due to the range attack 213
16.7 Infection Percentage vs. Malicious Nodes and Restart Probability 215
16.8 Successful Routing vs. Malicious Nodes and Restart Probability 215
16.9 Switching Percentage vs. Malicious Nodes and Restart Probability 216
16.10 Facebook social network graph of 1000 nodes. 218
16.11 Node Degree Distribution 219
16.12 Clustering Coefficient Distribution 220
16.13 Routing Failure Ratio 220
16.14 Routing Length 221
16.15 Wikipedia Credibility Extension 225
16.16 Wikipedia Credibility Proxy 226
16.17 Wikipedia Verifier Tag Extension 227
16.18 Wikipedia showing a credible enhanced text 227

17.1 A self-management architecture built with Kompics 246
17.2 A feedback loop 251

18.1 State diagram of a user 269
18.2 On the left, the user is in *Asking for locks* state. On the right, the user is in *Got locks* mode. 270
18.3 User interface for managing the peer 271
18.4 Example of objects 272
18.5 Selection of an object 273
18.6 Selection of an object already locked 273
18.7 Structure of the application for Android 274
18.8 DeTransDrawid in application launcher 275
18.9 Application is starting 275
18.10 Example of objects drawn with DeTransDraw and displayed by DeTransDrawid 276

18.11Minimum of code for the activity class 277
18.12Sample of code to invoke mozart 278
18.13Main function to receive messages from Java. 278

List of Tables

7.1 Comparison of Simple Ring Unification and Gossip-based Ring Unification. 65

14.1 Decomposition of quality factors in quality criteria in ISO/IEC 9126 standard 122
14.2 Qualitative assessment grid for elementary AB 129
14.3 Qualitative assessment grid for global ABs 131
14.4 Scale and levels description of autonomic computing maturity 132
14.5 Joining peer assignment 134
14.6 Failing peer assignment 137
14.7 Overlay Optimization Process 139
14.8 Joining Node 143
14.9 Periodic stabilization of successor's predecessor 144
14.10 Periodic stabilization 146
14.11 Periodic stabilization 147
14.12 Fingers 148
14.13 Load-balancing 150
14.14 Replica maintenance 153
14.15 Predecessor update on join 157
14.16 Predecessor and successors initialization on join 160
14.17 Successor update on join 161
14.18 Handling peer leaving/failure - Correction-on-change 163
14.19 Failure recovery of fingers enhanced with correction-on-use . . 166
14.20 PeerTV: qualitative assessment of global ABs 171
14.21 Scalaris: qualitative assessment of global ABs 173
14.22 The gPhone application: qualitative assessment of global ABs 175
14.23 Synthesis of qualitative assessment on local ABs coming from Selfman applications 176

15.1 Quantitative assessment grid for elementary AB 185

Chapter 1

Introduction

This document presents all the deliverables of the final year of the SELFMAN project except the Periodic Activity Report which is submitted as a separate document. Each deliverable is a separate chapter in this document. Relevant published papers are included as appendices, which may be referenced from several deliverables. In its third year, SELFMAN research has matured:

- The SicSim discrete event simulator, Kompics component model, Scalaris transactional store, and Mozart version 1.4.0 are publicly released.
- The PeerTV product has been released and Peerialism is the subject of a pending acquisition by GGF.
- Other software has been developed as well, to be released later: the Beernet transactional store, the WorkflOz/FructOz/LactOz self-configuration suite, and the MyP2PWorld application-level network and concurrency emulator.
- Three demonstrator applications have been written using the SELFMAN technology, including a new one running on a network of mobile devices (gPhones).
- In-depth research has been done on self-* services, for self configuration, self healing, self tuning, and self protection.
- In-depth evaluations have been done of qualitative and quantitative autonomic mechanisms in the demonstrator applications and of security mechanisms.
- A set of guidelines has been written for building self-managing applications.

We have extended the project by four months in order to continue the momentum of SELFMAN and to prepare a successor project.

Chapter 2

D3.1c: Final report on formal models for transactions over structured overlay networks

2.1 Executive Summary

Application developers using a storage system such as a relational database or a file-system requires well-defined semantics for reading and writing data. In a database storage layer this also include transaction functionality where read and write over multiple data entries are Atomic, Consistent, Isolated and Durable (ACID). We aim to develop a self-managing and scalable storage layer supporting transactions based on Structured Overlay Networks. A system with these properties will enable applications with higher requirements on data consistency and transaction support. An example is the Wikipedia demonstrator application presented in D5.2b. In this deliverable we present the final version of the transaction algorithm including results from a performance evaluation and we present the design of a component architecture for the transactional DHT algorithms, using the Kompics component model.

2.2 Partners Contributing to the Deliverable

ZIB(P5), KTH(P2) and UCL(P1) have contributed to this deliverable.

ZIB(P5) ZIB has contributed on the transaction model and the DHT consistency model. The largest focus during the report period was to perform evaluation of the transaction model implementation. ZIB collaborated with KTH to represent the transactional DHT as components using the Kompics component model.

KTH(P2) KTH contributed by providing expertise on Kompics and on component-based software design. KTH has collaborated with ZIB on designing a component architecture for the transactional DHT algorithms, using the Kompics component model.

UCL(P1) UCL contributed to empirically validate the advantages of Paxos consensus atomic commit over Two-phase commit. UCL and ZIB worked together on the implementation of both algorithms and demonstrated them on a conference. UCL also adapted and extended Paxos algorithm to allow synchronous collaborations of users.

2.3 Introduction

A SON-based DHT is a self-managing storage layer providing basic item manipulation primitives such as *put(key, value)*, for inserting a new (key, value)-pair and *get(key)*, for retrieving a value associated with a given key [75]. Traditionally, DHTs have been used by applications with immutable state or weak consistency guarantees. Applications with higher requirements on data consistency and interface flexibility are increasingly demanding easier to manage and more scalable storage layers than what current systems can provide [20, 65].

The transaction processing framework initially presented in D3.1a and D3.1b enables updates and/or reads over multiple data items stored in a DHT. In this deliverable we present the algorithm using the concepts developed in the Kompics component model. Additionally, the algorithms are evaluated in an experimental setting using the Scalaris key/value-store. A detailed description of the algorithm and the results is available in the technical report found in Appendix A.6.

2.3.1 Transaction algorithm described using components

We designed a component architecture for the transactional DHT algorithms using the Kompics component model [2, 7]. Entities in the system are modeled as Kompics components. Components local to one system node communicate by passing events, through well-defined ports. Ports specify the types of events communicated by components. Components residing on different system nodes communicate by messages. Messages are events that a network component on the source node marshals and sends to the network component on the destination node which unmarshals and delivers them to other components residing on the destination node.

We present the component architecture of the system nodes, the components with the protocols they implement and the messages they exchange. We also enumerate the possible failure scenarios and how they should be handled. Please consult Appendix A.7 for the full text of the design report.

2.3.2 Evaluation

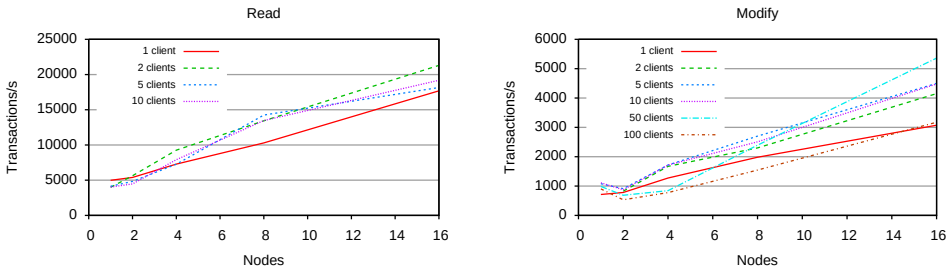
We tested the performance of Scalaris and the transaction algorithm implementation on an Intel cluster up to 16 nodes. Each node has two Quad-Core E5420s (8 cores in total) running at 2.5 GHz and 16 GB of main memory.

The nodes are connected via GigE and Infiniband; we used the GigE network for our evaluation.

On each physical node we were running one multi-core Erlang virtual machine. Each virtual machine hosted 16 Scalaris nodes. We used a replication degree of four, that is, there exist four copies of each key-value pair.

We tested two operations: a read and a modify operation. The read operation reads a key-value pair. The modify operation reads a key-value pair, increments the value and writes the result back to the distributed Scalaris store. To guaran- tee consistency, the read-increment-write is executed within a transaction. The read operation, in contrast, simply reads from a majority of the keys. The benchmarks involved the following steps:

- Start watch.
- Start n Erlang client processes in each VM.
- Execute the read or modify operation i times in each client.
- Wait for all clients to finish.
- Stop watch.



(a) Read throughput (b) Write throughput

Figure 2.1: Performance of Scalaris: (a) Read operation, (b) Modify operation for different numbers of local threads and cluster sizes.

Figure 2.1 shows the results for various numbers of clients per VM (see the colored graphs). In the read benchmarks depicted in Fig. 2.1(a), each thread reads a key 2000 times while the modify benchmarks in Fig. 2.1(b) modify each key 100 time in each thread.

As can be seen, the system scales about linearly over a wide range of system sizes. In the read benchmarks (Fig. 2.1(a)), two clients per VM produce an optimal load for the system, resulting in more than 20,000 read

operations per second on a 16 node (=128 core) cluster. Using only one client (red graph) does not produce enough operations to saturate the system, while five clients (blue graph) cause too much contention. Note that each read operation involves accessing a majority (3 out of 4) replicas.

The performance of the modify operation (Fig. 2.1(b)) is of course lower, but still scales nicely with increasing system sizes. Here, the best performance of 5,500 transactions per second is reached with fifty load generators per VM, each of them generating approximately seven transactions per second. This results in 344 transactions per second on each server.

Note that each modify transaction requires Scalaris to execute the adapted Paxos algorithm, which involves finding a majority (i.e. 3 out of 4) of transaction participants and transaction managers, plus the communication between them. The performance graphs illustrate that a single client per VM does not produce enough transaction load, while fifty clients are optimal to hide the communica- tion latency between the transaction rounds. Increasing the concurrency further to 100 clients does not improve the performance, because this causes too much contention. Note that for the 100-clients-case, there are actually 16*100 clients issuing increment transactions. Overall, both graphs illustrate the linear scalability of Scalaris.

2.3.3 Visualization of the Transaction Algorithm

The focus of this demonstrator is on the study of algo- rithms for implementing transactions on peer-to-peer net- works. Their visualization contributes to the analysis and test of the protocols, verifying their tolerance to failures. In particular, we show a DHT running two-phase commit and the Paxos consensus algorithm.

2.3.4 Eager locking for synchronous collaboration

We have observed how Paxos consensus algorithm for atomic transactions on DHTs (see Appendix A.6) is extremely useful for building systems with decentralized storage based on symmetric replication. The protocol works very well for applications such as Wikipedia on Scalaris [65, 73] or the recommendation system Sindaca (see Chapter 13). These systems are designed to support asynchronous collaboration between application's users. The fact that Paxos consensus protocol works with optimistic locking fits well asynchronous collaboration. However, this locking strategy limits the functionality of synchronous collaborative applications such as DeTransDraw, a collaborative drawing tool (see Chapter 18).

DeTransDraw has a shared drawing area where users actively make updates and observe the changes made by other users. If two users make modifications to the same object at the same time, at the end of the their work, when they decide to commit, only one of them will get her changes committed, and the other one will loose everything. Because users are working synchronously, the probability that this happens is much larger than in applications such as Wikipedia or Sindaca. This is why a pessimistic approach with eager locking is needed.

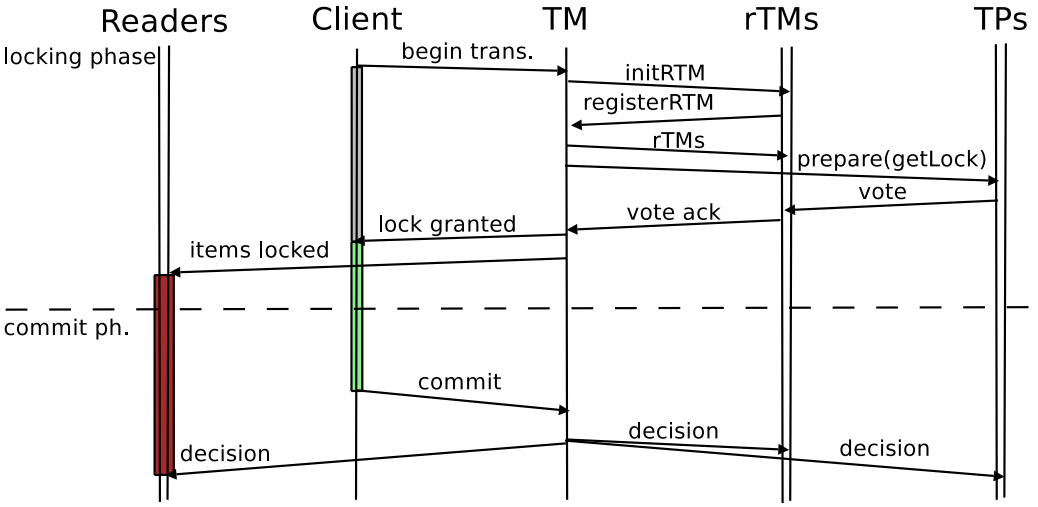


Figure 2.2: Adapted Paxos consensus algorithm with eager locking, no read-phase, and eager notification to readers.

We have adapted Paxos to support eager locking adding a notification mechanism for the registered readers of every shared item. We have implemented this new protocol in Beernet [54] with the possibility of dynamically choosing between the two Paxos protocols. Like that, the application can decide the protocol to be used depending on the functionality that is provided.

Figure 2.2 depicts the adapted protocol with eager locking. A new role has been added: the readers. The read-phase and commit-phase from the original protocol has been replaced by *locking-phase* and *commit-phase*. The read phase disappears because the transaction manager tries eagerly to get the needed to lock to proceed with the transaction. Once the locks are collected, the client is informed of the result, and all other readers are informed too. The goal is to prevent users from trying to start working on items that are already locked. The client of the transaction starts working on the changes on the items as soon as the transaction begins. Starting to work on an item

is actually the trigger of the transaction. The thicker grey line on the client represents the working time of the client. The line turns green if the locks are granted after the voting process is finished. The read line on the readers of the item prevents them to work on the locked items as long as the client has not finished the modifications.

When the user stops making modifications, it triggers the commit-phase. The transaction manager can take the decision immediately because the majority of the votes have been already collected at this stage. The decision is propagated no only to the client, the replicated transaction managers and transaction participants, as in the original Paxos algorithm, but also to the readers. Like this, the readers get notified about the release of the lock and they get the update of the modified items. Because there is no read-phase, it is important that the decision is transmitted together with the new state of the item, and not only a *commit/abort* message.

Using this adapted protocol we have successfully implemented DeTransDraw (see Chapter 18), and we will demonstrate it [56] during the extension period of the project.

2.4 Papers and publications

Transactional DHT Algorithms

> We present a framework for transactional data access on data stored in a DHT. It allows to atomically read and write items and to run distributed transactions consisting of a sequence of read and write operations on the items. Items are symmetrically replicated in order to achieve durability of data stored in the SON. To provide availability of items despite the unavailability of some replicas, operations on items are quorum-based. They make progress as long as a majority of replicas can be accessed. Our framework processes transactions optimistically with an atomic commit protocol that is based on Paxos atomic commit. We present algorithms for the whole framework with an event based notation. Additionally we discuss the problem of lookup inconsistencies and its implications on the one-copy serializability property of the transaction processing in our framework (see Appendix A.6).

Decentralized transactional collaborative drawing

When multiple users collaboratively edit a vector image, avoiding conflicts requires synchronizing exclusive access to the objects of the image. This synchronization needs a true concurrency control algorithm. One of the most common strategy to achieve this synchronization is to use a centralized architecture where a single server becomes the transactional manager. Unfortunately, a central point of control is also a single point of failure. This paper proposes a decentralized architecture based on a peer-to-peer network providing decentralized transactional support with replicated storage. As a consequence, there is a gain in fault-tolerance and the transactional protocol eliminates the problem of network delay improving usability and network transparency. The same result can be applied to text edition and other collaborative editing tasks (see Appendix A.22).

Visualizing Transactional Algorithms for DHTs

The focus of this demonstrator is on the study of algorithms for implementing transactions on peer-to-peer networks. Their visualization contributes to the analysis and testing of the protocols, verifying their tolerance to failures. In particular, we show a DHT with symmetric replication running two-phase commit and the Paxos consensus algorithm. We validate the advantages of Paxos by introducing failures on the transaction managers in both protocols, where Paxos is the only one being able to tolerate such failure (see Appendix A.5).

The Design of a Transactional Key-Value Store Using the Kompics Component Model

We document the architectural design of a transactional key-value store, based on a distributed hash-table (DHT). The DHT is built on top of the Chord$^{\#}$ overlay network. Data items are replicated within the DHT using a key-based replication scheme, namely symmetric replication. Replica agreement for committing transactions is achieved using the Paxos Commit protocol. The system assumes a trusted deployment infrastructure, like a data-center or a set of connected data-centers. We model the system entities as Kompics components. We also describe the messages exchanged by the components of the system as well as enumerate

the possible failure scenarios and how they should be handled (see Appendix A.7).

Chapter 3

D3.2b: Report on replicated storage service over a structured overlay network

3.1 Executive summary

This deliverable reports on the replicated storage service over structured overlay networks. It is based on Chord$^{\#}$ and the transaction framework developed in WP3. It implements so called symmetric replication to increase failure tolerance and includes basic load-balancing schemes based on the self-tuning techniques developed in WP4. This work is complemented by the replicated storage service developed on Beernet based on the relaxed-ring, which also offers a notification layer and an eager locking protocol.

3.2 Contractors contributing to the Deliverable

ZIB(P5) and UCL(P1) have contributed to this deliverable.

ZIB(P5) ZIB has contributed on the design and implementation of the replicated storage service: Scalaris.

UCL(P1) UCL has contributed with the design and implementation of the replicated storage service: P2PS/Beernet.

3.3 Results

The replicated storage service is based on Chord# and the transaction framework developed in WP3. It is completely developed in Erlang and released under an Apache License as Open-Source. The latest release is available at `http://code.google.com/p/scalaris`.

The current release implements so called symmetric replication with a configurable replication degree r. Every item is stored under r keys on the ring and the transactions can be performed on an item as long as the majority of replicas is available.

Some of the self-tuning techniques developed in WP4, especially load balancing, were implemented. In contrast to Chord, Chord# is a non-hashing structured overlay. Whereas Chord can rely on hashing to evenly spread the load over all participating nodes, Chord# has to implement explicit load-balancing.

For future version, we are investigating how to implement replica placement policies. These policies can be used to place replicas nearby the users to decrease access latency. At the same time, the placement policies can be used to host the replicas in different failure domains to minimize the effects of correlated node failures.

We are also investigating, how to design the overlay so that it can autonomously infer the network topology, i.e. how many datacenters are there and where are they. Once the network topology is known, the overlay can adapt itself to the environment to maximize failure tolerance and minimize maintenance cost.

Further details about the service including the user and developer guide are included in the appendix.

Complementary, we have also implemented a symmetric replication storage service on Beernet [54] influenced by the work developed in Scalaris. This work is released as Free Software under X11/MIT License, and is available at `http://beernet.info.ucl.ac.be` . The main difference of this implementation is that it also includes two other protocols: Two-phase commit and Paxos with eager locking.

The reason for including Two-phase commit is purely academic. We have shown already in Chapter 2 and Appendix A.6 that two-phase commit does not provide enough fault-tolerance in peer-to-peer networks because of relying too much on the survival of the transaction manager. This implementation allowed us to evaluate and validate our theoretical analysis, and we demonstrate it in Appendix A.5.

Beernet release does not include techniques for load balancing, but it provides an adaptation of the Paxos consensus algorithm with a pessimistic

approach. This provides eager locking that helps building functionalities for synchronous collaborative applications. This protocol, as reported in more details in Chapter 2, is enriched with a notification layer that sends notifications to a set of readers whenever an item is updated. Two implementations were implemented on this replicated storage service: Sindaca, reported in Chapter 13, and DeTransDraw, reported in Chapter 18.

3.4 Papers and publications

3.4.1 Scalaris: Users and Developers Guide

Florian Schintke, Thorsten Schütt. Scalaris: Users and Developers Guide (see A.8).

3.4.2 Scalaris: Reliable Transactional P2P Key/Value Store

Thorsten Schütt, Florian Schintke, Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. ACM SIGPLAN Erlang Workshop, September 2008 (see A.9).

> We present *Scalaris*, an Erlang implementation of a distributed key/value store. It uses, on top of a structured overlay network, replication for data availability and majority based distributed transactions for data consistency. In combination, this implements the ACID properties on a scalable structured overlay.
>
> By directly mapping the keys to the overlay without hashing, arbitrary key-ranges can be assigned to nodes, thereby allowing a better load-balancing than would be possible with traditional DHTs. Consequently, Scalaris can be tuned for fast data access by taking, e.g. the nodes' geographic location or the regional popularity of certain keys into account. This improves Scalaris' lookup speed in datacenter or cloud computing environments.
>
> Scalaris is implemented in Erlang. We describe the Erlang software architecture, including the transactional Java interface to access Scalaris.
>
> Additionally, we present a generic design pattern to implement a responsive server in Erlang that serializes update operations on a common state, while concurrently performing fast asynchronous read requests on the same state.
>
> As a proof-of-concept we implemented a simplified Wikipedia frontend and attached it to the Scalaris data store backend. Wikipedia is a challenging application. It requires—besides thousands of concurrent read requests per seconds—serialized, consistent write oper ations. For Wikipedia's category and backlink pages,

keys must be consistently changed within transactions. We discuss how these features are implemented in Scalaris and show its performance.

3.4.3 A Scalable, Transactional Data Store for Future Internet Services

Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Seif Haridi.A Scalable, Transactional Data Store for Future Internet Services. EU Future of the Internet Conference, May 2009 (see A.10).

Future Internet services require access to large volumes of dynamically changing data records that are spread across different locations. With thousands or millions of distributed nodes storing the data, node crashes or temporary network failures are normal rather than exceptions and it is therefore important to hide failures from the application.

We suggest to use peer-to-peer (P2P) protocols to provide self-management among peers. However, today's P2P protocols are mostly limited to write-once/read-many data sharing. To extend them beyond the typical file sharing, the support of consistent replication and fast transactions is an important yet missing feature.

We present *Scalaris*, a scalable, distributed key-value store. Scalaris is built on a structured overlay network and uses a distributed transaction protocol. As a proof of concept, we implemented a simple Wikipedia clone with Scalaris which outperforms the public Wikipedia with just a few servers.

3.4.4 Visualizing Transactional Algorithms for DHTs

Boris Mejías, Mikael Högqvist, Peter Van Roy. Demonstrator at the Eighth International IEEE Peer-to-peer Conference, September 2008.

This demonstrator evaluates the advantages of Paxos consensus algorithm over Two-phase commit. It is implemented using the replicated storage service of P2PS/Beernet (see A.5).

3.4.5 DeTransDraw: Decentralized transactional collaborative drawing

Boris Mejías, Jérémie Melchior, Yves Jaradin. Demonstrator at the Internet of Services 2009 Collaboration Meeting.

> This demonstrator is a collaborative drawing tool where users work synchronously, taking advantage of the eager locking mechanism implemented in P2PS/Beernet, together with the notification layer (see A.23).

Chapter 4

D3.3b: Report on simple database query layer for replicated storage service

4.1 Executive summary

This deliverable present the APIs for Scalaris. They allow applications to run arbitrary transactions on the key-value pairs stored in the database. A simple API allows to read and write single key-value pairs. A second API allows to specify custom transactions. This deliverable also presents the API in Mozart to make use of the replicated storage service implemented in P2PS/Beernet.

4.2 Contractors contributing to the Deliverable

ZIB(P5) and UCL(P1) have contributed to this deliverable.

ZIB(P5) ZIB has contributed on the simple database query layer for Scalaris.

UCL(P1) UCL has contributed on Beernet's replicated storage API.

4.3 Results

4.3.1 Scalaris

Scalaris provides two different ways for accessing data stored in the database:

- the Java API and
- the JSON API.

The former is a Java library which can talk directly to Scalaris using the Erlang native network protocol. The latter uses the so called JSON-RPC mechanism to talk to Scalaris. JSON-RPC uses HTTP for the transport layer and JSON to serialize data. There are JSON libraries for most major programming languages.

For Java, the Java API will perform better because the serialization overhead is smaller than for JSON. But JSON is supported by more languages.

The Scalaris class The `de.zib.scalaris.Scalaris` class provides methods for reading and writing values, publishing topics, subscribing to urls and getting a list of subscribers with both erlang objects (`com.ericsson.otp.erlang.OtpErlangObject`) and Java `java.lang.String` objects.

Example:

```
try {
  Scalaris sc = new Scalaris();
  String value = sc.read("key");
} catch (ConnectionException e) {
  System.err.println("read failed: " + e.getMessage());
} catch (TimeoutException e) {
  System.err.println("read failed with timeout: " + e.getMessage());
} catch (UnknownException e) {
  System.err.println("read failed with unknown: " + e.getMessage());
} catch (NotFoundException e) {
  System.err.println("read failed with not found: " + e.getMessage());
}
```

See the `de.zib.scalaris.Scalaris` class documentation for more details.

The Transaction class The `de.zib.scalaris.Transaction` class provides means to realise a scalaris transaction from Java. After starting a transaction, there are methods to read and write values with both erlang objects (`com.ericsson.otp.erlang.OtpErlangObject`) and Java `java.lang.String` objects. The transaction can then be committed, aborted or reset.

Example:

```
try {
  Transaction transaction = new Transaction();
  transaction.start();
  String value = transaction.read("key");
  transaction.write("key", "value");
  transaction.commit();
} catch (ConnectionException e) {
  System.err.println("read failed: " + e.getMessage());
} catch (TimeoutException e) {
  System.err.println("read failed with timeout: " + e.getMessage());
} catch (UnknownException e) {
  System.err.println("read failed with unknown: " + e.getMessage());
} catch (NotFoundException e) {
  System.err.println("read failed with not found: " + e.getMessage());
} catch (TransactionNotFinishedException e) {
  System.out.println("failed: " + e.getMessage());
  return;
}
```

See the `de.zib.scalaris.Transaction` class documentation for more details.

JSON API The JSON API is described in detail in the Scalaris user and developer guide (see A.8).

4.3.2 Beernet

Currently, nodes on Beernet [54] are not created with transactional support by default. Therefore, the API for getting access to the storage service of Beernet begins with the creation of nodes. The notation inherits systax from P2PS [66] because the system is still in migration state. The following code is an example for creating a node with transactional support.

```
import
   Beernet at 'P2PSNode.ozf'
define
   Node = {Beernet.newP2PSNode args(transactions:true)}
```

The most basic support provided by Beernet corresponds to the DHT operations *put* and *get*. This operations do not replicated the value of the item, but they are also part of the implementation of the transactional layer which actually realizes the replication. What follows is an example of how put and get can be used.

```
{Node put(key value)}
Value = {Node get(key $)}
```

To use the transactional layer, the user must write a procedure with one argument, typically named Obj. This argument represents a transactional object, which is an instance of the transaction manager that triggers the transaction. The object receives the operations *read* and *write*, which are almost equivalent to *put* and *get*. The main semantic difference between the operations is that if the transaction is aborted, *write* has no effect on the stored data. And if the transaction succeeds, the value is written at least on the majority of the replicas. Other operations received by the transactional object are *commit* and *abort*, to explicitly trigger those actions on the protocol.

To run the transaction, user must invoke the method $executeTransaction$, which receives three arguments. The procedure containing the operations, a port to receive the outcome of the transaction, and the protocol to be used for running the transaction. Note that at the creation of the node, we did not specify the protocol to be use by every transaction. This is because the protocol can be chosen dynamically, allowing the users to choose the best suitable protocol for every functionality. What follows is a complete example for writing two items with key/value pairs: $hello/"Charlotte"$ and foo/bar. The outcome of the transaction appears on variable $Stream$, which is the output of port $Client$.

```
declare
Stream Client
Trans = proc {$ Obj}
            {Obj write(hello ‘‘Charlotte’’)}
            {Obj write(foo bar)}
            {Obj commit}
        end
{NewPort Stream Client}
{Node executeTransaction(Trans Client paxos)}
if Stream.1 == commit then
   {Browse ‘‘transaction succeeded’’}
end
```

To retrieve the values the user passes a variable which has no value yet. The value is given by the transactional object. The next example shows how to retrieve the values stored under keys *hello* and foo.

```
declare
V1 V2
Trans2 = proc {$ Obj}
            {Obj read(hello V1)}
            {Obj read(foo V2)}
         end
{Node executeTransaction(Trans2 Client paxos)}
{Browse ‘‘for hello I got’’#V1}
{Browse ‘‘for foo I got’’#V2}
```

Note that it is not necessary to catch exceptions using Beernet, because the outcome is reported on the stream of the client's port. If there is a failure on the transaction, the outcome will be abort, and the user will be able to take the corresponding failure recovery action.

4.4 Papers and publications

4.4.1 Scalaris: Users and Developers Guide

Florian Schintke, Thorsten Schütt. Scalaris: Users and Developers Guide (see A.8).

4.4.2 Scalaris Java Interface

Nico Kruber, Thorsten Schütt. Scalaris Java Interface (see A.11).

Chapter 5

D4.1b: Second report on self-configuration support

5.1 Executive summary

The work on self-configuration support during the third year of the Selfman project has covered several inter-related aspects:

1. The formal definition in the Alloy specification language of a reference model for component deployment.

2. The continued development of an Oz/Mozart framework for the construction of self-configurable and self-deployable components, now comprising the FructOz, LactOz and WorkflOz libraries.

3. The development and implementation of self-configuration mechanisms in the Kompics component model, which provides a specialization for event-based programming of the Fractal model, also used by FructOz.

4. The development of an self-configuration mechanisms in the PeerTV platform.

5. The development of an infrastructure for dynamic deployment in a WAN-based overlay network. Part of this infrastructure includes a NAT-resilient gossip-based peer-sampling service, and a protocol for dynamic slicing of resources under application profiling constraints.

The last item will be dealt with in the final deliverable on self-configuration due at the end of the project extension.

5.2 Contractors contributing to the Deliverable

INRIA(P3), Peerialism(P6), and KTH(P2) have contributed to this deliverable.

INRIA(P3) INRIA has worked on providing self-configuration mechanisms in the Fractal component model.

KTH(P2) KTH worked on providing self-configuration mechanisms in the Kompics component model.

Peerialism(P6) Peerialism has worked on providing self-configuration mechanisms for connectivity in the PeerTV platform.

5.3 Results

This section is dedicated to report on the results of year three in this deliverable.

5.3.1 A formal specification of the Fractal deployment model

In deliverable D4.1a, the first report on self-configuration produced in year two of the project, we had described informally a "reference model" for deployment which was broadly supported by the FructOz implementation (also described in D4.1a). We have now developed a formal specification for this programming-language-independent deployment model, developed in the Alloy specification language. This specification refines considerably the informal one, adding in particular crucial concepts to ensure that the model is general yet precise enough to model existing software deployment tools and concepts. The specification builds on the formal specification in Alloy of the Fractal component model, reported in deliverable D2.3b, produced in year two of the project.

We have developed the Fractal deployment model with three specific objectives in mind:

1. To ensure that one could describe, using the model, *heterogeneous* deployment processes, i.e. deployment processes involving executables in different programming languages, relying on different deployment tools at different software layers (e.g. deploying Java applications using OSGI bundles for Java code and RPM packages for the supporting C libraries in a Linux environment).

2. To make explicit relations between different entities involved in a deployment process so as to be able to monitor them and to control them in a self-managed distributed environment, where configuration management extends as much to running components than to the executables they depend upon.

3. To develop a formal model that can ultimately be used to reason about deployment processes, deployment-related functions and abstractions, and to characterize correctness conditions associated to such functions and processes.

Interestingly, although there have been numerous works on software deployment, especially, during the past decade, on architecture-based approaches

to deployment (approaches that exploit software architecture descriptions to drive deployment processes), there are few works that address the above objectives. The three works most closely related to the Fractal deployment model are the Buildbox model [50], the OpenRec framework [86], and the Deployware framework [22]. The OpenRec framework uses the Alloy specification to formally characterize component configurations and to describe reconfiguration operations, however it supports only a non-hierarchical component model, and does not provide an analysis of deployment concepts and operations. The Builbox model provides a formal analysis of key deployment operations by means of a labelled transition system which are close to those described in our deployment model, but the Buildbow model does not consider deployment in a distributed context, and does not provide a modular analysis of the key components involved in a deployment process, a key requirement to address our second objective above. The Deployware framework addresses heterogeneous deployment processes, and relies on a UML meta-model to describe key deployment abstractions, which are modelled, as in our work, as Fractal components. The Deployaware metamodel is not formally specified, however, and provides an insufficiently detailed analysis in comparison to our model to meet our second objective.

The main concepts in the Fractal deployment model belong roughly to three main categories: software unit concepts, that capture the notion of executable software; transformer concepts, that capture basic forms of operations that can be applied to executables in the process of delivering them in a distributed environment; and support concepts, that capture key functionality required to enact the actual deployment of executables in a distributed environment. All these different concepts are specified as Fractal components, which makes the model fully recursive: components that implement a deployment process can themselves be deployed and configured, using the same abstractions and supporting mechanisms.

The software unit concepts are: software unit (SU), software unit system (SUS), and descriptor. A *software unit* corresponds to some software executable. Each software unit belongs primitively to a *software unit system*, i.e. a set of rules, APIs, or tools that define an executable format, and the operations that can be applied to it. For instance, an RPM package [1] corresponds to a software unit, while the set of tools, conventions, format rules, etc that define how RPM packages are formed and how they can be manipulated (mostly implicitly by the RPM tools), constitute a software unit system. Ditto for OSGI bundles (software units) and the OSGI specifications and tools (software unit system) [63]. From the point of view of formal model, one distinguishing feature of a software unit system is that it constitutes a *naming context*, as defined by the Fractal specification. This

allows to freeely combine within the same deployment process, software units from different software unit systems, with no risk of confusion.

A software unit is charecterized primarily by its set of *descriptors*. These are typed interfaces, i.e. access points for interaction with a component, according to the Fractal model, and can be server interfaces or client interfaces. The server descriptors of a software unit define the elements that are provided by a software unit (its exports – which can be as simple as sets of procedures or values), whereas the client descriptors define the elements it depends upon (its imports – which will be provided by other software units). As with general Fractal components, software units can be bound with other components (which may or may not be other software units), and can be composites i.e. contain subcomponents (typically, other software units). Descriptor types abstract constraints attached to imports, such as versioning constraints.

The transformer concepts encompass the following components: launcher, installer, and resolver. A *launcher* takes as input a set of software units and produces a running component. An *installer* takes as input a set of software units and produces other software units. A *resolver* takes as input a set of software units and binds the client and server descriptors of these software units, thus (possibly only partially) resolving the dependencies between the given software units. Deploying an executing component thus typically involves a combination of installers, resolvers, and launchers. The software architecture of an application, as envisaged with the Fractal deployment model, thus encompasses both the relations (binding and containment) between running components, but also the relations between running components and software units that have been transitively involved in their launch.

The last concepts of the Fractal deployment model are: (deployment) node and (software unit) repository. A *node* is an abstraction of a set of computing resources, that are capable of executing components. We require a node to comprise at least a binding factory, to support some form of remote communication with its environment (other computing nodes), and a launcher, to support the creation of executing components from some software units. A *repository* is a store of software units, possibly from different software unit systems, and that provides access to software units through their names or their types.

5.3.2 Self-configurable dynamic architectures in Oz

The FructOz and LactOz Oz/Mozart libraries, described in deliverable D4.1a from year two of the project, have been refined and evaluated in particular in relation to the work which appears closest to it, the Smartfrog system

[29]. They are described in detail in C. Taton PhD thesis (in French). To complement these two libraries, we have developed the WorkflOz library, also developed in Oz/Mozart, and that builds on FructOz. WorkflOz can be understood an Oz-based framework for defining component-based workflows. The main goal of WorkflOz is to enable programmers to write succinct descriptions of complex processes, involving both control flow and data flow, in a compositional and intuitive style. To achieve this goal, WorkflOz provides direct support for all common workflow patterns, which represent widely used, recurring constructs in modern workflow management systems and languages. Unlike most workflow management systems and languages, WorkflOz can be extended with abstractions that capture new patterns, which can be assembled to express arbitrary workflow situations.

WorkflOz is based on the concept of a task, which represents a unit of computation that can be in different states (e.g., executing, terminated). A task has a set of input/output pins through which it receives/emits data values while it is executing. WorkflOz enables composing tasks using operators that capture workflow patterns. For example, consider the operator Seq that captures the sequence pattern. The following expression defines a composite task T3 that represents the sequence of tasks T1 and T2.

```
T3={Seq T1 T2}
```

WorkflOz provides a set of operators similar to Seq (e.g., Sync, Parallel) which can be combined to express complex workflows and corresponding component configurations. For example, the following simple expression creates the component configuration depicted in Figure 5.1:

```
T5={Seq {MultiMerge {Sync T1 T2} T3} T4}
```

Primitive tasks can be created using helper functions, such BasicWrapper. This function takes as input an Oz unary function and creates a primitive task with one input pin and one output pin. When this task is executed, it invokes the Oz function with the input pin value, and emits the result to the output pin. Using BasicWrapper, T1 and T2 could be defined as follows:

```
T1={ BasicWrapper fun {$ X}  X+1 end}
T2={ BasicWrapper fun {$ X}  X*2 end}
```

T3 can be executed or cancelled using the following calls:

```
{Execute T3}
{Cancel T3}
```

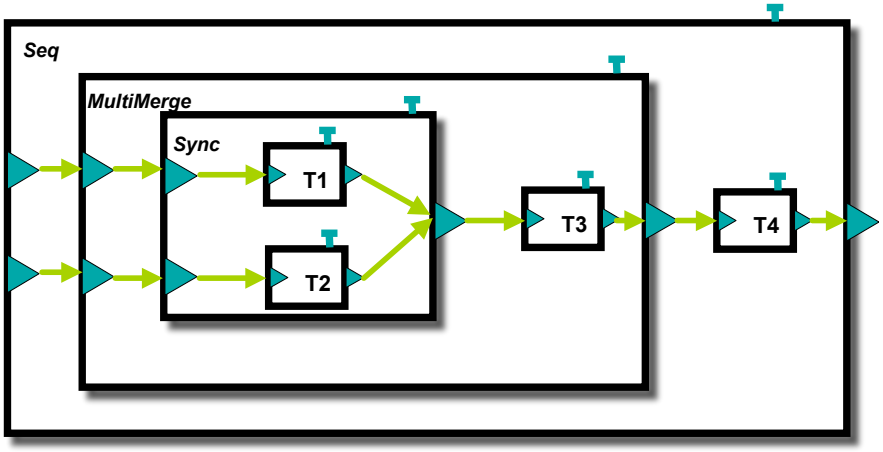


Figure 5.1: A composition of task components.

Pins can be accessed directly to send or read data. For example, one can send the value 8 to the single input pin of T3 (the input pin identifier is 1) through the call:

```
{Send T3 1 8}
```

Tasks are realised as FructOz components with a particular type of controller (i.e., the *task controller*) and specially-marked server and client interfaces representing input and output pins respectively. Composite and primitive tasks correspond to composite and primitive FructOz components. Input and output pins are realised respectively as server and client interfaces. Following the FructOz design, pins are connected through bindings and hold a potentially unbounded list of values (i.e., an Oz stream). Input and output pins are identified using their order (from 1 to the number of input/output pins). When a task is executing, it continuously reads values from its input pins and writes values to its output pins, thus supporting streaming-style data flow. Tasks may be primitive or composite, containing any number of interconnected sub-tasks.

Tasks expose a task controller interface through which their lifecycle is managed. A basic task controller is defined in WorkflOz, in which tasks can

be in one of four states: Ready, Executing, Cancelling, Terminated. The task controller exposes operations to trigger state changes (i.e., execute and cancel), to obtain the current state of the task, and to subscribe/unsubscribe to events corresponding to state changes (e.g., from Cancelling to Terminated). These operations are invoked by components external to the task (e.g., task controllers of containing components). The transitions from Executing and Cancelling to Terminated are triggered by the task itself when its computation or the cancellation process are terminated.

WorkflOz provides operators for creating different types of composite tasks, each type corresponding to a particular workflow pattern. A type of composite task (e.g., sequence type) specifies data- and control-flow dependencies between its sub-tasks. Specifically, it specifies the number of pins of the composite and the binding structure among sub-tasks and the composite. Moreover, it specifies control and lifecycle dependencies between the composite and its sub-tasks (e.g., the composite terminates when any of its subtasks terminate). Those dependencies are implemented by the task controller of the composite task using the controllers of the sub-tasks. In the previous sequence example, the task controller of T3 uses the task control interface of T1 to register for state change events; when a termination event is received, the controller triggers the execution of T2 through the T2 task controller.

Extending WorkflOz with support for a new pattern involves (1) defining the control- and data-flow semantics of the corresponding composite task, and (2) implementing the associated task controller and any supporting functionality. This implementation is facilitated by utility classes for managing the task lifecycle, sending events, and accessing and connecting pins.

5.3.3 Self-configuration mechanisms in the Kompics component model

Here we describe the Kompics mechanisms that enable the construction of self-configuring systems. Other details of the Kompics component model are provided in appendixes A.12 and A.13 which contain published papers describing Kompics. An initial Kompics programming manual is included in Appendix A.14.

Kompics components communicate with other components in their environment by asynchronously sending and receiving events through bidirectional typed ports. A simple “Hello World” example is given in Figure 5.2.

Kompics components are configured within the implementation language. Configuration operations are fundamentally part of the model. They allow

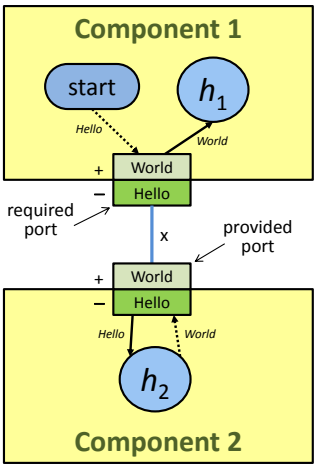


Figure 5.2: Component1 *requires* a HelloWorld port which allows it to send out Hello events and receive World events. Component2 *provides* a HelloWorld port which allows it to receive Hello events and send out World events. Since their HelloWorld ports are connected by channel x, Component1 and Component2 can communicate. The *start* handler of Component1 triggers a Hello event on its HelloWorld port. This Hello event is received by Component2 and handled by its h_2 handler which triggers a World event. This World event is received by Component1 and handled by its h_1 handler.

the architecture of the system to evolve dynamically. A few example of component architectures are given in Figure 5.3. The basic component configuration operations are: `create`, `destroy`, `connect`, `disconnect`, `subscribe`, and `unsubscribe`.

The `create` operation takes a component definition and creates a new component instance, as a child component of the component invoking `create`. This enables the dynamic creation of components inside a running system, in the same way as new objects are created in an object-oriented language runtime. When a new component is created from a component definition, the declared required and provided ports are automatically created, and they are visible in the scope of the creating component (the parent component). None of the new component's ports are connected immediately after the component is instantiated. At the time of instantiation, the new component's constructor is executed. This may lead to the recursive creation of subcomponents inside the newly created component.

Component instances are explicitly destroyed using the `destroy` operation. This is used by a parent component to destroy one of its children components. All external ports of the child component need to be disconnected and the parent component must have no event handler subscribed to any of these ports. The destruction of the child component is disallowed if these conditions are not met. The destruction of a component leads to the recursive destruction of all subcomponents of the destroyed component.

For two sibling components to be able to communicate, they must have compatible ports of opposite polarity, and these ports must be connected by a channel. For example, in the left architecture of Figure 5.3, components C1 and C2 can communicate through their ports of type Q, because these ports are connected by channel x1. Immediately after the Main component has

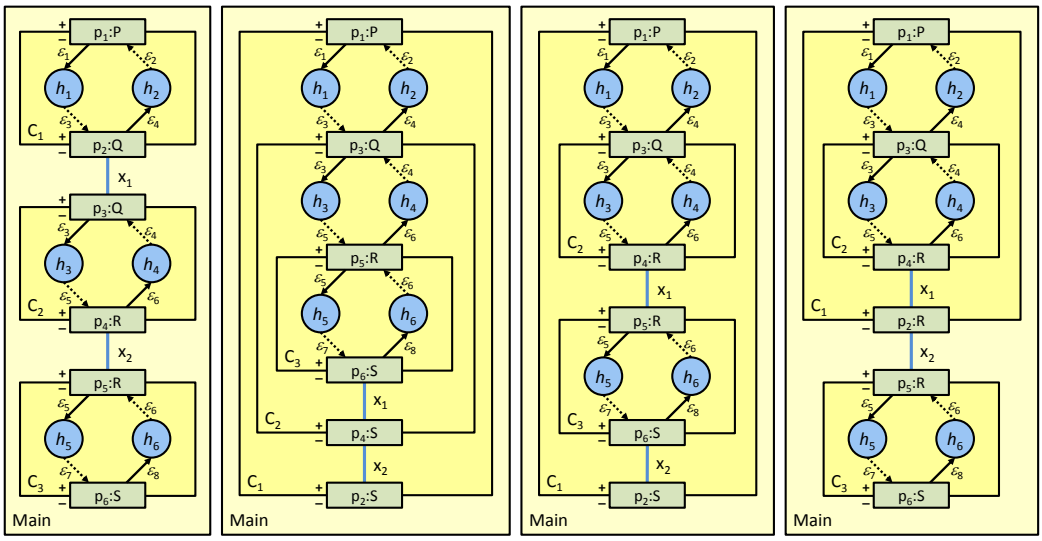


Figure 5.3: Examples component architectures.

created C1 and C2, they are disconnected. Main would execute a `connect` operation to connect ports p2 and p3, and hence enable the communication between C1 and C2. A `connect` operation can be used to connect both the compatible ports of two sibling components (as in the example above), and a port of a child component with a compatible port of the parent component (for example ports p4 and p2 in the architecture on the right of Figure 5.3). A `disconnect` operation disconnects two connected ports.

To subscribe one of its event handlers to a port visible within its encapsulation scope, a component executes a `subscribe` operation. A `subscribe` operation for port `p` and handler h is allowed only if the type of events handled by h are flowing through port `p` in the direction of the component (towards the component). A component executes an `unsubscribe` operation to unsubscribe an event handler from a port to which it was previously subscribed.

The `create`, `destroy`, `connect`, `disconnect`, `subscribe`, and `unsubscribe` operations enable components to dynamically change the architecture of a system by adding and/or removing components and re-wiring their interaction. These operations provide basic primitives for implementing self-configurating Kompics systems.

5.3.4 Self-configuration mechanisms in PeerTV

The main Self-Configuration aspect in the PeerTV platform is the one entitled with the task of coordinating the connection establishment process

between peers. This task is particularly challenging as peers in consumer networks present many connectivity limitations due mainly to the presence of Network Address Translators and Corporate Firewalls. We hereby provide a short summary about the work done by Peerialism(P6)to tackle this issue.

Dealing with Network Address Translators (NATs) and Firewalls is nowadays an essential need for any P2P application. The techniques used to deal with those have been more or less "coined" and there are several widely-used methods[23][?]. Some of them are rather a defacto standard like STUN[?], TURN[69], ICE[68]. In the context of our a P2P live video streaming application PeerTV, we are mainly concerned with media streaming using UDP and therefore the scope of this paper is UDP NAT traversal. Moreover, we are strictly interested in solutions that do not use relay, such as TURN for instance, due to the high bandwidth requirements of video streaming. We have found lots of of previous work on the subject that aims to answer the following question: *For every t in the set of NAT types $\mathcal{T}$, which s in the set of traversal strategies $\mathcal{S}$ should be used to traverse t? The answer is of the form $f : \mathcal{T} \rightarrow \mathcal{S}$.* i.e. the following is an example with a couple of types f : { Simple Hole Punching, Port-Prediction } $\rightarrow$ { Full-Cone, Symmetric} [95]. However, the point which we found not gaining enough attention is that the presence of a feasible traversal technique that enables two peers behind NAT to communicate depends on the "combination" of the NAT types and not on the type of each peer separately. Thus, the question should be: *"Given 2 peers p_a and p_b with respective NAT types $t(p_a)$ and $t(p_b)$, which traversal strategy s is needed for p_1 and p_2 to talk? The answer is of the form $f : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{S}$"*, i.e we need to analyze traversable combinations rather than traversable types.

Most works contain a few examples of combinations for explanation purposes [95][?]. However, we have failed to find any comprehensive analysis that states, for every possible combination of NAT types, whether *direct* (i.e. with no relay) connectivity is possible and how. The analysis is more topical given that NAT community is switching from the classical set of NAT types $\mathcal{T}_{classic}$ = { Full-Cone, Restricted-Cone, Port-Restricted, Symmetric} [?] to a more elaborate set that defines a NAT type by a combination of three different policies, namely, port mapping, port allocation and port filtering [?].

In this work, we have conducted a comprehensive analysis of what combinations of NAT types are traversable. We have shown that using a semi-formal reasoning that covers all cases and we provided a slightly augmented versions of the well-known traversal techniques and shown which ones are applicable for which combinations.We have shown also that about 80% of all possible combinations are traversable.

5.4 Papers and publications

The work reported in this deliverable has been published as follows:

1. Cosmin Arad and Seif Haridi. Practical Protocol Composition, Encapsulation, and Sharing in Kompics. In the *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW '08)*. IEEE Computer Society, October 2008.

2. Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, Simulating, and Deploying Peer-to-Peer Systems using the Kompics Component Model. In the *Fourth International Conference on COMmunication System software and middlewaRE (COMSWARE '09)*. ACM SIGSOFT, ACM SIGAPP, June 2009.

3. Roberto Roverso, Sameh El-Ansary, and Seif Haridi NATCracker: NAT Combinations Matter To appear in proceedings of *The 18th International Conference on Computer Communications and Networks*, IEEE Communications Society, August 2009.

4. Anne-Marie Kermarrec, Alessio Pace, Vivien Quéma and Valerio Schiavoni. NAT-resilient Gossip Peer Sampling. 29th International Conference on Distributed Computing Systems (ICDCS), IEEE Computer Society, June 2009.

Chapter 6

D4.1c: Self-configuration support (software)

6.1 Executive summary

In this deliverable, we present the Oz implementation of the WorkflOz library described in Chapter 5. WorkflOz is built using the FructOz library, which was described in deliverable D4.1a, produced in year 2 of the project.

Additional software for self-configuration support is available with the Kompics component model implementation, described in Chapters 5 and 7.

The Java implementation of the Fractal deployment model described in Chapter 5 is not yet completed and is not reported here.

6.2 Contractors contributing to the Deliverable

INRIA(P3)has contributed to this deliverable.

INRIA(P3) has developed the WorkflOz library, which builds on the FructOz library that was developed during year two of the project and that is described in deliverable D4.1a.

6.3 The WorkflOz library

6.3.1 Design

WorkflOz offers a framework for composing tasks and a functional language for creating task configurations.

A task is a FructOz software component representing a unit of computation. Tasks have a set of input and output pins, realised respectively as server and client interfaces. Following the FructOz design, pins are connected through bindings and hold a potentially unbounded list of values (i.e., an Oz stream). Input and output pins are identified using their order (from 1 to the number of input/output pins). When a task is executing, it continuously reads values from its input pins and writes values to its output pins, thus supporting streaming-style data flow. Tasks may be primitive or composite, containing any number of interconnected sub-tasks.

Tasks expose a task controller interface through which their lifecycle is managed. Tasks can be in one of four states: Ready, Executing, Cancelling, Terminated. The task controller exposes operations to trigger state changes (i.e., execute and cancel), to obtain the current state of the task, and to subscribe/unsubscribe to events corresponding to state changes (e.g., from Cancelling to Terminated). These operations are invoked by components external to the task (e.g., task controllers of containing components). The transitions from Executing and Cancelling to Terminated are triggered by the task itself when its computation or the cancellation process are terminated.

WorkflOz provides operators for creating different types of composite tasks, each type corresponding to a particular workflow pattern. A type of composite task (e.g., sequence type) specifies data- and control-flow dependencies between its sub-tasks. Specifically, it specifies the number of pins of the composite and the binding structure among sub-tasks and the composite. Moreover, it specifies control and lifecycle dependencies between the composite and its sub-tasks (e.g., the composite terminates when any of its subtasks terminate). Those dependencies are implemented by the task controller of the composite task using the controllers of the sub-tasks.

6.3.2 Patterns

This section examines how WorkflOz supports a number of common workflow patterns, using Oz functions.

The signatures of the functions are shown as:

```
{Seq Ts ?T}
```

The last argument (here T) is the output of the function. The types of the arguments are indicated by their name, using the following abbreviations:

- I Integer
- T Task
- Ts List of tasks
- BF Nullary function that evaluates to a boolean
- TF Nullary function that evaluates to a task
- M Milestone

We use indices such as T1 and T2 to denote several occurrences of arguments of the same type. For each operator, we detail its syntax, and its effects in terms of data flow (how inputs and outputs of composed tasks are connected), and control flow (how execution proceeds among the composed tasks).

Sequence

- *Syntax*

  ```
  {Seq Ts ?T}
  ```

- *Data flow.* The tasks in the list Ts are bound sequentially with the outputs of one bound to the inputs of the following. The created task T is bound to the first and last sub-task.
- *Control flow.* When a sub-task terminates (i.e., enters the Terminated state), the next sub-task is executed. Task T terminates when the last sub-task has terminated.

Parallel split

- *Syntax*

  ```
  {Parallel Ts ?T}
  ```

- *Data flow.* The task T has a pin corresponding to each pin of the sub-tasks; all tasks in Ts are bound directly to T.
- *Control flow.* The subtasks are executed in parallel. Task T terminates when all the sub-tasks have started executing.

Synchronisation

- *Syntax*

  ```
  {Sync Ts ?T}
  ```

- *Data flow.* The task T has an input pin corresponding to each input pin of the sub-tasks. It has an output pin corresponding to all the output pins of the sub-tasks of the same order (i.e., all n-th output pins of tasks correspond to the n-th output pin of T). In other words, the outputs of the tasks in Ts are merged into the output of T.

- *Control flow.* The sub-tasks are executed in parallel. Task T terminates when all the sub-tasks have terminated.

Exclusive choice

- *Syntax*

  ```
  {ExChoice BF T1 T2 ?T}
  ```

- *Data flow.* The input pins of T1 and T2 are merged into the input/output pins of the composite (i.e., the n-th input pin of composite is linked to both the n-th input pin of T1 and T2), and similarly for the output pins.

- *Control flow.* The boolean function BF is evaluated at execution time, and depending on the result, one of the two tasks is executed. Task T terminates when both T1 and T2 have terminated.

Simple merge The pattern is supported implicitly by combing ExChoice with Seq. For example, the following expression describes a simple merge between T1 and T2:

```
{Seq {ExChoice F T1 T2} T3}
```

When the tasks to be merged can run in parallel, the MultiMerge operator should be used (see next).

Multi-choice

- *Syntax*

  ```
  {MultiChoice BFTs ?T}
  ```

  where BFTs is a list of pairs composed of a nullary boolean function and a task.

- *Data flow.* The input pins of the tasks are merged into the input pins of the composite. Their output pins are linked to separate output pins of the composite.

- *Control flow.* When T executes, it triggers all the tasks whose corresponding function evaluates to true.

Synchronising merge No new construct is required for supporting this pattern. It can be expressed by combining Seq and MultiChoice. For example, the following expression describes a synchronising merge between T1 and T2:

```
{Seq {MultiChoice [F1#T1 F2#T2]} T3}
```

Multi-merge

- *Syntax*

  ```
  {MultiMerge T1 T2 ?T}}
  ```

- *Data flow.* The tasks T1 and T2 are bound to each other and to T, similarly to the sequence pattern.

- *Control flow.* If any of the sub-tasks of T1 terminates, then task T schedules execution of T2. Scheduled executions of T2 are performed atomically. Task T terminates when T1 has terminated and all scheduled executions of T2 have terminated. The MultiMerge operator is designed to be combined with a T1 created by a split pattern, such as Parallel or MultipleChoice. For example, in MultiMerge Parallel [T1 T2] T3, T3 will be triggered two times, after T1 terminates and after T2 terminates.

Structured Discriminator

- *Syntax*

```
{StructDiscriminator T1 T2 ?T}
```

- *Data flow.* T1 and T2 are bound to each other and to T similarly to the sequence pattern.
- *Control flow.* When the first of the sub-tasks of T1 terminates, task T executes T2 and waits for the termination of all remaining sub-tasks of T1. Task T terminates when T2 and all the sub-tasks of T1 have terminated. Similarly to MultiMerge, this operator is designed to be combined with a split pattern such as Parallel or MultipleChoice.

Multiple instances without synchronisation

- *Syntax*

```
{MultiInst  TF I ?T}
```

- *Data flow.* The created task T contains I task instances, created by invoking function TF. The single input pin of T receives lists, whose elements are distributed in a round-robin fashion among the input pins of the I sub-tasks. The values from their output pins are collected in a list which is passed to the single output pin of T.
- *Control flow.* T terminates when the I sub-tasks have started executing.

Multiple instances with a priori design-time knowledge

- *Syntax*

```
{MultiInstD  TF I ?T}
```

- *Data flow.* The created task T contains I task instances, created by invoking function TF. The single input pin of T receives lists, whose elements are distributed in a round-robin fashion among the input pins of the I sub-tasks. The values from their output pins are collected in a list which is passed to the single output pin of T.
- *Control flow.* T terminates when the I sub-tasks have terminated.

Multiple instances with a priori run-time knowledge Supported by the same operator MultiInstD, seen previously.

Multiple Instances without a priori run-time knowledge

- *Syntax*

```
{MultiInstNR  TF I ?T}
```

- *Data flow.* The created task T contains I task instances, created by invoking function TF. T has two input pins. The first pin receives lists whose elements are distributed to the input pins of the I sub- tasks. The second pin receives values that are sent to dynamically created task instances; every value results in an additional instance. T has an output pin that collects to a list the values from the output pins of the (initial and any additional) sub-tasks.
- *Control flow.* T terminates when all (initial and additional) sub-tasks have terminated.

Deferred Choice

- *Syntax*

```
{DeferredChoice Ts ?T}
```

- *Data flow.* The input pins of the tasks in Ts are merged into the input pins of the created task. Task T has also a special input pin which receives the identifier of the task to be activated. The output pins of the tasks in Ts are merged into the output pins of T.
- *Control flow.* When task T is executed, it waits for a value k in its special pin, and then executes the corresponding task (i.e., the k-th task in the list). The composite task terminates when the selected sub-task has terminated.

Interleaved parallel routing

- *Syntax*

```
{InPar Ts TTs ?T}
```

where TTs is a list of pairs of tasks and specifies the partial order of Ts tasks.

- *Data flow.* The created task T has a pin corresponding to each pin of the sub-tasks.
- *Control flow.* The tasks in Ts are executed one at a time in an order that conforms to the partial order. T terminates when all sub-tasks have terminated.

Milestone This pattern is supported by the following functions which create and use milestone entities.

- *Syntax*

```
{CreateMilestone ?M}
{MilestoneStart M T1 ?T}
{MilestoneEnd M T1 ?T}
{Milestone M T1 ?T}
```

- *Data flow.* The tasks created by MilestoneStart, MilestoneEnd, and Milestone have the same number and type of pins as their arguments and the corresponding pins are connected together (i.e., they are simple wrappers that do not affect data flow).
- *Control flow.* A milestone entity can be in two states: set and cleared. The functions MilestoneStart and MilestoneEnd set and clear the milestone M when their enclosed task (T1) terminates. When executed, the task created by Milestone checks its argument M and, if M is set, executes T1.

Cancel activity / case Supported for all tasks through their task controller, which can be invoked through the function Cancel T. Cancelling a task cancels also its sub-tasks.

Arbitrary cycles Arbitrary cycles are supported using the Call function.

- *Syntax*

```
{Call TF ?T}
```

where TF is a function that evaluates to a task.

The Call function creates a placeholder task T. When T is executed, it invokes the TF function to obtain a new task Tnew, binds T to Tnew (effectively replacing T by Tnew), and delegates execution to Tnew. Invocations of Call are useful for repeating workflow fragments captured by the TF function.

For example, the arbitrary cycle example in Figure 6.1 can be modelled as follows (A, B, C, D, E, F, End are tasks):

```
FD=fun {$} {Seq D {Call FE}} end
FE=fun {$}  {Seq E {ExChoice BF2 End {Seq F {Call FD}}}} end
Task={Seq A {ExChoice BF1 {Seq B {Call FD}}{Seq C {Call FE}}}}
```

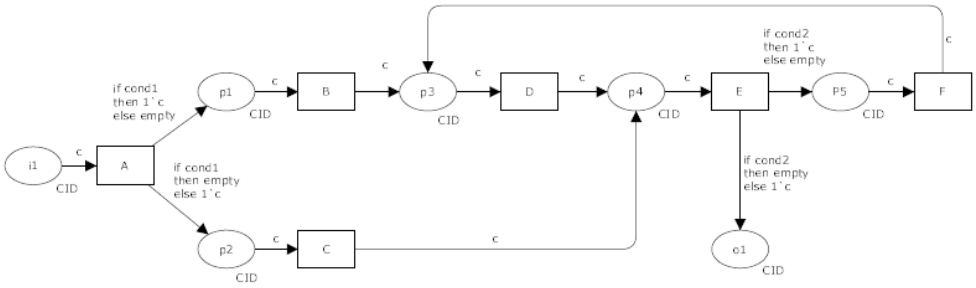


Figure 6.1: An example of cycle in task control flow

Chapter 7

D4.2b: Second report on self-healing support

7.1 Executive summary

This deliverable reports on work done to provide self-healing support. The work on self-healing has been done on two fronts. First, self-healing mechanisms have been developed to tolerate network partitions in structured overlay networks. Second, self-healing mechanisms have been explored in component models, specifically in Kompics and Fractal.

7.2 Contractors contributing to the Deliverable

KTH(P2) and INRIA(P3) have contributed to this deliverable.

KTH(P2) KTH worked on two fronts: (1) providing self-healing functionality in structured overlay networks to heal from physical network partitions and mergers, and (2) providing self-healing mechanisms in the Kompics component model.

INRIA(P3) INRIA enhanced their work on their component model, Fractal, to provide self-healing components. The work on components, Fractal and Kompics, was done in collaboration between INRIA and KTH.

7.3 Results

This section is dedicated to report on the results of year three in this deliverable. We describe the work in three parts, the work done on the self-healing of structured overlay networks, the work done on self-healing mechanisms in the Kompics component model, and the Fractal component model.

7.3.1 Self-healing in structured overlay networks

In our work for this deliverable, we have motivated that handling underlying network partitions and mergers is a core requirement for structured overlays. We argue that since fault-tolerance, self-healing, scalability and self-management are the basic properties of overlays, they should tolerate network partitions and mergers.

Our contribution is two-fold. First, we propose a mechanism for detecting a scenario where a partition occurred and later, the underlying network merged. Second, we propose two algorithms for merging overlays, *simple ring unification* and *gossip-based ring unification*. In our solution, one the partition and merger is automatically detected, the merging algorithm is invoked. Thus, the structured overlay network becomes self-healing under network partitions and mergers.

Simple ring unification is a low-cost solution with respect to the number of messages sent (message complexity), yet it suffers from two problems: (1) slow convergence time ($O(N)$ time for a network size of N), and (2) less robustness to churn.

Gossip-based ring unification addresses both short-comings of simple ring unification, i.e. it has a high convergence rate ($O(\log N)$ time for a network size of N), and is robust to churn, yet it is a high-cost solution in terms of message complexity. In our solution, we provide a *fanout* parameter that can be used to control the trade-off between message and time complexity in gossip-based ring unification. A comparison of the two algorithms is given in Table 7.3.1.

We have evaluated both algorithms extensively. Furthermore, we compared our solution to a self-stabilizing algorithm presented by Shaker *et. al.* [78], as, a self-stabilizing algorithm can be used to merge multiple overlays. The comparison is presented in our journal paper [77], which also appears as Appendix A.2. The comparison shows that our solution consumes lesser time and messages compared to the self-stabilizing algorithm.

Lookups made after the merge is complete perform normally. An interesting issue is the behaviour of lookups made during the merger of the overlays. Such lookups may not always succeed in finding the related data item, and

thus, some keys may temporarily appear unavailable. A trivial solution to this problem is that when n learns that the key is currently unavailable, it retries the lookup after a while.

	Simple Ring Unification	Gossip-based Ring Unification
Time Complexity	High	Low
Message Complexity	Low	High
Resilience to Churn	Low	High

Table 7.1: Comparison of Simple Ring Unification and Gossip-based Ring Unification.

7.3.2 Self-healing mechanisms in the Kompics component model

Here we describe the Kompics mechanisms that enable the construction of self-healing systems. Other details of the Kompics component model are provided in appendixes A.12 and A.13 which contain published papers describing Kompics. An initial Kompics programming manual is included in Appendix A.14.

Kompics components communicate with other components in their environment by asynchronously sending and receiving events through bidirectional typed ports. A simple "Hello World" example is given in Figure 7.1.

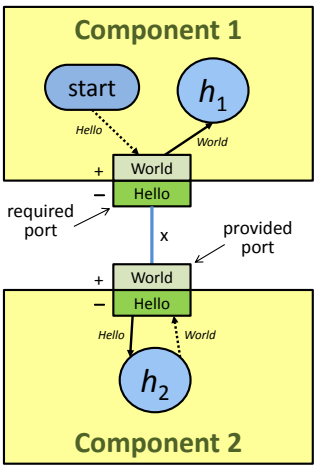


Figure 7.1: Component1 *requires* a HelloWorld port which allows it to send out Hello events and receive World events. Component2 *provides* a HelloWorld port which allows it to receive Hello events and send out World events. Since their HelloWorld ports are connected by channel x, Component1 and Component2 can communicate. The *start* handler of Component1 triggers a Hello event on its HelloWorld port. This Hello event is received by Component2 and handled by its h_2 handler which triggers a World event. This World event is received by Component1 and handled by its h_1 handler.

In addition to any functional ports, Kompics components have a control port. The control port is used to transmit life-cycle commands to the component, as well as to notify the parent component of uncaught software

faults that occurred inside the component during event handler execution. We extend our "HelloWorld" example in Figure 7.2 to show the control ports of the two components.

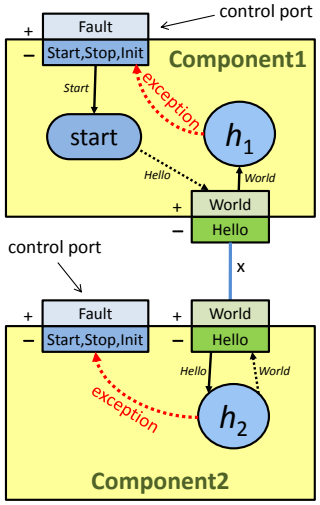


Figure 7.2: Each Kompics component has a control port. A component can use the control port to subscribe to life-cycle events like Start, Stop, and Init. This is exemplified in Component1 which subscribed its *start* handler to the control port. The control port of a component is also used by the runtime system to report uncaught exceptions that occur during the execution of an event handler. An uncaught exception occurring in handler h_1 of Component2, is caught by the runtime system and wrapped into a Fault event which is triggered on Component2's control port. The parent of Component2 (not shown) can handle the Fault event.

Kompics components form a containment hierarchy rooted at a Main component. A few examples of composite components are given in Figure 7.3. A component can subscribe a fault handler to the control channel of its children components. Thus, whenever a fault occurs in a child component, the parent component can handle it and take some repairing action. A parent component needs not subscribe a fault handler for its children components. In that case, the default behavior is to forward faults up the containment hierarchy, i.e., the runtime system traverses the containment hierarchy until it finds a parent component with a subscribed fault handler. If no such component is found and the traversal reached the Main component, a system-default fault handler handles the Fault by printing it out to the standard output and halting the computation.

This mechanism enables any component to act as a supervisor for its children components, and take healing actions once faults are detected. The supervisor component can manage/reconfigure the faulty component, for example by replacing it with a new instance. Hence this fault management mechanism provides basic primitives for implementing self-healing Kompics systems. Given any component, a self-healing wrapper component can be configured around it.

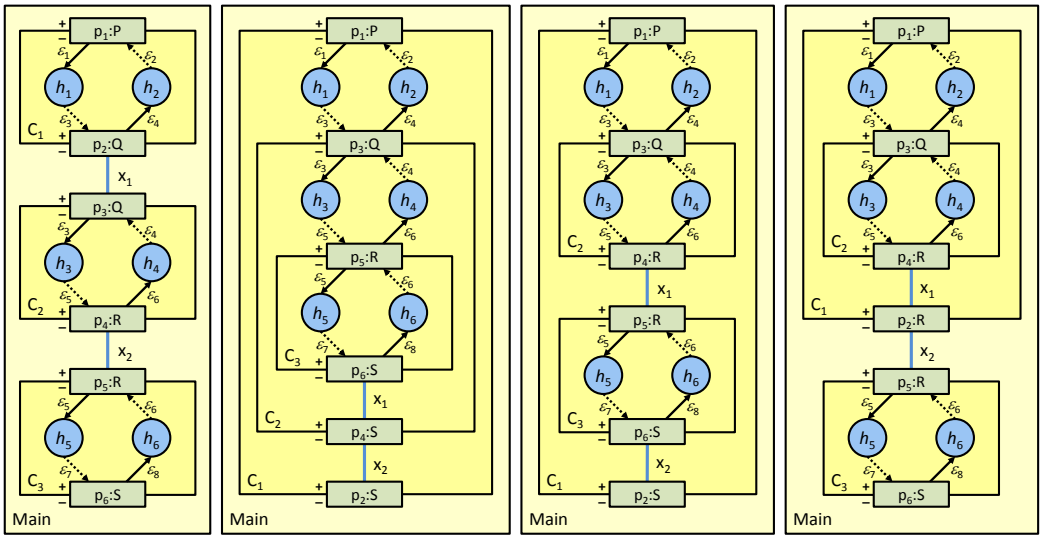


Figure 7.3: Examples of hierarchies of composite components rooted at Main.

7.3.3 Self-healing mechanisms in cluster systems using Fractal

As part of the work on this deliverable, we have developed using the Fractal model an architecture for the provision of self-repair mechanisms in cluster-size systems, such as Web application servers or file system servers. This architecture builds on the Jade framework developed at INRIA for the construction of self-managed systems as explicit control loop structures acting on component-based architectures. The original aspect of this work is the fact the self-repair capabilities extends to the management subsystem itself, through the use of replication. The uniform atomic broadcast replication protocol we adopted limits the use of our design to systems of a cluster size (extending our design to larger scale, WAN-based environments, is for further study). We describe here the main ideas behind our design for self-repair in cluster systems.

In our approach, a managed system appears as a distributed Fractal component structure. Control interfaces of components corresponding to managed elements (either developed natively with Fractal, or wrapped by Fractal components in the case of legacy systems) provide the required effectors to act on a managed system. Control interfaces of managed elements (or interfaces on components implementing suitable event detection protocols) provide the required sensors to monitor a managed system. A managed system in our approach primarily appears as a collection of components, called *nodes* that correspond to abstractions of physical machines (typically, a set of PCs in

a cluster environment). Managed elements executing on these physical machines are thus subcomponents of nodes. The set of nodes together with their subcomponents constitutes a management domain, i.e. a set of entities under the control of a single management authority and asociated set of policies.

To support repair of a managed system requires in our control-based approach to set up a control loop with feedback that monitors managed elements, analyzes reported state changes, plans a response to these changes, and executes it. In the case of repair, which requires recovering from failures of managed elements, some knowledge must be maintained of the runtime configuration of the system that persists even in presence of failures. We call *System Map* the knowledge of a managed system runtime configuration that needs to be maintained for the purpose of self-repair. The System Map is actually maintained and exploited (for analysis and planning purposes) by a set of components called *manager* components, which are responsible for the analysis of observations on the managed system, the planning and (ultimately) execution of management operations in response to observations and according to management objectives and policies. Manager components execute on a subset of nodes in a managed system, which are called *manager nodes*. Manager nodes are an integral part of the managed system in our design, which is a necessary condition for self-repair. We call *management subsystem* the set of manager nodes together with their manager subcomponents. The System Map is actually an active structure that serves as an intermediate between manager components and managed components, and that carries out reconfiguration operations originating with manager components. Thus a repair control loop in our framework, is constructed as a distributed software architecture which connects managed components to manager components and the System Map.

Constructing the System Map of a managed system litterally corresponds in our approach to building an explicit and causally connected representation of the managed system. A self-repairable system is thus a doubly reflective system:

- Each managed system element, as a Fractal component, provides introspection and meta-level capabilities through its control interfaces.
- The System Map provides introspection and meta-level capabilities to control a managed system as a whole.

Support for self-repair in our approach can be understood as constructing logically a hierarchy of meta-level feedback loops to ensure that any failure occurring in the set of manager nodes can be properly recovered, or, equivalently, as a way to ensure that the System Map (i) is indeed made persistent

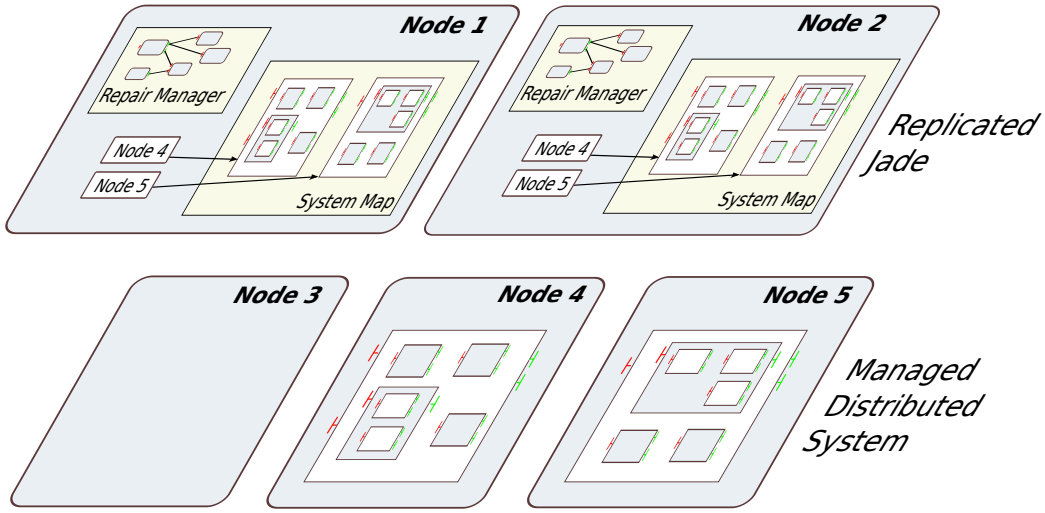


Figure 7.4: A managed system and its replicated management subsystem.

even in presence of failures among manager nodes, and (ii) properly reflects the whole managed system runtime configuration, including the configuration of manager nodes. This is realized by replicating components including manager nodes, and through an appropriate organization of the System Map.

Figure 7.4 illustrates our Jade self-repair architecture. In the figure, nodes 4 and 5 are managed nodes, nodes 1 and 2 form the management subsystem (node 3 is a spare node that can be used for repairing the system in case of a node failure in the managed system or the management subsystem). Nodes 1 and 2 are actively replicated, and each maintains a copy of the System Map, which itself reflects the architecture of the overall system.

This self-repair architecture has been used to demonstrate self-repair in JEE Web servers. We have also applied it to the repair of NFS servers in a cluster, demonstrating that our design is effective and applicable to legacy systems as well as native Fractal-based systems.

7.4 Papers and publications

The work described in this deliverable has been published as follows:

1. Tallat M. Shafaat, Ali Ghodsi, Seif Haridi. Dealing with Network Partitions in Structured Overlay Networks. *Journal of Peer-to-Peer Networking and Applications (PPNA),* 2009 (To appear). DOI: 10.1007/s12083-009-0037-7

2. Tallat M. Shafaat, Ali Ghodsi, Seif Haridi. Managing Network Partitions in Structured P2P Networks. Book Chapter in X. Shen, H. Yu, J. Buford, and M. Akon, editors, *Handbook of Peer-to-Peer Networking.* Springer-Verlag, July 2009.

3. Tallat M. Shafaat. Dealing with Network Partitions in Structured Overlay Networks. *Licentiate Thesis - KTH, The Royal Institute of Technology, Sweden.* ISBN 978-91-7415-290-6. May, 2009.

4. Tallat M. Shafaat, Ali Ghodsi, Seif Haridi. Handling Network Partitions and Mergers in Structured Overlay Networks. In *Proceedings of the 7th International Conference on Peer-to-Peer Computing (P2P'07),* pages 132–139. IEEE Computer Society, September 2007.

5. Cosmin Arad and Seif Haridi. Practical Protocol Composition, Encapsulation, and Sharing in Kompics. In the *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW '08).* IEEE Computer Society, October 2008.

6. Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, Simulating, and Deploying Peer-to-Peer Systems using the Kompics Component Model. In the *Fourth International Conference on COMmunication System software and middlewaRE (COMSWARE '09).* ACM SIGSOFT, ACM SIGAPP, June 2009.

7. Fabienne Boyer, Noel de Palma, Olivier Gruber, Sylvain Sicard, and Jean-Bernard Stefani. A self-repair architecture for cluster systems. To appear in Architecting Dependable Systems 6. R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, M. H. ter Beek eds. Springer,2009.

Chapter 8

D4.2c: Self-healing support (software)

8.1 Executive summary

In this deliverable, we present a software implementation of the structured overlay network self-healing algorithms presented in D4.2b (see Chapter 7).

The algorithms presented in Chapter 7 for self-healing in structured overlay networks have been designed for healing from multiple overlays into one. A typical scenario being a physical network partition and later, merger. The algorithms had to implemented, tested and evaluated. Unfortunately, Off-the-shelf simulators, like PeerSim [4] and P2PSim [3], do not provide functionality to simulate multiple networks. Thus, in this deliverable, we implemented a discrete event-based simulator with the functionality of simulating multiple physical networks.

8.2 Contractors contributing to the Deliverable

KTH(P2)has contributed to this deliverable.

KTH(P2) KTH proposed algorithms for self-healing in structured overlay networks, presented in D4.2b. These algorithms were implemented in a discrete event-based simulator that was implemented by KTH, called *SicSim*.

8.3 The SicSim Architecture

SicSim allows for various types of overlays to be implemented. Currently, we have implementations of DKS [27], Chord [81], Chord# [74], Kademlia [52] and Cyclon [85]. The simulator has a separate layer simulating the network. A physical partition in the network can be simulated by this layer by dividing the nodes into different components based on user supplied parameters. The messages sent from one component are not allowed to reach another component. Once the user wants to merge the physical network, all the components are merged into one. Thus, all nodes can send messages to all other nodes.

Further functionality of simulating network bandwidth has also been added to SicSim. This is useful to simulate applications that depend on network bandwidth, such as live streaming.

The SicSim discrete event-based simulator is available as a public release at *http://www.sics.se/ amir/sicsim*.

The main modules of SicSim are as follow:

- **Peer:** A peer models a node of the network and can be of any customized type.
- **Link:** Each peer connects to the system through a link. The links are referred to as physical link in Figure 8.3.
- **Core:** Each message, which is sent from one peer to another, is passed through core network. The core modules is aimed to model Internet, while abstracting away from the underlying layers of the network.
- **Network:** Network contains all the peers in the system, no matter whether they have already joined the overlay or not.
- **Overlay:** It contains the peers who have joined the overlay.
- **Monitor:** Monitor is an auxiliary object that have a global view of peers in network and can monitor the behaviour of each peer and structure of the whole overlay.
- **Failure Detector:** For the sake of simplicity we assume there is one failure detector in the system. Each peer can register for the peers of its interest. Upon failure of a peer, all the peers who have registered for that peer, will be notified.

Figure 8.3 shows each of these modules and their relation. Details on the architecture of SicSim, along with a step-by-step example can be found at *http://www.sics.se/ amir/sicsim*.

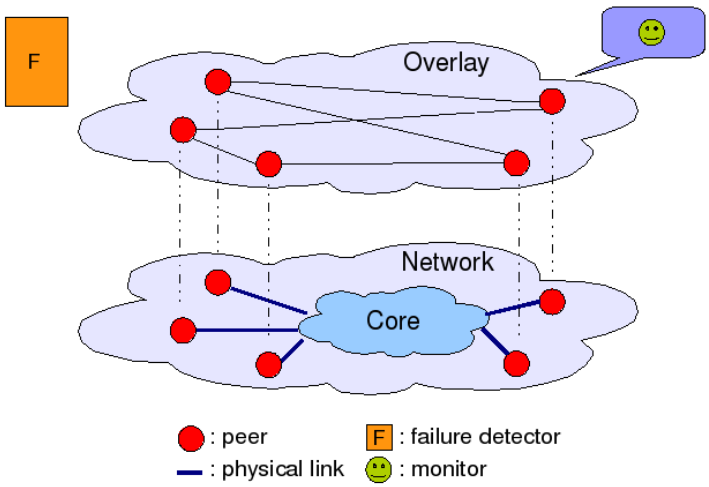


Figure 8.1: SicSim main modules

Chapter 9

D4.3b: Second report on self-tuning support

9.1 Executive summary

Self-tuning of large systems, such as Structured Overlay Networks (SONs), is complex since the overall system must be optimized through the interaction of many independent components. The work on self-tuning is focused on two parts. First, we introduce two variations on decentralized load balancing algorithms used in the context of the replicated storage service developed in SELFMAN. Second, we present a technique for SON size estimation.

9.2 Contractors contributing to the Deliverable

KTH(P2) has developed an algorithm for estimating the network size in a structured overlay. The algorithm is completely decentralized, and the estimate can then be used for tuning different parameters of the overlay.

ZIB(P5) has performed the work on load balancing. The main contribution is a simulated analysis of a replicated storage service.

9.3 Results

In this Deliverable we report on our work related to Self-tuning of the replicated storage service (Deliverable 3) and Structured Overlay Networks (SONs). First, we present a decentralized load balancing algorithm used to even out the load between contributing nodes. This is important in order to avoid excessive overload resulting in for example denied write requests.

Second, we provide an algorithm for estimating the network size in a structured overlay. The algorithm is designed for ring-based overlays, such as DKS [27], Chord [81] and Chord# [74].

9.3.1 Load Balancing in a Distributed Key/Value store

Ring-based Structured Overlay Networks (SONs) provide algorithms for node membership management and efficient look-ups for finding a node responsible for a key. The overlay construction in a SON requires that a range of subsequent keys belong to a node. In a distributed key/value store, where a value is associated with a key, this tight coupling between the data layer and the overlay has direct implications on node utilization. In particular, a stored key uses a node's storage capacity while access to the key requires bandwidth and processing to handle the request. The storage and access load can differ significantly when comparing the nodes. Our goal with balancing the load is fairness, i.e. the load of individual nodes should be as close as possible to the average system load.

A common approach to load balancing in SONs is to use hashing, which ensures that the system storage imbalance is at most $O(\log N)$, where N is the number of nodes [42]. The system imbalance is defined as the ratio between the highest loaded node and the average load. By using hashing, both key IDs and node IDs are derived from the same uniform hash function. However, using a hash function that generate uniform IDs from a key makes range queries impossible, a requirement of the replicated storage service described in Deliverable 3. Therefore we cannot use hash-based load balancing of keys.

Without hash-based key balancing, keys and the load will follow the distribution of the data. To balance the system, the nodes must therefore also follow this distribution. In this work we have focused on randomized balancing algorithms which efficiently adapt the position of nodes to better fit the key distribution. These algorithms allow a node to take balancing decisions based on information available either locally or collected from a small subset of the nodes. Generally, randomized algorithms have the following phases:

Collect data Data is collected by periodically contacting a set of random

nodes asking them for their state, e.g. used storage or current workload. Additionally, data can be collected through an out-of-band mechanism such as a gossiping protocol calculating the average system load.

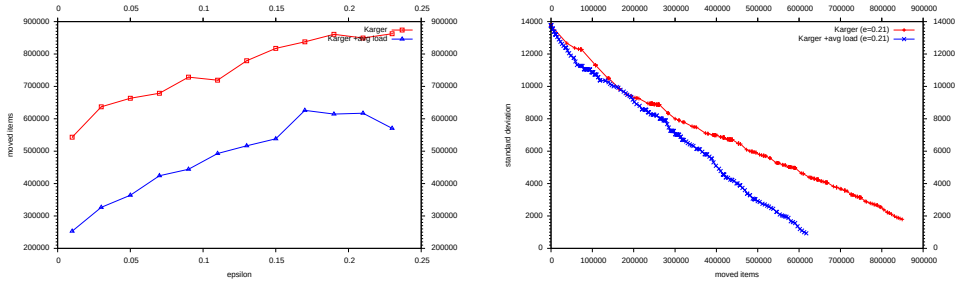
Decide on balancing operation By analyzing the collected data, a node can make a decision to either do nothing or to balance with an overloaded node. In our model, nodes are limited to a single operation resulting in a leave followed by a re-join of the overlay at a new position.

Execute balancing operation The node executes the balancing operation by either failing or leaving gracefully and then re-joining the system at a new location.

We have worked on two parts of the load balancing algorithm within the last year. First, we made the assumption that the system would require less bandwidth when knowing the average load for balancing. Second, we designed a node placement function that considers several load attributes when selecting its new position.

Using Global Information

In A.16, we argue for the benefits of introducing approximations of global knowledge for DHT load balancing algorithms with the goal of reducing the network utilization. To support this claim, we extended a randomized load balancing algorithm to use the average storage load.



(a) Moved items with increasing balancing factor (b) Moved items vs. standard deviation

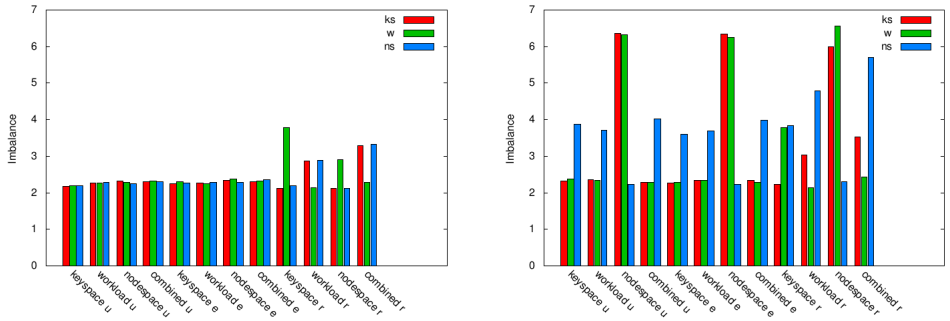
Figure 9.1: Impact of global information on a randomized load balancing algorithm.

The results of this enhancement showed two things. First, the number of moved items was significantly reduced (see Fig. 9.1(a)). Second, the

load imbalance improved faster compared to the algorithm that did not have knowledge about the average load (see Fig. 9.1(b)).

Placement Function

In a distributed key/value-store, a node contributes storage for key/value-pairs and network and CPU-time for accessing the stored items. The goal of load balancing is to improve the fairness of resource usage between the contributing nodes. Imbalance between nodes mainly occurs due to; 1) non-uniform key distribution, 2) skewed access frequency of keys and 3) node heterogeneity.



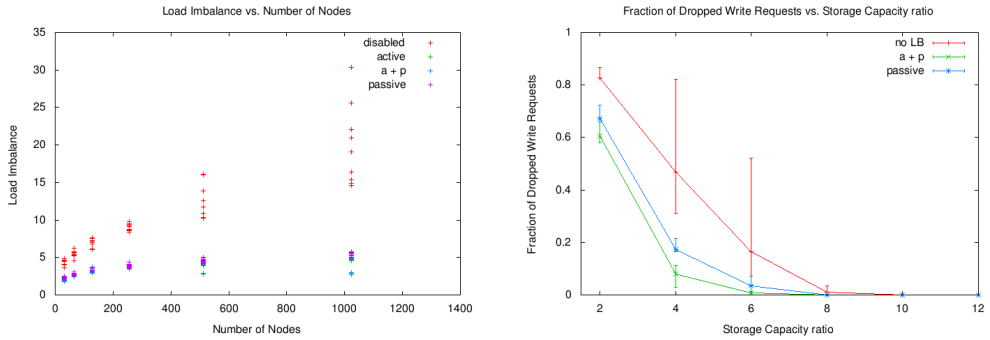
(a) Uniform key distribution

(b) Dictionary key distribution

Figure 9.2: The effect of different access workloads and key distributions.

Figure 9.2 shows how different workloads and key distributions affects the imbalance of the storage, workload and node responsibility size. In Figure 9.2(a) a uniform key distribution was used while in Figure 9.2(b) we used a dictionary-like distribution derived from a subset of the Wikipedia article titles. The access workload on the keys was either uniform (u), exponential (e) or range (r), where a set of subsequent keys were accessed. In these results we see that a uniform key distributions have minimal impact on the imbalance. However, a non-uniform distribution can in this case create a difference in the imbalance up to a factor 3. Thus, we conclude that a balanced system must consider more than a single metric when balancing.

In this work, we have developed a node placement function that take multiple load metrics into account. The placement itself occurs when a node is joining the system. A new node first samples a set of possible nodes where it can join, it then greedily selects the node with highest load. Using the placement function, this node then calculates the best position before the new node joins.



(a) Increasing nodes and items

(b) Capacity

Figure 9.3: Imbalance of the system using difference balancing strategies and increasing the system size.

Results from a simulation-based evaluation show that the proposed algorithm improves the load balance by a factor 6-12 in a system with 1000 nodes (Fig. 9.3(a)). In addition, we also show that load balancing reduces the extra storage capacity necessary in an overloaded system from a factor 10 to 8 (Fig. 9.3(b)). Further results and a detailed description of the algorithms can be found in the Appendix A.17.

9.3.2 Network Size Estimation

The network size is a global variable which is not accessible to individual nodes in the system as they only know a subset of the other nodes. This information is, nevertheless, of great importance to many structured p2p systems, as it can be used to tune the rates at which the topology is maintained. Moreover, it can be used in structured overlays for load-balancing purposes [28], deciding successor-lists size for resilience to churn [48], choosing a level to determine outgoing links [51], and for designing algorithms that adapt their actions depending on the system size [12].

Due to the importance of knowing the network size, several algorithms have been proposed for this purpose. Out of these, gossip-based aggregation algorithms [41], though having higher overhead, provide the best accuracy [59]. Consequently, we focus on gossip-based aggregation algorithms. While aggregation algorithms can be used to calculate different aggregates, e.g. average, maximum, minimum, variance etc., our focus is on counting the number of nodes in the system.

Although Aggregation [41] provides accurate estimates, it suffers from a few problems. First, Aggregation is highly sensitive to the *overlay topology*

that it is used with. Convergence of the estimate to the real network size is slow for non-random topologies. On the contrary, the majority of structured p2p overlays have non-random topologies. Thus, it is not viable to directly use Aggregation in these systems. Second, Aggregation works in rounds, and the estimate is considered converged after a predefined number of rounds. This can be problematic. Finally, Aggregation is highly sensitive to node failures.

In our work, we suggest a gossip algorithm based on Aggregation to be executed continously on every node to estimate the total number of nodes in the system. The algorithm is aimed to work on structured overlay networks. Furthermore, the algorithm is robust to failures and adaptive to the network delays in the system.

The network size estimation algorithm

A naive approach to estimate the network size in a ring-based overlay would be pass a token around the ring, starting from, say node i and containing a variable v initialized to 1. Each node increments v and forwards the token to its successor i.e. the next node on the ring. When the token reaches back at i, v will contain the network size. While this solution seems simple and efficient, it suffers from multiple problems. First, it is not fault-tolerant as the node with the token may fail. This will require complicated modifications for regenerating the token with the current value of v. Second, the naive approach will be quite slow, as it will take $O(n)$ time to complete. Since peer-to-peer systems are highly dynamic, the actual size may have changed completed by the time the algorithm finishes. Lastly, at the end of the naive approach, the estimate will be known only to node i which will have to broadcast it to all nodes in the system. Our solution aims at solving all these problems at the expense of a higher message complexity than the naive approach.

Our goal is to make an algorithm where each node tries to estimate the average inter-node distance, Δ, on the identifier space, i.e. the average distance between two consecutive nodes on the ring. Given a correct value of Δ, the number of nodes in the system can be estimated as $\frac{N}{\Delta}$, N being the size of the identifier space.

Every node p in the system keeps a local estimate of the average inter-node distance in a local variable d_p. Our goal is to compute $\frac{\sum_{i \in \mathcal{P}} d_i}{|\mathcal{P}|}$. The philosophy underlying our algorithm is the observation that at any time the following invariant should always be satisfied: $N = \sum_{i \in \mathcal{P}} d_i$.

We achieve this by letting each node p initialize its estimate d_p to the distance to its successor on the identifier space. In other words, $d_p = succ(p) \ominus p$,

where $\ominus$ represents subtraction modulo N. Note that if the system only contains one node, then $d_p = N$. Clearly, a correctly initialized network satisfies the mentioned invariant as the sum of the estimates is equal to N.

Details of the algorithm, along with extensive evaluations, can be found in Appendex A.3.

9.4 Papers and publications

Using Global Information for Load Balancing in DHTs

In A.16, we argue for the benefits of introducing approximations of global knowledge to DHT load balancing algorithms with the goal of reducing the network utilization. To support this claim, we have extended a randomized load balancing algorithm to use the average storage load.

Node Placement in Distributed Key/Value-stores

Distributed key/value stores are a basic building block for large-scale Internet services. Support for range queries introduces new challenges to load-balancing since both the key and workload distribution can be non-uniform.

We build on previous work based on the power of choice to present algorithms suitable for active and passive load-balancing that adapts to both the key and workload distribution. The algorithms are evaluated in a simulated environment, focusing on the impact of load-balancing on scalability and in an overloaded system.

Network Size Estimation in Structured Overlays

A Practical Approach to Network Size Estimation for Structured Overlays. *Tallat M. Shafaat, Ali Ghodsi, Seif Haridi.* In the Third International Workshop on Self-Organizing Systems (IWSoS'08), December, 2008, Austria.

This paper has been included as Appendix A.3. It describes in detail the importance of knowing an estimate of the current network size, the functioning of the algorithm, and a detailed evaluation of the proposed algorithm.

Chapter 10

D4.3c: Self-tuning support (software)

10.1 Executive summary

This deliverable describes the implementation of self-tuning support in the replicated storage service (Deliverable 3). The introduced mechanism continuously tunes the system load balance towards an even load over the participating nodes. Load balancing is necessary to handle non-uniform distribution of the items stored by the system as well as avoiding extreme load conditions for certain nodes. The implementation itself is part of the Scalaris source distribution available at http://scalaris.googlecode.com/.

10.2 Contractors contributing to the Deliverable

ZIB(P5) Provided an implementation of load balancing in the replicated storage service.

10.3 Results

In this deliverable we present the software implementation for self-tuning support. The implementation is a load balancing algorithm part of the replicated storage service (Scalaris) presented in Deliverable 3. The algorithm is based on Karger's item-balancing scheme [42].

This deliverable is related to Deliverable 9 where we presented two techniques for load-balancing and one for estimating the network size estimation. These techniques have been verified through simulation. A simulation environment was necessary to simplify studies showing the impact of different algorithm parameters such as the frequency of performing load balancing. The result of these studies can be found in Section 9.3 and the accompanying papers of Deliverable 9.

10.3.1 Algorithm

Classical DHTs such as CAN, Pastry and Chord [81, 70, 67] use a hash function to distribute the keys evenly among the nodes in the system. However, by using a hash function it is not possible to support range queries since the order of the stored keys would be destroyed. A direct issue when removing the hash function is that the storage load on the nodes depend on the key distribution (a storage load of a node is the number of items stored on it). In the worst case, this imbalance can lead to network congestion and unresponsive nodes. In order to avoid this, Scalaris has been extended with an algorithm (build upon Karger's Algorithm [42]) that balances the storage load fairly over the nodes based on the item distribution. The implementation of this load balancing algorithm is the core of this deliverable. It can be found in `src/cs_lb.erl` as part of the Scalaris open source distribution[1]. In the following, we describe this implementation at the algorithmic level.

Karger's Algorithm. Karger et al. [42] present a randomized item balancing scheme where each node contacts another random node periodically. If the load of the nodes differs by more than a factor $0 < \epsilon < \frac{1}{4}$, they share each others load by either jumping or sliding. Karger provides a theoretical analysis of the protocol, but does not evaluate the algorithms in an experimental or real-world setting.

We start with description of our model. A DHT consist of N nodes and an identifier space in the range $[0, 1)$. This range wraps around at 1.0 and can be seen as a ring. A node, n_i, at position i has an identifier n_i^{ID}, a part of the

[1] http://scalaris.googlecode.com/

ID space. Each node n_i has a *successor*-pointer to the next node in clockwise direction, n_{i+1}, and a *predecessor*-pointer to the first counter-clockwise node, n_{i-1}. The last node, n_{N-1}, has the first node, n_0 as successor. Thus, the nodes and their pointers create a double linked list where the first and last node are linked.

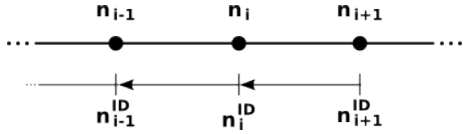


Figure 10.1: A node n_i with successor and predecessor and the ranges of responsibilities

Each node in the DHT stores a subset of items, $I(n_i)$, where each item has a key in the range $[0, 1)$ and a uniform weight of one. A node n_i is *responsible* for a key iff it falls within the node's key range $(n_{i-1}^{ID}, n_i^{ID}]$. Each node has a load $l(n_i)$ indicating the number of stored data items. Figure 10.1 shows three nodes and their respective responsibilities.

The two fundamental operations in the Karger's Algorithm - *jump* and *slide* - are described in the following.

Jump allows a node to move to an arbitrary position in the ID space. A jumping node n_i first leaves its current position and re-joins at a new position, $n_{i'}$, with $n_{i'+1}$ as its successor. Data is moved two times. First, the items in the range $(n_{i-1}^{ID}, n_i^{ID}]$ are transferred to n_{i+1}. Second, when $n_{i'}$ joins at its new position, all data in the range $(n_{i'-1}^{ID}, n_{i'}^{ID}]$ is transferred from $n_{i'+1}$ to $n_{i'}$.

Slide is a specialized form of jumping where n_i moves to an ID in the range $(n_{i-1}^{ID}, n_{i+1}^{ID}]$. Since a node does not need to leave and re-join the system, which results in extra data transfer, sliding is always preferred over jumping.

We rely on the heuristics introduced for Karger's item balancing algorithm. Expressed in our notation, a load-balance operation is only performed between any pair of nodes n_i, n_j, iff $l(n_i) < \epsilon l(n_j)$ or $l(n_j) < \epsilon l(n_i)$, $0 < \epsilon < \frac{1}{4}$. Under these conditions and assuming $l(n_i) > l(n_j)$, the following cases are possible:

Case 1 $i = j + 1$, n_i is the successor of n_j. Slide n_j towards n_i, letting n_j take responsibility for $\frac{l(n_i)-l(n_j)}{2}$ of n_i's items.

Case 2 $i \neq j+1$, if $l(n_{j+1}) > l(n_i)$, set $i = j+1$ and go to case 1. That is, when the load of n_j's successor is larger then the load of n_i, slide n_j towards the overloaded node n_{j+1}. Otherwise, n_j jumps to an ID in the range (n_{i-1}^{ID}, n_i^{ID}), taking half of $l(n_i)$.

```
1  def balance_decision(n_i, n_j):
2      if n_i == n_{j+1}:
3          # j increases its ID so that the (l_i -
4          # l_j)/2 items with lowest keys get
5          # re-assigned from n_i to n_j
6          return (SLIDE, n_{j+1})
7      # case 2: load(n_{j+1}) > load(n_i)
8      elif load(n_{j+1}) > load(n_i):
9          # set i = j + 1 and go to case 1
10         # Assume that this means that we should move j
11         # to take half of its successor's load since
12         # this is more than the load of node i.
13         return (SLIDE, n_i)
14     else:
15         # load(n_j) < load(n_i)
16         # and load(n_i) > load(n_{j+1})
17         # move j to the middle of i's item space
18         return (JUMP, n_i)
19
20 def balance(n_i):
21     for n_j in random_nodes():
22         if load(n_j) ≤ load(n_i)*ε:
23             execute(balance_decision(n_j, n_i))
24         elif load(n_i) ≤ load(n_j)*ε:
25             execute(balance_decision(n_i, n_j))
```

Figure 10.2: Pseudo code of Karger's algorithm.

Figure 10.2 contains pseudo-code describing Karger's balancing algorithm. `balance` is called with a node n_i that represents the local node. Thereafter, a set of random nodes are sampled in the overlay. If any of the sampled nodes differ by a factor of ϵ from the local load, the algorithm executes a jump or a slide operation. By varying ϵ, we can trade-off the bandwidth used for balancing vs. the achieved system imbalance.

Chapter 11

D4.4b: Self-protection support

11.1 Executive summary

This deliverable describes the software developed for the self-protection. Self-protection focuses on the robustness of Small World Networks (SWN) from attacks on the overlay network and protection from identity attacks using social networks. The software consists of two SWN simulators. One has visualization and is targeted at experiments on artifical SWNs. The other is a more efficient simulator for simulating experiments on social networks. In order to work with real social networks, we developed a crawler to extract the social network graph from popular social network sites, namely, *Facebook*, *Live Spaces* and *MySpace*.

11.2 Contractors contributing to the deliverable

NUS(P7) has contributed to this deliverable.

NUS(P7) implemented the small world network simulators and the social network crawler.

11.3 Software for small world network and social network experiments

The self-protection software developed in WP4 consists of Small World Network (SWN) simulators and a crawler to extract social networks. The following software is described in the remainder of this chapter:

- SWN simulator for synthetic SWNs:
  This simulator described in Section 11.4 supports routing, failure, self-tuning and self-protection experiments on synthetic SWNs which are embeddable in a ring. It supports animated visualizations of the experiment.

- SWN simulator for social networks:
  This simulator described in Section 11.6 is a more efficient simulator meant for experimenting on real social networks. It also has specific functionality for social networks such as extracting specific sub-graphs of the social network graph and visualizing the graph.

- Social network crawler:
  The crawler described in Section 11.5 extracts the social network graph from several social networking sites.

11.4 Initial SWN simulator

We have built two Small World Network (SWN) simulators. The first SWN simulator was built to investigate and explore the feasibility of SWN as a form of semi-structured overlay network. In what follows, we use n to denote the number of nodes in the SWN or the network. The SWN simulator was designed to investigate the following questions (see Section 16.3.1):

- Evaluated the performance of basic routing of SWN under different number of edges which follow power law distributions.
- Measure the performance and the robustness SWN under static and dynamic settings (churn).
- Evaluate the effect of node and link failures and their effect on the routing success rate.
- Evaluate the Sandberg self-tuning method for optimizing the SWN.
- Investigate self-tuning issues such as simulate effectiveness of self-tuning under poisoning attack and protection against poisoning.

As we wanted to investigate how SWNs behave, we developed a GUI interface which shows graphically what is happening in the network as the simulation progresses. The simulator also provides a snapshot of the current state of the SWN in the form of a ring visualization. The various visuals in the simulator are animated and change as time progresses in the simulation.

Figure 11.1 shows the interface of the simulator. The top left is a visualization of the network which is mapped onto a 2-dimensional ring. This visualization is meant to show the state of the SWN with respect to the node identifiers which is relevant to the performance of decentralized routing, in particular, greedy routing. A perfect SWN state is when all the node identifiers are positioned correctly on the ring so that the edge distance between nodes according based on their positions follow a power law distribution. The visualization displays a perfect SWN state by coloring the nodes so that the ring forms a continuous red-green-blue ring color as in Figure 11.2. A SWN with random node identifiers is equivalent to shuffling the perfect SWN state and is shown in Figure 11.3. The effect of the randomization can be seen as breaking up the smooth color continuity of the ring. The self-tuning tries to re-optimize the shuffled SWN node positions to be more similar to the perfect state by reorganizing the node positions. A recovered ring structure is shown in Figure 11.4. The outer ring shows the imbalance of the particular node that resides in that position. The imbalance is calculated by how

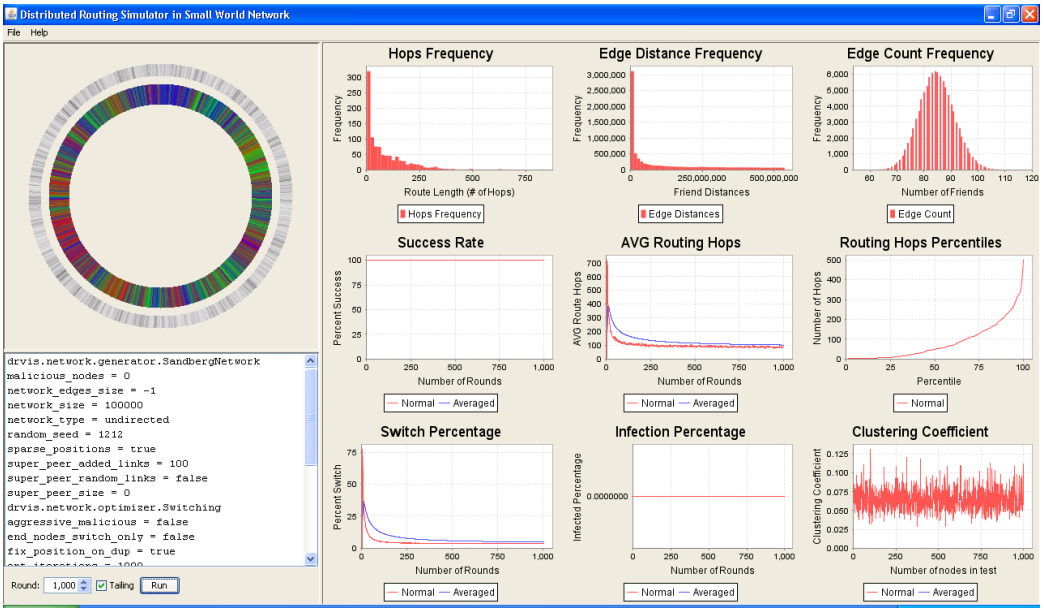


Figure 11.1: Simulator Interface

many edges are pointing to the left of the ring (in a clockwise manner, this is the lower node identifier) and how many edges are pointing to the right of the ring. The greater the difference between left and right, the darker the outer ring for that particular position. This essentially means that the node residing in that position is dissimilar to its correct position in the perfect state because a good state should be roughly balanced between the number of left and right edges. The nodes in the inner ring are colored using shades of red-green-blue and it shows relative positions of the current state with respect to the perfect state. The closer the self-tuning gets to the perfect state, the more continuous will be the color in the inner ring. The visualization is animated so that it is shown with the colors possibly changing as the self-tuning progresses as the nodes swap positions and hence colors.

The right side of Figure 11.1 shows nine visualizations of statistics to do with the SWN simulation for monitoring the other properties of the SWN as the self-tuning or poisoning experiment is run. The statistics also changes from round to round providing a realtime visualization of the current state of the network. The hops frequency histogram tells us the distributions of the routing length. The edge distance frequency shows the edge distributions. This example shows a power law distribution — there are more edges that have a short distance and fewer edges that have a long distance. The edge count frequency shows the number of friends' distributions. The success rate

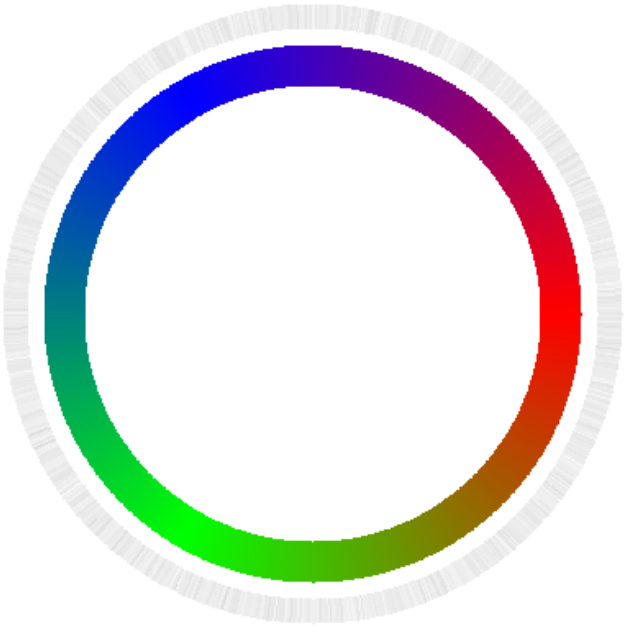


Figure 11.2: Ring Structure - Perfect SWN node positions

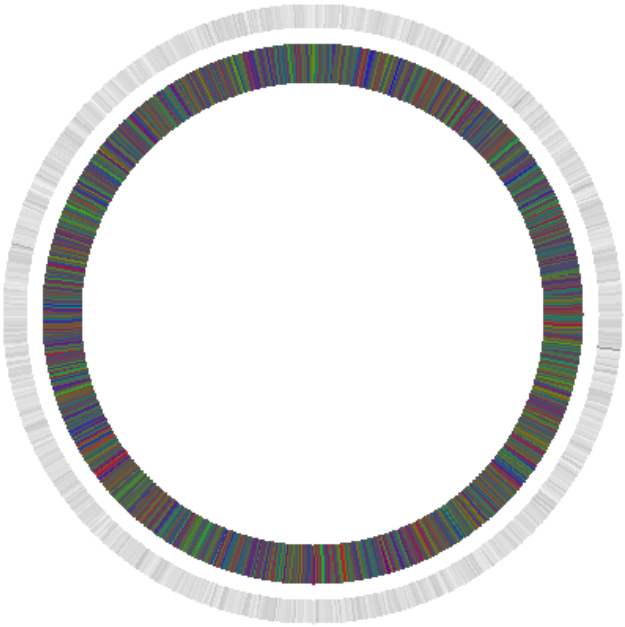


Figure 11.3: Ring Structure - Shuffled node positions

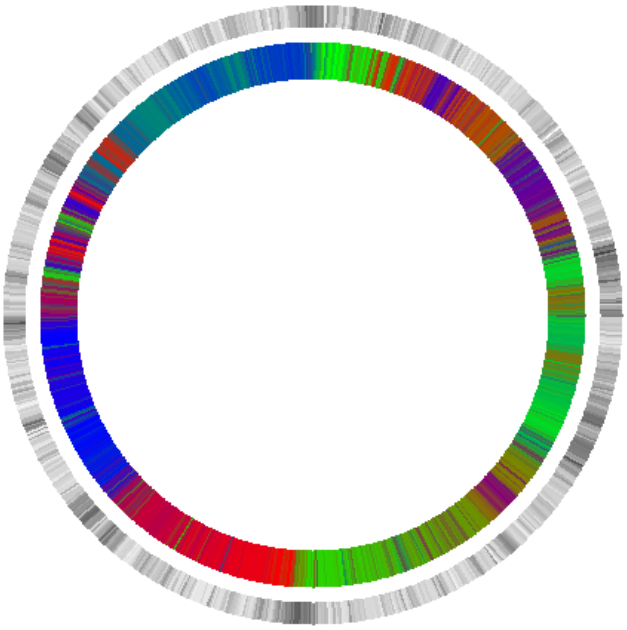


Figure 11.4: Ring Structure - Recovered node positions

shows out of 1000 routing tests, how many are successful (complete in less than $log^2(n)$ hops). The average (AVG) routing hops gives the average of the successful routing length for 1000 routing tests. The routing hops percentile gives the distribution of the routing length. The switch percentage shows the percentage of nodes that are involved in switching operations within a round. The infection percentage shows the percentage of nodes that are poisoned by a malicious node. The clustering coefficient shows the clustering coefficient of individual nodes.

The input for the simulator is parameters that can be entered directly in the bottom left part of the UI in Figure 11.1. Examples of the parameters that can be changed in the configuration text are:

- Network generation algorithm: The choice of the network being analyzed. Currently it can generate 3 different networks (Chord, SWN, and Random).
- Network optimization algorithm: The self-tuning optimization works to reshape the network identifiers to follow a power law distribution. There is one option which is the Sandberg self-tuning algorithm based on switching node positions.

- Network routing algorithm: There are many ways to do routing in the network as well as different decision whenever a cycle is found. Several variants of greedy routing are possible.
- Network parameters such as the number of nodes, number of edges, the edge type (directed/undirected), the length of the random walk for the network optimizer, number of routing tests, number of rounds.
- Poisoning experiment parameters such as the number of malicious nodes, the infection rate of a malicious node, and the restart probability for the node positions per round (this is used to reduce the effect of malicious nodes).

The current round is shown at the bottom left textbox and we can see the state of the previous round by changing the value of the textbox. The "tailing checkbox" is to disable the animation when we are looking for a particular round. By checking the checkbox, the animation will update the entire visualizations to show the latest round.

The ring visualization and the statistics help in deciding the correct parameters for the self-tuning algorithm and for picking the correct strategy to reduce the impact of the poisoning attack.

11.5 Social network crawler

In order to simulate our SWN described in Section 16.3.1 on real social networks, we have developed website crawlers for the following social networks: *Windows Live Spaces*, *MySpace*, and *Facebook*. We chose these three social networks because they are among the largest social networks publically available. The three crawlers are very similar except for the HTML analysis part, thus we only describe the Facebook one. The HTML analysis is used for extracting the friend list from the HTML data downloaded from the web server.

In Facebook, by default, users' friend list is visible to all users logged in to Facebook. Users can "close" their profile by changing the policy to only allow viewing from their friends. In this case, our crawler cannot view the friend list. We found out that 87.7% of the users' friend list is publicly visible in Facebook.

In MySpace, friend list is by default visible publicly to everyone including non-logged in users. Thus our crawler for MySpace does not have to perform login before crawling. After crawling a few hundred thousand users, we found out that the average number of friends per user is very high. Having more than 5000 friends is quite common in MySpace. We believe that it is a culture in MySpace to add friends even if people do not know each other. Thus the social network in MySpace can be very different from real life. We decided not to continue crawling MySpace.

In Windows Live Spaces, before November 2008, the policy was similar to MySpace, i.e. friend list was by default visible to even non-logged in users. After November 2008, the policy was changed to be similar to Facebook to only allow viewing by logged in users. Similar to Facebook, users can "close" their profile as well. We found out that 67.0% of the users' friend list is publicly visible in Windows Live Spaces.

The crawler is implemented using a combination of `bash` scripts, `awk` scripts, and some UNIX text processing tools. The actual downloading is handled by `wget`.

```
30 Tono%20Aguilar 28 29 53 58 189 192 209 ...
171 Qian%20Zhang 117 190 201 824 949 967 ...
216 Egle%20Cekanaviciute 112 327 444 450 ...
220 Helen%20Chou 48 56 60 85 89 120 149 ...
233 Michael%20Blickstead 37 40 42 44 48 ...
```

Figure 11.5: An example of a Social Network Database

The crawled social network database is stored in *tab separated variable* format. Figure 11.5 shows an example of the social network database from Facebook. Each line represents a downloaded user in Facebook. The first column is the user ID of the downloaded user. The second column is the user's name. User names are not used in our simulation program. They are stored to make it easier to recognize users. The rest of the columns are the friends' IDs of the downloaded user. All user IDs in Facebook are represented using integers.

The crawler consists of three bash scripts: `crawl.sh`, `login.sh` and `getfl.sh`.

- `crawl.sh` is the main script which executes the other two scripts. It executes `login.sh` and then forks a number of processes to execute `getfl.sh` in parallel to download a list of users. After all users are downloaded, it reads the database to determine the new users to be downloaded in the next round and restart the procedure.

- `login.sh` takes a user name and password and then it logins to Facebook. It checks if the login is successful. The cookie which contains the login session ID is that stored so that it can be used by `getfl.sh`.

- `getfl.sh` takes a user ID and download the friend list of the user and append to the database file. The friend list is downloaded from the URL `http://www.facebook.com/friends/?id=[uid]&s=[page]`, where *uid* is the user ID in decimal integer and *page* is the page number starting from 0. (Facebook lists 400 users per page). The friend's user IDs is then extracted from the downloaded webpages.

To make the crawling server-friendly, i.e. not to cause significant load to Facebook servers or appear to be an attack, we applied some rate limiting constraints. Firstly, `crawl.sh` creates no more than 60 downloading threads regardless of the number of users to be downloaded and downloading speed. Secondly, we use linear backoff in the case of download failure. The first failure causes a sleep of one second, the second causes two seconds and so on. The average download speed we obtained is about 6 users/second or 500 KB/s.

Statistics of the crawled Facebook graphs can be found in Section 16.3.1.

11.6 A more efficient SWN simulator for social networks

The initial Java Simulator described in Section 11.4 was suitable to simulate the "synthetic" SWN behavior under various environments and under poisoning attack. In order to run simulation on a real social network that is collected using the crawler (from Section 11.5), we developed a new version of the simulator in C++. There are several reasons why we needed to develop the second simulator. Firstly, it needs to deal with real social networks. Secondly, the first simulator was meant for exploration but the visualization is less useful and becomes to slow for large networks. Furthermore, the ring visualization cannot be used to visualize the progress of the self-tuning with a real social network as it relies on knowing the correct synthetic SWN which is then shuffled. Thirdly, as visualization is not effective for social networks and difficult on large networks, a new implementation which runs self-tuning until the network is sufficiently restored is more efficient. Lastly, we added a number of features to the new simulator to support social networks. These include pruning the graph to reduce the number of edges and adding another attack model for the network such as the consecutive range ID attack (see Section 16.3.1).

The second simulator does not use a graphical user interface. All simulations are controlled from the command line. We briefly describe how to use the simulator. The simulator accepts the following options from the command line:

- $-T$ denotes number of routing tests per round (default is 1000)

  For each self-tuning round, we do T routing tests to measure performance of routing on the selected graph. Statistics are given per round based on the routing tests. These include how many routing tests are successful and their average routing length.

- $-H$ denotes maximum number of hops (default $log^2(n)$)

  Each routing test is done by picking 2 random nodes in the network and trying to route from one to another. The maximum number of hops behaves as a TTL (time to live) for the message. If the routing uses more than H hops, then routing is stopped and considered as failed.

- $-L$ denotes random walk length (default 10)

  The self-tuning algorithm requires two nodes to switch node positions for the optimization. In order to find a partner to switch a node needs to do a random walk of length L.

- $-R$ denotes the number of self-tuning rounds (default 10000)
- $-X$ denotes the percentage of node failure (randomly picked)

  Node failure affects the routing performance. Whenever a routing arrived at a node that is marked as failed, the message will be bounced back to the previous node, which cost 2 routing steps for visiting a failed node.
- $-Y$ denotes the percentage of link failure (randomly picked)

  Link failure behaves similarly with the node failure.
- $-A$ denotes the consecutive range ID attack enabled

  This is similar to the node failure parameter X except that there are A consecutive failed nodes. This parameter is used to simulate the attack model of a consecutive range ID attack.
- $-D$ random seed for the simulator to run the experiments with a different random seed
- $-G$ output the graph in Graphviz dot file format

  To visualize the real social network, we need to translate the input format to the dot file format that is supported by Graphviz.[1]
- $-C$ the maximum clustering coefficient for each node (default 1.0)

  Facebook has more than 200 million users. Since we only analyze a subset of the real social network graph, the proportion of the number of nodes and the number of edges in the subgraph may not be balanced. Therefore we need to adjust the number of edges in respect to the number of nodes. The maximum bound on the clustering coefficient is used to limit the maximum number of edges that can exist in the neighbours of a node.

A sample of run of the new simulator is as follows:

```
$ ./sim < facebook-data.txt

N = 3996, Connected = 3996
F = 124.028
EP = 33 57 69 80 92 106 123 144 171 215 768
CCP = 0.093 0.288 0.344 0.397 0.437 0.480 0.521 0.566 0.617 0.683 0.977
```

[1] http://www.graphviz.org

```
1 1640 487.808000 0.112
2 1044 679.570000 0.820
3 889 917.364000 0.790
...
many rounds elapsed
...
9997 23 50.613000 0.003
9998 41 62.379000 0.003
9999 23 78.613000 0.002
```

The input for the simulator data is a social network graph in the format produced by the social network crawler in Section 11.5. The output of the simulator is the result of running the self-tuning algorithm on the input graph. The first line shows the number of nodes N in the real social network followed by the number of nodes that are reachable from the first node. This is used to check that the network is fully connected — the value should be the same as the number of nodes in the network. The second line denotes the average number of friends F in the network. The third line shows the distribution of the number of edges over all nodes displayed as percentiles. The percentiles shown in the output are from 0 to 100th percentile with an interval of 10. The fourth line gives the distribution of the clustering coefficient over all nodes also in percentile form. The remaining lines gives the self-tuning results formatted as 3 numbers per line: the first number is the number of the round (or iteration), the second number is the number of switches in that round, the third number is the average routing length using T routing tests, and the last number is the failure ratio of the routing tests. This gives the fraction of the T routing test that are failed (i.e. those routing tests that takes more than H hops).

Chapter 12

D5.2b: Demonstrator application for J2EE (software)

We decided to implement a distributed Wikipedia-clone based on Scalaris instead of a more traditional J2EE application. As we describe later, the application's architecture is similar to traditional J2EE applications but the tools are more lightweight.

12.1 Executive summary

We implemented a Wikipedia-clone with Scalaris as the database. The presentation layer is completely written in Java. A part of the business logic is written in Java. The lower levels, which directly interact with the database, were written in Erlang.

For demonstrations, we loaded Wikipedia dumps into our system and presented basic features like browsing, page histories, and editing.

The presentation layer as well as the business logic are completely stateless. All data is stored in Scalaris. This architecture allows us to easily scale these two layers by simply adding more servers. Adding fault-tolerance and self-* properties to these layers is also simple. For scaling the database layer we rely on Scalaris.

12.2 Contractors contributing to the Deliverable

ZIB (P5) has contributed to this deliverable.

ZIB (P5) ZIB has contributed on the simple database query layer for Scalaris.

12.3 Results

We implemented a Wikipedia-clone with Scalaris as the database. The presentation layer is completely written in Java. A part of the business logic is written in Java. The lower levels, which directly interact with the database, were written in Erlang. Fig. 12.1 shows this architecture.

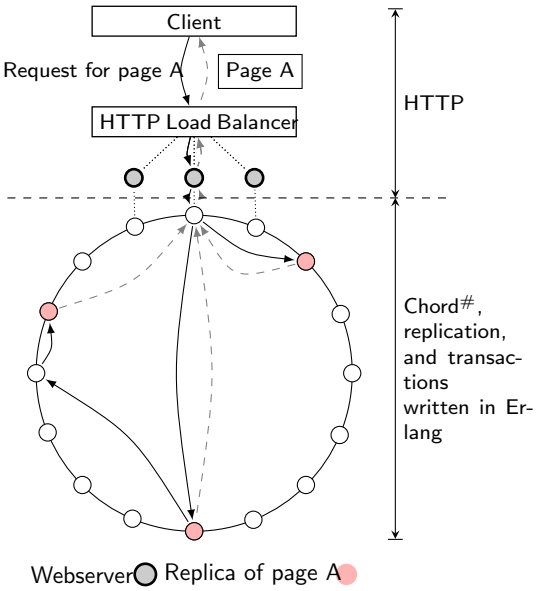


Figure 12.1: Wikipedia on Scalaris.

The public Wikipedia uses PHP to render the Wikitext to HTML and stores the content and page history in MySQL databases. Instead of using a relational database, we map the Wikipedia content to Scalaris [65]. We use the following mappings, using prefixes in the keys to avoid name clashes:

	key	value
page content	title	list of Wikitext for all versions
backlinks	title	list of titles
categories	category name	list of titles

The page rendering of the Wikitext is done in Java in the web servers (see Fig. 12.1) running jetty. For that we had to modify the Wikitext renderer of the *plog4u* project.

Using this data layout, users may view pages by typing the URL, they can navigate to other pages via hyperlinks, they can edit pages and view the history of changes, and create new pages (see the screenshot in Fig. 12.2). Since the Wikipedia dumps do not include images, we render a proxy image at the corresponding positions instead. Moreover, we do not maintain a full text index and therefore full text search is not supported by our implementation. This could easily be performed by external crawling and search indexing mechanisms.

When modifying a page, a transaction over all replicas of the responsible keys is created and executed. The transaction includes the page itself, all backlink pages for inserted and deleted links, and all category pages for inserted and deleted categories.

Dissemination We published several papers on Scalaris and the Wikipedia-clone. Sec. 12.4 shows the new papers for this reporting period. Additionally, we presented the demonstrator at two industrial conferences (Google Conference on Scalability and Erlang eXchange) and participate successfully in the first IEEE Scale Challenge.

- T. Schuett, M. Moser S. Plantikow F. Schintke A. Reinefeld. A Transactional Scalable Distributed Data Store: Wikipedia on a DHT. IEEE Scale, May 2008. First Prize.
- T. Schtt. Scalable Wikipedia with Erlang. Google Conference on Scalability, June 2008.
- A. Reinefeld. Building a transactional distributed data store with Erlang. Erlang eXchange, June 2008.

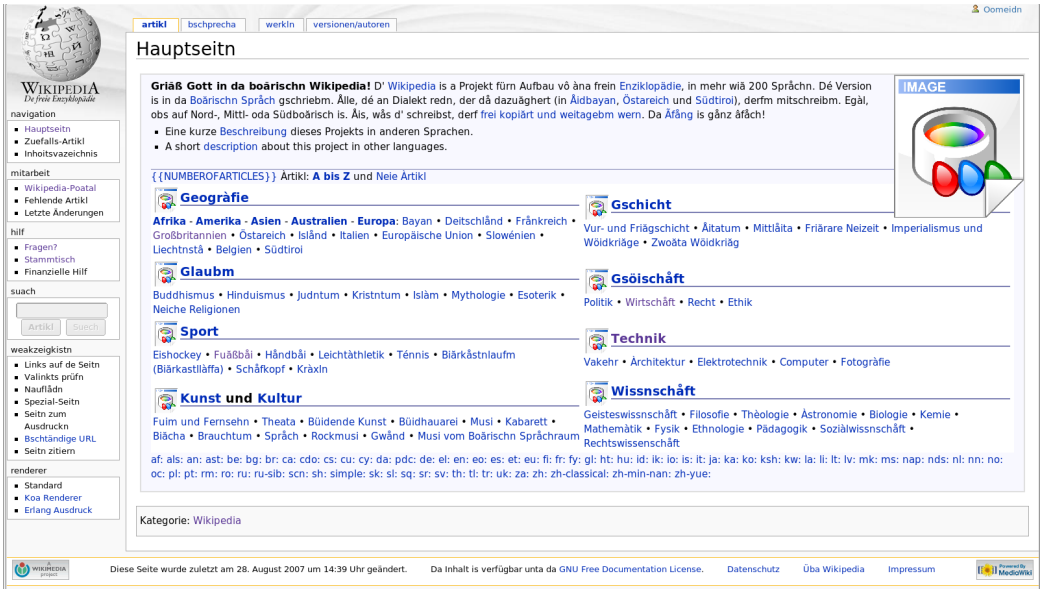


Figure 12.2: Screenshot of the Bavarian Wikipedia on Scalaris. Images are not included in the dump.

12.4 Papers and publications

12.4.1 Scalaris: Reliable Transactional P2P Key/Value Store

Thorsten Schütt, Florian Schintke, Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. ACM SIGPLAN Erlang Workshop, September 2008 (see A.9).

> We present *Scalaris*, an Erlang implementation of a distributed key/value store. It uses, on top of a structured overlay network, replication for data availability and majority based distributed transactions for data consistency. In combination, this implements the ACID properties on a scalable structured overlay.
>
> By directly mapping the keys to the overlay without hashing, arbitrary key-ranges can be assigned to nodes, thereby allowing a better load-balancing than would be possible with traditional DHTs. Consequently, Scalaris can be tuned for fast data access by taking, e.g. the nodes' geographic location or the regional popularity of certain keys into account. This improves Scalaris' lookup speed in datacenter or cloud computing environments.

Scalaris is implemented in Erlang. We describe the Erlang software architecture, including the transactional Java interface to access Scalaris.

Additionally, we present a generic design pattern to implement a responsive server in Erlang that serializes update operations on a common state, while concurrently performing fast asynchronous read requests on the same state.

As a proof-of-concept we implemented a simplified Wikipedia frontend and attached it to the Scalaris data store backend. Wikipedia is a challenging application. It requires—besides thousands of concurrent read requests per seconds—serialized, consistent write oper ations. For Wikipedia's category and backlink pages, keys must be consistently changed within transactions. We discuss how these features are implemented in Scalaris and show its performance.

12.4.2 A Scalable, Transactional Data Store for Future Internet Services

Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Seif Haridi.A Scalable, Transactional Data Store for Future Internet Services. EU Future of the Internet Conference, May 2009 (see A.10).

Future Internet services require access to large volumes of dynamically changing data records that are spread across different locations. With thousands or millions of distributed nodes storing the data, node crashes or temporary network failures are normal rather than exceptions and it is therefore important to hide failures from the application.

We suggest to use peer-to-peer (P2P) protocols to provide self-management among peers. However, today's P2P protocols are mostly limited to write-once/read-many data sharing. To extend them beyond the typical file sharing, the support of consistent replication and fast transactions is an important yet missing feature.

We present *Scalaris*, a scalable, distributed key-value store. Scalaris is built on a structured overlay network and uses a distributed transaction protocol. As a proof of concept, we implemented a simple Wikipedia clone with Scalaris which outperforms the public Wikipedia with just a few servers.

Chapter 13

D5.3: Demonstrator application for Mozart (software)

13.1 Executive summary

The objective of this software deliverable is to implement a service that can benefit for the results presented in other work packages, mainly WP1 and WP3. We have developed a community-driven recommendation system called Sindaca. The system is built on a peer-to-peer network which uses the relaxed-ring topology to self-organize the peers that hold the state. The relaxed-ring is a result from WP1. The state is consistently and symmetrically replicated across the network using the transactional layer designed in WP3.

We have implemented a web interface to interact with the Sindaca peer-to-peer network. It provides basic functionality for a recommendation system with explicit collection of data. Users can recommend titles to the community, which are voted by other users. The state of every recommendation is also displayed as part of the feedback to the users.

13.2 Contractors contributing to the Deliverable

This software deliverable is the result of the work of partner UCL(P1).

UCL(P1) implemented the peer-to-peer network using the relaxed-ring topology. It also implemented the transactional layer with contribution of ZIB(P5) as reported in in deliverable D3.1c, Chapter 2. UCL(P1) also implemented the web pages to interact with the application running on the peer-to-peer network.

13.3 Sindaca recommendation system

13.3.1 Introduction

This deliverable presents a community-driven recommendation system named Sindaca, which stands for Sharing Idols N Discussing About Common Addictions. The name spots the main functionality of this application which is making recommendations on music, videos, text and other cultural expressions. It is not meant for file sharing to avoid legal issues, but for providing links to official sources of titles. The whole system is implemented using a peer-to-peer network self-organized with the relaxed-ring protocol, which is a result from WP1, see Appendix A.4. The state of the system is symmetrically replicated on the network using the transactional layer for decentralized storage management, presented in Chapter 2, being a result of WP3.

We have implemented a web interface to access the application. All requests done through the web interface are transmitted to a peer in the network which triggers the corresponding operations on the peer-to-peer network. The results are transmitted back to the web server, which presents the information in HTML format as in any web page. The way the application is interfaced through web and the way the information is stored in the network is similar to the architecture developed to implement the Wikipedia using Scalaris [65], see Chapter 12. We have extended this architecture with a notification layer which allows eager information updates. This layer is also used in the DeTransDraw application [56], reported in Chapter 18. However, this eager notification feature is not provided on the web interface.

To generalize the similitudes and differences between Sindaca and the above mentioned applications, we can say the following: the Wikipedia on Scalaris uses *optimistic* transactions using the Paxos consensus algorithm. DeTransdraw uses *pessimistic eager-locking* transactions using Paxos consensus algorithm with a *notification layer*. Sindaca is a combination of theses strategies. It uses *optimistic* transactions with Paxos extended with the *notification layer*.

Sindaca is available for demo testing at url:

`http://beernet.info.ucl.ac.be/sindaca`

Use the following login information to enter the system:

- Username: selfman
- Password: sindaca

Any modifications done will be stored in the network, but they are not persistent to the reinitialization of the network. In case of problems during the test, please check contact information on the web page.

Figure 13.1 shows Sindaca's welcome page with the sign in form on the left of the page, together with the menu. The screenshot shows user *fbrood* logging in.

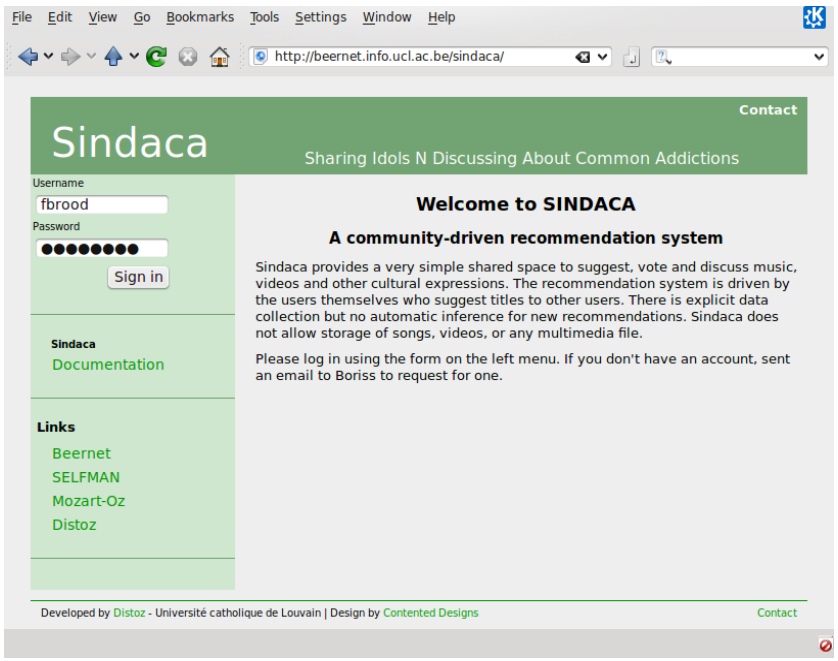


Figure 13.1: Sindaca's welcome page with sign in form

13.3.2 After sign-in and voting

If username and password are successfully provided, the user is taken to the profile page where information about the recommendations stored in the system is displayed. Figure 13.2 is a screenshot of the web page displayed after user *fbrood* has signed in. There is a welcome message both in the menu and in the center of the content. What follows is a list of recommendations suggested by members of the Sindaca community. This recommendation could have been made by other members or by the user itself. The recommendation is composed by a title, the name of the artist, and a link where the title can be found. As mentioned before, Sindaca does not provide storage for content preventing legal issues.

The listed recommendations are only those that has not received a vote from the user. A radiobutton is provided to express the preference which goes from no good to good. We have actually chosen a scale from *No beer* to *Beer*,

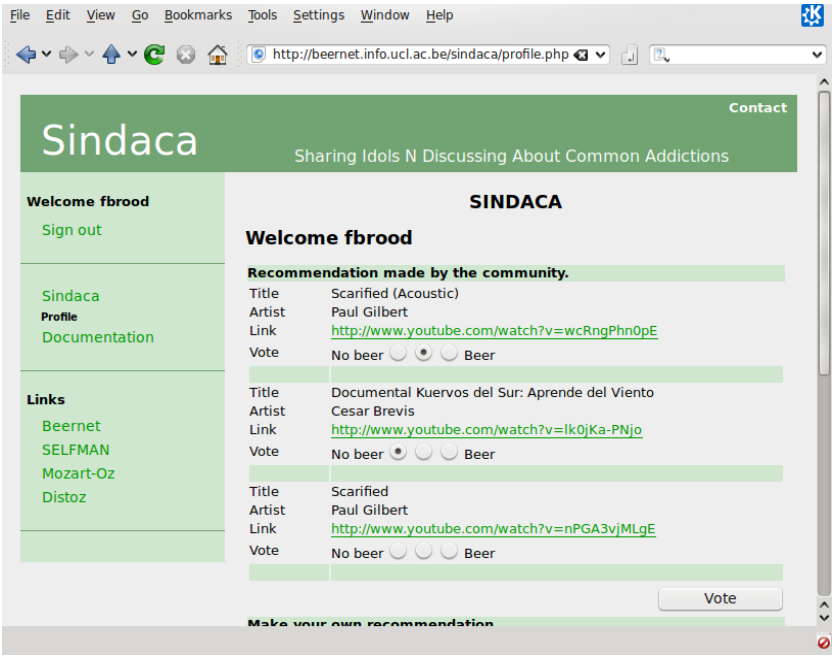


Figure 13.2: After sign in, users can vote for suggested recommendations.

because Sindaca is implemented on Beernet citeBeernet. The vote are submitted to the network when the user press the *Vote* button. Once the voting submission is sent, a transaction is triggered to modify the item that stores the recommendation. There are more items involved in this transaction, but the details will be explained in section 13.3.4.

13.3.3 Making a recommendation

The form to add a new recommendation is presented in the same page where the recommendations to be voted are displayed. The form can be seen in Figure 13.3 where the data for a new recommendation is already completed. The data to be filled in corresponds to the title, author, and link to the title. Once the data is submitted by pressing the button *Recommend*, a new item will be created in the network to store the recommendation, and this one will be associated to the user that creates it.

Every user has a list of recommendations she has made. This list is displayed in the same profile page, below the form for adding new recommendations. Therefore, the full profile page displays from top to bottom: welcome message, list of recommendations to be voted, form to add a new recommendation, and the list of recommendations already made by user.

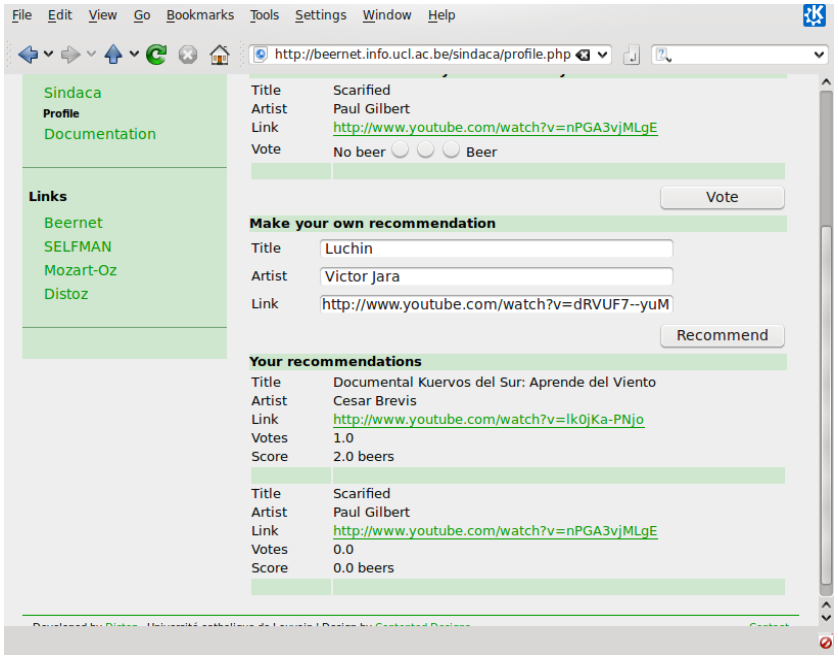


Figure 13.3: Adding a new recommendation.

Figure 13.4 shows how the last list is presented. Apart from the above mentioned fields, namely title, artist and link, the information contains two other fields being part of the state of every recommendation: the amount of votes, and the average score. The screenshot we display in Figure 13.4 was taken after the addition of the recommendation made in Figure 13.3. For that item we can observe that no vote is registered, and therefore there is no average score either.

13.3.4 Data storage structure

In the previous sections we described the functionalities of Sindaca, and we briefly introduced the effects of every action in the storage of the network. This section is dedicated to explain more about the details on the implementation of Sindaca.

First of all, it is important to remark that Sindaca it is not implemented on top of a database supporting SQL queries. Sindaca is implemented on top of a transactional distributed hash table with symmetrically replicated state. Therefore, the basic unit for storage is the key-value pair, which is what it is called *item*. The information of every user is stored as one item. The *value* of such item is a record with the basic information: user's id,

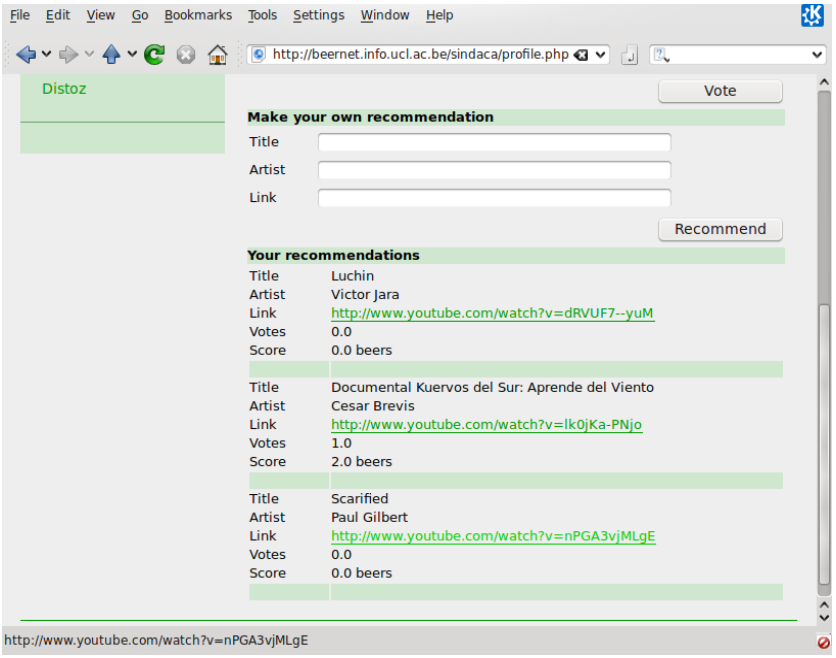


Figure 13.4: State of recommendation proposed by the user.

username and password. We have chosen a very minimal record to build the prototype, but the value can potentially store any data such as user's real name, contact information, age, description, etc. The *key* of the item is an Oz name [62], which is unique and unforgeable, acting as a capability reference. This strategy provides us with a certain level of security, because only programs that are able to map usernames with their capability can have access to the key, and therefore to the item. The username-capability mapping is only available to programs holding the corresponding capability to the mapping table.

The functionality of adding a new recommendation, shown in Figure 13.3, makes it clear that a recommendation belongs to a user. Therefore, every user item contains a list of capabilities which are references to recommendations. The functionality of voting also implies that every user item hold a list of capability references to votes. The relational model is depicted in Figure 13.5, and we can observe that a user can have multiple recommendations and multiple votes. What it is also stored in user's item is the list of recommendations already voted. That list will allows us to filter all other recommendations, presenting to the user only those she still have not voted yet.

From the relational model we can also observe that every recommendation

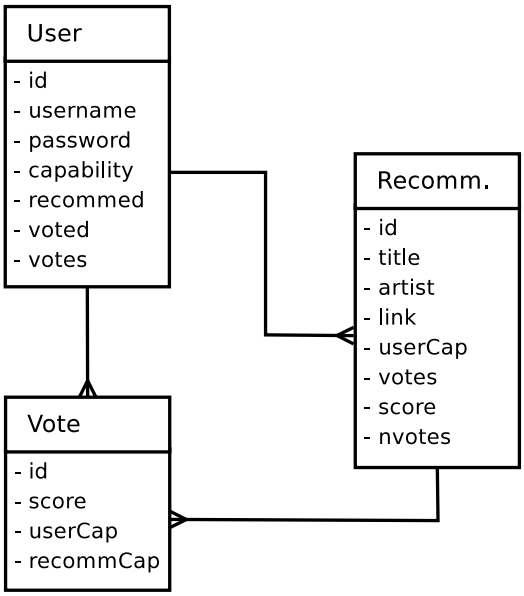


Figure 13.5: Sindaca's relational model

has a list of votes associated to it. Every vote contains information about the score, the user who made the vote, and the voted recommendation. What it is not shown in the relational model is how to find all the items on the network. There are two other items which store the list of all user's keys and all recommendation keys. Every time a new user or recommendation is created, these global items are modified. There is no global item for votes, because votes are accessible through the users and the recommendations.

The most complex transaction is triggered with the voting functionality. First of all, a new vote item is created. Then, the item of the voting user is modified in two fields. The list of voted recommendations is increased with the capability of the voted recommendation. The capability of the newly created vote is added to the list of votes of the user. Three fields of the voted recommendation are affected. The created vote is added to its list of votes, and the vote counter is increased by one, and the score average is recomputed taking the new value into account.

13.3.5 Configuration

The current version of Sindaca available for demo testing is configured with a peer-to-peer network of 42 nodes. All of the nodes are currently running on the server hosting Beernet's web page. The current state is a proof of

concepts, and we are planing to deploy the service on different machines. Some initial information is stored in the network in order to bootstrap the network and run the test. This information includes the creation of user *selfman*, which is available for testing. We have successfully run the service for several days allowing other users in our department to test the functionalities of the system by providing their own recommendations and voting on the recommendations of other users. The data provided by the testers is not persistent to the failure of the system. If the network needs to be restarted, the data provided by the testers is lost.

To transmit the information from the web interface to the network, we have running Mozart [62] process that listens to the Apache [25]-PHP [31] module which is reading the web requests. This Mozart process connects to a peer in the network in order to trigger the corresponding transaction. The implementation of the peer-to-peer network is done with P2PS/Beernet rev396 [54] or later, which is available for downloading on Beernet's web site, under the download section.

Chapter 14

D5.4a: Qualitative evaluation of autonomic features of Selfman applications

14.1 Executive summary

Many experimental works on autonomic computing aim at showing that such a disruptive technology can improve information systems in terms of quality of service and, de facto, in terms of return on investment as well. However still little research has attempted to demonstrate this intuition. Moreover such works can be characterized by their qualitative and fragmented approach. Thus there is still no generic (i.e. independent from business domain, technology, architecture and implementation choices) autonomic computing benchmarking tool for evaluating and/or comparing autonomic systems in terms of efficiency, reliability, availability, costs, etc.

The task T5.4 of Selfman WP5 aims at carrying out the first step toward a benchmark for evaluating autonomic computing in a technical and economical perspective. Its goal is to assess autonomic features both qualitatively and quantitatively. Thus it is worth noticing from the Selfman DoW that the goal is to demonstrate the effectiveness of autonomics capabilities of information systems. This goal will be achieved by comparing Selfman applications (provided by Peerialism(P6), ZIB(P5) and UCL(P1)) with and without autonomic features. So the purpose consists in focusing only on autonomic features in these applications and not in evaluating them functionally.

This deliverable intends to provide a methodology and process for evaluating qualitatively the self-management features. Section 14.3 provides a definition of these methodology and process. It is divided into two subsec-

tions.

- First section 14.3.1 describes the existing background by putting forward the state of the art related to autonomics benchmarking.
- Then section 14.3.2 defines the specificities of the approach that has been applied to Selfman applications.

Section 14.4 consists in carrying out assessment grids in order to apply this process on the Selfman applications (Peerialism(P6)'s PeerTV, ZIB(P5)'s Wiki on Scalaris and the UCL(P1)'s gPhone application). Section 14.5 and section 14.6 focus on the results obtained thanks this evaluation process. Finally section 14.7 aims at analyzing the methodology itself.

14.2 Contractors contributing to the Deliverable

Peerialism(P6), ZIB(P5), UCL(P1) and FT R&D(P4) participate to the writing of this deliverable.

FT R&D(P4) defined the evaluation framework (methodology and process) and provided support for applying it to the Selfman applications.

Peerialism(P6), ZIB(P5) and UCL(P1) provided feedback to the design of the evaluation framework, and then applied the assessment methodology on their own application (by fulfilling assessment grids).

14.3 Methodology and process

14.3.1 Background under discussion

Autonomic computing context

[40] and [43] define autonomic computing thanks to four main characteristics (self-configuration, self-healing, self-protection and self-optimization) and four secondary characteristics (self-awareness, context-awareness, openness and anticipation). The autonomic features provided by a system are embedded into autonomic managers. Each autonomic manager is responsible for a set of managed resources. Its processing can be divided into four stages [40].

Monitoring: the autonomic manager collects data coming from the constituents with which it is in charge (resources, probes, etc.) It can eventually handle these data with simple technical process like aggregating or filtering.

Analyzing: collected data are then analyzed regarding management policy.

Planning: according to analyzing results, the autonomic manager decides if a set of actions has to be executed and build a reaction plan.

Execution: in this last stage, the autonomic manager carries out the plan built in planning phase for replying to the original stimulus.

This decomposition into four stages is named MAPE-loop (for "Monitoring Analyzing Planning Executing-loop") or control loop. The term loop highlights the fact that the autonomic manager continuously and sequentially repeats these four stages. Such a decomposition is purely conceptual. In existing autonomic systems some phases can be merged together like monitoring with analyzing or analyzing with planning or else planning with execution. In the scale of autonomic computing maturity (see Table 14.4) for instance, the managed level consists of both monitoring and analyzing implementation. A MAPE-loop is independent from each different autonomic computing main characteristic and can be applied similarly to self-configuration, self-healing, self-protection and self-optimization. Each main autonomic characteristic –and each autonomic computing system as well– consists of many MAPE-loops that can interoperate together. A hierarchy can also exist between these MAPE-loops.

In the following, autonomic behavior (AB) designates the autonomic feature that is implemented by a MAPE-loop. Many ABs can so be associated with each main autonomic characteristic.

Models and metrics

The idea that underlies the definition of a global evaluation model of autonomic computing, consists in declining the eight autonomic characteristics from the well-known Factors-Criteria-Metrics (FCM) model [24]. Such FCM models enclose three levels of abstraction.

Factors (to specify): these characteristics contribute actively to the software quality by giving an external view (e.g. a user perception) of it. They can be directly or indirectly linked to the costs they imply.

Criteria (to build): these are system internal attributes that participate to quality factors estimation and definition (developer level). Each criterion can participate for estimating one or many factors. Each criterion is associated with one metric. A criterion can be objective (quantitative) or subjective (qualitative).

Metrics (to control): a metric provides a scale and a method for measuring the value of the associated criterion.

Each factor is made up of multiple criteria, each criterion being the result of a combination (heuristic) function of one or many metrics.

Since autonomic computing aims essentially at improving the QoS of systems, [71] and [98] try to define an autonomic computing evaluation model based on the standardized ISO/IEC 9126 software quality model [60]. The ISO/IEC 9126 software quality model defines six factors specifying external user views and about twenty criteria describing the internal view of the system. However no metrics are defined for evaluating these criteria due to the tight coupling between software quality and business domain. Table 14.1 describes the relationships between factors and metrics in ISO/IEC 9126 standard.

[71] and then [98] define the relationships between the eight autonomic computing characteristics and the six quality factors of ISO/IEC 9126 (see figure 14.1).

Moreover [49] intends to give a first standardized and quantifiable definition of autonomic computing. It notices effectively that for now research works only focus to answer to requirements without any industrial process. This lack of common and quantifiable definition leads to a collection of heterogeneous autonomic behaviors unable to interoperate in order to reach a common target. Thus [49] bases its approach on the Quality Metrics Framework (QMF) defined in IEEE 1061-1998 specification. Its purpose is to define some quality factors that would be evaluated thanks quality metrics. The

Table 14.1: Decomposition of quality factors in quality criteria in ISO/IEC 9126 standard

Quality factors	Quality criteria
Functionality	Suitability
	Accuracy
	Interoperability
	Compliance
	Security
Reliability	Maturity
	Recoverability
	Fault-tolerance
	Compliance
Usability	Learnability
	Understandability
	Operability
	Attractiveness
	Compliance
Efficiency	Time behavior
	Resource behavior
	Compliance
Maintainability	Stability
	Analyzability
	Changeability
	Testability
	Compliance
Portability	Installability
	Replaceability
	Adaptability
	Conformance

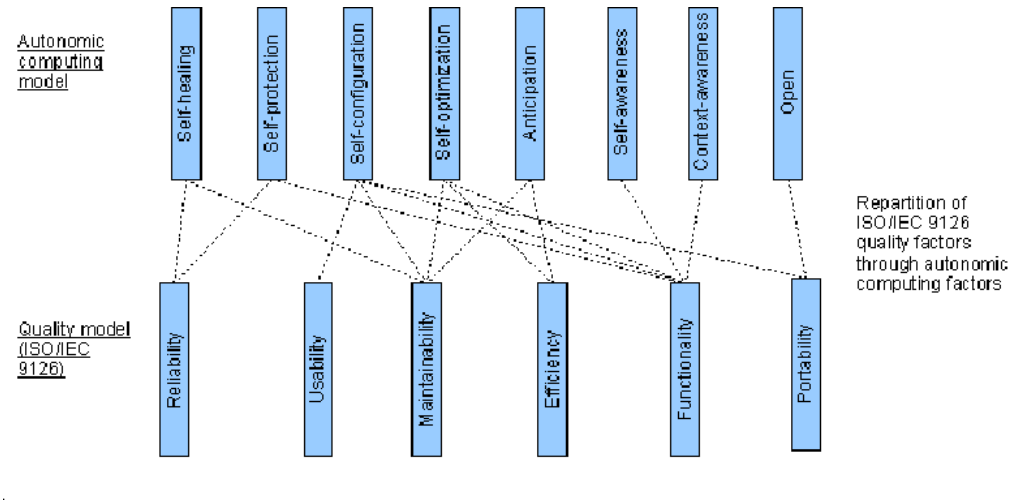


Figure 14.1: Organization of autonomic computing characteristics based on ISO/IEC 9126 standard quality factors (inspired from [71] and [98])

main result that this paper exposes is a hierarchy between autonomic computing characteristics (see figure 14.2).

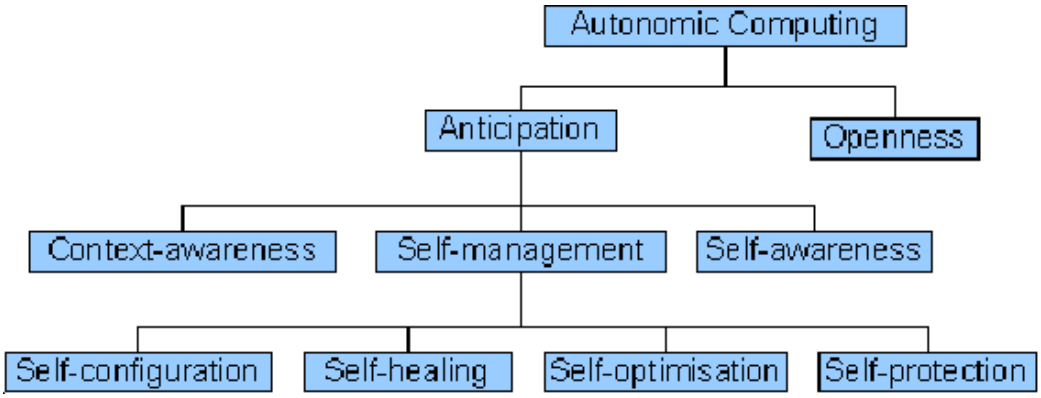


Figure 14.2: Hierarchy between autonomic computing characteristics (extracted from [49])

Finally concerning criteria evaluation, [53], [17] and [19] propose a non-exhaustive set of metrics which are not related to any evaluation standard such as ISO/IEC 9126. For instance [53] suggest the evaluation of two specific durations for measuring autonomics sensitivity. These metrics are the adaptation time (between the change detection and the while the adaptation has been done) and the reaction time (between the change occurrence and the while the system adaptation can start). Comparably [19] defines a

third metric which is the stabilization time (between adaptation start and the while the system is back to a stable state). See figure 14.3 for more details.

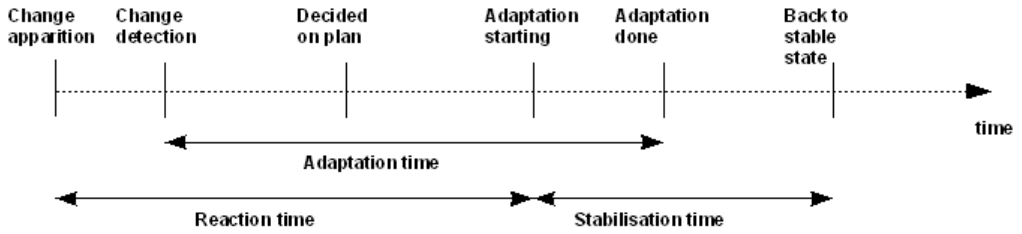


Figure 14.3: Definition of adaptation time, reaction time and stabilization time

All these works concerning models and metrics constitute indispensable steps toward providing a full autonomics benchmark, but they can be characterized by a qualitative and fragmented approach. [71] and [98] do not provide a quantitative composition of the factors for evaluating experimentally the autonomic characteristics, nor an adaptation of ISO/IEC 9126 model to autonomic computing field. [49] is not based on any FCM model and still has to be quantitatively validated. Finally, most of the metrics listed by [53], [17] and [19] are qualitative: their evaluation remains subjective.

14.3.2 Approach specificities

The main purpose of benchmarking is to estimate the improvement provided by a technology. Concerning autonomic computing benchmarking, a two-stage approach will be adopted.

- The first step consists in providing a technical benchmark for measuring the impact of ABs on the efficiency of the system in terms of availability, reliability and quality of service. This tool should be generic in order to be coupled with any convenient QoS benchmarking tool, according to the business domain of the evaluated system.

- The second step consists in coupling these technical measurements with known (industrial) management costs models in order to infer economical gains.

This approach intends to providing an autonomics benchmarking tool as independent as possible from the architecture and from the business domain of

the system under test (SUT). Thus it will be applied on many applications (differing according their business domain and/or their software architecture).

The task T5.4 of Selfman project aims at providing a qualitative and quantitative evaluation of ABs. Then future works will use these results for carrying out the technical and thereafter the economical benchmarks.

So we developed a hierarchical and extensible[1] model for characterizing autonomic capabilities. This model defines the whole constituents (i.e. factors, criteria and metrics) of an hybrid ISO/IEC 9126 model for the autonomic computing field (see figure 14.4), similarly to the adaptation of ISO/IEC 9126 model proposed by [97] and addressing the concrete case of test specification. We adopted a bottom-up approach integrating, supplementing and adapting the existing works (see section 14.3.1).

At the same time we identified a set of metrics (by using and enriching the list defined by [53], [17] and [19]) participating in the empirical evaluation of the model. These metrics must be generic, measurable and quantifiable –for those implied in the definition of quantitative high-level indicators–:

Generic (or business independent), in order to carry out an evaluation and comparison tool applicable to any business domain;

Measurable i.e. these metrics can be assessed independently of the choices related to the system in terms of architecture, design, implementation or technologies;

Quantifiable for metrics processed by composition functions as inputs in order to calculate quantitative higher-level indicators. Quantitative metrics are de facto quantifiable. Concerning qualitative metrics they are characterized by the subjectivity of their evaluation. However some of them are composed of discrete but orderable values (for instance, the level of maturity of an AB) whereas others are made of non orderable values (like the list of standards or technologies to which a component conforms). Only the first ones are quantifiable;

Let us shortly focus on some metrics mentioned in figure 14.4.

1. Monitorability, analyzability, planning capability and changeability are four time measurements mapped on each phase of the MAPE-loop (see figure 14.5). This decomposition results from the widely accepted con-

[1] We are conscious that autonomic benchmarking approach presents limits or difficulties because of some business domain specificities. However we would try to exhibit a model with generic criteria and metrics (macroscopic properties) and domain specific criteria and metrics as well e.g. for P2P systems, J2EE systems, etc. (microscopic properties). Hence the notion of an extensible model.

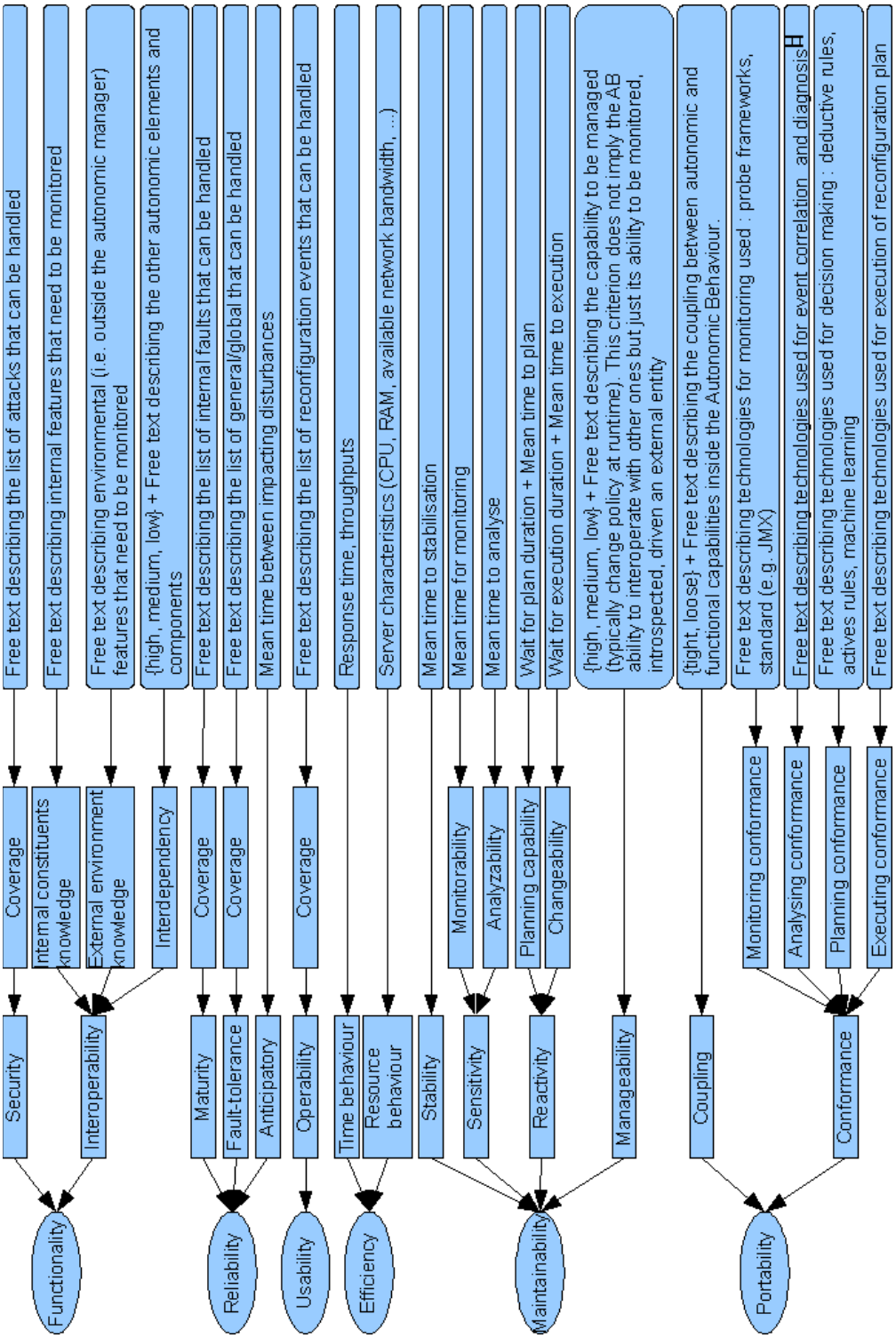


Figure 14.4: Adaptation of ISO/IEC 9126 to autonomic computing field

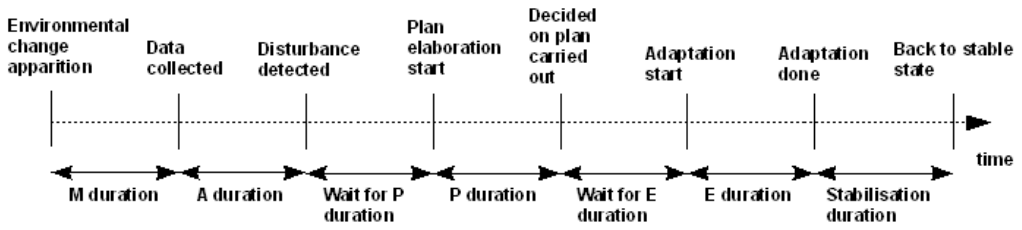


Figure 14.5: Mapping between MAPE-loop stages and their duration

cept of control loop. Moreover autonomic systems offer different levels of maturity that are mapped on the MAPE-loop as well. So it will be possible to quantify each duration according to the system maturity level.[2]

2. Anticipatory is the criterion in charge of evaluating the system ability for dealing with events, without modifying its QoS efficiency. Thus, among the whole disturbances the system is able to compute, the impacting ones are those which modify system efficiency.

3. Stability tends to measure the time the system needs to return to a stable state (i.e. where QoS values are stable).

[2]Similarly, this would have needed an important adaptation of time metrics proposed by [53]

14.4 Assessment process

For assessing ABs qualitatively, we defined two evaluation grids.

- The first one consists in grouping together the qualitative metrics defined in the hybrid ISO/IEC 9126 model (see figure 14.4). It allows to assess each AB independently.
- The second one contains higher level metrics for giving a synthesized view of the autonomic features of a given application.

14.4.1 Qualitative assessment of elementary AB

Table 14.2 allows the evaluation of each AB independently: one table / AB.

Table 14.2: Qualitative assessment grid for elementary AB

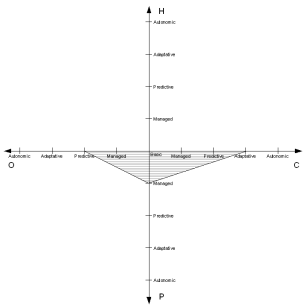
Autonomic Behaviour title (Bandwidth Optimization in P2PTV for example)		
Criterion	**Description**	**Example**
Related to self-* characteristic	{self-configuration, self-healing, self-optimization, self-protection}	self-optimization
Description of autonomic behaviour	Free text. Short description of the autonomic behaviour.	The bandwidth optimization in P2PTV consists in...
Internal constituents knowledge	Free text. Internal features that need to be monitored for this AB	
External environment knowledge	Free text. Environmental (i.e. outside the AB) features that need to be monitored for this AB	
Level of automation	{-, M, MA, MAP, MAPE}	MAPE (full loop)
Monitoring conformance	Free text. Technologies for monitoring used : probe frameworks, standard (e.g. JMX)	
Analysing conformance	Free text. Technologies used for event correlation and diagnosis	
Planning conformance	Free text. Technologies used for decision making : deductive rules, actives rules, machine learning	Explicitly programmed
Executing conformance	Free text. Technologies used for execution of reconfiguration plan	

Coupling	{Tight, Loose}. Free text. Description of coupling between autonomic and functional capabilities inside the Autonomic Behaviour.	Tight. The AB can not be enabled / disabled independently from the business features
Manageability	{High, Medium, Low}. Free text. Capability to be managed (typically change policy at runtime). This criterion does not imply the AB ability to interoperate with other ones but just its ability to be monitored, introspected, driven an external entity.	Low. The AB is inline coded. No possibility to change at runtime.
Interdependency	{High, Medium, Low}. Free text. Description of the other Autonomic Behaviours and components the current Autonomic Behaviour is depending on.	Low. This AB has no dependencies and no collaborations with other ABs
Coverage	Free text. List of disturbances with which the AB is able to deal	Peer joining and peer leaving

14.4.2 Qualitative assessment of global ABs

Table 14.3 provides a synthesized view of the whole autonomic features of an application: one table / application.

Table 14.3: Qualitative assessment grid for global ABs

Application name (Application1 for example)		
Criterion	**Description**	**Example**
"CHOP radar"	On a radar chart with four axis, each self-* main characteristic (Configuration, Healing, Optimization, Protection) is reported [19]. Each axis has for scale its associated level of maturity (see table 14.4)	
Self-configuration maturity level[3]	{Basic, Managed, Predictive, Adaptive, Autonomic}	Adaptive
Self-healing maturity level[3]	{Basic, Managed, Predictive, Adaptive, Autonomic}	Basic
Self-optimisation maturity level[3]	{Basic, Managed, Predictive, Adaptive, Autonomic}	Predictive
Self-protection maturity level[3]	{Basic, Managed, Predictive, Adaptive, Autonomic}	Managed
Support for interacting control loops	{Yes, No}. Free text. Description of ABs interactions	No

[3]Self-* characteristic maturity is evaluated thanks the scale proposed by IBM [92] (see table 14.4). At the AB level (local) we kept a scale with a better adequation to MAPE-lopp stages. Here are the equivalences between these two scales : (- ⇔ Basic, M ⇔ <not applicable>, MA ⇔ Managed, MAP ⇔ Predictive, MAPE ⇔ Adaptative, <not applicable> ⇔ Autonomic)

Support for stability	{Yes, No}. Have a stabilisation mechanism been developed in order to prevent from system instability or oscillations.	No

Table 14.4 contains the scale with the description of the five corresponding levels of autonomic computing maturity that have been proposed by IBM [92] and that are widely accepted.

Table 14.4: Scale and levels description of autonomic computing maturity

Level	**Description**
Basic	The product and environment expertise resides in human minds
Managed	Scripting and logging tools automate routine execution and reporting. Autonomic systems so carries out monitoring and analyzing phases from MAPE-loop providing human specialists with synthesized data for building action plans.
Predictive	Autonomic system is able to raise warning flags when predefined threshold are tripped. It also is able to propose some reaction plans according to the content of its centralized knowledge base.
Adaptive	Building on the predictive capabilities, the adaptive system takes action itself based on the situation.
Autonomic	Management policy drives system activities within a framework dealing with priorities

14.5 Experimental assessment of local ABs

This section contains the results obtained by applying the qualitative assessment grids for evaluating each local AB of each Selfman application.

14.5.1 PeerTV (Peerialism(P6))

The structure of the PeerTV platform comprises of a number of entities, as seen in Figure 14.6, but only four major ones are involved in content distribution: the Client, the Source, the Tracker and the Optimization engine (Opto). Clients are responsible of both retrieving content and distributing it to other clients, the source can be considered as a client which has all the content and can always provide other clients. The tracker is the central point of reference for all clients, it receives requests and issues responses following directions given by the optimization engine. The latter instead decides how to organize the overlay network such that the content is distributed efficiently to the clients. The Optimization engine keeps an overview of the overlay network state. Such information is built and maintained upon reception of periodic update messages from the clients. The network overview is vital for all decision processes which happen in the Optimization Engine. Three ABs are active in the PeerTV platform: a first one deals with the event of a new peer joining the system, i.e. of the self-configuration type, a second one of self-healing type is in charge with peer leaving or failure and the third one which is responsible of optimizing the flow of content over the overlay network which is a self-optimization AB.

Overlay Creation and Maintenance

When a new Client intends to join the overlay network, it contacts the Tracker on a well-known address. The Tracker receives the request and hands it out to the Optimization Engine which decides which peer(s) should the new client contact in order to receive content, being that a source or a normal client which has already received such content. The decision regarding the latter assignment is made taking into account a number of different factors, such as: bandwidth availability of provider peers, connectivity constraints observed on the underlying network and the preservation of content locality. Information which is included in the network image that has been built over time in the Optimization Engine. The aforementioned self-configuration mechanism ensures that all joining clients are provided with content and consequently that the overlay network is connected, i.e. all peers have a path to the source. Such Autonomic behavior cannot be switched off for evaluation purpose since

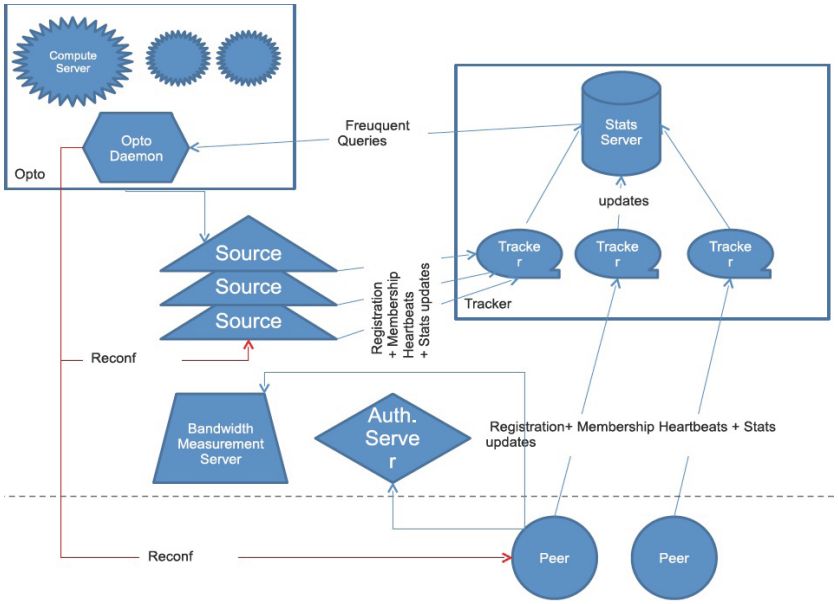


Figure 14.6: PeerTV System Architecture

it has vital role for maintaining the structure and functionality of the peer-to-peer network.

Another autonomic behavior of the self-healing type makes sure that all peers that experience problems during the receiving of the content are re-assigned to a new peer. This process is triggered by the client reporting a delivery failure to the tracker/opto. Such a failure might be caused by congestion of the network or a malfunction of providing peers. This AB cannot be disabled, since it would compromise the ability of the system to serve all peers, even those which experience problems over time.

Table 14.5: Joining peer assignment

Related to self-* characteristic	self-configuration

Description of autonomic behavior:	This AB is embedded in the Optimization Engine. It aims at assigning a set of provider peers P to a joining peer n, such that n is able to receive the requested content from peers in P. Switching o this behavior will prevent any new node to enter the delivery system. Peers are not authorized to provide content unless previously notified by the Tracker on behalf of the Optimization Engine, this prevents joining peers to contact directly content providers and makes the joining process essential. Upon event $<$ **join** $>$ event from n: • Check if n is authorized to enter the system • Choose a set of peers P to assign to n using overview O, which represents the state of the overlay network. • Update overview O with the outcome of the new assignment. • Forward set P to Tracker and then to n.
Internal constituents knowledge	Overview (O) built with information from actual members of the overlay network
External environment knowledge	The identifier and characteristics (Bandwidth capacity, connectivity constraints, public/private address, etc...) of peer (n).
Level of automation	MAPE
Monitoring conformance	Proprietary implementation of a Rendez-vous server (Tracker). Proprietary communication protocol between the server and the peers.
Analyzing conformance	Proprietary Processing engine for interpreting information provided by peers.
Planning conformance	Proprietary Assignment Algorithm.
Executing conformance	Proprietary communication protocol.

Coupling	This AB is built-in in the Optimization Engine. The system cannot accept any new peer if the AB is disabled .
Manageability	The AB can be stopped manually if needed but it will prevent new peers to join the system .
Interdependency	It does not depend on any other AB. Instead, it's the other self-optimization AB which depends on this one. This because the self-optimization process involves only peers that have already joined the system.
Coverage	This AB deals with the < **join** > event, corresponding to the occurrence of a new peer joining the system

Table 14.6: Failing peer assignment

Related to self-* characteristic	self-healing
Description of autonomic behavior:	This AB is embedded in the Optimization Engine. It aims at substituting the peers in set P serving n which have not succeeded to provide content. Switching o this behavior will prevent any failing node to re-enter the delivery system. Upon event $<$ **failure** $>$ event from n: • Check if n is authorized to enter the system • Choose a set of peers P to assign to n using overview O, which represents the state of the overlay network. • Update overview O with the outcome of the new assignment. • Forward set P to Tracker and then to n.
Internal constituents knowledge	Overview (O) built with information from actual members of the overlay network
External environment knowledge	The identifier of peer (n) and updated information on its status, e.g. buffers' status.
Level of automation	MAPE
Monitoring conformance	Proprietary implementation of a Rendez-vous server (Tracker). Proprietary communication protocol between the server and the peers.
Analyzing conformance	Proprietary Processing engine for interpreting information provided by peers.
Planning conformance	Proprietary Assignment Algorithm.
Executing conformance	Proprietary communication protocol.
Coupling	This AB is built-in in the Optimization Engine. The system cannot recover from any failure if the AB is disabled.

Manageability	The AB can be stopped manually if needed but it will create a big disruption in the system as peers fail and are discarded.
Interdependency	It does not depend on any other AB.
Coverage	This AB deals with the < **failure** > event, corresponding to the occurrence of an existing peer reporting a failure in the content delivery.

Content Distribution Optimization

The third AB in the PeerTV system is of the self-optimization type and performs the task of reorganizing the structure of the overlay network to optimize content distribution. This optimization process is executed periodically after T number of seconds, where T may vary according to the state of the overlay network, e.g. if the distribution load is not balanced among peers, the optimization process is executed more often, hence smaller values of T are chosen. The optimization process consists in re-organizing the assignments between peers, taking into account the following characteristics of the distribution network: bandwidth availability, connectivity constraints and issues, data locality, network delays and peering costs. Note that this information is contained in the overview that the Opto has built using detailed state messages from the client. The optimization process returns as a result the new assignments that must be enforced between the peers.

Table 14.7: Overlay Optimization Process

Related to self-* characteristic	self-optimization

Description of autonomic behavior	This is a full-fledge Autonomic Behavior where the MAPE loop can be easily identified. Monitoring: • Monitoring phase is carried out by both Client and Tracker entities, as the former sends information about its local state and the latter aggregates it before handing it out to the Opto. Analysis: • When the period T is expired, the Opto engine aggregates all information provided previously by the peers during the monitoring phase in a matrix structure (the previously mentioned network overview O). Processing: • The matrix built during the Analysis phase is given as input to the Optimization engine. The Optimization engine solves the linear optimization problem associated with the assignment of peers to each others. Execution: • The Optimization engine returns a set of new assignments. Note that not all assignments are changed but only the ones considered not optimal. The actual sending of the messages is performed by the tracker on behalf of the Opto.
Internal constituents knowledge	Overview (O) built with information from actual members of the overlay network acquired over time
External environment knowledge	Overview (O) of the network status can also be considered as external knowledge, since it represents the status of the peers.
Level of automation	MAPE

Monitoring conformance	PeerTV's proprietary communication protocol
Analyzing conformance	Proprietary Processing engine for interpreting information provided by peers.
Planning conformance	Proprietary Assignment Algorithm.
Executing conformance	Proprietary communication protocol.
Coupling	This AB can be disabled at any time, although this will lead to poor performance of the whole system.
Manageability	The AB can be stopped manually. There is no automatic way to stop such mechanism
Interdependency	This self-optimization AB depends on the overlay maintenance one. This because the self-optimization process considers only peers that have already joined the system.
Coverage	This AB deals with periodic self-configuration process of the system

14.5.2 Scalaris (ZIB(P5))

Note: Scalaris and the gPhone application are built on top of structured overlay networks (SONs) providing distributed hash tables (DHT), Chord# and the relaxed-ring respectively. Some of the ABs overlap in the two projects whereas others are implemented differently.

Ring Maintenance

Two ABs are in charge of the ring maintenance regarding respectively Chord# and the relaxed-ring. The first one is related to self-configuration and deals with new node joining the network. The second one is related to self-healing and fixes the ring when a peer leaves the network. Both of these ABs provide ring maintenance under churn. Each AB works slightly differently on each network.

Peer joining process in the Chord# ring. All nodes in Chord# provide look-up functionality for finding the node succeeding an identifier. A new peer joining the overlay contacts a bootstrap node already part of the overlay for determining its successor using a look-up (see fig. 14.7). We do not describe the bootstrap node as part of the AB since it is not involved further than providing the basic look-up functionality.

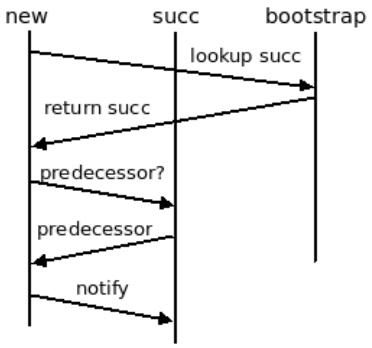


Figure 14.7: Sequence diagram of a join and stabilization.

When the new node has located its successor, it uses the stabilization protocol (see paragraph concerning Chord# stabilization in section 14.5.2) to repairs the ring (i.e. ensures that a node has both successor and predecessor). This is not done by a central point of control but by local arrangement of peers in the neighborhood of the joining peer. This stabilization mechanism cannot be switched off because it role is essential for maintaining the

coherence of the peer-to-peer network. A node consist of an overlay ID and a network address.

At the joining node.

Table 14.8: Joining Node

Related to self-* characteristic	self-configuration
Description of autonomic behavior	The node starts with an empty predecessor. This is updated as part of the stabilization. The successor is retrieve by performing a look-up to the bootstrap node. **Upon** *lookup_response* **event:** Set our successor with the returned successor.
Internal constituents knowledge	Internal state: successor node, bootstrap node, local node
External environment knowledge	The identifier of the bootstrap node used to find a successor
Level of automation	MAPE
Monitoring conformance	General DHT common agreement. I.e. proprietary protocols and event-handling.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	None.
Interdependency	Stabilization finalizes the join.
Coverage	*lookup_response*

Stabilization of a Chord# ring This behavior is used to correct the successor/predecessor of nodes due to failures, joining nodes and leaves. When the stabilization executes on a node, it first asks for the successors predecessor. If this node is not the local node, it changes the successor to this node. The last step is to notify the successor that the local node thinks it is the predecessor (see fig. 14.7).

Check correctness of the successor's predecessor.

Table 14.9: Periodic stabilization of successor's predecessor

Related to self-* characteristic	self-healing
Description of autonomic behavior	This AB is embedded in each peer. It aims at checking regularly that the predecessor p of the successor s of the considered node n is still the correct node in the ring. Switching off this behavior will quickly lead to an inconsistent ring since any node uses its predecessor to know its responsibility range. A *stabilize* event is triggered periodically by the node (e.g. called first time the node is started and then re-sent with a periodic delay at the end of the handling of the *stabilize* event) **Upon** *stabilize* **event:** send *predecessor*? to s in order to retrieve p from s. Resend a *stabilize* event to ourselves with some delay (configuration parameter). **Upon** *pred_response* **event:** This event contains p. If p is between the overlay identifiers of n and s, the successor of n is set to p. Thereafter, tell s that we think we are its predecessor by sending $notify$ to s. The last step is used by s to set p.
Internal constituents knowledge	Local node (n) and the successor (s)

External environment knowledge	Predecessor p of the successor (p)
Level of automation	MAPE
Monitoring conformance	General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	None.
Interdependency	Notify at successor, Predecessor request
Coverage	*stabilize* and *pred_response* events

Notify at successor.

Table 14.10: Periodic stabilization

Related to self-* characteristic	self-healing
Description of autonomic behavior	Make sure that the considered node n has the correct predecessor p. Waits for a notification from a node that thinks is our predecessor, p'. Updates the local predecessor variable if it is better than the current predecessor p. **Upon** $notify$ **event:** If the overlay ID of p' (that sent $notify$) is between the current p and n, update $p = p'$.
Internal constituents knowledge	Predecessor identifier (p)
External environment knowledge	Predecessor candidate identifier (p')
Level of automation	MAPE
Monitoring conformance	General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	None.
Interdependency	Check correctness of the successors predecessor.
Coverage	$notify$ event

Predecessor request.

Table 14.11: Periodic stabilization

Related to self-* characteristic	self-healing
Description of autonomic behavior	Returns a predecessor p as requested by a possible predecessor p'. **Upon** *pred_request* **event:** Return the current value of p to p' via the *pred_response* event.
Internal constituents knowledge	Predecessor identifier (p)
External environment knowledge	Predecessor candidate identifier (p')
Level of automation	MAPE
Monitoring conformance	General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	None.
Interdependency	Check correctness of the successors predecessor.
Coverage	*pred_request* event

Periodic maintenance of Chord# fingers The finger table contains long-range pointers used to reduce the number of hops required to find a node responsible for a given key. The finger table is updated periodically with a similar mechanism as described for the ring maintenance (see paragraph concerning Chord# stabilization in section 14.5.2). The table itself contains a mapping from an ID to a node and is local state at each node in the system. The maintenance mechanism tries to find the node in the system that is the next closer to the ID.

Fix finger.

Table 14.12: Fingers

Related to self-* characteristic	self-optimization, self-healing
Description of autonomic behavior	Let n represent the local node and $n.id$ the local node's ID. Note that the ID is part of a larger ID space, typically in the range $[0, 2^k)$, where $k = 256$ for example. $fix\_fingers$ is called periodically. **Upon** $fix\_fingers$ **event:** the peer generates a set of IDs which are increased exponentially with the local ID as reference, e.g. $x = n.id + 2^i$ for i in the range $[0, k-1]$. For each of these IDs, a *lookup_request* is sent by the local node n containing the ID and i identifying the position in the finger table. **Upon** *lookup_response* **event:** This event contains a remote node n' and i. Let f_i be node i in the lookup table. If $n'.id$ is closer in the ID space to $x = n.id + 2^i$, then set $f_i = n'$.
Internal constituents knowledge	Local node n
External environment knowledge	Finger table
Level of automation	MAPE

Monitoring conformance	General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Loose. System works without, but look-ups will be $O(n)$ instead of $O(\log N)$
Manageability	None.
Interdependency	Standard look-up functionality of DHT, however this is not detailed since it is not an AB .
Coverage	$lookup\_response$, $fix\_fingers$

Global Data Storage

The storage service uses the SON for membership maintenance and to assign data items to nodes. An item is mapped into the identifier range used by the overlay and is stored at the node responsible for the item. Each item is replicated to a set of other nodes in the overlay according to a predefined scheme such as successor list or symmetric replication. Below we describe the MAPE-loops for load balancing and replica maintenance.

Load-balancing Due to key distribution skew, data access (read/write) and churn (node join/leave/failure) nodes in the system can become overloaded. The goal of load balancing is to even out the load over the nodes according to the current system state and the nodes capacity. The protocol is periodically triggering the local node to request the load of a set of other nodes, typically log(N).

Table 14.13: Load-balancing

Related to self-* characteristic	self-optimization

Description of autonomic behavior	This AB is embedded in each peer. It deals with two kinds of event: **Upon event** *load_request* requests the local node to provide its load attributes to the sender of this event; **Upon event** *load_response* contains the load attributes of a node to a *load_request* event. Periodically (triggered by a local timed event loop) the local node starts by sending *load_request* events to a set of other nodes. The members of this set are determined randomly. Each remote node sends a *load_response* event containing information about its load. This stage lasts until the local node get the responses of all remote nodes or till the expiration of a time out. Thereafter the load attribute from the random sampling and from the local node load are compared. According to the results from all *load_response* messages, the AB greedily find the node that can be used to lower the imbalance the most. It decides if to perform either a jump (leave/re-join), slide (leave/re-join within the range of the predecessor and successor) or no operation. Upon *load_request* event reception, the AB triggers a *load_response* event containing information on its load. This event is addressed to the node whose identifier is contained in that *load_request* event.
Internal constituents knowledge	Load state and node identifier
External environment knowledge	State of other node's load. IDs of nodes are retrieved by performing a look-up on a randomly generated ID in the ID space of the SON.
Level of automation	MAPE
Monitoring conformance	General DHT common agreement.

Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	loose, System works without this AB, but it leads to worse performance and reliability due to overloaded nodes.
Manageability	None.
Interdependency	None
Coverage	*load_request* and *load_response* events (i.e. balancing of data)

Replica Maintenance The replica maintenance is a continuous process ensuring that a node stores the replicas it is responsible for. That is, all items which are in the range between a node's successor and itself. In addition, each node is storing replicated items from an identifier range according to a predefined schema, e.g. symmetric over the ring. We call the nodes storing replicas of a range its replica set. The periodic maintenance of replicas described below is orthogonal to the placement schema. To simplify, we also ignore details on consistency management such as item locking and versions. The algorithm first synchronizes the set of items stored in the replica ranges. In the second step the node fetches any items which are missing in the local store. For a detailed description of replica maintenance, we refer to deliverables 9 and 3.

Table 14.14: Replica maintenance

Related to self-* characteristic	self-healing
Description of autonomic behavior	Each node is having a local store in memory or persistently on disk that store items, (key, value)-pairs. A key is part of the overlay ID space, i.e. $[0, 2^k)$, where $k = 256$ for example. **Upon event** *replica_range_request* contains a range $[r_{start}, r_{end}]$ and a source node n'. Return all keys in the specified range found in the local storage to n' via the *replica_range_response* event. **Upon event** *replica_range_response* Compare the difference between the local store and the result of the *replica_range_response*, any items missing locally is fetched by using the look-up mechanism of the SON. **Upon event** *replica_maintenance* Generate *replica_range_request* messages for the nodes in the replica set according to the replication scheme (see deliverable 9).
Internal constituents knowledge	Local data store

External environment knowledge	The nodes storing replicas and the stored items.
Level of automation	MAPE
Monitoring conformance	General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. There will be data loss without replica maintenance
Manageability	Rate of maintenance
Interdependency	None
Coverage	*replica_range_request*, *replica_range_response*, *replica_maintenance*

14.5.3 The gPhone application (UCL(P1))

Note: Scalaris and the gPhone application are built on top of structured overlay networks (SONs) providing distributed hash tables (DHT), Chord# and the relaxed-ring respectively. Some of the ABs overlap in the two projects whereas others are implemented differently.

Ring Maintenance

Two ABs are in charge of the ring maintenance regarding respectively Chord# and the Relaxed-Ring. The first one is related to self-configuration and deals with new node joining the network. The second one is related to self-healing and fixes the ring when a peer leaves the network. Both of these ABs provide ring maintenance under churn. Each AB works slightly differently on each network.

Peer joining process in the relaxed ring. A new peer q joins in between two nodes p and r, where p, q and r are identifiers in a circular address space where $p < q < r$. We say that p is the predecessor of q and q is the successor of p. The identifiers of a peer and its predecessor (respectively its successor) constitute the both ends of an interval of hash-keys in a Distributed Hash Table (DHT). Thus each peer is responsible for the keys included in the interval delimited by its predecessor identifier (excluded from the interval) and its own one (included in the interval). its own one.

Three peers are interacting in order to provide the full process dealing with a new peer joining the network. A distinct role is associated to each of them: *successor*, *predecessor* and *new peer*. The *successor* role aims at dealing with a *new predecessor*, the *predecessor* role handles with a *new successor* and the *new peer* role gets its first successor and predecessor pointers, performing a *join*. Figure 14.8 shows the sequence of events resulting on the insertion of the new peer in the Relaxed Ring. Each role participating to a

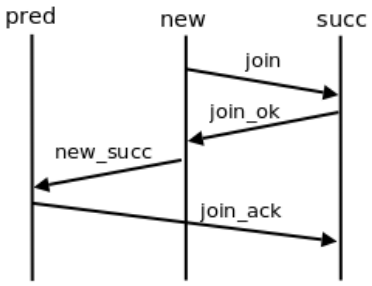


Figure 14.8: Sequence diagram of the join AB

new peer joining consists in a local AB embedded in the corresponding peer. Figures 14.9, 14.10 and 14.11, depicts the feedback loop of each AB.

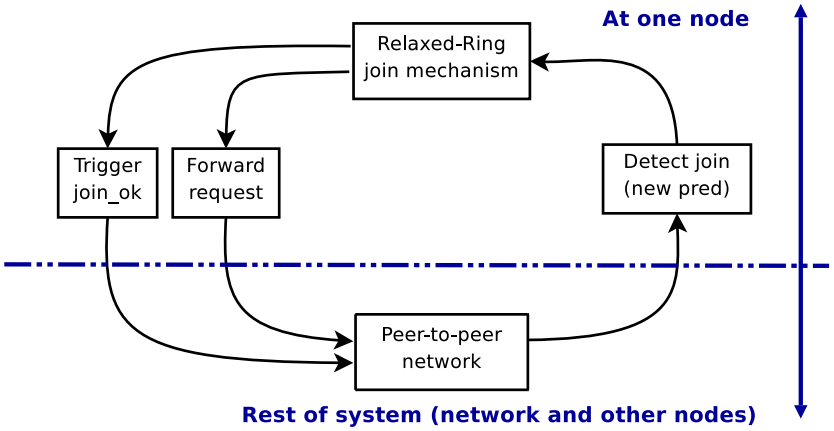


Figure 14.9: Ring maintenance as a feedback loop. New peer join as new predecessor of the current responsible of its key

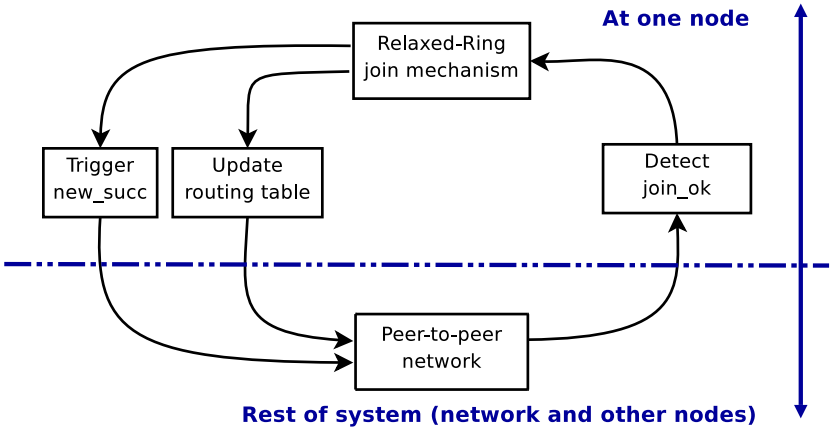


Figure 14.10: Ring maintenance as a feedback loop. New peer is accepted to join between p and r, and becomes the new successor of peer p

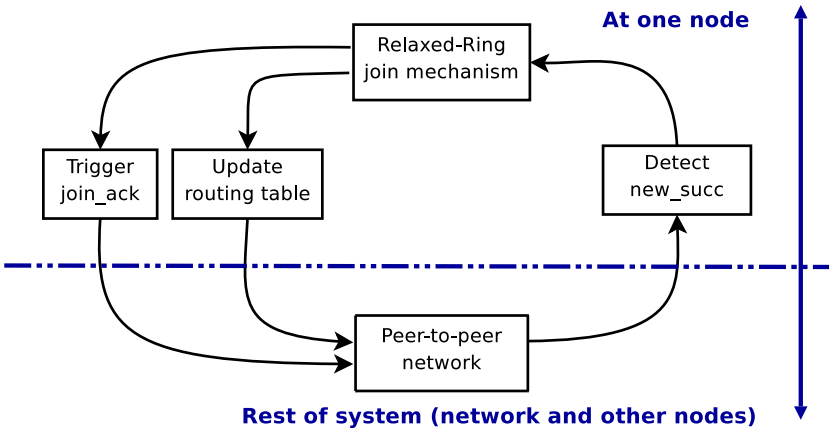


Figure 14.11: Ring maintenance as a feedback loop. A peer is notified about its new successor

***newpred*: new predecessor.**

Table 14.15: Predecessor update on join

Related to self-* characteristic	self-configuration

Description of autonomic behavior	When a new peer joins the relaxed ring it sends a request to a peer randomly selected. Then this one determines if the new peer is a better predecessor. In such a case, it replaces its old predecessor with this new peer. Otherwise, the joining request if forwarded to another peer. Role *succ* in figure 14.8 illustrates this AB. **Upon** *join* **reception:** the successor peer determines if the new peer is a better predecessor, i.e. if the new peer owns an identifier included between its own identifier and the identifier of its predecessor. If this is the case, the current peer accepts its request by sending it a *join_ok* event. Otherwise the current peer forwards the *join* event to another peer. Another reason to accept a *join* request is when the current predecessor is detected to have failed. In such case, an alive predecessor is always better than a dead one. **Upon** *join_ok***:** such an event consists in the final acknowledgment sent by the old predecessor to the successor in order to indicate the normal end of the join process. However the lack of this last event does not avoid the relaxed ring to properly work.
Internal constituents knowledge	Internal state: own identifier, predecessor identifier
External environment knowledge	Joining peer identifier
Level of automation	MAPE
Monitoring conformance	General DHT common agreement [4]

[4]Existing peer-to-peer networks implementing Distributed Hash Tables (DHT) use the notion of peer joining and leaving the network, and exchanging messages between them in

Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	None.
Interdependency	*newpred* depends on *join*. It depends on *newsucc* as well, but not strongly. Independently from the *join_ack* event that is 1) an output of *newsucc* AB and 2) an input of *newpred* AB, the relaxed-ring can still continue to works
Coverage	*join* and *join_ack* events

order to handle these events. The messages exchanges, such as *join*, *new_succ*, etc., are the commonly accepted and understood by the community. We take the same approach

***join*: new peer joins the ring.**

Table 14.16: Predecessor and successors initialization on join

Related to self-* characteristic	self-configuration
Description of autonomic behavior	This is the AB carried out by the new peer joining the relaxed ring. The new peer first contacts the successor candidate by triggering a *join* event. If it succeeds, the new peer will get a *join_ok* back, with the needed information to contact the predecessor. Then it will trigger a *new_succ* to the old predecessor (of the successor) notifying it that it becomes its new successor. The role *new* in figure 14.8 illustrates this AB.
Internal constituents knowledge	Internal state: own identifier
External environment knowledge	Successor identifier and predecessor identifier
Level of automation	MAPE
Monitoring conformance	General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	None.
Interdependency	*join* depends on *newsucc*
Coverage	*join_ok* event

***newsucc*: new successor.**

Table 14.17: Successor update on join

Related to self-* characteristic	self-configuration
Description of autonomic behavior	A peer in the relaxed ring is contacted by a new joining peer: it receives a notification (i.e. *new_succ* event) indicating that this new peer requests for becoming its new successor. Then it checks the identifier of the new peer and compares it with the identifier of its own current successor. If the new joining peer is a better successor, it accepts it, and notifies the old successor with a *join_ack*. A better successor is a peer whose identifier is ranged between the identifier of the current peer and the identifier of its own successor. The role *pred* in figure 14.8 illustrates this AB.
Internal constituents knowledge	Internal state: own identifier, successor identifier
External environment knowledge	Joining peer identifier (new successor)
Level of automation	MAPE
Monitoring conformance	General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	None.
Interdependency	*newsucc* depends on *join*
Coverage	*new_succ* event

Peer leaving and failing process in the relaxed ring We do not handle gentle leaves on the relaxed-ring, because in case of failure during the process of leaving, the failure mechanism would still be necessary. Therefore, double work would be needed. By handling correctly failure recovery, both cases are covered at once.

When a peer q crashes, it is detected by its predecessor and its successor. Only q's predecessor reacts to this crash event by sending a *join* message to q's successor. This join message triggers the join loop (see paragraph concerning peer joining in section 14.5.3). The reaction to failure detection is described as a sequence diagram in Figure 14.12. Because the failure recovery triggers the join autonomic behavior to fix the ring, the full feedback loop for failure recovery is shown in Figure 14.13.

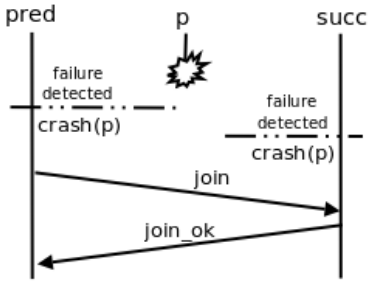


Figure 14.12: Failure recovery sequence diagram

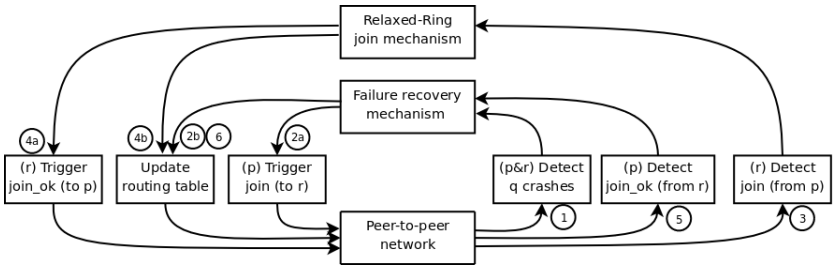


Figure 14.13: Failure recovery as a feedback loop

Table 14.18: Handling peer leaving/failure - Correction-on-change

Related to self-* characteristic	self-healing

Description of autonomic behavior	The failure of a peer is detected by the successor and the predecessor by monitoring the *crash* event. Only the predecessor takes action in order to fix the ring. If the peer is falsely suspected, an event *alive* will be monitored, rolling back the changes. This AB is embedded in each peer. **Upon** *crash* **event:** the local peer determines if the crashed peer is its successor. In this case, the AB runs the failure recovery by removing crashed peer from routing table and then triggering *join* on successor candidate in order to start self-configuration loop. **Upon** *alive* **event:** if this events contains the identifier of the 'supposed' crashed peer (i.e. peer was false suspected), the AB undoes changes: routing table is fixed. **Upon** *join_ok* **event:** the AB is informed that the failure recovery mechanism succeed.
Internal constituents knowledge	Its own identifier
External environment knowledge	Identifier of the crashed peer and successor's identifier
Level of automation	MAPE
Monitoring conformance	Eventually perfect failure detector: strongly complete and eventually accurate. Strongly complete means that all crashed peers will be detected. Eventually accurate means that if an alive node is falsely suspected of having crashed, this inaccuracy will eventually be corrected.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.

Executing conformance	General DHT common agreement.
Coupling	Tight. This AB is built-in the code of each peer. Moreover the system can not work if this AB is disabled.
Manageability	Parameter of failure detector can be tuned: • keep alive rate: How often keep alive messages are sent to other peers. • timeout value: How long is the time to wait for an acknowledgment of the keep alive message.
Interdependency	It strongly depends on *newpred* and *join* (see paragraph concerning peer joining in section 14.5.3)
Coverage	*crash* and *alive* event (i.e. peer leaving the relax-ring, peer failure in the relax-ring and false suspected (to leave/fail) peer

Finger maintenance with correction-on-use. The following evaluation grid describe the autonomic behavior in charge of maintaining the finger table for routing on the relaxed-ring. The behavior is divided in two parts: failure handling and correction-on-use.

A failure detector constantly monitors fingers. A change on the fault state of a finger will be triggered to the upper layer which will try to fix the broken finger.

The traffic that goes through a particular peer is also monitored. If new peers that can be better fingers are discovered, then, the finger table can be updated accordingly.

The combination of both monitoring actions combined into the maintenance of the finger table is depicted in the feedback loop of figure 14.14.

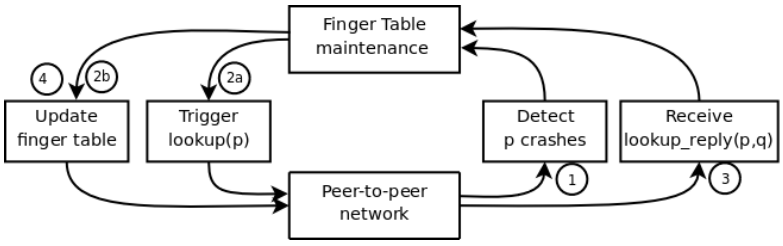


Figure 14.14: Finger maintenance with failure detection and correction-on-use

Table 14.19: Failure recovery of fingers enhanced with correction-on-use

Related to self-* characteristic	self-healing

Description of autonomic behavior	In order to maintain the finger-table as correct as possible, the fingers are constantly monitored to detect failures on them. The traffic going through the peer is also monitored in order to discover new peers in the neighborhood of the fingers. See figure 14.14. Monitoring: • Failure detector constantly checks the state of every fingers. A suspicion of failure is triggered as a *crash* event. • Communication layer receives messages and trigger them to the upper layers as events. This is the way to monitor traffic. Analysis: • upon event *crash(p)* and p is in finger-table, then prepare finger recovery. • upon event *lookup_reply(p, q)*, test finger p with q. Processing: • if finger p crashes, update finger table and trigger lookup for key p in order to find a better finger. • when *lookup_reply(p, q)* is received, fix finger p with q. Execution: • trigger *lookup(p)* to find a replacing finger. • update finger tables upon crashes and lookup replies (fixes).
Internal constituents knowledge	Finger-table

External environment knowledge	• fault-state of fingers. • Identifier of new peers detected monitoring the traffic.
Level of automation	MAPE
Monitoring conformance	• Eventually perfect failure detector: strongly complete and eventually accurate. Strongly complete means that all crashed peers will be detected. Eventually accurate means that if an alive node is falsely suspected of having crashed, this inaccuracy will eventually be corrected. • General DHT common agreement.
Analyzing conformance	General DHT common agreement.
Planning conformance	General DHT common agreement.
Executing conformance	General DHT common agreement.
Coupling	Semi-tight. It is not meant for correctness of the system, but for efficiency. But, if the fingers are not efficient enough, the whole system could become unusable in practice.
Manageability	Parameter of failure detector can be tuned: • keep alive rate: How often keep alive messages are sent to other peers. • timeout value: How long is the time to wait for an acknowledgment of the keep alive message.
Interdependency	It does not depend on other ABs

Coverage	Finger-table (routing-table) maintenance.

Global Data Storage

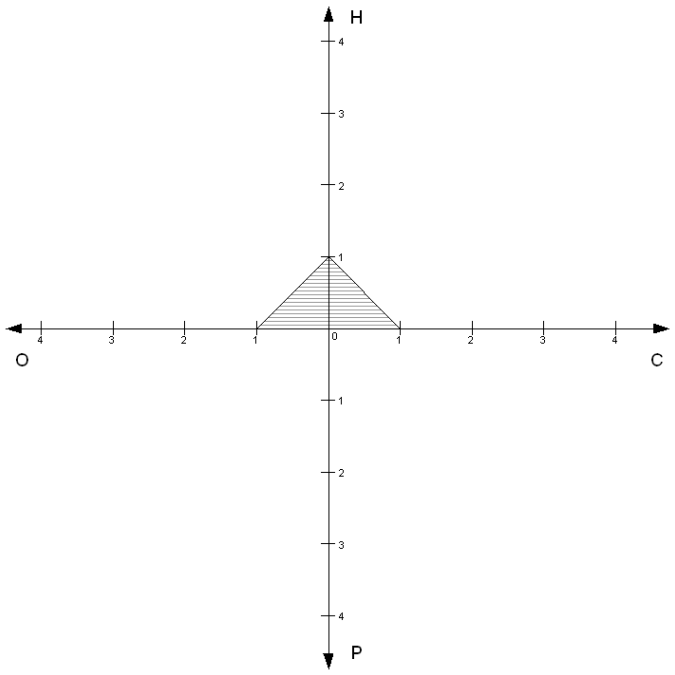
Refer to section 14.5.2

14.6 Experimental assessment of autonomics in Selfman applications

This section provides the global and qualitative evaluations of autonomics obtained for each Selfman application. In the following grids, the "CHOP radar" has been a bit modified. It shows the number of local ABs relative to each autonomic main characteristic. This allows to assess the distribution of ABs on each characteristic for a given application. However, notice that such a type of figures does not include the events coverage of each AB. So they can not offer a comparison of autonomy level between two applications. As written before they just show the distribution among the four autonomic main characteristics.

14.6.1 PeerTV (Peerialism(P6))

Table 14.20: PeerTV: qualitative assessment of global ABs

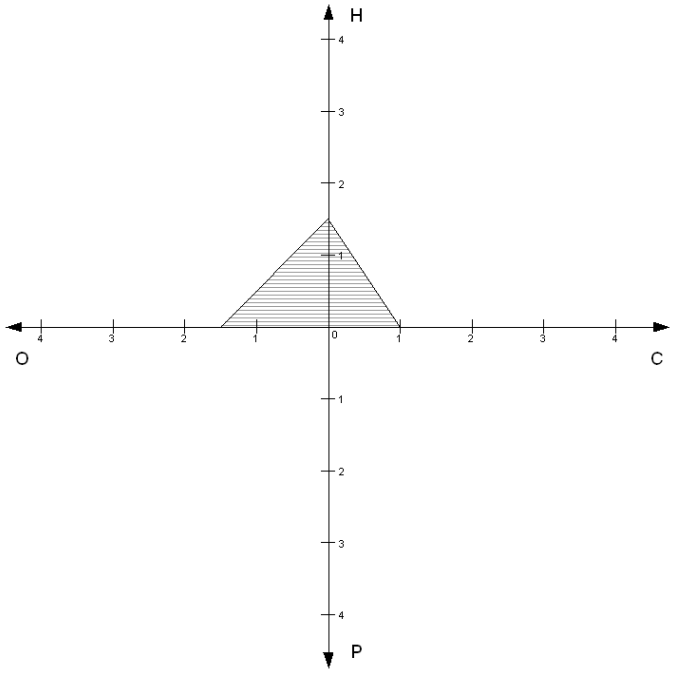
"CHOP radar"	
Self-configuration maturity level	Adaptive. One AB among 3 concerns self-configuration. It offers a MAPE maturity level without any knowledge management.

Self-healing maturity level	Adaptive. One AB among 3 concerns self-healing. It offers a MAPE maturity level without any knowledge management.
Self-optimization maturity level	Adaptive. One AB among 3 concerns self-optimization. It offers a MAPE maturity level without any knowledge management.
Self-protection maturity level	Basic. No AB concerns self-protection.
Support for interacting control loops	Yes (see section 14.5.1)
Support for stability	Not completed by Peerialism(P6)

14.6.2 Scalaris (ZIB(P5))

Notice: the number of ABs concerned by a criterion can be sometimes a float number because some AB concerns many autonomic characteristics simultaneously

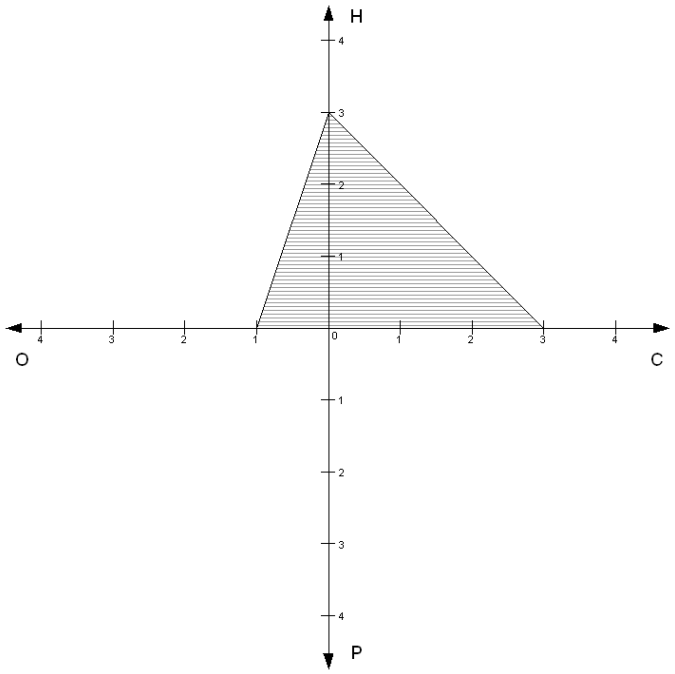
Table 14.21: Scalaris: qualitative assessment of global ABs

"CHOP radar"	
Self-configuration maturity level	Adaptive. One AB among four concerns self-configuration. It offers a MAPE maturity level without any knowledge management.
Self-healing maturity level	Adaptive. 1.5 ABs among four concerns self-healing. It offers a MAPE maturity level without any knowledge management.
Self-optimization maturity level	Adaptive. 1.5 ABs among four concerns self-optimization. It offers a MAPE maturity level without any knowledge management.
Self-protection maturity level	Basic. No AB concerns self-protection.
Support for interacting control loops	Yes (see section 14.5.2)

Support for stability	Yes. For example, the load balancing algorithm has a target load range to avoid oscillations.

14.6.3 The gPhone application (UCL(P1))

Table 14.22: The gPhone application: qualitative assessment of global ABs

"CHOP radar"	
Self-configuration maturity level	Adaptive. Three ABs among seven concerns self-configuration. It offers a MAPE maturity level without any knowledge management.
Self-healing maturity level	Adaptive. Three ABs among seven concerns self-healing. It offers a MAPE maturity level without any knowledge management.
Self-optimization maturity level	Adaptive. One AB among seven concerns self-optimization. It offers a MAPE maturity level without any knowledge management.
Self-protection maturity level	Basic. No AB concerns self-protection.
Support for interacting control loops	Yes (see section 14.5.3)
Support for stability	Every feedback loop is designed as a negative loop. Combination of loops have also been tested to present a negative loop behaviour.

14.7 Discussion

14.7.1 Experimental results

Table 14.23 summarizes the main tendencies highlighted by the qualitative evaluations of local and global ABs.

Table 14.23: Synthesis of qualitative assessment on local ABs coming from Selfman applications

Criterion	**Tendency**
Related to self-* characteristic	None of the assessed ABs is related to self-protection. Moreover, the distinction between self-optimization and self-configuration is not self-evident.
Coverage	All of these ABs can be considered elementary: they only deal with one or two low-level events
Interdependency	Most of these ABs interoperates with another one but this interaction is limited to a single query/reply and does not consist in a structured dialog
Internal constituents knowledge	The whole of these ABs run according to the monitoring data they get from the resources they manage
External environment knowledge	The whole of these ABs have very little (or even no) knowledge of their external environment
Level of automation	However all are fully autonomous regarding their level of maturation: none of them claims to be autonomic without fully implementing the four stages of the MAPE-loop
Monitoring conformance	Concerning their conformance to a standard and/or a wide spread / opened technology, all ABs propose highly proprietary implementation for this MAPE-loop stage
Analyzing conformance	Concerning their conformance to a standard and/or a wide spread / opened technology, all ABs propose highly proprietary implementation for this MAPE-loop stage

Planning conformance	Concerning their conformance to a standard and/or a wide spread / opened technology, all ABs propose highly proprietary implementation for this MAPE-loop stage
Executing conformance	Concerning their conformance to a standard and/or a wide spread / opened technology, all ABs propose highly proprietary implementation for this MAPE-loop stage
Coupling	Only two ABs are loosely coupled with the business functionalities in term of implementation and execution.
Manageability	Less than 30% of the evaluated ABs include manageability capabilities. However the management policy elements they can get are quite rudimentary (like a timer value or a combination of simple conditions). This illustrates the difficulty for converting high-level management policies into simpler rules understandable by the autonomic managers

Among these strong tendencies, the lack of self-protection ABs is explained by the systems editors as follows: some of them assume that their solution runs in a safe environment whereas others estimate that security has to be delegated to another system contributor. Unfortunately, these hypotheses seem to be invalid especially regarding open architectures such as peer-to-peer approach. A more likely explanation is that self-protection ABs have to deal with more complex events and thus can not be elementary behaviors, exclusively focusing on their internal resources. They might result from the composition (implying sophisticated interoperability) between different lower-level ABs. Thus the composition of ABs becomes an important research domain. However the lack of conformance to open standards, which characterizes elementary ABs, prevents any progress on their interoperability.

Concerning the tight coupling between autonomic features and business functionalities, it results essentially from:

1. the systems architecture of Selfman applications that lie on a wide distributed multi-agents architecture. Each of these agents is supposed to handle independently and must so embed autonomic behaviors;

2. the preexistence of managed resources. Selfman applications have been

developed from scratch and do not consists in the adding of an autonomic overlay on existing applications.

14.7.2 Evaluation methodology

This section analyses the proposed qualitative methodology for evaluating autonomic computing whereas section 14.7.1 focuses on presenting and analyzing the results coming from the technical benchmark.

The methodology for qualitatively assessing autonomic features is quite generic. It has been applied simultaneously to many systems differing according their business domain, their architecture and their implementation:

1. the three Selfman applications;
2. an application dealing with workload management in an environment composed of J2EE application servers.

In both cases this evaluation process highlighted some strong, macroscopic and common tendencies concerning the autonomic capabilities. It provides a quite wide and easy-to-measure set of criteria. However the composition of these criteria for obtaining higher level indicators –relative to the entire application for example– is difficult because of their qualitativeness (even if some of them are measurable (see section 14.3.2)). This is the main reason why a quantitative methodology has been provided as well: it tends to be more business specific whereas the qualitative one demonstrates its generic approach.

Chapter 15

D5.4b: Quantitative evaluation of autonomic features of Selfman applications

15.1 Executive summary

Many experimental works on autonomic computing aim at showing that such a disruptive technology can improve information systems in terms of quality of service and, de facto, in terms of return on investment as well. However still little research has attempted to demonstrate this intuition. Moreover such works can be characterized by their qualitative and fragmented approach. Thus there is still no generic (i.e. independent from business domain, technology, architecture and implementation choices) autonomic computing benchmarking tool for evaluating and/or comparing autonomic systems in terms of efficiency, reliability, availability, costs, etc.

The task T5.4 of Selfman WP5 aims at carrying out the first step toward a benchmark for evaluating autonomic computing in a technical and economical perspective. Its goal is to assess autonomic features both qualitatively and quantitatively. Thus it is worth noticing from the Selfman DoW that the goal is to demonstrate the effectiveness of autonomics capabilities of information systems. This goal will be achieved by comparing Selfman applications (provided by Peerialism(P6), ZIB(P5) and UCL(P1)) with and without autonomic features. So the purpose consists in focusing only on autonomic features in these applications and not in evaluating them functionally.

This deliverable intends to provide a methodology and process for evaluating quantitatively the self-management features. Section 15.3 provides a definition of these methodology and process. It is divided into two subsec-

tions.

- First section 15.3.1 describes the existing background by putting forward the state of the art related to autonomics benchmarking.
- Then section 15.3.2 defines the specificities of the approach that has been applied to Selfman applications.

Section 15.4 consists in carrying out assessment grids in order to apply this process on the Selfman applications (Peerialism(P6)'s PeerTV, ZIB(P5)'s Wiki on Scalaris and the UCL(P1)'s gPhone application). Section 15.5 focuses on the results obtained thanks this evaluation process. Finally section 15.6 aims at analyzing the methodology itself.

15.2 Contractors contributing to the Deliverable

Peerialism(P6), ZIB(P5), UCL(P1) and FT R&D(P4) participate to the writing of this deliverable.

FT R&D(P4) defined the evaluation framework (methodology and process) and provided support for applying it to the Selfman applications.

Peerialism(P6), ZIB(P5) and UCL(P1) provided feedback to the design of the evaluation framework, and then applied the assessment methodology on their own application (by fulfilling assessment grids).

15.3 Methodology and process

15.3.1 Background under discussion

Autonomic computing context

Refer to section 14.3.1.

Models and metrics

Refer to section 14.3.1.

Methodologies and tools for quantitative benchmarking

[13] underlines the challenges and pitfalls related to the development of an autonomic computing benchmark. A "classical" performances benchmark consists in a system under test (SUT) that is deployed in a stable environment (i.e. without any disturbance). A workload is then injected in the SUT. This workload is as representative (realistic) as possible compared to the load the SUT will have to deal during the exploitation phase. No management intervention occur during the benchmarking process. The response of the SUT to the injected work-load consists in evaluating some QoS metrics characterizing its efficiency (duration, throughput, etc.). The knowledge of the profile of the injected workload and the guarantee of environmental stability confer on the benchmark reproducibility.

The main goal of autonomic computing is to improve the system QoS. Thus the evaluation of autonomics efficiency consists in measuring the impact on system QoS metrics when injecting a disturbance[1]. An autonomic computing benchmarking tool so consists of the system under test (SUT), a workload injector and a disturbance injector. It provides both business specific (QoS) metrics and autonomic computing measures.

A "classical" performance benchmark and an autonomic computing benchmark differ along three main axes according [13].

Environment stability that is questioned for autonomic computing by the injection of disturbances.

[1] A disturbance consists in:

- an event implying configuration change for self-configuration;
- a workload variation for self-optimizing;
- faults for self-healing;
- attack for self-protection.

Management interactions that must not occur in a classical performance benchmark, but that constitute the AB as well.

the antagonism between test realism (regarding the representativeness of the workload or the disturbances load the system under test will have to face) and the benchmark requirements, especially in terms of reproducibility, cost and legality.

Among the works dealing with the experimental evaluation of autonomics efficiency, [14] seems to be one of the most advanced. It describes one of the first autonomic computing benchmark dealing with self-healing evaluation. It details its experimental protocol for validating the tool. This consists in measuring the impact of thirty different classes of disturbances on two metrics. The disturbances are sequentially injected.

The three main specificities of an autonomic computing benchmark (highlighted by [13]) illustrate the addition of a second dimension into an autonomic computing benchmark compared to a classical QoS benchmark. Each benchmark can indeed be associated with a function that get injection profiles as inputs and that returns a vector of evaluated metrics. Thus a classical QoS benchmark is a function with a single input (i.e. the injected workload profile) whereas an autonomic computing benchmark consists in a function getting two inputs (i.e. the injected workload profile and the injected disturbance profile). Concerning this last type of benchmark, constraints relative to costs, reproducibility and legality, which are antagonistic to test realism, can be declined into constraints on the injection profiles (synchronization of workload and disturbances injections, stability of workload injection, etc.). In other words an autonomic computing benchmark consists of two coupled evaluation tools. The first one is dedicated to the QoS measurement: it is business specific and is essentially made of quantitative metrics. The second one focuses on the evaluation of self-management behaviors. It should include only generic (i.e. business independent) aspects. Concerning the experimental assessment works like [14], [64], [90], although they tend to demonstrate that autonomic features improve information systems efficiency, they do not achieve to define a scale offering a synthesized and absolute[2]view. Thus [14] and [17] define a value ranged between 0 (non autonomic) and 1 (fully autonomic) for assessing self-* features. However this indicator is obtained by restraining the number of different disturbances.

[2]By absolute we designate a scale that could classify the results from a system without any autonomic features to an idealistic system that could autonomously anticipate or deal with any disturbances (without any delay and any impact on the quality of service)

15.3.2 Approach specificities

Refer to section 14.3.2.

The scope of this benchmark approach can be divided as follow.

- In the ideal situation the single autonomic system under test (SUT) offers the capability:
  - to easily enable / disable its autonomic capabilities and then the bench will allow the comparison of the SUT with or without autonomic behaviors;
  - to configure the policy (rules) applied to an AB. Thus it will be possible to evaluate the efficiency of different policy of a given AB in the unique SUT.
- Otherwise, if autonomic features are built-in the system and not configurable thanks to policy rules:
  - the benchmark can provide the comparison of an autonomic SUT to another one that is non-autonomic but functionally equivalent;
  - it can also be interesting to compare autonomic systems as black boxes, i.e. to compare the similar or common autonomic capabilities of two autonomic SUT that are not functionally equivalent. However this last benchmark should not only compare autonomic capabilities but global features.

Concerning quantitative evaluation process, in order to face to the difficulties linked to autonomic computing benchmarking (see section 15.3.1), an evaluation process –similar to those used by [14] concerning self-healing– has been adopted. It consists in a three-step test process for evaluating ABs.

1. First a workload, that will be maintained constant during all the three stages, is injected in the system under test (SUT). This step lasts until the system reaches a stable state regarding its QoS metrics.
2. Then a single disturbance is injected in order to trigger an autonomic reaction of the SUT.
3. The last step consists in observing metrics (related to QoS and autonomic computing) until the SUT returns in a stable state (possibly different from the initial one) regarding the QoS metrics.

15.4 Assessment process

For assessing ABs quantitatively, we defined one evaluation grid. It consists in grouping together the quantitative metrics defined in the hybrid ISO/IEC 9126 model (see section 14.3.2). It allows to assess each AB independently.

15.4.1 Quantitative assessment of elementary AB

Table 15.1 allows the evaluation of each AB independently: one table / AB.

Table 15.1: Quantitative assessment grid for elementary AB

Autonomic Behaviour title (AB1 for example)			
Criterion / Metric	**Sub-division**	**Description**	**Example**
Evaluation process		Free text	Load injection on 3 systems under test (SUT): **SUT1:** PeerTV **SUT2:** Implementation of PeerTV with random overlay **SUT3:** Implementation of PeerTV with no overlay
Experimental settings		Free text	Load injection has been conducted N times...
Efficiency	Initial latency	QoS metrics	SUT2 is x% better than SUT1 and SUT1 x% better than SUT3 on this metrics
	Throughput		SUT1 is x% better on this metrics

Sensitivity	Monitor-ability	Mean time for monitoring (see figure 14.5)	
	Analyz-ability	Mean time to analyze (see figure 14.5)	
Reactivity	Planning capability	Wait for plan duration + Mean time to plan (see Figure 14.5)	
	Change-ability	Wait for execution duration + Mean time to execution (see figure 14.5)	
Anticipatory		Mean time between impacting disturbances	
Stability		Mean time to stabilisation	
Composition		Function on these metrics to get one aggregated decisional metric	SUT1 is x% better than SUT2. SUT 2 is x% better than SUT3

15.4.2 Quantitative assessment of global ABs

Evaluating the overall autonomic features of a complete application from a quantitative perspective does not make sense. However future work will consists in composing quantitative elementary evaluations for obtaining global evaluation a different abstraction levels, like composite ABs or else autonomic characteristics.

15.5 Experimental assessment of local ABs

This section contains the results obtained by applying the quantitative assessment grids for evaluating each local AB of each Selfman application.

15.5.1 PeerTV (Peerialism(P6))

PeerTV is an application developed by Peerialism(P6) Inc. It is a platform for Content Live Streaming based on Peer-To-Peer technologies. It utilizes proprietary overlay network algorithms and optimization techniques. In this context, we will examine the performance of the Self-configuration Autonomic Behaviour presented in 14.5.1, which it is entitled with the task of reorganizing the structure of the overlay network connecting all client hosts according to a number of criteria, such as bandwidth utilization.

The Optimization Process

In Section 14.5.1, we introduced the entities which constitute the PeerTV platform and their roles. As mentioned in that context, peers periodically report their status to the Tracker. This periodic update includes some general information such as the playback point and the number of packets in the playing buffer, plus some more detailed ones about each ongoing transfer with other peers, e.g. observed connectivity issues such as packet loss and excessive delay. The data received from the peers is processed and aggregated to form a matrix, called the "Happiness Matrix" $A(i, j)$, which represents all possible interconnections between peers in the system and whose inner values express the worthiness of such combinations. An "Happiness Value " in the matrix is a weighted sum of all characteristics observed and/or expected from a certain combination. The resulting value can be considered as a grade for a certain combination. If many combinations for a certain peer are available, the one with highest grade of all will be chosen. Which means that., for a certain peer A, a providing peer B will be selected which is expected to provide the best performance in the future transfer between A and B. Such mechanism of assigning peers to one another has been modeled as a Linear Programming Optimization problem of the type Linear Sum Assignment. The task of solving the optimization problem is carried out by the Optimization Engine. Once a result is produced, the Tracker notifies the peers so that the new transfers can be established.

There are two sensible steps in this process: the calculation of the "Happiness" values and the actual solving of the Optimization Problem. In the first case, it's trivial to understand that the choice of the happiness values

will directly impact the performance of the system. For instance, one of the parameters in the calculation of the Happiness values is the "Stickiness ", i,e. how importartant is the fact that there already exists an ongoing transfer between two peers.. If high Stickiness values are chosen, the system might result more stable but the overall bandwidth utilization might be affected, since the system will give higher priority to the preservation of successful connections than to load balancing.

The other sensible step in the Optimization process is the actual solving of the linear optimization problem. In fact, the computation associated with it might take long time to execute since the number of potential peer combinations is typically quite large. In presence of high churn, disruptions in the network which happen during the calculation might totally change the initial information which the ongoing computation is based upon. This will cause the results of the calculation to be totally erroneous. It's therefore vital for the calculation to happen as fast as possible to avoid such situation. For this purpose, PeerTV uses Auctioning algorithms to solve the Optimization problem, which are known to be the most performant for solving complex Linear Sum Optimization Problems. Bersekas et al.[11] propose a state-of-the-art parallel/distributed auctioning approach which provides optimal results and the best performance for Linear Sum Optimization Problems. However once implemented in our system, the algorithm turned out to perform poorly given the size of the problem to be solved. Consequently, a new heuristic for auctioning based on Bersekas' approach has been developed by Peerialism(P6) which guarantees 98% results optimality but it is up to four times faster than Bersekas'.

Optimization Levels

To test the performance of the aforementioned self-optimization behaviour we proceeded defining three test configurations which we will call "Levels".

- **Level 0**. In this configuration setting, provider and requester peers are associated to each others randomly given that providers have enough data to serve the requester peers. No method is used to improve connectivity among peers or to guarantee that the overall bandwidth utilization is balanced among the participants of the overlay network. However, the Optimizer does always make sure that all requester peers are associated to enough peers to receive the content.

- **Level 1**. This setting is very similar to Level 0, a part from the fact that the Stickiness optimization technique is used in this case. As

previously mention, the Stickiness makes sure that ongoing successfull transfers are kept through different optimization processes.

- **Level 2** All optimizer's features are enabled in this setting. The following characteristics are considered in the Optimization process.

  Static Characteristics, which are supposed to remain constant for a certain pair of peers:

  - Connectivity Issues (Routing, Nat Constraints)
  - Bandwidth Capacity
  - Average Delay
  - ISP Friendliness, i.e. the distance between two peers' ASs.

  Dynamic Characteristics, which vary in time for each pair of peers (i, j):

  - Experienced Connectivity Issues, experienced issues observed by i or j in a previous connection attempt.
  - Stickiness, i.e. how stable were the previous transfers between i and j.
  - Buffer State.

Results

In Figure 15.5.1, we can observe the result of the tests regarding saving percentage at the Source, i.e. how much bandwidth was spared at the Source with respect to the situation where all peers would receive the content directly from the Source and not from other peers. Results from 5 different simulation scenarios are shown in the same graph. The simulation scenario's difficulty is given by a number of parameters such as: number of peers in private networks, delay and bandwidth distributions, Internet Autonomous Region distances between peers, etc.. As we can observe, the performance across all scenarios of *Level 0* decreases substantially as the scenario difficulty increases. This is trivial to understand, since its' clear that the choice of random peers becomes less and less effective as peers present more peculiar features in the more complicate scenarios which might hinder their performance. The same kind of result is observed for *Level 1*, where Stickiness just partially compensates for the random assignment of peers. *Level 2* instead constantly shows better performance than the other settings. This difference is manifest in the worst case scenario where not only *Level 2* performs better

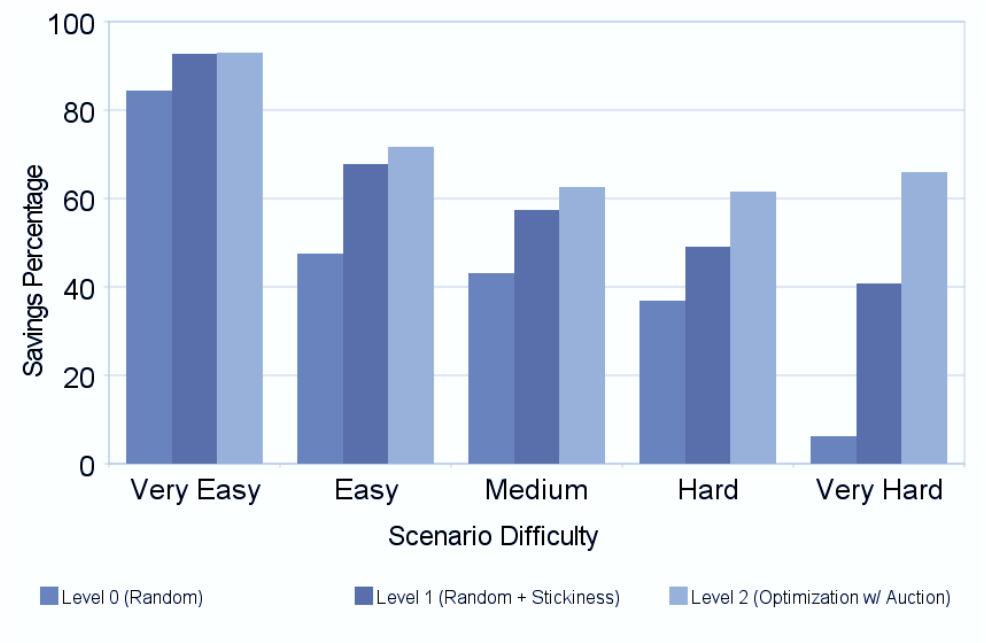


Figure 15.1: Effect of Self-Optimization Level on Saving Percentage

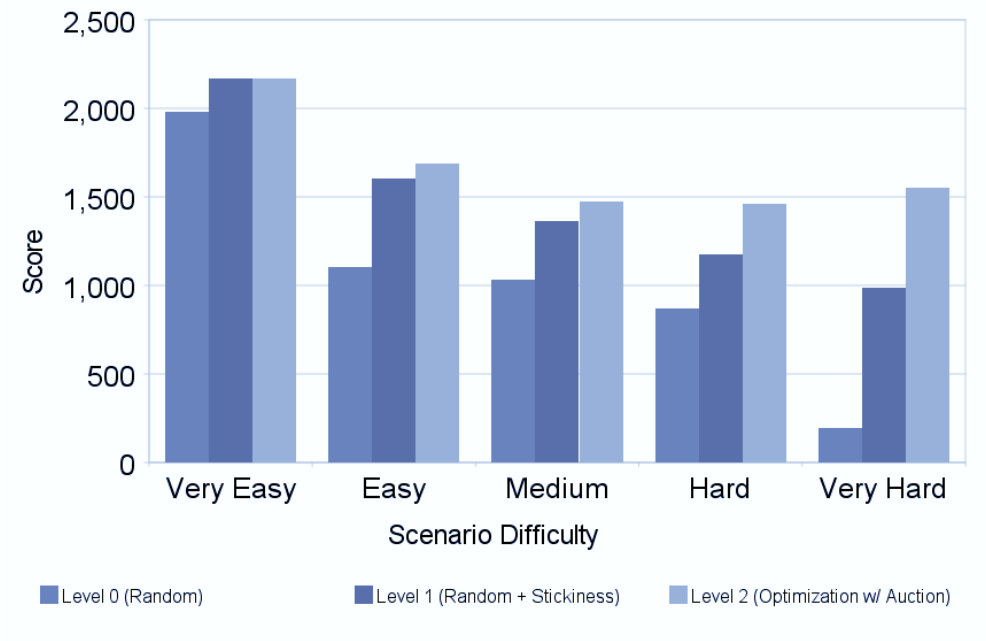


Figure 15.2: Effect of Self-Optimization Level on Total Performance Score

than the other levels but it also shows performance comparable to the easier simulation scenarios. In Figure 15.5.1 are shown the results of the same series of tests presented in 15.5.1, in this case however the evaluation metric for the system is the "Score", which is a weighted sum of the following measured metrics: Max/Average Initial buffering, Max/Average Playback buffering, Total Successfull streaming sessions and Total Failed streaming sessions. The "Score" is therefore a measure which tries to entail the user average experience of the system. As shown in 15.5.1, the results are very similar to the ones based on the savings metric, showing that *Level 2* always performs better than the other levels and that the performance of the system stabilizes as the difficulty of the scenario increases.

All tests have been obtained using our emulation environment *MyP2PWorld* previously published in Selfman SASOW 2008.

15.5.2 Scalaris (ZIB(P5))

Scalaris is the implementation of the replicated storage service presented in Deliverable 3. The quantitative evaluation of the autonomic behaviors in Scalaris include the performance of transaction and the impact of the finger table. Chord# is the name of the SON that Scalaris implements.

Note that part of these results also can be found in other Deliverable 2 and 9.

Impact of the Finger Table

Rowstron et al. [70] introduced in 2001 with Pastry a DHT which supports routing in $\log_k N$ hops where k is a system parameter. The lookup performance increase was bought by increasing the size of the routing table from $\log_2 N$ to $(k-1)\log_k N$. DKS [27] presented a similar idea as a generalization of Chord. In the following, we compare DKS, as a representatitve of Chord-style overlays, with Chord# for $k > 2$.

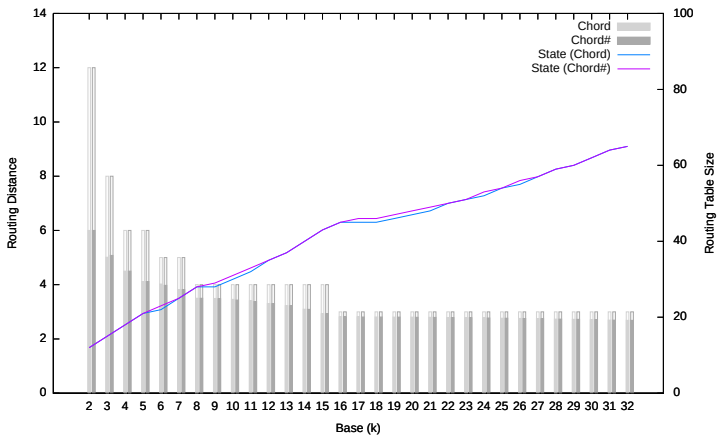


Figure 15.3: K-ary routing in Chord and Chord#

For Fig. 15.3, we simulated the respective overlays. It shows the differences between Chord and Chord# are negligible for larger k. But it also shows the decreasing returns for larger k. In many applications, the increased maintenance overhead for the larger routing tables will outweigh the improved routing performance. Proximity routing could be more beneficial.

The drops/steps in the maximum path length are near powers of two in Fig. 15.3 because the size of the system is a power of 2. In general, the maximum path length is $\lceil \log_k N \rceil$. The steps in the graph are the steps of this function.

Here are the steps for a chordal ring with 4096 nodes:

k	max hops
2	12
3	8
4	6
6	5
8	4
16	3
64	2

Transaction Performance

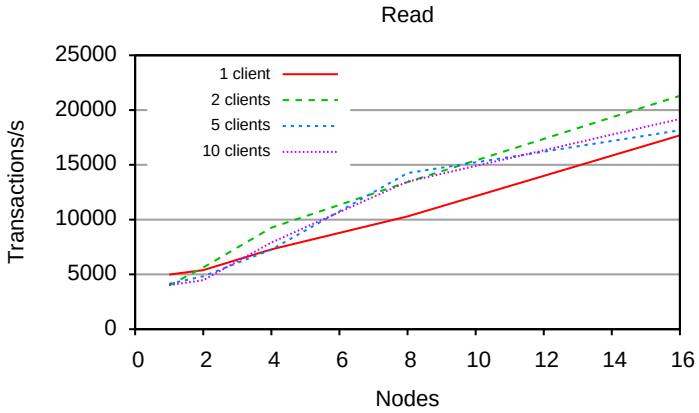
We tested the performance of Scalaris and the transaction algorithm implementation on an Intel cluster up to 16 nodes. Each node has two Quad-Core E5420s (8 cores in total) running at 2.5 GHz and 16 GB of main memory. The nodes are connected via GigE and Infiniband; we used the GigE network for our evaluation.

On each physical node we were running one multi-core Erlang virtual machine. Each virtual machine hosted 16 Scalaris nodes. We used a replication degree of four, that is, there exist four copies of each key-value pair.

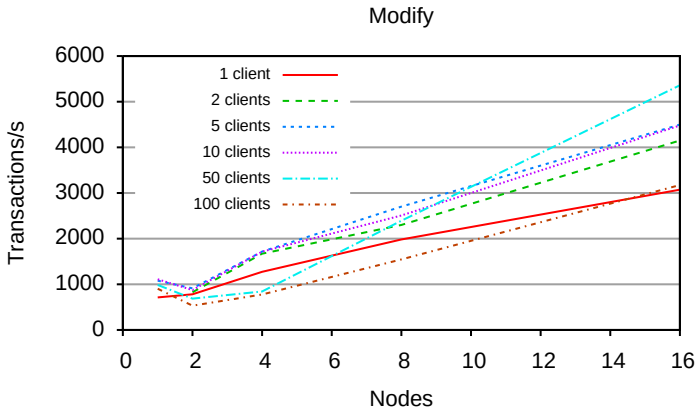
We tested two operations: a read and a modify operation. The read operation reads a key-value pair. The modify operation reads a key-value pair, increments the value and writes the result back to the distributed Scalaris store. To guarantee consistency, the read-increment-write is executed within a transaction. The read operation, in contrast, simply reads from a majority of the keys. The benchmarks involved the following steps:

- Start watch.
- Start n Erlang client processes in each VM.
- Execute the read or modify operation i times in each client.
- Wait for all clients to finish.
- Stop watch.

Figure 15.4 shows the results for various numbers of clients per VM (see the colored graphs). In the read benchmarks depicted in Fig. 15.4(a), each thread reads a key 2000 times while the modify benchmarks in Fig. 15.4(b) modify each key 100 time in each thread.



(a) Read



(b) Write

Figure 15.4: Performance of Scalaris: (a) Read operation, (b) Modify operation for different numbers of local threads and cluster sizes.

As can be seen, the system scales about linearly over a wide range of system sizes. In the read benchmarks (Fig. 15.4(a)), two clients per VM produce an optimal load for the system, resulting in more than 20,000 read operations per second on a 16 node (=128 core) cluster. Using only one client (red graph) does not produce enough operations to saturate the system, while five clients (blue graph) cause too much contention. Note that each read operation involves accessing a majority (3 out of 4) replicas.

The performance of the modify operation (Fig. 15.4(b)) is of course lower, but still scales nicely with increasing system sizes. Here, the best performance of 5,500 transactions per second is reached with fifty load generators per VM, each of them generating approximately seven transactions per second. This

results in 344 transactions per second on each server.

Note that each modify transaction requires Scalaris to execute the adapted Paxos algorithm, which involves finding a majority (i.e. 3 out of 4) of transaction participants and transaction managers, plus the communication between them. The performance graphs illustrate that a single client per VM does not produce enough transaction load, while fifty clients are optimal to hide the communica- tion latency between the transaction rounds. Increasing the concurrency further to 100 clients does not improve the performance, because this causes too much contention. Note that for the 100-clients-case, there are actually 16*100 clients issuing increment transactions. Overall, both graphs illustrate the linear scalability of Scalaris.

15.5.3 The gPhone application (UCL(P1))

The gPhone application developed by UCL(P1), see Chapter 18, is built on top of the relaxed-ring, which results where presented in the first two years of the project, and it uses the transactional support for DHTs presented in Chapter 2.

Apart from the analysis done during the design and implementation of the relaxed-ring, we also made qualitative and quantitative evaluation of it. This report is about the quantitative evaluation. Qualitative evaluation is presented in chapter 14.

The main focus of the quantitative evaluation we performed was on the branches that were created by our ring maintenance algorithm. We also evaluated cost-efficiency by measuring the amount of messages generated by the relaxed-ring during the ring maintenance, and we compare it with the periodic stabilization technique of the Chord [81] ring.

Branches on the relaxed-ring

The relaxed-ring topology does not require a perfect successor-predecessor chain along the ring. That is why it is called relaxed. This relaxation improves lookup consistency with respect to other ring-based networks such as Chord [81]. However, it implies a small degradation on the lookup complexity that goes from $log(N)$ to $log(N) + b$, where b is the size of the branch where the responsible node is to be found. If there is no branches, the routing algorithm is exactly the same.

Branches appear on the relaxed-ring only if there are problems on the quality of the connectivity between peers. Therefore, we have perform a quantitative evaluation of the size of the branches for many different network sizes with different quality of connectivity. Figure 15.5 presents the results

obtained for networks going from 1000 to 10000 nodes. Each size was tested with a connectivity factor c, representing the probability of establishing a connection between any pair of peers, where $c \in \{0.9, 0.95, 1\}$. We can observe that the average amount of branches follows linearly the quality of the connectivity of the simulated network.

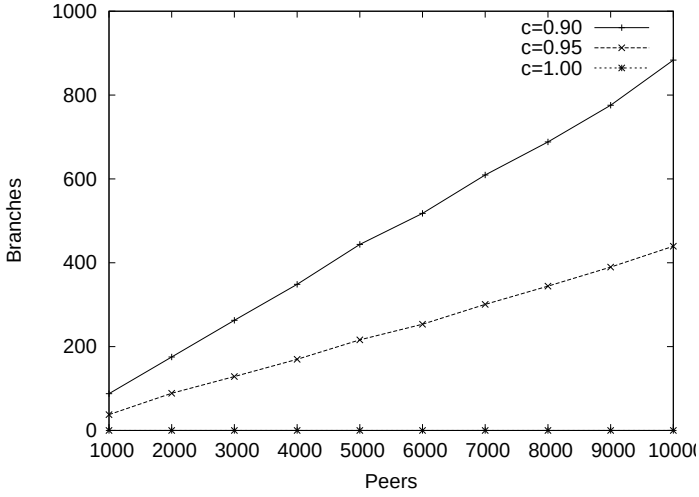


Figure 15.5: Average amount of branches depending on the size of the network and the quality of the connectivity.

Figure 15.6 presents the average size of branches in different networks following the same size and connectivity factors as in Figure 15.5. The data of the networks with perfect connectivity has been removed, because there are no branches in such scenario. We can observe on the two above lines that average size of branches is constantly bellow 2, independent of the size of the network so the topology scale very well. To conclude something about the complexity of the routing algorithm, we have to observe the amortized average size of branches, which is represented by the two lower lines. We can observe that they are constantly bellow 0.5 independent of the size of the network, meaning that empirically we could eliminate b from the complexity of the routing algorithm, keeping it to $log(N)$.

Bandwidth consumption

It is known by the peer-to-peer research community that periodic stabilization is a reasonable solution to constantly fix the ring, but it is very expensive in terms of bandwidth usage. It basically generates too many messages constantly. Therefore, one of the effort of the relaxed-ring was to reduce this

problem by fixing the ring with correction-on-change instead of by a periodical check. Figure 15.7 shows the load of messages in Chord due to periodic stabilization, compared to the load of the Relaxed-Ring maintenance with bad connectivity. Y-axis is presented in logarithmic scale. We have chosen only the worse case we studied for the relaxed-ring which still performs much better than Chord. The different values for Chord are obtained by tuning the frequency of the periodic stabilization. This frequency is expressed as a relative value with respect to the modifications on the network. For instance, a stabilization rate of 4 means that every node performs a periodic stabilization round after a total amount of 4 joins/leaves have occurred in the whole ring.

One of the results presented in year two of the project was a self-adaptable topology called PALTA, which stands for Peer-to-peer AdaptabLe Topology for Ambient intelligence. It is an adaptation of the relaxed-ring in order to provide a better routing algorithm for small networks. In small networks, the ring presents a fully connected network, which is much faster to perform lookups, but it is more expensive to bootstrap and to maintain every time a new node join. As soon as the network reaches a size of a pre-defined value ω, the routing table adapts itself to behave as a regular peer-to-peer ring. Because of the high bandwidth consumption of fully connected network, it was necessary to evaluate the amount of messages exchanged within the network.

Figure 15.8 does not show the marginal cost of joining a network, but the total amount of messages generated to construct a fully connected network, a relaxed-ring, and two PALTA networks with ω values equal to 100 and 200. We can see that with less active connections, as in PALTA or the relaxed-ring, the number of messages remains small, generating less network traffic. The curve of the fully connected network increases quadratically, generating $n * (n - 1)$ messages, with n being the size of the network, we can conclude that this network cannot scale.

The curve of the relaxed-ring shows a constant and controlled increment in the amount of messages, keeping them at a very low rate, showing that it scales very well. Now, the results obtained from experiments with PALTA are very interesting because both perform better than the ring for larger networks. One can observe that PALTA with $\omega = 100$ and $\omega = 200$ increases quadratically the amount of messages, as in a fully connected network. This happens only until the network reaches a size of ω peers. Then, the amount of messages increases slower that in a ring, and furthermore, after a certain size of the network, both PALTA networks remain at better values that the relaxed-ring. The explanation for this is that when a new peer join in the network, it needs less messages to find the k fingers. This is because PALTA

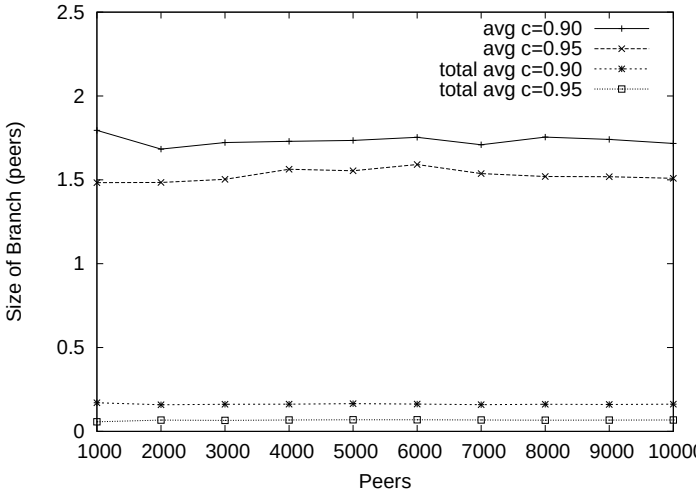


Figure 15.6: Average size of branches depending on the quality of connections.

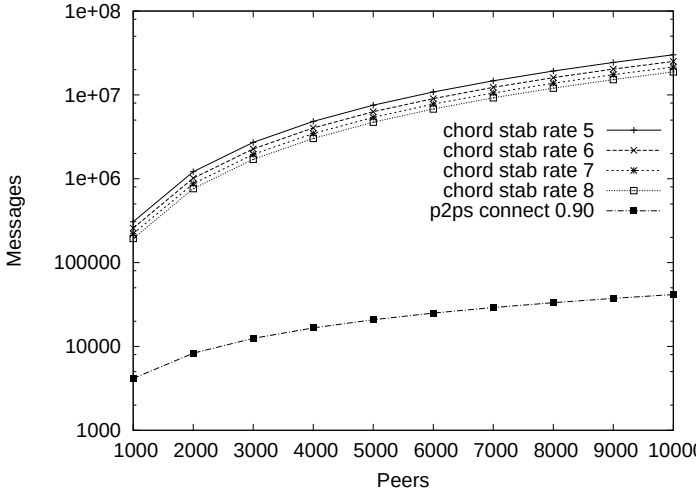


Figure 15.7: Bandwidth consumption of ring maintenance in Chord and the Relaxed-Ring.

has ω peers with a larger routing table ($\omega > k$), making a more efficient jump during the routing process. We study this further in the following figure.

This means that the cost of maintaining a small fully connected network can help a larger network to be more efficient for routing, generating less network traffic.

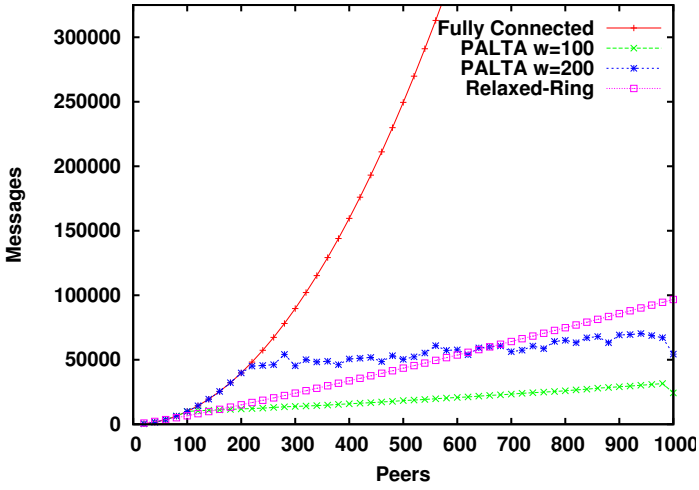


Figure 15.8: Bootstrapping bandwidth usage on different networks.

In order to confirm our conclusions from the previous experiment, we decided to measure the average amount of hops needed for a message to reach its destination. This is known as a *lookup operation* in a ring. This experiment does not consider fully connected networks, because there is no concept of responsibility is such systems. In addition, because of its characteristics, peers in a fully connected network reach any other peer in the network in 1 hop.

In Figure 15.9 we can observe the results obtained. The relaxed-ring shows that the number of needed hops increase logarithmically when the network size increases. PALTA performs better than the relaxed-ring due to fact that some peers have a larger routing table, confirming the results from the previous experiment. In both cases, PALTA presents an average number of hops slightly smaller than 2 if the network consist of less than ω peers. This is because the network is fully connected, and therefore, in can reach the predecessor of the responsible of the looked up key in only one hop. The second hop is needed to reach the responsible. The average is smaller than 2 because the randomized experiments sometimes generates lookups where the responsible is the peer triggering the lookup.

After the value of ω is reached, the average increases faster in PALTA

with $\omega = 100$ that with $\omega = 200$. This is clearly due to the amount of peers having a larger routing table. We observe that in both cases the system behaves much better than the ring. We expect that for larger networks the value would converge to the curve of the ring, but still performing better. What we cannot currently explain is the behavior of PALTA with $\omega = 100$ when the network is in between 100 and 200 nodes. It seems to perform even better than a $\omega = 200$.

15.6 Discussion

15.6.1 Experimental results

A major pitfall we had to face for carrying out the quantitative evaluation of autonomics in Selfman applications was to measure the efficiency of ABs themselves and not only the QoS of the different systems.

Thus the results reported in section 15.5.2 are highly interesting but focus especially on the global system efficiency –in terms of QoS– of Scalaris and not on the ABs embedded in the system. The numbers of handled read and/or write requests (QoS indicators) are evaluated according to different configurations (i.e. node numbers) instead to be measured either by (refer to section 15.3.2):

1. enabling / disabling the AB under test;
2. modifying the configurable settings of the AB under test;
3. comparing the AB under test with a similar one –or even none– coming from a business equivalent SUT.

Concerning the experimental results obtained in section 15.5.3, they have been reported by providing some synthetic figures. This allows to avoid from a huge number of quantitative evaluation grids. These results try to compare similar ABs (i.e. the behaviors relative to the ring maintenance) of two different structured overlay networks (SONs), Chord and the relaxed ring. The main consequence of such a comparative approach is to obtain results that put together autonomics and QoS measurements, i.e. that should be hardly extended to other business domain. Thus the major limitations of this assessment work lies in:

- the "aggregated" evaluation of every ABs in charge of ring-maintenance, which tend to limit fine-grained comparison;
- the lack of generic metrics for evaluating ABs quantitatively like they have been defined in the assessment methodology. However, this second restriction is essentially due as well to the inadequacy of the quantitative evaluation methodology to the peer-to-peer domain (see section 15.6.2).

Although these two difficulties tend to constraint the quantitative benchmark in terms of business independence and re-usability, they would not prevent from a future economical evaluation by binding a cost model to the technical benchmark.

15.6.2 Evaluation methodology

As mentioned in section 15.6.1, the quantitative evaluation methodology, which has been proposed, is still highly business domain and architecture specific. It questions in particular the possibility to apply the test process (see section 15.3.2) to any system under test.

1. This evaluation methodology basically consists in measuring the impact of the execution of an AB on the system QoS. So it implies that the QoS metrics and the autonomic indicators are evaluated at the same abstraction level. This is the main reason why, it was impossible to apply it to the Selfman applications (as shown in section 15.5): the QoS metrics were global (i.e. macroscopic level) whereas the autonomic indicators were local to the AB (i.e. microscopic level). In such a situation, an injected disturbance triggers an autonomic adaptation that has no measurable effect on the QoS (due to the important number of agents/peers contributing to the QoS). [3]

2. The definition of anticipation and stabilization durations independently of the business domain and architecture is also a challenge for a quantitative autonomics benchmark. We have experimented indeed that for now, in some cases, the stabilization duration can be difficult or even impossible to measure (for example in an AB running periodically without any "real" triggering event and for which this period duration cannot be configured).

Concerning the first limitation, a solution could consist in defining a methodology for composing the results obtained at the agent/peer (local) level in order to extrapolate the impact of ABs at a global level. Another approach consists in decomposing global QoS into many local values. However this last solution is business domain specific.

[3]However this methodology is adapted for evaluating systems for which QoS and autonomic metrics are evaluated at the same abstraction level: it has been successfully applied on an application dealing with workload management in an environment composed of J2EE application servers.

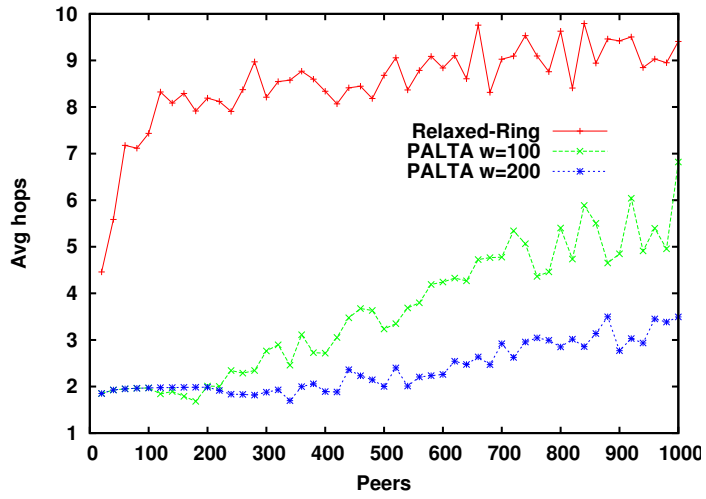


Figure 15.9: Average number of hops vs number of peers.

Chapter 16

D5.6: Evaluation of security mechanisms

16.1 Executive summary

This deliverable reports on the work in D4.4b which is on self-protection mechanisms and D5.6 which is on application-level security mechanisms. Self-protection mechanisms center on the use of Small World Networks (SWN) and social networks as an alternative to structured overlay networks. We develop a SWN simulator to show that SWNs can be effective. SWNs can approach the efficiency of structured overlay networks while being robust to network failure/churn. In particular, SWNs are more resilient to targeted attacks. Self-protection mechanisms are also needed for special attacks such as self-tuning attacks on the SWN. We also collect real world social networks and show that the SWN techniques can work with real social networks which approximate the structure of SWNs.

Application-level security focuses on a security infrastructure for Wikipedia. In Wikipedia, the attackers are the users rather than the server infrastructure and the attack is on the Wikipedia content. One of the main problems with Wikipedia is to ensure that open and anonymously submitted content is credible. Our infrastructure is usable with Wikipedia and Wikis using WikiMedia and serves to inject credibility information into a Wiki.

16.2 Contractors contributing to the deliverable

Peerialism(P6), NUS(P7) and ZIB contributed to this deliverable.

Peerialism(P6) contributed to refining the application self-protection issues pertinent to SELFMAN.

NUS(P7) performed the work on self protection in Deliverable D4.4b which is also reported here. The main contribution is the use of small world and social networks as a form of overlay network which is more robust to attacks. Self protection issues at the application level were analyzed in this deliverable. An infrastructure which leverages on social networks and third parties to add trust was developed as a credibility mechanism for Wikipedia.

ZIB(P5) contributed to the refining the application self-protection issues pertinent to SELFMAN and Wikipedia.

16.3 Self-protection support & mechanisms

This deliverable also reports on the work in WP4, Deliverable 4.4b. Deliverable 4.4b focuses on the use of small world and social networks as a self-protection mechanism to mitigate identity and routing attacks in structured overlay networks. It also looks at some new security issues which are raised by small world networks when self-tuning is employed to optimize routing. The software developed is described in Deliverable 4.4b (Section 11).

A small extension is also reported on the self-protection mechanism in Deliverable D1.3b (from Year 2). As much of the SELFMAN self-configuration work is in languages such as Java, the software component authentication mechanism is extended to deal with non-binary files such as Java class files.

16.3.1 Small world networks (SWN) as a kind of SON

Introduction and motivation

Small World Networks (SWN) have properties which help to counter some of the drawbacks of Structured Overlay Networks (SON). SONs rely on the assumption that most of the nodes are non-malicious. A Sybil-type attack gives the attacker the opportunity to control many nodes which breaks the assumption. Furthermore as the SON is self-organizing but it requires maintenance to preserve a particular network topology. This maintenance is vulnerable various attacks, e.g. the eclipse attack [80] and attacks which exploit churn and network failures.

SWNs are a model for social networks which have trust and identity relationships which mitigate the serious problems caused by Sybil-type attacks in SONs. Sybilguard [96] provides a method to limit the number of Sybil nodes a malicious user can create based on the number of his friends. Link maintenance is also cheaper than a SON because the links are much more static in a SWN. They are also more robust because they have a stochastic element which makes the network somewhere in between a structured network and a random one. The random properties make it harder to disconnect a SWN during churn. The network partitioning and merging described in Section 7.3.1 can utilize SWN to make it more robust under churn.

Figure 16.1 shows the topology of three graphs: a SON (a Chord network), a SWN, and a Random Network. All the graphs have the same number of nodes and links. Throughout our discussion, we will use n to refer to the number of nodes/peers in the graph/network. The leftmost graph is a SON (in this case a Chord ring [81])[1] where the links are very structured with inter-

[1]Since n is not a power of two, the longest edges do not cross the center.

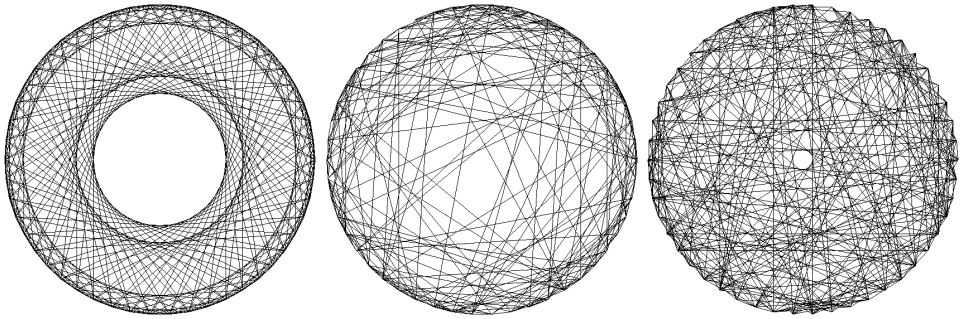


Figure 16.1: Network topology of Chord, SWN, and Random Network

node distances given in powers of two. The middle graph is a SWN where the links are created according to power-law distributions. The rightmost graph is a Random Network where the links are randomly connected between any two nodes. Although each of the three graphs has the same number of nodes and edges, they look rather different because of how they are structured.

The Sindaca recommendation system described in Deliverable D5.3 (Section 13.3) can suffer from Sybil attacks. In such an attack, a malicious user tries to manipulate the votes by creating a large number of user accounts in the system. SWNs based on social networks can be used to prevent such Sybil attacks. An additional benefit of using social networks in such recommendation systems is that social information may give more accurate recommendation. This is because people in a social group tend to have similar interests.

Routing in SWN

One of the first SWN models was introduced by Watts and Strogatz as a network that have a high clustering coefficient and low diameter [87]. The clustering coefficient C_i of node i is the fraction of edges between the neighbors of node i divided by the number of possible edges that could possibly exists between them. The clustering coefficient C of the network is the average of the clustering coefficient for each node C_i. Kleinberg showed that a drawback of SWNs based on the Watts and Strogatz model is that decentralized algorithm for routing will not have small expected routing length [45]. The insight is that the missing geometric information can be obtained by using a power law edge distribution on the adjacent nodes. A power law edge distribution means that there are many links to nearby nodes and few links to far away nodes according to some functions. Assuming a power law edge distribution, decentralized routing can achieve a small expected routing

length of $O(log^2(n))$.

We are interested in purely decentralized algorithms for routing which are attractive as they are based on local knowledge rather than relying on global knowledge. Usually this is done with greedy-based routing — node s routes the message to node t by forwarding it to its neighbor whose node position is the closest to the target node t. A cycle avoidance measure is for the node to remember the message and forward it to a not yet forwarded next closest neighbor. In what follows, we will assume that routing will be considered to be failed when it reaches more than $O(log^2(n))$ steps. (The experiments, actually use a bound of $log^2(n)$).

We propose the use of SWN as a kind of semi-structured SON where the edge connections are more relaxed. Preliminary results show that SWN can achieve good routing given more edges. Adding more edges in SWN doesn't directly contribute to larger overhead.[2] Experiments in [35] show that having $(O(log(n)))$ connections/friends in the SWN leads to comparable performance with DHTs and routing lengths of $(O(log(n)))$ is observed.

In order for a SWN to be usable as a semi-structured SON, it should also be robust in the face of node failures. Figure 16.2 shows experiments which investigate the robustness of the SWN under node failures for $n = 10^5$ nodes. A route is considered successful if the routing length falls below $log^2(n)$. We see that as long as the SWN has a reasonable number of edges, the probability of successful routing is high. When there is node failure, the probability of routing success is still very high even with 60% of the nodes failing.

Both SWN and Chord use a form of greedy routing, i.e. the message is forwarded to the neighbor which has ID that is the closest (has the most similar) to the target node ID. Routing is more robust when there are more alternative paths. One way of getting more alternative paths is to vary the greediness of the routing. Note that this requires some minor changes to the Chord lookup algorithm.

We consider the greediness percentage G as a parameter is used during routing. A message will be forwarded with probability G to the neighbor that has the closest node ID to the destination otherwise, the next closest node will be picked with probability G, and so on. The lower the G, the larger the effect of randomness in selecting a neighbour.

Figure 16.3 shows the tradeoff between routing performance and robustness for a network based on a SWN versus a Chord SON where both networks have $n = 10^5$ nodes. As the greediness approaches 100%, we get the expected result that the Chord SON gives shorter routing lengths com-

[2]Note that a SWN would typically have more edges than a SON, however, maintenance for a SWN is simple under churn.

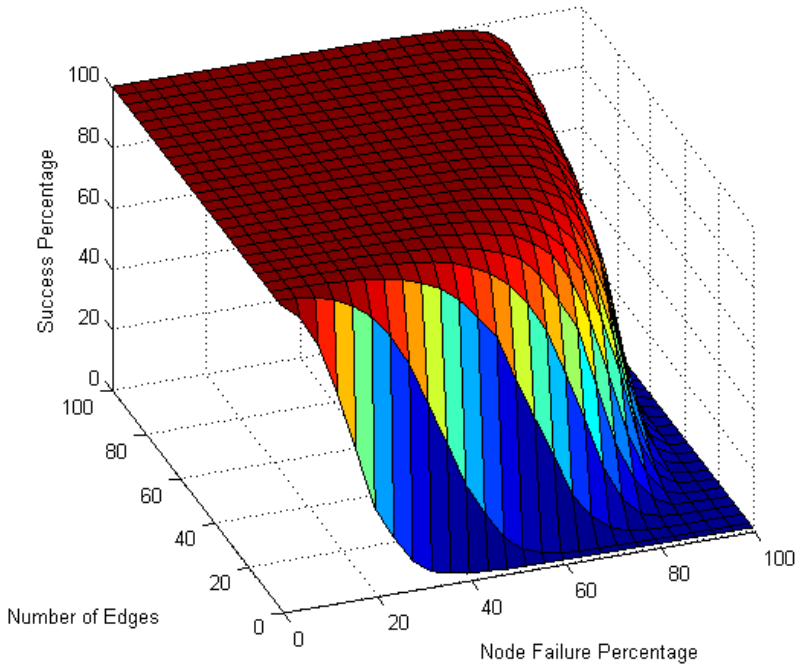


Figure 16.2: Routing Success and Node Failure

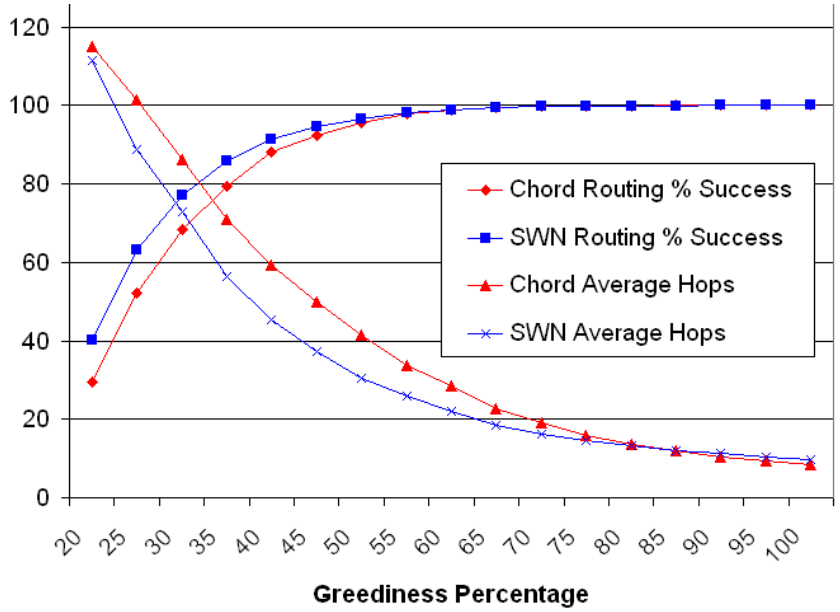


Figure 16.3: SWN vs. Chord on Greediness

pared to the SWN. However with less greediness (less than 80% greediness), SWN still maintain good routing performance while Chord degrades quicker. This shows that SWN has more routing alternatives than Chord. As for the percentage of successful routing, SWN has more successful routing over the range of greediness values.

Another way to look at the robustness is to examine the distribution of the routing lengths as G varies. Figure 16.4 shows the standard deviation and mean of the routing with varying greediness on the same network as Figure 16.3. The error bars show one standard deviation for routing length across 1000 routing tests while the line gives the average. We can see that the SWN has shorter routing over Chord (the lower line in the graph) and the standard deviation is also less (the shorter error bars in the graph). This shows that SWN routing is more robust than Chord if less greedy routing is used.

As SWNs have much less structure than SONs like Chord, they can be more robust to an adversarial churn model or a selective attack. In a highly structured network, it is easy to know the location of the nodes in the network. This means that a selected node in the network can be the subject of a denial of service if some other selected nodes also fail.

Assume the following attack model, the adversary is able to bring down a consecutive region of nodes. That is, the attacker can cause nodes with

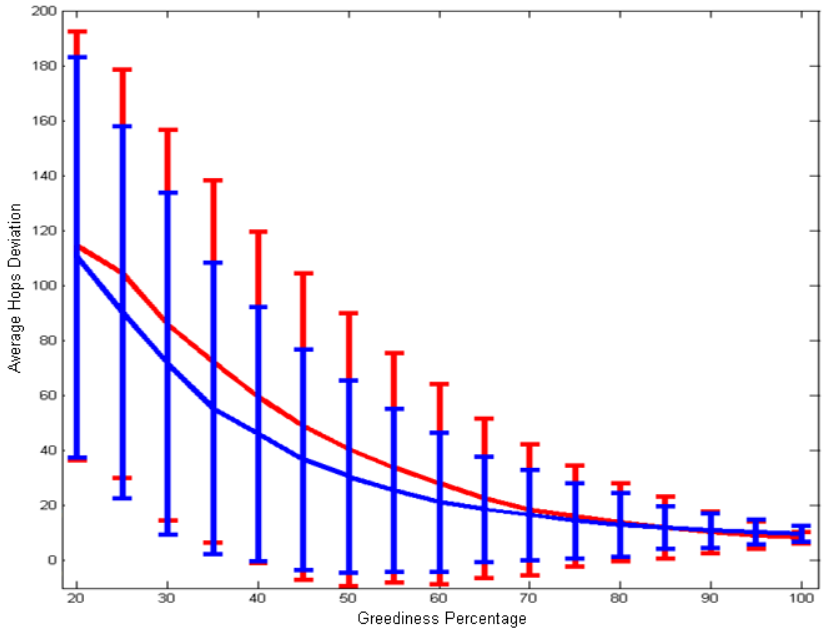


Figure 16.4: Average and Deviation in Routing Length vs. Greediness

consecutive IDs in the network to fail. In such a scenario, for Chord, the next node after the largest consecutive ID that fail will suffer the most in terms of successful routing to that node, followed by the next node with less severe damage and so on. This attack does not have much effect on an SWN because of the more flexible network and routing.

Figure 16.5 shows that SWN can withstand a consecutive range ID attack. The figure shows the average routing length for 1000 routing tests on a Chord and SWN network with $n = 10^5$ nodes. We marked a range of 50 consecutive node identifiers in both networks as "failed nodes" (i.e. nodes with identifiers $i-1, i-2, \ldots, i-50$ are marked as failed). The experiment then measures routing to nodes with identifiers $i, i+1, \ldots, i+49$ to measure the impact of the selective failure on the routing to the most affected nodes. The figure shows the condition before and during the attack for $i = 0$. For Chord, the impact is severe for the immediate nodes after the failed nodes (in this case, it is node i and the following nodes) and the impact become smaller further out. We see that while routing on the SWN is longer before the attack, the impact is considerably smaller and for the immediate nodes around i, the effect is much less than with Chord.

To more clearly see the impact of the attack, Figure 16.6 shows the difference in the average hops from a normal non-attack scenario to what

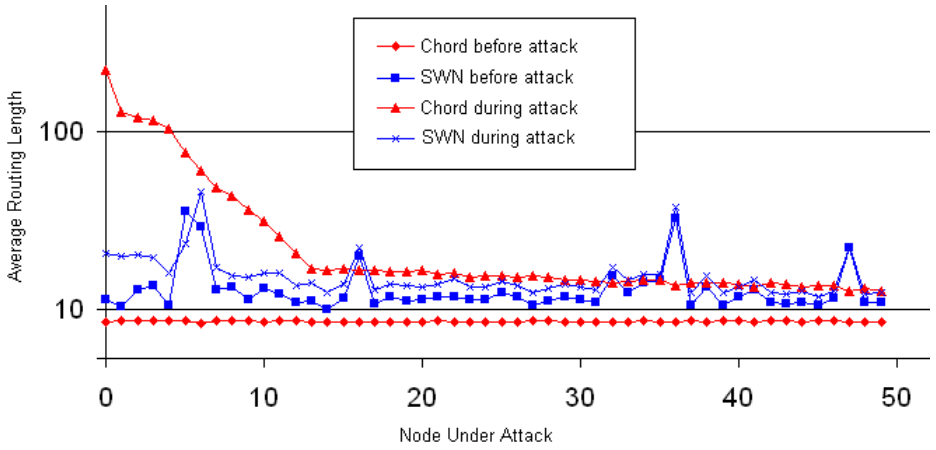


Figure 16.5: Consecutive range ID attack

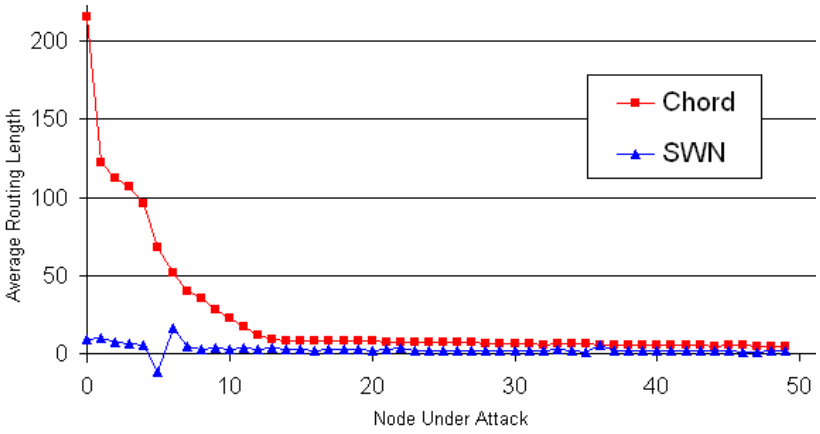


Figure 16.6: Increase in the average number of hops due to the range attack

happens during the attack. The increase in the average hops is the average routing length during the attack subtracted from the average routing length before the attack. For Chord, the node that has the most increase in average routing length during the attack is node i since most of the incoming nodes are down (i.e. node $i-1, i-2, i-4, i-8, i-16, i-32$) while the SWN has no node that has significant increase in routing hops during the attack. This showed that a SWN can be more robust than Chord under this attack model.

We remark that SONs need more complex self-healing to deal with network partitioning because of the interaction with self-stabilizing for ring maintenance (see Deliverable D4.2b, Section 7.3.1). It might be the case that the robust nature of a SWN could be useful in a hybrid network to deal with network merging after a network partition occurs.

SWN security issues

In order to be able to use greedy routing which only has local knowledge, the node identifiers should reflect the geometric information in the power law edge distribution. If the node identifiers are instead randomly assigned, greedy routing will not be successful. A feasible self-tuning approach is to reorganize the node identifiers in the network to form a power law distribution [72]. We show that this self-tuning approach fails when there are malicious nodes which do not obey the self-tuning protocol [34]. The problem is that the self-tuning requires all nodes to be honest about their node identifier. Malicious nodes can attack this assumption by lying about its node position. This effect propagates throughout the network because of the self-tuning protocol and after some number of self-tuning rounds, the entire network can be poisoned.

We devised two self-protection mechanisms. The first is a simple distributed self-tuning based protection against the poisoning attack. The node positions are reset periodically such that the routing performance can still be good while reducing the effect of the poisoning. Figure 16.7 shows the effect of this mechanism on the poisoning attack with different numbers of malicious nodes and the restart probability. The network used has $n = 10^5$ nodes. We see that poisoning easily spreads to the whole network and poisons all nodes when there is no self-protection (restart probability = 0). When the number of malicious nodes is small, e.g. 0.1% (100 nodes), poisoning occurs but the number of poisoned nodes is controlled. Figure 16.8 shows that the successful routing percentage is still reasonable for 0.1% malicious nodes.

In summary, the distributed protection mechanism works for small number of malicious nodes (e.g. 0.1% * n). We observe that with 0.008 restart

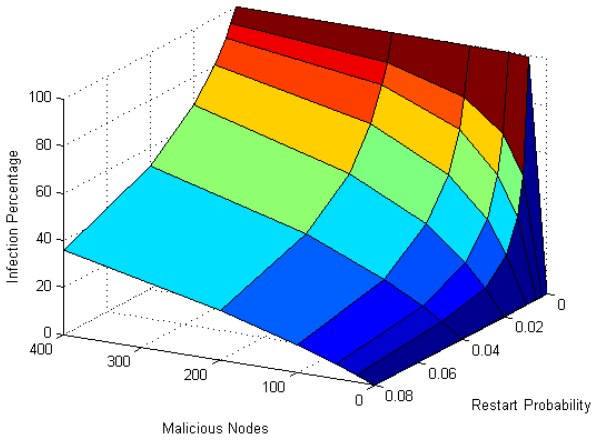


Figure 16.7: Infection Percentage vs. Malicious Nodes and Restart Probability

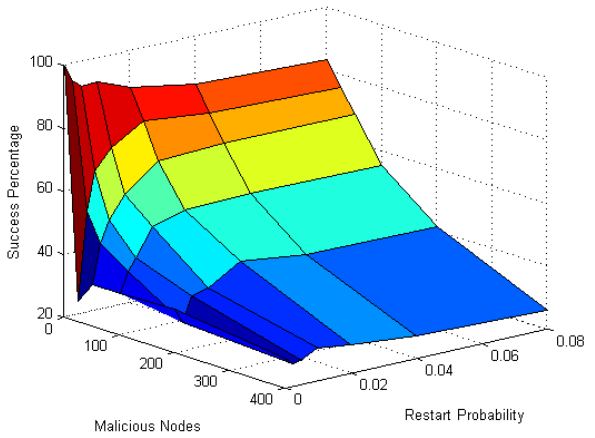


Figure 16.8: Successful Routing vs. Malicious Nodes and Restart Probability

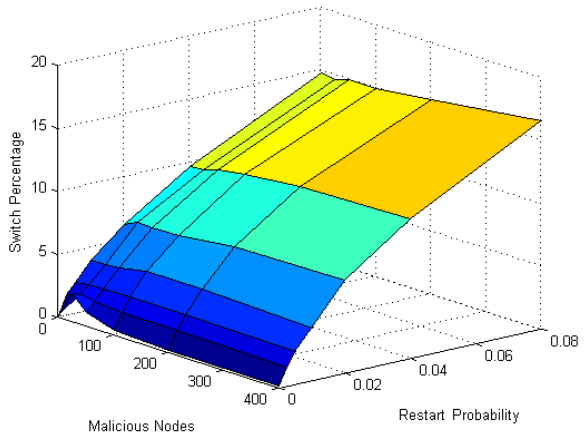


Figure 16.9: Switching Percentage vs. Malicious Nodes and Restart Probability

probability of node restarting the self-tuning each round, a small infection percentage and switch percentage (as shown in Figure 16.9) while maintaining a high successful routing percentage.

The second mechanism is to directly deal with how the poisoning attack works when there are malicious nodes. Nodes can monitor whenever its own token (which is a derived by a function on the position) is being swapped. The requirement is that the owner of the token has to be available (node is alive) during switching so that the owner can verify and certify the switching as a transaction. This incurs some overhead for switching. By keeping track of the token, a malicious node will not be able to cause a switch with the incorrect node identifier. This protection mechanism prevents the malicious nodes from lying but incurs more overhead since every node switch requires three nodes rather than two nodes which means more communication costs. Unlike the first mechanism which can work in a darknet where the overlay network is also the underlay network, this needs the SWN to be an overlay network on top of some other network which is able to route between all nodes.

Real social networks

We have studied SWNs as a way of understanding real world social networks. We employ the fact that a social network allows us to have additional trust mechanisms which are not present in an arbitrary network like a SON. We investigated how the SWN assumptions and routing behave on a real social

network.

We choose to use a network extracted from Facebook's social network as our real social network. In Facebook, each user is identified by a integer. Each user has a list of friends which are users in Facebook as well. We downloaded about 6 million users' friend list. The whole list contains about 114 million different users. The average number of friends per user (i.e. average node degree) is 135.89. This is not the entire Facebook network for two reasons. Firstly, some profiles are closed, so they cannot be collected. Secondly, it takes too long to query and crawl through the social network because of network restrictions and we do not want to be appear to be attacking Facebook.

We started by downloading 100 users, some are randomly selected, others are the users we know. Then we did breadth-first search by downloading the friends of the 100 users, then friends' friends and so on.

Although the graph we obtained is a portion of the entire social network, there are some computational difficulties in using the data because it is too large to be computationally practical to run experiments both in time and space. Thus, we selected some subgraphs. The selected subgraph has to be well connected and satisfy small world network properties, thus random selecting some nodes will not work. The algorithm to select a subgraph of n nodes from a larger graph G is based on the one in [72]:

1. Initially the node set S is empty.
2. Select a random node in G and add to S.
3. For the all nodes which are directly connected to nodes in S, select the node with the most number of edges to nodes in S and add it to S. If there are no such nodes, the algorithm terminates with fewer than n nodes. If there is more than one node having most number of links, randomly select one of them.
4. Terminate if there are n nodes in S.
5. Go to Step 3.

The above method selects a group of nodes which are well connected. Figure 16.10 shows a subgraph with 1000 nodes selected using this algorithm where the dots are the users and the edges denote the friend relationship. Note that in Facebook, the friend relationship is not symmetric. That means "A is B's friend" does not necessarily mean "B is A's friend". However, our SWN model assumes an undirected graph. Thus we have converted the Facebook

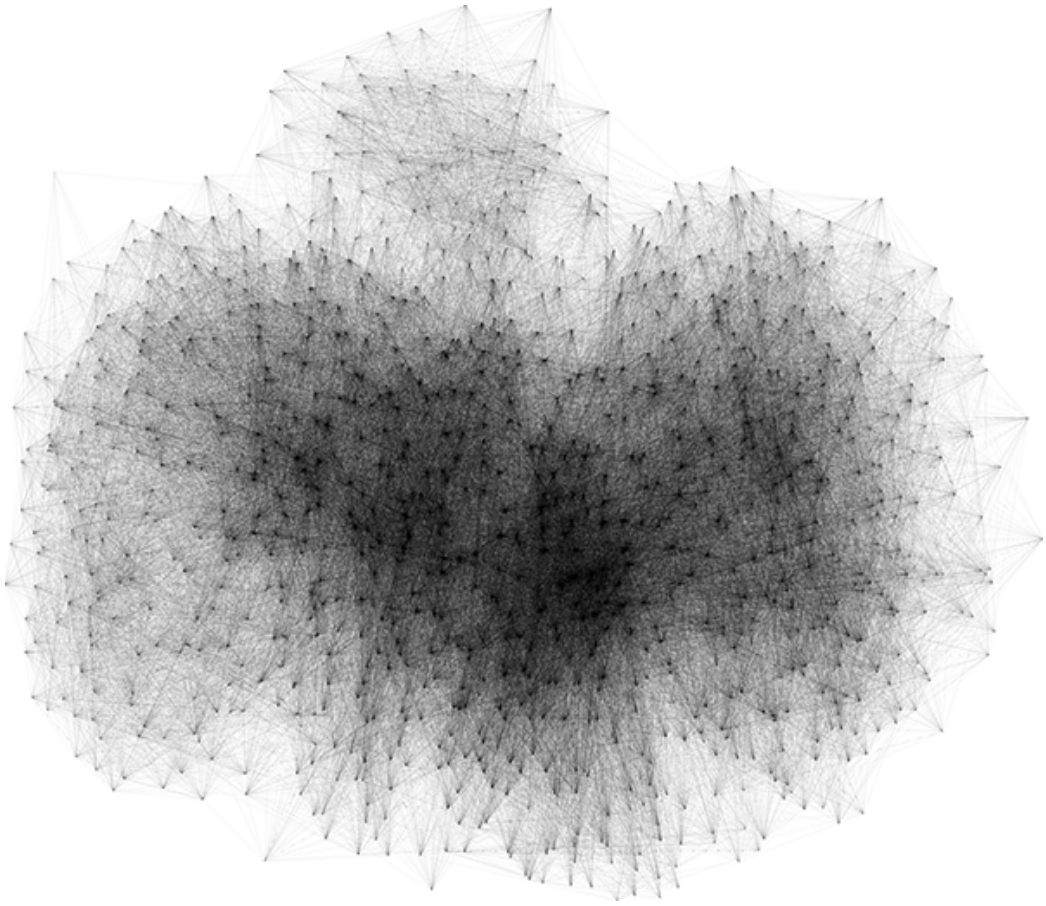


Figure 16.10: Facebook social network graph of 1000 nodes.

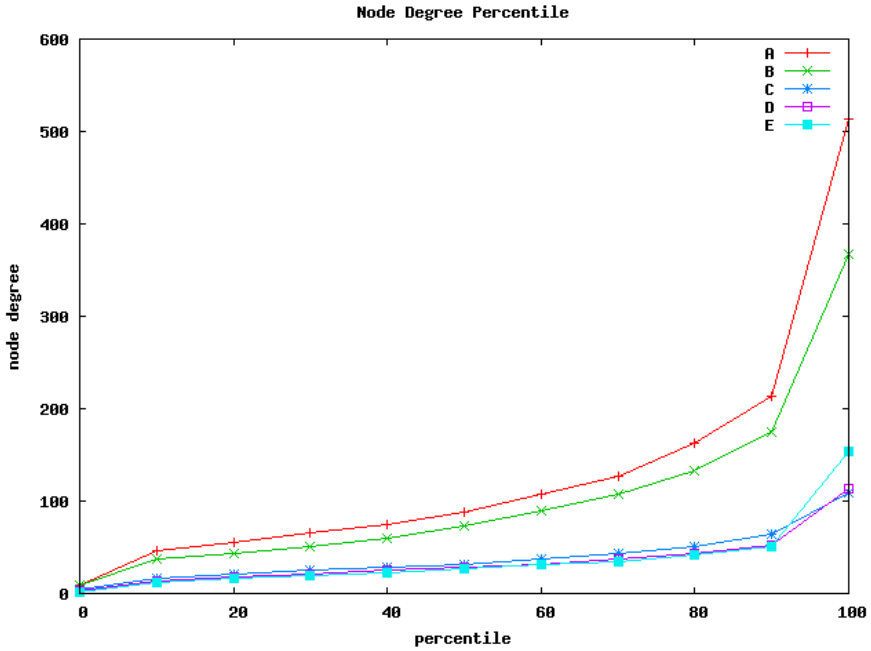


Figure 16.11: Node Degree Distribution

data to an undirected graph by defining an edge between A and B as "A is B's friend *or* B is A's friend".

The average degree of the graph is 110.97, which is very high comparing to DHTs. To study graphs with smaller degree, we removed some edges to get 4 other graphs. In the remainder of this section, we call the original graph A, and the 4 other graphs as B, C, D, and E.

Figure 16.11 and 16.12 shows the node degree and node clustering coefficient distribution for the five graphs. Graph A's maximum degree is about 500, which means there some nodes which connects to a lot (in this case, half) of nodes in the graph. This is very common in social networks. Some people are very popular in the community and some are not. Also note that although the node degree distribution of Graph C and D are similar, their clustering coefficients are different.

Figure 16.13 and 16.14 shows the routing failure ratio and routing length of the 5 graphs during the self-tuning process which remaps the node identifiers. The routing failure ratio is the number of routes which fail over all routes tested. All the graphs reach a steady state after about 300 iterations. This shows that routing in an actual social network can be more difficult than in am artificially constructed SWN. However, the failure ratio is small,

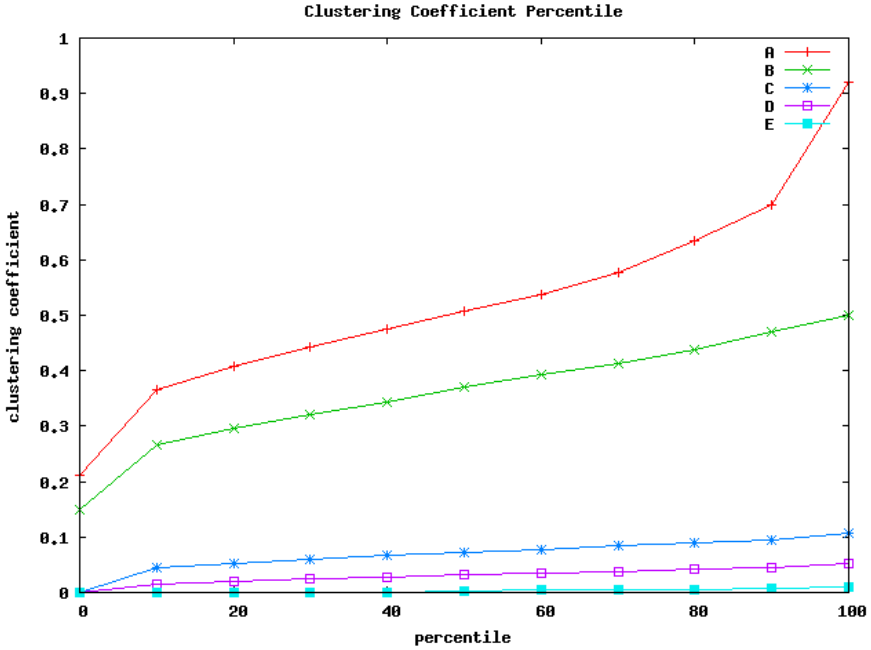


Figure 16.12: Clustering Coefficient Distribution

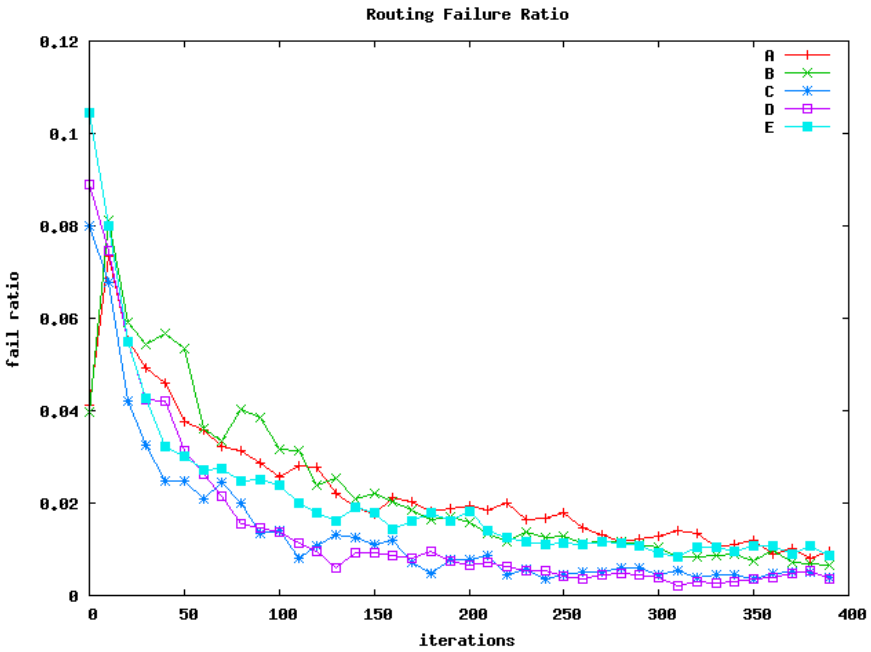


Figure 16.13: Routing Failure Ratio

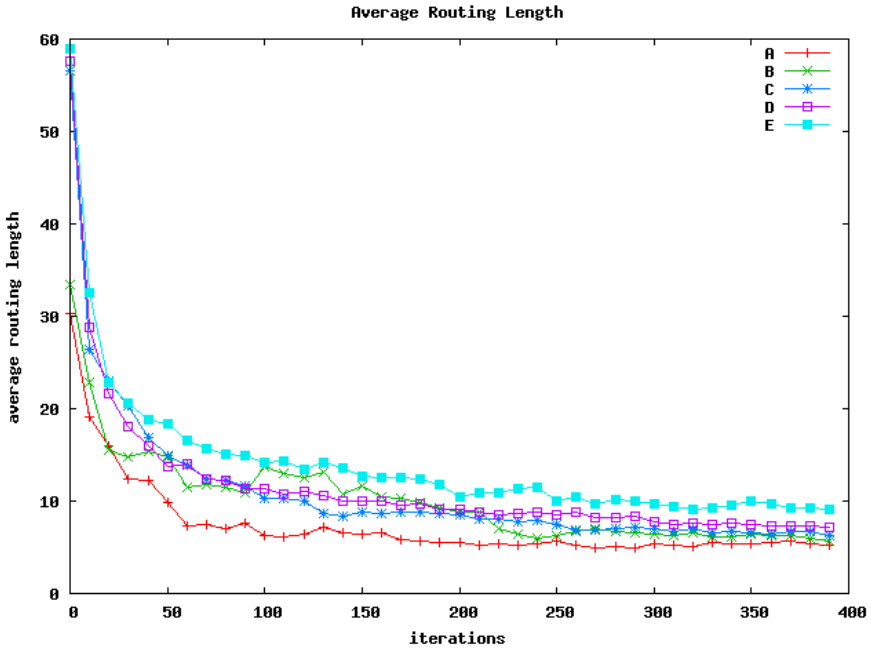


Figure 16.14: Routing Length

under 2%. The average routing length is larger than a SON with that number of nodes and edges. This is partly because the distribution of friends is skewed towards friends which are clustered together. However, it shows that routing length is still small enough to be reasonable.

16.3.2 Software component security

SELFMAN is concerned also with self-configuring software components. This means that the code and possibly data for the components needs to be updated. In order to make software update scalable, the dissemination mechanism should be distributed. This leads to security problems since attackers can attempt to attack or manipulate software updates and self-configuration as another way of attacking a system.

In the case software in Java, authentication can be achieved by signing the software archives. However, this security is only as secure as the JVM implementation. Integrity of distribution of software components is more secure at the operating system level [33]. We extended the authentication in BinAuth [33, 93] which was developed in D1.3b to also authenticate Java classes and archives. A policy file is used to specify the classes and archives

which can be used Java. The BinAuth extension provides a transparent way of ensuring mandatory verification of the integrity of every file loaded by the Java Virtual Machine as per a policy.

The following example demonstrates a policy to authenticate Java bytecode coming from `.class` and `.jar` files except for project `foo`. This policy can be shared by many Java programs such as `java.exe`, `javac.exe`, etc.

```
allow  E:\\projects\\foo\\.*\.class
allow  E:\\projects\\foo\\.*\.jar
verify .*\.class
verify .*\.jar
```

16.4 Application level security

Deliverable D5.6 was envisaged to be about the evaluation of security mechanisms from WP1 and WP4 in the context of the SELFMAN applications. There are some differences between the proposal and the work in D5.6. Firstly, due to changes in partners, SELFMAN had different demonstrator applications and any security considerations depend on the precise nature of the application. Secondly, while some of the security mechanisms from WP1 and WP4 are relevant, the main consideration would be the security as defined in the particular application. The resulting developed security mechanisms in D5.6 were developed based on these constraints.

Peerialism(P6) joined SELFMAN only in the middle of the project. Consequently there was only sufficient resources for Peerialism(P6) to focus on one of the self-* aspects. Peerialism(P6) focused on autonomous evaluation and while we had discussion on security issues, no person-months were allocated to security mechanisms and evaluation in the context of PeerTV.

The main application developed for SELFMAN is the distributed Wikipedia-clone based on Scalaris (see D5.2b). As such, the focus in D5.6 was on security mechanisms appropriate for Wikipedia which would fit with the Scalaris enhanced version of Wikipedia. The question then is what kinds of security issues and mechanisms are appropriate for Wikipedia. Both Scalaris and Wikipedia servers are data-centric focused. They assume that the machines in the data center(s) are secure which also implies that the right question is not on the infrastructure security at the heart of WP1 and WP4 but other Wikipedia specific security issues. Nevertheless, it made sense to look at what security problems and corresponding solutions which apply both to Wikipedia as well as other contexts.

An appropriate attack model is one where the attackers are the contributors to Wikipedia content. As Wikipedia is intended to be both open and anonymous, an attacker can abuse Wikipedia by deleting useful content and adding useless or irrelevant content. A related problem is to prevent spammers from taking over Wikipedia content. Wikipedia uses human editors to police such content attacks. We develop in D5.6 automatic infrastructure which addresses such attacks. The focus in D5.6 is a general purpose mechanism for adding trust to Wikipedia content and thereby making the information more credible. The problem of spam prevention is a related one which will be addressed in D5.11.

The mechanisms developed here for Wikipedia can also be seen as an extension of the issues developed in WP4 since a major problem in a self-managing system may be the lack of global trust. While the security infrastructure here is tailored for Wikipedia, the problem being addressed is a more

general one, namely generating trust and preventing spam when there are virtual identities. For example, the credibility mechanism is also useful for the demonstrator application for Mozart, the community-driven recommendation system for Beernet (D5.3). In the absence of global voter identities, such voting mechanisms can also be attacked by Sybil or other identity attacks.

16.4.1 Security issues for Wikipedia

The philosophy of Wikipedia is to generate content which is written by the world at large. It is intended to be: (a) open so that anybody can contribute; and (b) anonymous so that it is not censored or restricted as to who can contribute content. One of the criticisms of Wikipedia is that the content is written by "unknown strangers of unknown qualifications". Since anybody can register and make an edit to the content directly, the credibility of the content becomes questionable to the readers. Moreover, authors of the edits are anonymous — they are known only by pseudonyms or IP address. The credibility of information written in Wikipedia is based on having references to other sources.

An alternative to a user generated content like Wikipedia is Google Knol. Google Knol uses a different philosophy where authors have identities which have been certified by Google. For example, an author could be certified by their credit card. This is rather different from the open and anonymous approach in Wikipedia.

From a Wikipedia perspective there are two main security threats and issues. The first is to ensure that the information content is credible or trustworthy even with anonymity and openness. The second is that to reduce or prevent spam in such an environment. We have developed an architecture to facilitate the information transference between applications in a secure fashion which addresses these issues in Wikipedia. Although our architecture is useful for Wikipedia, we remark that the issue of trust/credibility and spam reduction/prevention are more generic and is thus not limited to Wikipedia.

We also take advantage of and are compatible with open protocols for authentication such as OpenID and OAuth. OpenID is an open standard for user distributed authentication and access control. Another protocol which is supported is OAuth which can be used to transfer more information.

Adding trust to Wikipedia

We propose an architecture which transfers trust between one or more independent parties and application by sharing information securely and anony-

mously. This architecture is used to as a credibility enhancement mechanism which allows content in Wikipedia to be marked up with credibility information [36].

An author may have existing credibility details which are available from third parties. Some examples of such credibility information are author's profession, name, position, etc. Our Wikipedia mechanism allows authors to bring a subset of these information into Wikipedia in anonymous manner in order to write or edit an article. Figure 16.15 illustrates this process.

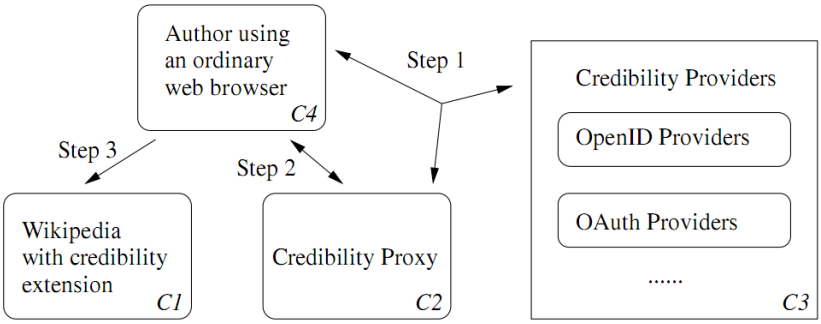


Figure 16.15: Wikipedia Credibility Extension

Figure 16.15 shows four components: C1 is the Wikipedia web server with our credibility extension installed. C2 is the credibility proxy. The Wikipedia web server stores a certificate of the credibility proxy so that Wikipedia can verify the proxy's signature using its public key. Credibility providers are the sources of credibility information specified by the author to the credibility proxy. C4 is the Wikipedia author using an ordinary web browser. The communication between the credibility proxy (C2), the credibility provider (C3) and the author (C4) is using widely used open protocols such as OpenID or OAuth.

The steps to make a credible (and anonymous edit) in a Wikipedia page is first to contact a trusted proxy (C2) and specify one of the credibility providers (C3) that the author (C4) want to use. The author is authenticated by the credibility provider and will be asked which kind of information to disclose to the proxy. We employ open protocols for the communication between the credibility provider and credibility proxy, e.g. OpenID and OAuth.

Once the proxy acquires the information of the author from the credibility provider, the author then adds the content which is to be added to a Wikipedia page. The proxy then signs the text together with the author's information from the provider. At this step, the author can select what

credibility information to be attached to the text. These steps are shown in Figure 16.16 which also shows a sample interface for using the credibility proxy for Wikipedia.

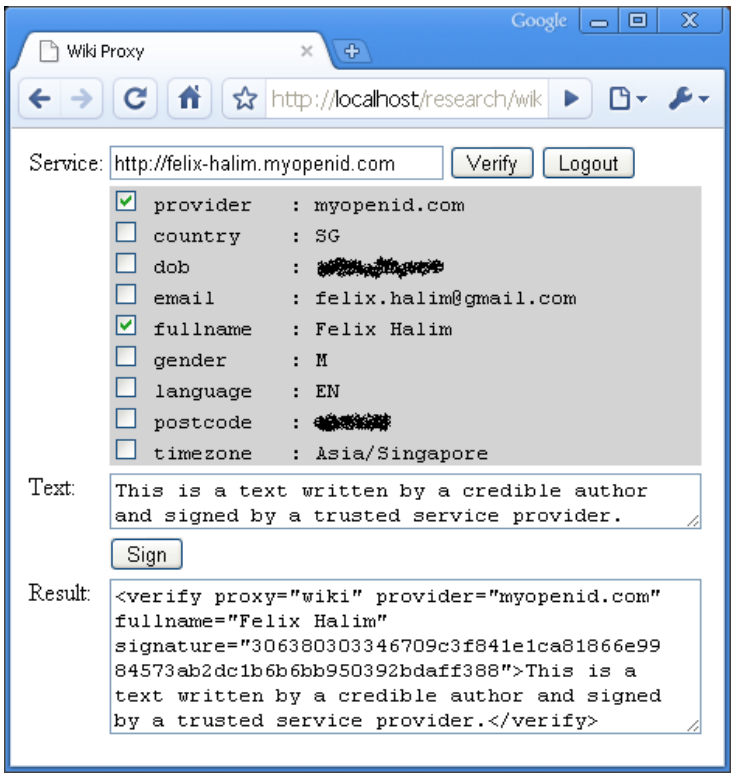


Figure 16.16: Wikipedia Credibility Proxy

The last step is to add the signed text to Wikipedia page just like the usual edit in Wikipedia as shown in Figure 16.17. The credibility information about the author should be displayed in a special way. For example, Figure 16.18 shows the end result of adding the content from Figure 16.16. More sophisticated user interfaces can also be easily employed.

To summarize, the Wikipedia credibility protection mechanism consists of a number of components: an infrastructure for transferring information securely between different parties; a credibility proxy which also serves as the interface for performing a credibility edit; various credibility providers which are the source of the credibility information; and a WikiMedia extension which deals with the credibility edit which is just normal data to WikiMedia.

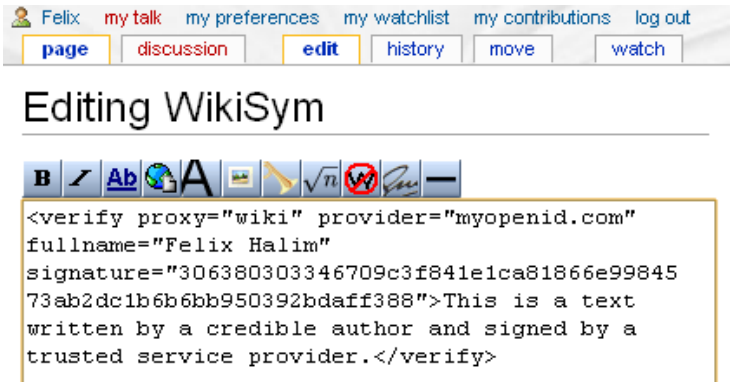


Figure 16.17: Wikipedia Verifier Tag Extension

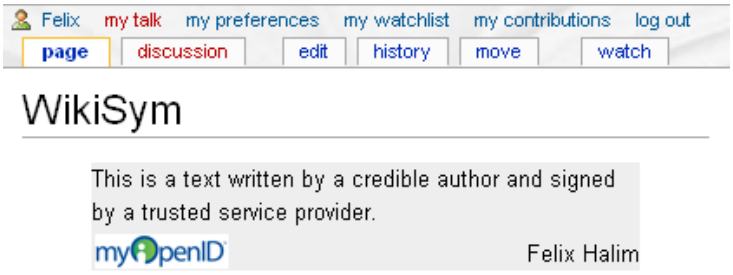


Figure 16.18: Wikipedia showing a credible enhanced text

Vandalism/spam attack prevention

The other application level security problem is one of vandalism or spam where there is a deliberate attempt to corrupt or maliciously change the content in Wikipedia. One approach is to analyze the content within Wikipedia or to analyze the history of the edits. We take a different approach which is more general and is applicable in more generic settings but still usable for Wikipedia.

We have developed infrastructure which can mediate between Wikipedia and other data sources such as the cloud in a secure fashion. We have already identified that social networks can be used as sources of trust and in particular, we have developed some infrastructure, which extracts information from Facebook. The work in the upcoming deliverable D5.11 ending at month 40 will focus on the use of social networks such as Facebook as a kind of credibility provider for controlling spam.

16.5 Papers and publications

Security issues in small world network routing

Felix Halim, Yongzheng Wu, Roland H.C. Yap, Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2008 (see A.18).

> Small World Networks (SWN) have been shown to be navigable – a short route can be found using efficiently using decentralized algorithms. This routing relies on nodes having a position to guide the routing such as its coordinates. Even in the absence of positional information such as node coordinates, by using local self-reorganization, it is possible to reconstruct a proxy for the node coordinates which still allows for efficient routing. This paper shows that in the presence of malicious nodes, the self-reorganization mechanism breaks down. We investigate self-protection mechanisms for such SWNs. Preliminary results using a simple restart mechanism for self-tuning shows that much of the effect of malicious nodes can be mitigated.

Small world networks as (semi)-structured overlay networks

Felix Halim, Yongzheng Wu, Roland H.C. Yap, Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASO), 2008 (see A.19).

> Recent research has shown that Small World Network (SWN) is navigable. In this position paper, we propose that SWN, for example those which are social networks, have nice properties which make them attractive as overlay networks. Such networks occupy a space between structured and unstructured overlay networks. Our thesis is that SWN may be attractive enough to be a replacement for traditional structured overlay networks which are usually based on Chord-style Distributed Hash Tables. Preliminary experiment results show that without node failure, the performance of greedy routing in SWN works very well and with additional links in SWN the robustness in routing can be improved as well as the resilience against node/link failure.

Establishing software integrity trust: a survey and lightweight authentication system for Windows

Yongzheng Wu, Sufatrio, Roland H.C. Yap, Rajiv Ramnath and Felix Halim, chapter in Trust Modeling and Management in Digital Environments: from Social Concept to System Development, Zheng Yan (ed.), IGI Global, Dec. 2009 (to appear).

> Malware causes damage by stealing confidential data or making other software unusable. Ensuring software trustworthiness is difficult because malware may disguise itself to appear benign or trusted. This chapter explores the problem of making software more trustworthy through the use of binary integrity mechanisms. We review the problem of devising an effective binary integrity protection, and discuss how it complements other operating system security measures. We analyze design factors for binary integrity and compare existing systems. We then present a prototype which exemplifies a mandatory binary integrity mechanism and its integration within an operating system. Our system, BinAuth, demonstrates a practical, lightweight in-kernel binary authentication system for Microsoft Windows. A system like BinAuth shows that mandatory authentication is practical on complex commodity operating system like Windows. To deal with various constraints in the user's environments, BinAuth uses a flexible scheme which does not mandate Public Key Infrastructure (PKI) although it can take advantage of it. We also combine the authentication with a simple software-ID scheme which is useful for software management and vulnerability assessment.

Wiki credibility enhancement

Felix Halim, Wu Yongzheng and Roland H.C. Yap, Fifth International Symposium on Wikis and Open Collaboration (WikiSym), 2009 (to appear, see A.20).

> Wikipedia has been very successful as an open encyclopedia which can be edited by anybody. However, the anonymous nature of Wikipedia means that readers may have less trust since there is no way of verifying the credibility of the authors or contributors. We propose to transfer external information from outside Wikipedia to Wikipedia pages. These additional information is meant to

enhance the credibility of the content. For example, it could be the education level, professional expertise or affiliation of the author. We do this while maintaining anonymity. In this paper, we present the design and architecture of such system together with a prototype.

Chapter 17

D5.7: Guidelines for building self-managing applications

17.1 Executive summary

This report outlines a methodology to build self-managing applications, consisting of the following steps:

- We assume that self-managing applications use an underlying architecture based on loosely coupled components and asynchronous communication. We show three acceptable variations of this architecture, namely the Kompics component model, the Erlang language (used for Scalaris), and the Oz language (used for Beernet).
- The self-managing application then consists of a set of self-managing managers, each of which is assigned a specific management task.
- Each manager is built as a feedback structure. A feedback structure is a set of interacting feedback loops.
- The application design now consists of two steps: decomposition (determining what managers are needed and what they do) and orchestration (handling interactions between the managers).
- We give one complete example of a self-management architecture. To justify each part of this architecture, we show step by step how it is built in the Kompics model.
- Since feedback structures are omnipresent, we give design rules for building them. The correct design of feedback structures is important for building self-managing managers.

- Finally, we focus on the four main self-management axes, namely self tuning, self protecting, self healing, and self configuring, and explain what to do for each case.

This methodology is based on our experience in SELFMAN augmented with important contributions from the GRID4ALL project and from the literature on self-managing systems. Many of our examples are based on decentralized applications that use a structured overlay network for communication and storage, since that is the area where SELFMAN has mainly worked on. We outline what remains to be done to flesh out our methodology.

17.2 Contractors contributing to the Deliverable

This deliverable was written by UCL(P1) (Peter Van Roy) based on SELFMAN results.

17.3 Introduction

How should self-managing systems be designed? For practical system design, it is important to have a methodology that is simple and that allows to design systems with desired global properties. To our knowledge, such a methodology does not yet exist. Most of the knowledge in this area is fragmented and deriving formal properties is difficult. In the SELFMAN project we have addressed a particularly interesting part of the self-management design space: structured overlay networks that survive in realistically harsh environments (with imperfect failure detection and network partitioning) and that provide a transactional storage interface. During the project we have accumulated experience in how to design self-managing systems. This report summarizes this experience. We have built libraries at different levels of abstraction and we have built application demonstrators and self-* services:

- A self-managing structured overlay network for the communications infrastructure. The Scalaris and Beernet libraries both have structured overlay networks.
- A component model for building self-managing applications. The component model supports isolated concurrent components, with hooks for observation and reconfiguration.
- A replicated storage layer and a transaction protocol for managing sharing and coordination for applications. The Scalaris and Beernet libraries both implement transactions on top of the structured overlay network, using a modified version of the Paxos uniform consensus protocol to handle atomic commit in the face of Internet failure model (permanent node failures and temporary communication failures, i.e., false failure suspicions are possible) [61].
- Application demonstrators, such as a Distributed Wiki and a Decentralized Collaborative Drawing Tool.
- Experiments with self-* services, such as load balancing, network partitioning handling, and security (small-world topologies, monitoring, and component security).

From this experience, this report collects a set of guidelines that we provide to future designers of self-managing applications. This report also takes ideas and examples from other sources, including the GRID4ALL project, which has addressed similar problems [6].

17.3.1 Context of this report

This report gives guidelines for the design of a specific kind of self-managing system, namely one that separates a design phase (done by human developers) and a self-managing execution phase (done by the system itself, which is able to adapt itself according to the abilities contained in the design). More general self-managing architectures exist, e.g., the robotic architecture of Gat [26] as applied to self management by Kramer and Magee [47], in which the system design is itself done by part of the system. In Gat's design there are three layers: a component control layer that consists of feedback loops, a change management layer that applies plans to select new control layers, and a top layer that creates new plans using time-consuming deliberation algorithms. In the present report we focus on the first two layers. The first layer corresponds to our feedback loop architectures and the second layer corresponds to our component reconfiguration. The third layer would redesign the components and devise the plan to implement this redesign (i.e., reconfiguration). The third layer is beyond the scope of this report.

17.3.2 General guidelines

A self-managing application consists of a set of interacting feedback loops. Each of these loops continuously observes part of the system, calculates a correction, and then applies the correction. Each feedback loop can be designed and optimized separately using control theory [38] or discrete systems theory [16]. This works well for feedback loops that are independent. If the feedback loops interact, then your design must take these interactions into account. In a well-designed self-managing application, the interactions will be small and can be handled by small changes to each of the participating feedback loop. This is the kind of design that we will focus on in this report.

It can happen that parts of the self-managing application do not fit into this "mostly separable single feedback loops" structure. We have encountered several examples of this in the SELFMAN project. We recommend the following approach:

- In the case of a large number of agents that collaborate, the best approach is to design a distributed algorithm [32] or a multi-agent system [79] to perform the task. For example, in SELFMAN we needed an algorithm to perform atomic commit for distributed transactions, in the face of possible node failures and communication interruptions (imperfect failure detection). We found that a modified version of the Paxos uniform consensus protocol was an essential part of the solution. This is a complex algorithm whose correctness is not trivial to prove [61].

Instead of trying to reinvent it in terms of interacting simple feedback loops, we used the existing knowledge about this algorithm.

- In the case when the feedback loop structure consists of more than one loop intimately tied together, the global behavior must be determined by analyzing the structure as a whole and not by trying to analyze each loop separately. To our knowledge, no general methodology for doing this exists. We have made progress on two fronts: design rules for feedback structures and patterns for common feedback structures. Section 17.6 summarizes some of the important design rules we have encountered. We are preparing a comprehensive survey of feedback loop patterns [15]. Many commonly occurring complex patterns, such as "Tragedy of the Commons" and "Communication Congestion Control", have been extensively studied and it is possible to take an existing pattern from the literature. Unfortunately, the literature is extremely fragmented. Studies of feedback loop systems exist in widely different disciplines, such as management [76], biology [44, 46], and computer science [16, 38]. A future research topic is to devise a methodology at a similar level of abstraction to an existing methodology of software construction (e.g., such as object-oriented programming).

The SELFMAN project has partly studied these two cases (multiple agents and complex feedback). But the domain is very rich and we can guide the designer only along the paths that we have explored ourselves.

17.3.3 Phase transitions

Feedback loop architectures often show abrupt changes in behavior. These shifts can be seen as *phase transitions*. A *phase* in a feedback structure is similar to a phase in thermodynamics: a situation in which the system has well-defined global properties and reacts in a well-defined way to external stimuli. There is a phase transition when the system's global properties and reactions to stimuli change abruptly. If the feedback structure is properly designed, then it reacts to an increasingly hostile environment by doing a *reversible* phase transition [83]. For example, when the node failure rate increases or the network has communication problems (e.g., a partition), then a large overlay network may become a set of disjoint smaller overlay networks. We can say that the overlay network has made a transition from a "liquid" phase (connected with changing set of neighbors) to a "gaseous" phase (disconnected).

This transition does not necessarily mean the end of the overlay network. It can be a normal part of the overlay network's behavior, if the system is

properly designed. When the failure rate decreases, these smaller networks will coalesce into a large network again if the system is properly designed, e.g., with a merge algorithm. The first practical merge algorithm for SONs was developed in SELFMAN [77]. Earlier SONs could not "condense" (move from a gaseous back to a solid phase) as failure rates decreased or communication was reestablished. They would boil (become disconnected) in a hostile environment and then stay disconnected forever. We conclude that network merge is more than just an incremental improvement that helps improve reliability. It is fundamental because it allows the system to sur vive any number of phase transitions. The system is reversible and therefore does not break. Without it, the system breaks after just a single phase transition.

The lesson for system designers is always to make a design that can perform reversible phase transitions. This implies designing algorithms for all pairs of phases for which a transition is possible, in both directions. In addition to this, we recommend to make a design that takes advantage of phase transitions by exposing them to the application as an API. For example, a transactional store built on an overlay network will become a set of smaller stores when there is a network partition. When the overlay network merges, the application needs to merge the data stored in each subnetwork. How to do this is application-dependent; the transaction layer can just provide an API to permit it to be done. The main design issues are to determine what this API should be and how it affects application design. Full answers to these issues are outside the scope of the SELFMAN project. We propose to address them in a follow-up project.

17.3.4 Interdisciplinary nature

It is clear that designing self-managing systems touches on many disciplines. We have encountered the following disciplines:

- Control theory [38]. This allows quantitative design of systems with one or two feedback loops, typically to optimize throughput or some other quantitative property.
- Discrete event systems [16]. This discipline includes control of automata and queueing theory. This generalizes control theory to discrete systems, in which each component is defined by an automaton. Control of these systems can be defined theoretically but is a combinatorially difficult problem in the general case.
- Multi-agent systems [79]. This discipline includes game theory, auction theory, and collective intelligence. This provides practical solutions to

well-defined collaboration problems, called "games" or "auctions".

- Distributed algorithms [32]. This discipline underlies the construction of distributed computing systems. The algorithms depend strongly on the failure model (e.g., whether false suspicions of failure are possible) and the synchronicity model (whether the system is asynchronous or synchronous). This discipline has reached sufficient maturity to provide algorithms for many practical design problems. In SELFMAN, we use the Paxos uniform consensus algorithm as the heart of the transaction commit algorithm.

- Various limited studies on biological or engineering systems that use feedback. There exist many successful biological systems (e.g., the human respiratory and endocrine systems) and engineering systems (e.g., TCP/IP, which can be formulated as a feedback structure) [82, 84]. These systems can be used as examples when designing new systems.

There is no one body of theory on how to build self-managing applications. For parts of the system we recommend to choose the appropriate discipline, as listed above. For the overall system, we recommend to follow general design rules, as presented in the rest of this report, and to take inspiration from existing successful self-managing systems, from biology, sociology, and engineering. For isolated parts of the system and depending on the application requirements, you may need to draw on one or more of these disciplines, as we have done in SELFMAN.

17.3.5 Structure of this report

This report gives an outline of a methodology to build self-managing systems. It is structured as follows:

- Section 17.4 proposes a general architecture for self-managing systems based on loosely coupled components with asynchronous communication. This architecture supports feedback structures, which are the basic design element in a self-managing system: a feedback structure is a set of interacting feedback loops.

- Section 17.5 gives three examples of self-managing systems that were built in SELFMAN using variants of this architecture, namely Kompics, Scalaris, and Beernet. For each of these systems, we explain how it performs self management. In particular, Section 17.5.1 gives a step-by-step outline how to build a self-management architecture with Kompics.

- Section 17.6 gives design rules for building feedback structures. This section just scratches the surface of an area that we intend to investigate further. Much of the literature in this area is fragmented; we are writing a survey to bring this information together [15].
- Section 17.7 then explains how to design the overall structure of a self-managing system, as a set of managers, each of which is implemented as a feedback structure. We design this structure by means of two techniques called decomposition (determining the managers' roles) and orchestration (handling interactions between managers).
- Section 17.8 targets these ideas to the four main self-management axes, namely self tuning, self protecting, self healing, and self configuring. We give specific examples and rules for each axis.
- Section 17.9 concludes by summarizing briefly the methodology presented in this report and by presenting two important unsolved problems for future work.

For all sections we take examples from SELFMAN and from the literature and we reference the appropriate SELFMAN publications.

17.4 The general architecture

We propose to build self-managing systems as loosely coupled sets of concurrent and distributed components, with an asynchronous communication mechanism between components. The default behavior is that components do not communicate. We then gradually add well-defined communications between components, to implement the algorithms in the design. Components with no communications defined between them cannot affect each other.

We have used this approach in designing the three main software artefacts of SELFMAN: the Scalaris library [73], the Kompics component model [21], and the Beernet library [54]. Scalaris is written in Erlang, a language based on asynchronous message passing between isolated processes. Erlang's failure detection model creates asynchronous messages whenever the failure of a process is detected by the system [8]. Kompics is written in Java and provides independent concurrent components that are connected using channels. A channel is a conduit for asynchronous typed events, and each component connected to a channel can subscribe to specific types of events. Components can be disconnected and reconnected, which is needed for reconfiguration.

Beernet is written in Oz and uses active objects with asynchronous message passing, and fault streams for failure detection [18]. Fault streams generalize Erlang's failure detection to handle also temporary failures (also known as false failure suspicions).

17.4.1 Three-layered architecture

The general architecture has three layers:

1. Components and events. This basic layer corresponds to the service architec- ture mentioned above: services based on concurrent components that interact through events. There can be publish/subscribe events, where any component that subscribes to a published type will receive the events. There is a failure detection service that is eventually perfect with suspect and resume events. There can be more sophisticated services, like a transaction service.

2. Feedback loop support. This layer supports building feedback loops. This is sufficient for cooperative systems. The two main services needed for feedback loops are a best-effort broadcast (for actuating) and a monitoring layer. Best-effort broadcast guarantees that nodes will receive the message if the originating node survives [32]. Monitoring detects both local and global properties. Global properties are calculated from local properties using a gossip algorithm [41] or using belief propagation [89]. The multicast and monitoring services are used to implement self management abilities.

3. Multiple user support. This layer supports users that compete for resources. This is a general problem that requires a general solution in the area of self protection. If the users are independent, one possible approach is to use collective intelligence techniques. These techniques guarantee that when each user maximizes its private utility function, the global utility will also be maximized. This approach does not work for Sybil attacks (where one user appears as multiple users to the system). No general solution to Sybil attacks is known. A survey of partial solutions is given in [94]. We cite two known partial solutions. One solution is to validate the identities of users using a trusted third party. Another solution is to use algorithms designed for a Byzantine failure model, which can handle multiple identical users up to some upper bound. Both solutions give significant performance penalties. It seems that the only solutions that make sense in a distributed context are ones that exploit graph properties of social networks. See Section

17.8.2 for more information on how to implement self protection using collective intelligence and the graph properties of social networks.

17.4.2 Combining structured overlay networks and components

The SELFMAN project is based on the observation that there is a synergy between structured overlay networks (SONs) and component models. This synergy has confirmed itself during the project as we have built the transactional store and the application demonstrators. This leads to a set of loosely coupled services built on top of a structured overlay network. Feedback loop structures are built within this framework. We recapitulate the initial reasoning:

- SONs already provide low-level self-management abilities. We are reimplementing our SONs using a component model that adds lifecycle management and hooks for supporting services. This makes the SON into a substrate for building services.

- The component model is based on concurrent components and asynchronous message passing. It uses the communication and storage abilities of the SON to enable it to run in a distributed setting. Because the system may need to update and reorganize itself, the components need introspection and reconfiguration abilities. We have designed a process calculus, Oz/K, that has these abilities in a practical form [23].

This leads to a simple service architecture for decentralized systems: a SON lower layer providing robust communication and routing services, extended with other basic services and a transaction service. Applications are built on top of this service architecture. The transaction service is important because many realistic application scenarios need it (see, e.g., the three SELFMAN demonstrator applications: Distributed Wiki, Sindaca recommendation system, and DeTransDraw collaborative drawing tool).

The structured overlay network is the base. It provides guaranteed connectivity and fast routing in the face of random failures [81]. It does not protect against malicious failures: in our current design we must consider the network nodes as trusted. We assume that untrusted clients may use the over- lay as a basic service, but cannot modify its algorithms. See chapter 16 for more on security for SONs and its effect on SELFMAN. We have designed and implemented robust SONs based on the DKS, Chord#, and P2PS protocols. These implementations use different styles and platforms, for example DKS is implemented in Java and uses locking algorithms for node join

and leave. P2PS is implemented in Oz and uses asynchronous algorithms for managing connectivity (which gives a relaxed ring topology, explained in Appendix A.4). We have also designed an algorithm for handling network partitions and merges, which is an important failure mode for structured overlay networks [77].

The transaction service uses a replicated storage service (Section 6). The transaction service is implemented with a modied version of the Paxos non-blocking atomic commit and uses optimistic concurrency control [30, 61]. This algorithm is based on a majority of correct nodes and eventual leader detection (the so-called partially synchronous model). It should therefore cope with failures as they occur on the Internet.

17.4.3 Failure detection

An important part of a self-management architecture is the approach used for failure detection. We find that using objects with RMI is difficult; it breaks abstraction boundaries and is generally hard to program with [55]. A much better approach is to use an independent failure detection component in the system that informs the application asynchronously. We have explored two variations of this idea:

a In Beernet, we use the Mozart failure detection architecture, which uses fault streams attached to distributed language entities. The fault stream is created by failure detection in the run-time system. The fault stream is read asynchronously by a part of the application independent of the failing part. The fault stream contains events for permanent failures, temporary failures (failure suspicions), and for resuming (lifting a suspicion).

b In Scalaris, we use the Erlang failure detection architecture, which is based on message passing. The run-time system creates messages when it detects failures. The messages are sent to part of the application, which can then take action. The messages detect permanent failures only. Erlang does not support failure suspicions.

This general approach works well in both these cases and we can recommend it for fault tolerance in new self-managing applications. The Erlang system is designed for working in cluster or cloud computing environments, in which there are only permanent failures. The Mozart system is designed for working in an Internet environment, in which there can be frequent failure suspicions (which are often unjustified, but which must be handled anyway).

Critique of RMI for building distributed systems

Our experience shows that RMI (remote method invocation) is the wrong primitive for building distributed systems, for two reasons: exceptions break transparency and synchronous communication adds a useless and slow dependency [55]. The classical view of distributed computing sees partial failure as an error. For instance, a RMI on a failed object triggers an exception. This goes against distribution transparency, because the programmer is not supposed to make the distinction between a local and a distributed entity. Therefore, an exception due to a distribution failure is completely unexpected, breaking transparency. Another issue is both that RMI and RPC are conceived as synchronous communication between distributed processes. Due to network latency, synchronous communication is not able to provide good performance because the execution of the program is suspended until the answer (or an exception) arrives. Synchronous communication also creates a dependency between the sender and the receiver. This dependency is often not needed by the application, but it adds a failure mode to the application that must be handled anyway.

Instead of RMI, we use asynchronous message sending and fault streams. Asynchronous messages introduce no extra dependency. Fault streams indicate errors independently of the rest of the application and hence do not break abstraction boundaries. The Scalaris application is built in Erlang and the Beernet application is built in Oz, both of which support asynchronous messages and asynchronous notification of faults.

More recent trends, such as ambient intelligence and peer-to-peer networks, see partial failure as an inherent characteristic of the system. A disconnection of a process from the system is considered normal behaviour, where the disconnection could be a gentle leave, a crash of the process, or a failure on the link. Together with asynchronous messages and fault streams, our experience shows that this approach leads to more realistic language abstractions to build distributed systems.

17.5 Examples of the general architecture

In SELFMAN we have built three significant software designs that use the general architecture of the previous section: the Kompics component model, the Scalaris library, and the Beernet library. Here we summarize these three designs and highlight the lessons they have taught us for building self-managing systems.

17.5.1 A self-management architecture built with the Kompics component model

We show how to build one self-management architecture with the Kompics component model. Kompics was designed to support the operations needed for self-managing systems [21] and the design of this section uses many of its abilities. Here we give a brief overview of Kompics followed by a presentation of the architecture that we have built. The architecture respects the following rules of thumb:

- Fault tolerance requires Isolation requires Concurrency
- Isolation requires Loose Coupling requires Event Passing
- Loose Coupling requires Publish-Subscribe Interaction
- Complexity Management requires Compositionality and Encapsulation
- Resource Reuse requires Sharing
- Self-management requires Online Reconfiguration requires Loose Coupling

We recommend to follow these rules in any self-managing application.

Overview of Kompics

The Kompics model consists of concurrent components that communicate with each other by asynchronously passing typed events on channels (for more information see the Kompics user manual [21]). The current implementation of Kompics is in Java, however, we have made an effort to make it language-independent. Events are different from messages in that they can be received by all the components that are connected to the channel. Components are loosely coupled: they do not know the type, availability, or identity of the components with which they communicate. Components can be nested and shared. Components react to events by atomically executing event handlers, which may trigger new events. Because components are independent, it was possible for us to make Kompics support multi-core scheduling, which allows it to transparently take advantage of multi-core processors.

Building the architecture step by step

Figure 17.1 gives the structure of a realistic self-management architecture that we have built in Kompics, with components for overlay networks and

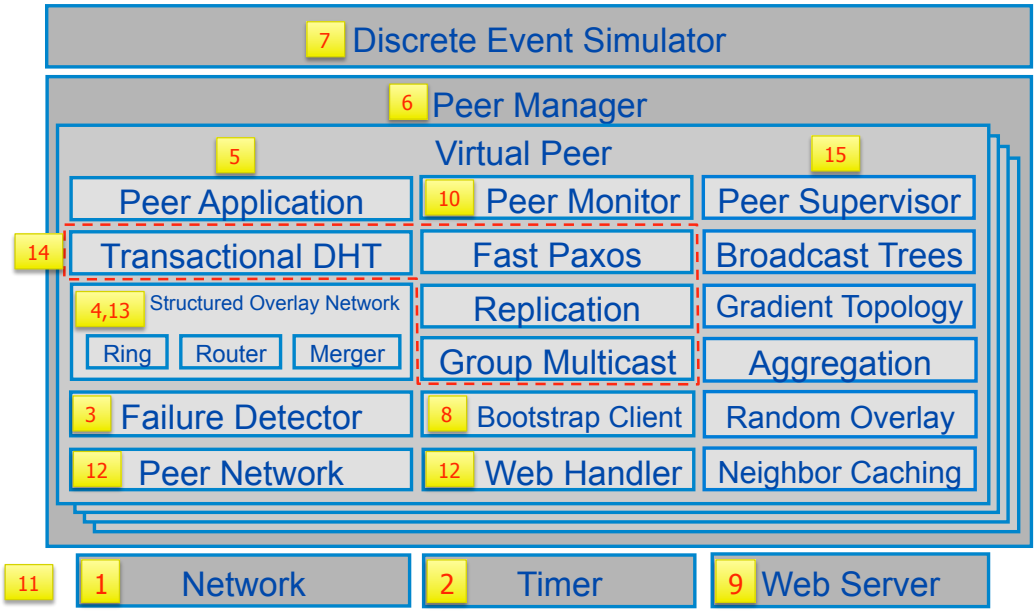


Figure 17.1: A self-management architecture built with Kompics

transactions and with various self-management services. This design seems complex, with a lot of components and nesting structure, but it is actually quite simple and coherent. We reconstruct the system in steps so that it is clear why each component is needed. We build the design with reusable components, instead of keeping it as a monolithic block. Here is a step-by-step explanation of the design (item numbers correspond with the numbers in Figure 17.1):

1. Encapsulate the communication primitives inside a Network abstraction.

2. Encapsulate the timeout and alarm primitives inside a Timer abstraction.

3. Encapsulate the failure detection primitives inside a Failure Detector abstraction.

4. Decompose the SON into its functional aspects, namely the Ring (with periodic stabilization), the Router (with topology maintenance), and the Merger (ring unification). All of these fit into the SON abstraction, which can be used as a building block for higher-level abstractions.

5. Encapsulate everything so far (Peer Application, SON, Failure Detector, Network, and Timer) into a Virtual Peer component. Making the

complete application so far into a component is an important prerequisite for reconfiguration.

6. Allow an enclosing Peer Manager component to add and remove Virtual Peers.

7. The Peer Manager is now generally available as a component in its own right; for example it can be driven by a Discrete Event Simulator component for complex tests.

8. Encapsulate the bootstrapping procedure into a separate component, the Bootstrap Client. This is again important for reconfiguration: the procedure is itself a component that can be replaced.

9. Enable Web-based peer state visualization and debugging with a Web Server component, added at the same level as Network and Timer.

10. Collect global state from local state pushed periodically by a monitoring component, the Peer Monitor, added at same level as Peer Application component (all inside Virtual Peer!).

11. Share the Network, Timer, and Web Server components among all Virtual Peers. Note that the communication part of Kompics allows the Virtual Peers to talk separately to each of these three components.

12. Inside Virtual Peer, keep peer subcomponents unchanged by adding a Peer Network component and a Web Handler component. These are actually proxies. Note that Peer Network can also act as a network simulator.

13. The three SON subcomponents can be replaced, for example the Ring component can be replaced by a Beernet relaxed ring [57] or a Chord# ring [74], the Router component can be replaced by a Chord# router. This can be done without changing anything else: it is a nice form of modularization using the reconfiguration ability.

14. Now add protocol components: Transactional DHT, Fast Paxos, Replication, and Group Multicast. These are dependent: the Transactional DHT uses the other three to implement the transactional store service.

15. We can also add a new pillar inside the Virtual Peer, of components that provide other useful services: Peer Supervisor, Broadcast Trees, Gradient Topology, Aggregation, Random Overlay, and Neighbor Caching. The Peer Supervisor is inspired by Erlang: it supervises peer components for *software* faults [8].

This gives a fairly complex, but flexible and manageable software architecture. Each part has its place and the whole works together well. Many of the subcomponents perform self-management tasks. We propose this architecture as a guideline: it shows one way to bring components together in the context of a structured overlay network. We have done many experiments with Kompics and with this architecture. We are confident that it performs well with no unexpected surprises.

17.5.2 Using self management to provide availability and scalability: the Scalaris example

Scalaris is an example application providing a self-managing data management service for Web 2.0 applications [73]. Web 2.0 initiated a business revolution: service providers offer Internet services for many activities, shopping, online banking, information, social networking, and recreation. In today's society Web 2.0 is no longer a convenience, but customers rely on its continuous availability, regardless of time and space. How to cope with such strong demands, especially in case of interactive community services that cannot be simply replicated? All users access the same Wikipedia, meet in the same Second Life environment and want to discuss with others via Twitter. Even the shortest interruption, caused by system downtime or network partitioning may cause huge losses in reputation and revenue. Web 2.0 services are not just an added value, but they must be dependable. Apart from 24/7 availability, providers face another challenge: they must, for a good user experience, be able to respond within milliseconds to incoming requests, regardless whether thousands or millions of concurrent requests are currently being served. Indeed, scalability is a key challenge. Any scalable service, to be affordable, somehow requires the system to be self managing.

Scalaris provides a comprehensive solution for self-managing scalable data management. Scalaris provides the traditional ACID properties of transactions in a scalable decentralized setting. Scalaris does not attempt to replace current database management systems with their general, fullfledged SQL interfaces. Instead our target is to support transactional Web 2.0 services like those needed for Internet shopping, banking, or multiplayer online games. Our system consists of three layers:

1. At the bottom, an enhanced structured overlay network, with logarithmic routing performance, provides the basis for storing and retrieving keys and their corresponding values. In contrast to many other overlays, our implementation stores the keys in lexicographical order. Lexicographical ordering instead of random hashing enablescontrol of data

placement which is necessary for low latency access in multidatacenter environments.

2. The middle layer implements data replication. It enhances the availability of data even under harsh conditions such as node crashes and physical network failures.

3. The top layer provides transactional support for strong data consistency in the face of concurrent data operations. It uses a fast consensus protocol with low communication overhead that has been optimally embedded into the structured overlay.

As a challenging benchmark for Scalaris, we implemented the core of Wikipedia, the "free encyclopedia, that anyone can edit". The Wikipedia on Scalaris is fast. Using eight servers it executes 2,500 transactions per second. All operations are performed within transactions to guarantee data consistency and replica synchronization. Adding more computers improves the performance almost linearly. The public Wikipedia, in contrast, employs ten servers to execute the 2,000 requests per second on its large master/slave MySQL database in Tampa.

For many Web 2.0 services, the total cost-of-ownership is dominated by the costs needed for personnel to maintain and optimize the service. Scalaris greatly reduces the operation cost with two builtin self-management properties:

- Self healing: Scalaris continuously monitors the hosts it is running on. When it detects a node crash, it immediately repairs the overlay network and the database. Management tasks such as adding or removing hosts require none or minor human intervention.

- Self tuning: Scalaris monitors the nodes's workload and autonomously moves items to distribute the load evenly over the system to improve the response time of the system. When deploying Scalaris over multiple datacenters, these algorithms are used to place frequently accessed items nearby the users.

In traditional database systems these operations require human interference which is error prone and costly. With Scalaris the same number of system administrators can operate much larger installations than with legacy databases.

17.5.3 Using a relaxed ring to simplify overlay maintenance: the Beernet example

The Beernet library is similar in its functionality and underlying implementation platform to Scalaris. Both Beernet and Scalaris use asynchronous message passing between lightweight "objects" to implement a replicated transactional store over a structured overlay network. There are three main differences between Beernet and Scalaris:

- Beernet is written in Oz (using the Mozart Programming System) [62] and Scalaris is written in Erlang [9]. The Oz language provides a network-transparent distribution layer that implements failure detection using fault streams. This can be seen as a generalization of Erlang's failure detection to detect temporary as well as permanent failures [18].

- Beernet uses a relaxed ring overlay network [58] and Scalaris uses a Chord# overlay network [73]. The relaxed ring differs from overlays descended from Chord, such as Chord# and DKS, in that it simplifies ring maintenance. The node join and failure algorithms need the agreement of only two nodes at each step instead of three. The latter requires the atomic update of three peers, which leads to increased perceived node failures when the update fails. In the relaxed ring, there is no need for periodic maintenance of the ring, because the ring remains correct after each step. The relaxed ring algorithms, failure detection and join, are presented as feedback structures in [57].

- Beernet extends the transactional store algorithm to do eager locking and to provide notifications of object updates also to nodes that do not participate in transactions. This extension is needed for the collaborative drawing application [56].

These differences are large enough to see Beernet as a different point in the design space. One of the lessons learned from Beernet is to avoid shared-state concurrency (i.e., threads accessing shared objects through monitors). We achieve this by encapsulating state, by doing asynchronous communication between threads and processes, by using single-assignment variables for dataow synchronization, and by serializing event handling with a stream (queue) providing exclusive access to the state. The language primitives needed, ports and lightweight threads, are also present in Erlang and are not specic to object-oriented programming. Single-assignment variables also appear in other languages such as E and AmbientTalk, in the form of promises.

In Beernet as in Scalaris, we organize the system in terms of active objects only, where an active object consists of a single thread reading a message

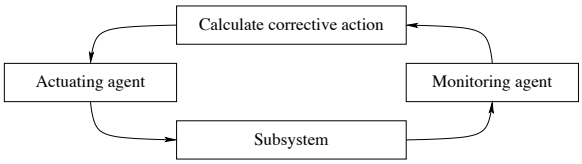


Figure 17.2: A feedback loop

queue and serializing messages to an internal object. We make no distinction in the send operation between a local and a remote active object. Transparency is respected by not raising an exception when a remote reference is broken. There is only one kind of entity, an active object, and only one send operation. The Kompics model is closely related to this: corresponding to active objects and message queues, there are event-driven components and typed event channels.

17.6 Design rules for feedback structures

When building a self-managing system, one of the basic building blocks is the feedback structure: a set of interacting feedback loops. This section gives some basic design rules for feedback structures. We first define what a feedback loop is. A feedback loop consists of three parts that interact with a subsystem (see Figure 17.2): a monitoring agent, a correcting agent, and an actuating agent. The agents and the subsystem are concurrent components that interact by sending each other messages. As explained in [82], feedback loops can interact in two ways:

- Stigmergy: two loops monitor and affect a common subsystem.
- Management: one loop directly controls another loop.

How can we design systems with many feedback loops that interact both through stigmergy and management? We want to understand the rules of good feedback design, in analogy to structured and object-oriented programming. Following these rules should give us good designs without having to laboriously analyze all possibilities. The rules can tell us what the global behavior is: whether the system converges or diverges, whether it oscillates or behaves chaotically, and what states it settles in.

To find these rules, we start by studying existing feedback loop structures that work well, in both biological and software systems. We have studied different kinds systems: artificial systems (Wiener's hotel lobby example),

computing systems (TCP/IP implementation), and biological systems (human respiratory and endocrine systems). We summarize the design rules that we have found. For more details on the particular examples we refer to the original papers [82, 84, 83].

17.6.1 Stigmergy should be used with care

In Wiener's hotel lobby example, two independent loops attempt to control the temperature of a hotel lobby: a thermostat connected to an airconditioner and a primitive savage lighting a bonfire. This is an example of uncontrolled stigmergy: the two loops will compete and this may lead to a runaway situation such as the hotel being set on fire. The solution in this case is to replace stigmergy by management: the primitive savage should control the temperature by manipulating the thermostat. This is a case where stigmergy is undesirable. In other cases, stigmergy can be used in a positive way. In the example of Section 17.7.4, two agents (a replica manager and a storage manager) communicate through stigmergy. One agent deliberately sets up a situation that will be detected and corrected by the second agent. This is a correct use of stigmergy.

17.6.2 Loop management corresponds to data abstraction

In the human respiratory system, breathing can be consciously controlled. This is modeled by a complex component that manages a respiratory loop. The respiratory loop handles the details of controlling the muscles of the human breathing apparatus and timing the breathing cycles. The complex component does not have to understand these details, but interacts through several parameters in the respiratory loop: the timing of the cycle and the depth of the breathing.

17.6.3 Loop management should control a natural parameter

In the TCP/IP implementation, an inner loop handles the sliding window protocol and an outer loop manages the inner loop to control congestion. The outer loop does this in a simple way: it changes the size of the sliding window. This changes the bandwidth needed by the inner loop, since the size of the sliding window determines the number of packets that can be "in transit" at any moment in time. This is an example of a natural management: the

outer loop adjusts a simple parameter of an inner loop. No other interaction is needed.

17.6.4 Take advantage of different time scales

Different parts of a system, such as two feedback loops, can take advantage of different time scales. For example, one loop can work at short times and another at long times, thus avoiding interference. Or one loop can gather information using short times, and then pass this information to another loop that works at long times. Norbert Wiener [88] gives a simple example of a human driver braking an automobile on a surface whose slipperiness is unknown. The human "tests" the surface by small and quick braking attempts; this allows to infer whether the surface is slippery or not. The human then uses this information to modify how to brake the car. This technique uses a loop at a short time scale to gain information about the environment, which is then used for regulation at a long time scale. The fast loop manages the slow loop.

17.6.5 Complex components should be sandboxed

In the human respiratory system, there is a conscious control of the breathing apparatus. This has the advantage that all the power of conscious reasoning can be brought to bear in the case of catastrophes. For example, if the person is in a car that falls into a river, the conscious control can stop breathing temporarily until the person is outside of the car. Conscious control is also dangerous, however: it can introduce instability. If the person decides to stop breathing (for example, because of a wager), then the system must somehow defend itself. This can be done by having an outer loop observe the conscious control. In the case of the human respiratory system, the brain falls unconscious if the blood oxygen level drops too low. When this happens, the conscious control disappears and the breathing apparatus starts working normally again. The general rule is that complex components can improve the power of the system (for example, they can stabilize an unstable system, like a pilot who stabilizes an unstable airplane) at the price of possibly introducing instability at other occasions. They must therefore always be observed by an outer loop that can take action when this happens.

With respect to stability, there is no essential difference between human components and programmed complex components; both can introduce stability and instability. Human components excel in adaptability (dynamic creation of new feedback loops) and pattern matching (recognizing new situations as variations of old ones). They are poor whenever a large amount of

precise calculation is needed. Programmed components can easily go beyond human intelligence in such areas. Whether or not a component can pass a Turing test is irrelevant for the purposes of self management.

17.6.6 Use push-pull to improve regulation

Many systems are regulated by simple negative feedback. When a parameter becomes too high, there is a reaction to reduce its value. The reaction is a initiated by a second parameter, which is a regulator. The effectiveness of this kind of regulation can be improved by making it "push-pull", that is, by having two regulators, one which actively increases the parameter and one which actively decreases it. In this way, the parameter can be changed quickly in both directions ("pushed" and "pulled").

We give two examples from biology [84]. In the first example, the glucose level in the blood stream is regulated by the hormones glucagon and insulin. In the pancreas, A cells secrete glucagon and B cells secrete insulin. An increase in the blood glucose level causes a decrease in the glucagon concentration and an increase in the insulin concentration, and conversely. These hormones act on the liver, which releases glucose in the blood. The second example is the calcium level in the blood, which is regulated by parathyroid hormone (parathormone) and calcitonine, both of which act on the bone but in opposite directions. The pattern here is of two hormones that act in opposite directions on the regulated substance. This allows improved regulation: quicker changes in the substance's concentration and faster convergence. This pattern is a generally useful one to improve control.

17.6.7 Handle failures with reversible phase transitions

The basic idea is that a system controlled by feedback loops may have several macroscopic (global) states, similar to "phases" in thermodynamics. If the system is exposed to a hostile environment, it may change its global state. In order for the system to survive such changes, they should be reversible. This is explained in detail in Section 17.3.3.

17.7 Overall design of a self-managing system

We now focus on the overall design of a self-managing system: how it is organized as a set of feedback structures. There are two main steps in determining the structure of a self-managing system: *decomposition* and *orchestration*. We first explain what each step is and then we give examples to

show how it is done.

This overall design technique is taken from [6]. In that article, there are two other steps: *assignment* (of tasks to autonomic managers) and *mapping* (of autonomic managers to nodes in the distributed environment). These two steps are not relevant for SELFMAN since the autonomic managers in SELFMAN are not separate entities in the system, but rather feedback loop structures that exist in distributed fashion.

17.7.1 Decomposition: defining the management tasks

The first step is to perform a decomposition, i.e., divide the management into separate tasks. Each task will be performed by a single manager, where a manager corresponds to a feedback loop structure. For example, in the distributed store, we can distinguish connectivity, routing, replicated storage, and transactions. Each of these is done by a different feedback structure. Connectivity is done through ring maintenance. Routing is done through finger table maintenance. And so forth for the other tasks.

To bring clarity into the decomposition, we can divide the tasks into five general areas: functionality and the four non-functional areas of self tuning, self protection, self configuration, and self healing. In the four tasks mentioned before, we distinguish the following non-functional areas:

- Connectivity management is self healing (correctness is the issue) (deliverable D4.2b, see chapter 7).
- Routing is self tuning (performance is the issue) (deliverable D4.3b, see chapter 9).
- Replicated storage is self healing (creating a new replica when one fails) and self configuration (coherent updating of the replicas) (deliverable D4.2b in chapter 7 and deliverable D4.1b in chapter 5).
- Transaction management is self healing (Paxos uniform consensus algorithm for coherent concurrency control [61]).

Each of these four tasks is studied in detail in the corresponding deliverable or paper.

17.7.2 Orchestration: handling the interactions

The second step is to perform the orchestration. This consists of handling the interaction between the management tasks. Each management task is done by a manager, which is a self-contained feedback loop structure that

maintains itself and pursues its own goals. But because all these tasks affect parts of the same system, there can be interactions between them. The possible interactions must be carefully studied and each management task must be written to take them into account.

We find that orchestration is the most challenging part of building a self-managing application. Ideally, the self-management tasks will be designed to have minimal interaction. But some interaction always exists, therefore some coordination is always necessary. For example, the finger table maintenance has to take the connectivity management into account. The replicated storage has to take the routing into account. If a node fails, then the replicated storage has to create a new replica, but taking the routing into account which must also be repaired. We give some rules of thumb to help the designer approach this ideal goal. One rule of thumb is to program each task using a monotonic function, which always increases as the task is performed. In that way, each task can be done with as little interference as possible from other tasks.

17.7.3 Forms of interaction

It is important to understand *how* managers can interact [5]. We identify three possible ways that interaction can happen:

- *Stigmergy*: This is the most ubiquitous and hardest to control. Stigmergy happens because managers make changes to a shared subsystem. Each change made by a manager may be sensed by another manager. Stigmergy is unavoidable. We address it in two ways: by using it to aid coordination when possible and when this is not possible by minimizing its ill effects.

- *Hierarchical management*: This occurs when a manager directly controls another manager. This situation occurs inside a feedback loop structure, when an outer loop controls an inner loop. For example, this situation occurs in the TCP structure or in the human respiratory system. We will not address this further as part of orchestration, but as part of the design of a single feedback loop structure.

- *Direct interaction*: This occurs when two managers interact directly with one another. It does not mean that a manager controls the other, but one manager may request something from another. We also call this peer-to-peer interaction since the managers are peers (not client and server). Direct interaction is sometimes needed since two independent loops managing the same resource may cause undesired behavior. It

must be handled carefully to avoid oscillation or other undesirable behavior. In our case, we handle this by giving each manager a monotonic function with a limiting value that corresponds to perfect behavior. If each manager increases its own function in discrete steps greater than some existing minimal increment then there will be no oscillation.

17.7.4 Examples of interaction

We give some examples of how managers work and interact. These examples are taken from storage management architectures developed in both the SELFMAN project and from other work (in particular, from the GRID4ALL project).

Replica management

This manager is responsible for maintaining the desired replication degree for each stored object in spite of nodes failing and leaving. One possible implementation is as follows. The manager consists of two agents, a replica aggregator and a replica manager. The aggregator subscribes to fail and leave events caused by any object's node. The manager then performs the restoration by creating a new replica.

Storage management

This manager is responsible for maintaining the total storage capacity and total free space of storage, in the presence of dynamism, to meet QoS (tuning) requirements. The dynamism comes from nodes failing or leaving (affecting both total and free storage space) or objects being created or deleted (affecting free storage space). The manager will reconfigure the total free space and/or the total storage space to meet the requirements. The reconfiguration is done by allocating free nodes and deploying additional storage components.

Direct interaction between replica and storage management

There is a race condition between the two above managers. If a node fails, the storage manager may start allocating more nodes and deploying components. Meanwhile the replica manager will be restoring the objects on the failed node. The replica manager might fail to restore the files due to space shortage if the storage manager is slower and does not have time to finish. This may prevent the users, temporarily, from creating objects.

The solution is to let the replica manager wait until the storage manager has completed its work. This can be done through direct interaction between the two managers. This direct interaction does not mean that one manager controls the other. For example, if there is only one replica available for an object, the replica manager may ignore the request to wait from the storage manager and proceed anyway.

Interaction between the replica and storage managers gives an example of stigmergy. When the utilisation of storage on the nodes drops, the storage manager can deallocate some nodes. This deallocation resembles a node failure at the replica manager, which will then restore replicas on other nodes. Because of stigmergy, the replica manager has done the right thing.

Availability of an object can be increased by changing the replication degree. This can be done by an "availability manager" that monitors the frequency of access to the object. If the frequency increases, the availability manager can decide to change the object's replication degree. Before doing this, the replica manager checks with the storage manager to see whether there is enough storage for the new replicas. In this example, the managers collaborate. It is not so that one manager controls another.

17.8 Design rules for the self-management axes

Now that we have explained how to build feedback structures and how to organize them into a self-managing system, we narrow our focus to the four main main self-management axes: self tuning, self protection, self healing, and self configuration. We illustrate the principles in the previous sections with examples taken from the SELFMAN project and from the literature on self management and feedback. We assume that all examples use the right programming model, i.e., concurrent components with asynchronous messages and fault streams instead of distributed exceptions. The examples given have been implemented in different systems, mainly Kompics, Erlang, or Oz, which all provide this basic programming model, with variations.

17.8.1 Making it self-tuning

We outline a load-balancing algorithm for structured overlay networks with good properties and explain how it interacts with replica management. See [39] for a precise definition of the algorithm and an evaluation of its behavior. The load-balancing algorithm is decentralized, i.e., each node acts independently depending on knowledgte at the node. Each node selects a set of nodes with which to balance. A balancing operation consists of an under-

loaded node leaving the network and rejoining at its new location. When the node leaves it first moves its data items to adjacent nodes. When it joins it takes part of the load of adjacent nodes. It takes a maximum of the average global load at its new location. This heuristic reduces the number of data items that are moved unnecessarily. The algorithm uses gossip to maintain an approximation to the average global load.

There is a dependency between the load balancing and the replica management since each node is responsible for a set of items depending on its position in the overlay. If a node decides to balance (leave and rejoin), then the replica manager must recreate the data according to the changes in responsibility. So there is a trade-off between how often to do load-balancing steps (this affects the convergence towards the balanced state) and the cost of replica maintenance. The rate of load balancing needed is mainly dictated by the churn.

To summarize, the replica manager and the load-balancing algorithm are independent of each other, but their parameters must be tuned to reduce impact on the system stability.

17.8.2 Making it self-protecting

Depends on threat model. We propose an approach based on three successive steps, for successively higher levels of security. Each step can work only if the previous step has been successfully implemented. SELFMAN has done experiments in these three steps but we have not built a full security architecture. We can give advice but we do not have user experience.

1. Assume nodes are trusted and network is not. After this step, we can forget about the network and think only about the nodes/users. Node authentication is performed here.

2. Handle non-collusion (nodes do not communicate with each other directly) We distinguish two cases: first where the nodes each use a service independently, and second where the nodes depend on each other. We explain each of the two cases below. The first case is the simplest and can be handled with local techniques. The second case can be handled with collective intelligence techniques.

3. Handle collusion (malicious nodes can communicate with each other directly) This is the most difficult case. In the case of structured overlay networks, changing the topology of the network to become a small-world network can help.

No collusion, with independent nodes

If there is no collusion in a system in which nodes each use a service independently, then end-to-end techniques and trust component techniques suffice. End-to-end security assumes that the nodes are trusted and the network is not. Communication between nodes is encrypted. Then, to assure the trusted nodes are correct, authentication is used to verify the integrity of installed components at the user nodes.

The Scalaris library is designed for working in a data center, which is implicitly considered secure. The Peerialism product is a closed network infrastructure with end-to-end security. So for both of these software products, no security mechanisms are identified. But this does not mean that security is irrelevant. For Scalaris, we have looked at Wikipedia-specific issues, which are relevant to applications using Scalaris. In Wikipedia there can be anonymous edits. Information may not be trustworthy. There may be spam. The question is how to increase the credibility of the content. One technique is to use citations. But this does not directly address credibility, since the cited documents may not be easily accessible or applicable. In the Google Knol application, anonymous edits are not possible. Edits are signed with Google accounts. Edits are signed and therefore put the author's reputation on the line. To enhance Wikipedia credibility (and any application using storage tools such as Scalaris), we propose to use author credentials which are attached to the edits and verified using a secure mechanism.

No collusion, with dependent nodes

It is possible for a system to have no collusion, but the global performance can still depend on the nodes collaborating in some way. A promising technique to achieve this is collective intelligence, which can give good results when the users are independent (no Sybil attacks or collusion). The basic question is how to get selfish agents to work together for the common good. Let us define the problem more precisely. We have a system that is used by a set of agents. The system (called a "collective" in this context) has a global utility function that measures its overall performance. The agents are selfish: each has a private utility function that it tries to maximize. The system's designers define the reward (the increment in its private utility) given to each of the agent's actions. The agents choose their actions freely within the system. The goal is that agents acting to maximize their private utilities should also maximize the global utility. There is no other mechanism to force cooperation. This is in fact how society is organized. For example, employees act to maximize their salaries and work satisfaction and this should benefit

the company.

A well-known example of collective intelligence is the El Farol bar problem [10], which we briefly summarize. People go to El Farol once a week to have fun. Each person picks which night to attend the bar. If the bar is too crowded or too empty it is no fun. Otherwise, they have fun (receive a reward). Each person makes one decision per week. All they know is last week's attendance. In the idealized problem, people do not interact to make their decision, i.e., it is a case of pure stigmergy! What strategy should each person use to maximize his/her fun? We want to avoid a "Tragedy of the Commons" situation where maximizing private utilities causes a minimization of the global utility [37].

We give the solution according to the theory of collective intelligence. Assume we define the global utility G as follows:

$$G = \sum_{w} W(w) \tag{17.1}$$

$$W(w) = \sum_{d} \Phi_d(a_d) \tag{17.2}$$

This sums the week utility $W(w)$ over all weeks w. The week utility $W(w)$ is the sum of the day utilities $\Phi_d(a_d)$ for each weekday d where the attendance a_d is the total number of people attending the bar that day. The system designer picks the function $\Phi_d(y) = a_d y e^{-y/c}$. This function is small when y is too low or too high and has a maximum in between. Now that we know the global utility, we need to determine the agents reward function. This is what the agent receives from the system for its choice of weekday. We assume that each agent will try to maximize its reward. For example, [91] assumes that each agent uses a learning algorithm where it picks a night randomly according to a Boltzmann distribution distributed according to the energies in a 7-vector. When it gets its reward, it updates the 7-vector accordingly. Real agents may use other algorithms; this one was picked to make it possible to simulate the problem.

How do we design the agents reward function $R(w)$, i.e., the reward that the agent is given each week? There are many bad reward functions. For example, Uniform Division divides $\Phi_d(y)$ uniformly among all ay agents present on day y. This one is particularly bad: it causes the global utility to be minimized. One reward that works surprisingly well is called Wonderful Life:

$$R_{WL}(w) = W(w) - W_{\text{agent absent}}(w) \tag{17.3}$$

$W_{\text{agent absent}}(w)$ is calculated in the same way as $W(w)$ but assuming the agent is missing (dropped from the attendance vector). We can say that

$R_{WL}(w)$ is the difference that the agent's existence makes, hence the name Wonderful Life taken from the eponymous the Frank Capra movie. We can show that if each agent maximizes its reward $R_{WL}(w)$, the global utility will also be maximized. Let us see how we can use this idea for building collective services. We assume that agents try to maximize their rewards. For each action performed by an agent, the system calculates the reward. The system is built using security techniques such as encrypted communication so that the agent cannot "hack" its reward.

Collusion

The approach of the previous sections does not work when there is collusion, i.e., when many agents get together to try to break the system. For collusion, one solution is to have a monitor that detects suspicious behavior and ejects colluding users from the system. This monitor is analogous to the SEC (Securities and Exchange Commission) which regulates and polices financial markets in the United States. Collective intelligence can still be useful as a base mechanism. In some cases, the default behavior is that the agents cannot or will not talk to each other, since they do not know each other or are competing. Collective intelligence is one way to get them to cooperate.

Collusion and the topology of the overlay network

One kind of collusion is the Sybil attack, where an attacker can assume identities of many users and gain a large influence on the system. SONs suffer from the Sybil attack. They are vulnerable to attacks that exploit churn and network failures. If Sybil attacks are to be expected, then the topology of the overlay must be changed. We propose to use a small-world network topology (SWN), which has a natural protection against Sybil attacks. It has low maintenance cost and exploits random graph properties that make it hard to disconnect. It is immune to a churn attack. Routing can be done well assuming that node IDs follow a power law distribution.

The SWN routing has a vulnerability in the self-tuning algorithm. A single malicious node can poison the entire network (the poison is propagated through acquaintances) causing the self-tuning algorithm to break down, since node IDs become incorrect. A decentralized self-protection algorithm can help to contain the attack (contain the infection rate to 10%). The idea is that each node resets its ID periodically, which corrects erroneous node IDs.

17.8.3 Making it self-healing

In SELFMAN we have three main examples of self-healing design:

- Replication of the storage over a structured overlay network. This example uses redundancy (each stored item exists multiple times) to allow self healing. Replication is used for keeping the storage intact (one replica always must exist for this to be true) and for the transaction algorithm (the Paxos algorithm requires a majority of replicas to exist to allow a commit).

- Relaxing the structural invariant of a structured overlay network. This is used in the relaxed ring underlying the Beernet library. The relaxed ring handles failure suspicions by using a two-level structure: there is a perfect ring at the center and there may be "bushes" sticking out of it because of failure suspicions. The bushes are either merged back into the perfect ring (when the suspicion goes away) or ejected from the relaxed ring (if the suspicion persists). A node may be incorrectly ejected if there are too many false failure suspicions; in that case the node will simply ask to reconnect with the ring. The connectivity manager is driven by the size of the bushes; as long as there is at least one node in one bush, it will be active.

- Make phase transitions reversible. A phase transition happens in the case of a network partition. The ring may be split into several smaller rings because of network partitions. If node communication in the smaller rings is restored, then the merge algorithm is activated to merge the separate rings into one ring [77].

Each of these three examples uses a different technique to achieve self healing: replication, relaxation of an invariant, and reversibility of a phase transition.

17.8.4 Making it self-configuring

To support self configuration, the system has to be built with components that provide the appropriate reflective primitives. In SELFMAN we have explored these primitives in two models: the Kompics component model and the more extensive WorkflOz/FructOz/LactOz libraries. It is interesting to look at both. Kompics provides primitives for monitoring and reconfiguration; it is sufficient for many basic self-configuration tasks as well as other tasks such as self healing (detection of failures and reconfiguration). The WorkflOz/FructOz/LactOz libraries provide more extensive abilities: they

support more sophisticated self-configuration structures. The basic operations needed for self configuration are the following:

- *Observation and navigation.* The LactOz library allows dynamic navigating and monitoring of the component structure.
- *Reconfiguration.* The FructOz library allows dynamic deploying and configuring of the component structure.
- *Workflow.* The WorkflOz library was designed in Year 3. It allows to create dependencies between different managers through which information can flow. This can simplify the system structure since managers are informed in a more uniform way of important events in the system.

17.9 Conclusions

We have given a first approximation of a methodology to build self-managing systems. We start by proposing a general architecture (Sections 17.4 and 17.5), with examples from Kompics, Scalaris, and Beernet. This architecture supports feedback structures, which are the basic design element in a self-managing system: a feedback structure is a set of interacting feedback loops. Section 17.6 gives design rules for building feedback structures. Section 17.7 then explains how to design the overall structure of a self-managing system, which consists of self-management managers implemented with feedback structures. We determine this structure by means of two techniques called decomposition and orchestration. Finally, Section 17.8 targets these ideas to the four main self-management axes. All these sections take examples from SELFMAN and from the literature and reference the appropriate SELFMAN publications.

17.9.1 Future work

SELFMAN has made some progress in understanding how to build self-managing systems but much remains to do. We have encountered several important unsolved problems that are outside the scope of the project. Specifically, here are two:

- To create a complete methodology for *program design with feedback structures* that is usable by practicing programmers. This means it should be at a similar abstraction level as existing methodologies, e.g.,

for object-oriented programming. Two important parts of this methodology are *feedback patterns* and *design rules*. In SELFMAN we have explored a few patterns and rules in the context of building self-managing applications, but we have only begun to formulate a methodology.

- To study how to build applications that can undergo *reversible phase transitions*. This is essential for building long-lived applications that can survive in realistically hostile situations. Any future research direction in fault tolerance must take phase transitions into account. In SELFMAN we have explored one case of reversible phase transitions, namely the effect of network partitions on structured overlay networks and the use of a merge algorithm to make the partition reversible.

These problems should be addressed in future research.

Chapter 18

D5.8: Self-managing distributed collaborative drawing tool on mobile devices

The development of a drawing tool for Android phones based on P2PS/Beernet [66, 54] is the result of researches on mobile devices. We introduce the first application running on mozart for Android.

18.1 Executive summary

DeTransDraw for Android runs as a drawing observer of a ring of P2PS/Beernet nodes. The first important step to this result was to port mozart on the device. Mozart runs on top of the unix kernel below the Java API. In order to create the user interface, we have implemented a bridge between mozart and this API. DeTransDraw is an application with two different parts, the computer application running as a peer is features complete and the mobile part which only observes the action. Both applications are implemented in OZ language [62].

18.2 Contractors contributing to the Deliverable

UCL(P1)has contributed to this deliverable.

UCL(P1) have ported mozart on the Android operating system, added an eager locking mechanism to P2PS/Beernet and developed the DeTransDraw applications.

18.3 Introduction

This software deliverable aims to be a real application of the concepts introduced in Selfman and a first appearance on mobile phones. With their small size, they are highly mobile and portable. But they have smaller memory capacities and slower processors than computers.

In a ring, they are not stable peers because they can enter and leave it quickly. It is a good test of the scalibity of the ring.

DeTransDraw is a decentralized collaborative vector-based graphical editor with a shared drawing area. It provides synchronous collaboration between users with graphical support for notifications about other users' activities. Conflict resolution is achieved with a decentralized transactional service with storage replication, and self-management replication for fault-tolerance. The transactional service also allows the application to prevent performance degradation due to network latency, which is an important feature for synchronous collaboration.

18.4 Specification

DeTransDraw is intended to be a drawing tool that allows multiple users to collaborate on creating and editing a vector image. Each user may join the network by creating a peer and joining the ring or by connecting to an existing peer of the network.

There are two kinds of users. The normal users can draw forms, edit the image and delete objects. While mobile users can only observe the system.

The application uses eager transactions to synchronize the modifications. This means that when a user try to edit some part of the image, it asks for locking this part. When the user have finished his work, he may commit his work and release the lock.

The transactions allow the user to ensure fast reaction while keeping coherence. On the state diagram of the Figure 18.1, the user have three distinct states. *No lock* when he is not editing the image though he can draw and observe. When he locks objects, he is in a new state waiting for an event such as *abort* or *locked*. After he has received all the events, he will be in the state *Got locks* if he got some locks or *No lock* if he got none. When all the locks are released, the user is back in the state *No lock*.

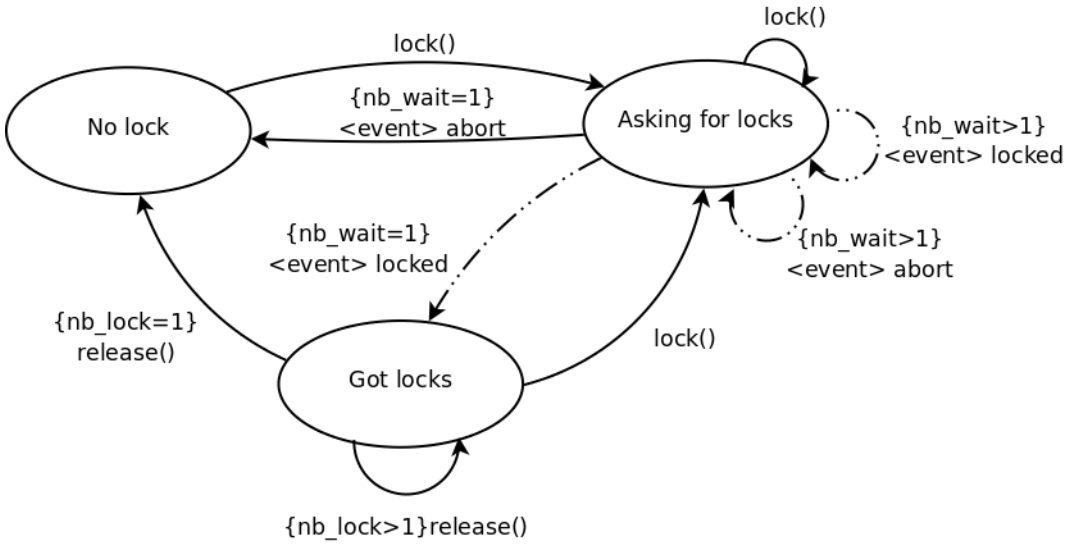


Figure 18.1: State diagram of a user

As the modification is made locally before comitting, the responsiveness is the same as a non-distributed drawing application. In all these states, the user can edit the image. In *No lock*, he is not editing the image but he can

ask for locks.

In *Asking for lock*, he can edit but is not sure that his changes will not be lost. He may lose the changes on the objects to which he will get an abort event. In *Get lock*, he can edit and he is sure that his changes will be saved. The coherence is maintained because of the locks on objects. An example of visual feedback of the difference between the two last states is in Figure 18.2

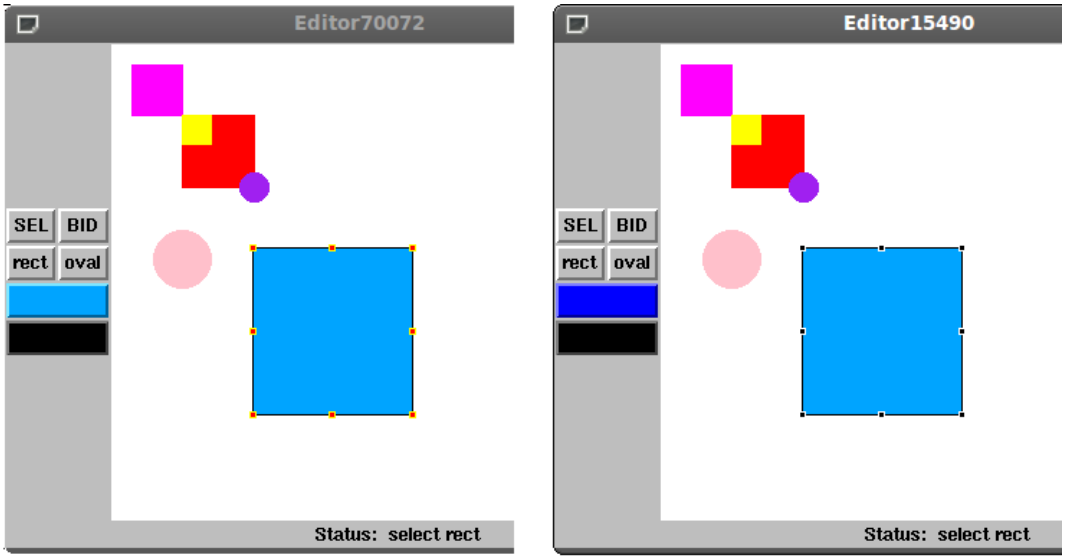


Figure 18.2: On the left, the user is in *Asking for locks* state. On the right, the user is in *Got locks* mode.

In ordrer to add or remove an object on the image, the user does not need the lock eagerly. This would lock the complete image and prevent users from adding or removing objects which is not necessary. The transaction will ask for the lock just before committing his changes. If it is aborted, the transaction is repeated until it is comitted.

18.5 Architecture

DeTransDraw is a two-part application. A computer part is a feature complete application for drawing while an android part is only observing what is happening on the system.

18.5.1 DeTransDraw for computers

DeTransDraw for computer is a drawing tool with the following features :

- draw objects such as rectangles, squares, ovals and circles
- change the colour that fill the object
- change the colour of the border of the object
- delete objects
- move objects
- collaborative drawing by joining a ring
- collaborative drawing by connecting to a peer of a ring

DeTransDraw offers a simple user interface to create a ring of P2PS/Beernet nodes. The Figure 18.3 is the manager window for the node 21797. Two buttons named *Save:* and *Save* allow the user to save the ticket, while *Load:* and *Join* help to connect to some already saved tickets. A ticket is a file used by DeTransDraw to connect to a ring. When the ring is ready, the editor window may be opened thanks to *Open editor* button.

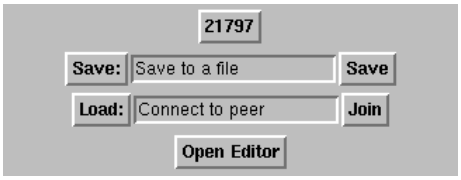


Figure 18.3: User interface for managing the peer

In the Figure 18.4, it is the drawing area. The area is divided in three parts : the toolbar, the drawing part and the status bar. Button *SEL* stands for the selection of an object or multiple object with *Shift* key pressed. The buttons *rect* and *oval* allows the user to draw rectangles and ovals. The two

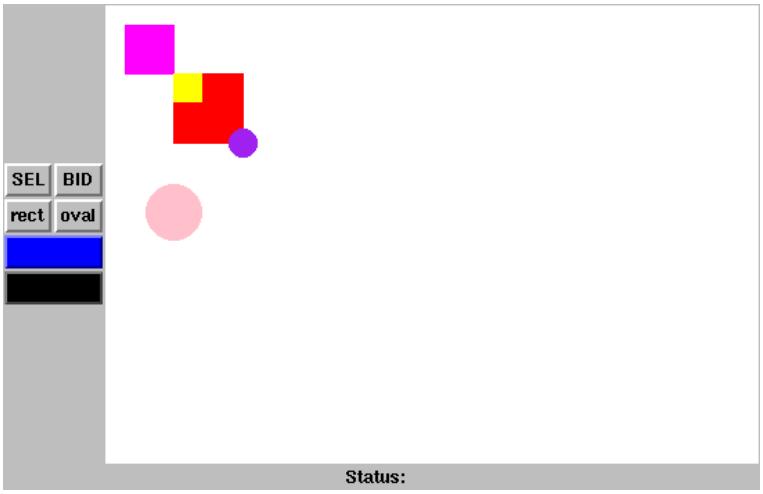


Figure 18.4: Example of objects

colored buttons represent the color of the object and its border. The status bar notifies the user of the action he is currently doing.

Each object drawn has a well defined record such as :

val(id:ID xb yb xe ye fill outline form)

Where *ID* is used as a key in the transaction manager where the value is the record itself. The value of another key, *dt*, is the list of all the keys stored in the transaction manager. It represents all the objects of the application.

In selection mode, the user is able to select either rectangles or ovals. A selected object appears with a virtual bounded box as in Figure 18.5. This box has nine dots. One for each vertices and center of edges.

As it appears in the Figure 18.6, the dots may be filled in black or red. The colour depends on the status of the lock for the object. Here, the user on the left try to select the cyan rectangle which is already selected by the user of the right window. When an object is selected, the node tries to lock it.

The locking mechanism is different than in usual transactions systems. The application asks for a lock before trying to commit any change. We have implemented this eager locking mechanism in P2PS/Beernet 2.3.4.

While trying to get the lock, the dots are in red meaning that the lock may not be accepted. The user on the left will see the object unselected and back to the original position as soon as the lock is refused. The user on the right has got the lock and the dots are black.

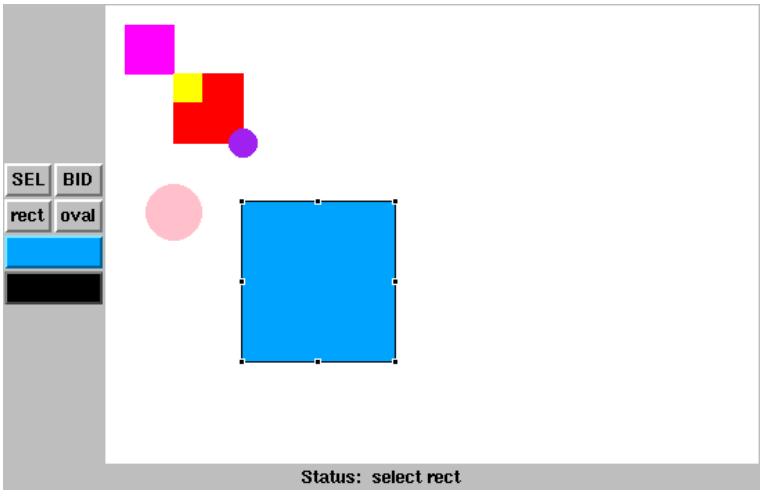


Figure 18.5: Selection of an object

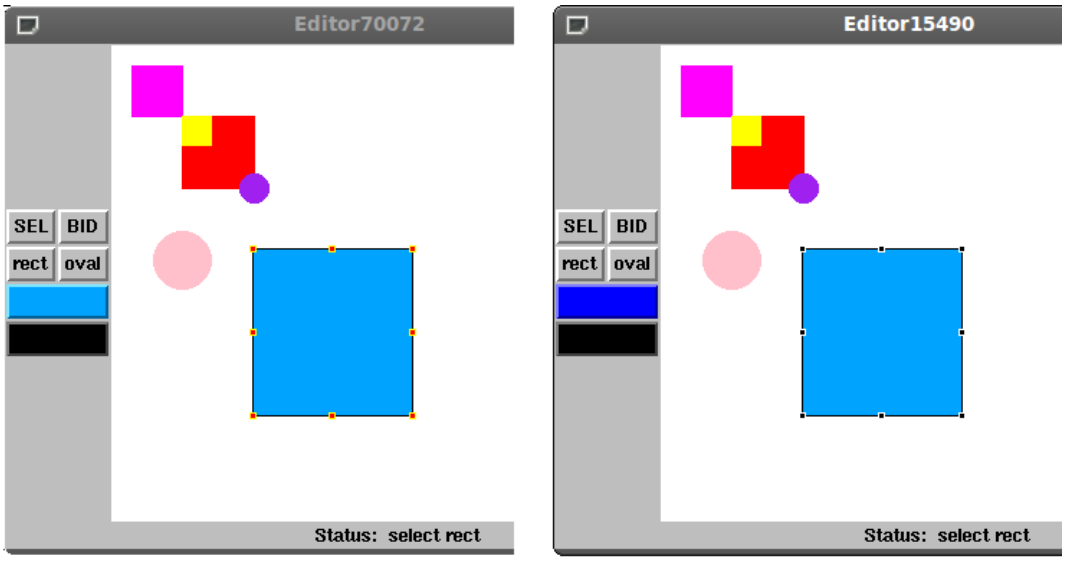


Figure 18.6: Selection of an object already locked

18.5.2 DeTransDrawid

DeTransDrawid is the mobile part of DeTransDraw. It runs on mobile phones with Android as operating system. We have implemented a ligher version of DeTransDraw for mobile phones, it allows users to observe what's going on the shared drawing area.

An application for Android is quite different from an application on a computer. Each application runs as a different user and extends the class *Activity*. The activity allows Android to tell the application when she is created, paused, resumed, and some other events. The application has to react to these events. In order to create an application in Mozart environment, there events are sent from Java side to the Mozart side.

As in Figure 18.7, there are two parts of DeTransDrawid running in a different level. The functional part is implemented in Oz language on the Unix kernel while the activity part is implemented in Java on the API provided by the Software Development Kit.

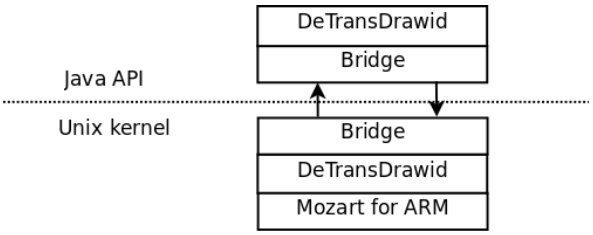


Figure 18.7: Structure of the application for Android

All the screenshots used in this section are taken from the emulator of the software developer kit. It allows better visibility than pictures of the mobile phone.

The Figure 18.8 illustrate the android launcher. Touching the icon *DeTransdraw* starts DeTransDrawid. The result is a black screen as in Figure 18.9 waiting for a ring.

When the ring has started drawing, the whole screen is filled by the drawing area as it appears in Figure 18.10.

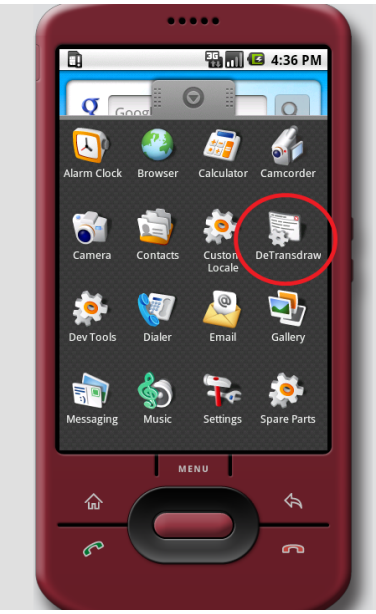


Figure 18.8: DeTransDrawid in application launcher

Figure 18.9: Application is starting

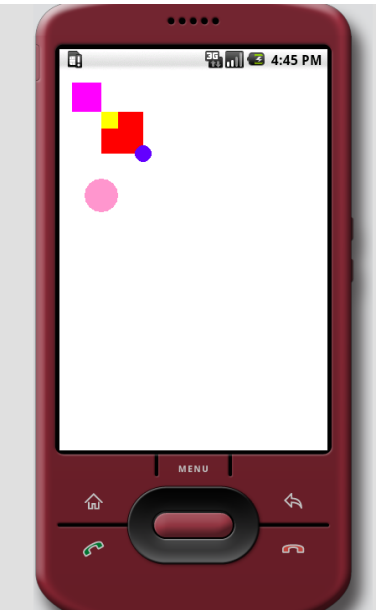


Figure 18.10: Example of objects drawn with DeTransDraw and displayed by DeTransDrawid

18.6 Implementation

18.6.1 DeTransDraw

Requirements : Mozart-trunk revision 17242 or newer

Installation : Download DeTransDraw binaries and extract it.

Run in real mode : Each user should execute DT.exe to get the manager window. If there are more than ten users, one of the user may save a ticket and send it to other users which will load it. A ring of nodes without drawing features may be created with the Bootstrap binary with option *dss*.

Run in simulation mode : First create the ring with Bootstrap binary with option *sim*. Each user can join the ring loading a ticket of this ring or connect with this command :

./DT.exe sim URL

Where *URL* is the location of a ticket of this ring.

18.6.2 DeTransDrawid

Requirements :

- Android 1.5 SDK or newer
- Eclipse and plugin for Android 1.5 SDK

Installation :

- Download DeTransDrawid sources
- Unzip DeTransDrawid
- Create a java project from existing sources (javaaccess directory from zip) and named it javaaccess
- Create an android project from existing sources (DeTransDraw directory from zip)
- Create and Android Virtual Device (AVD) or use a device with Android 1.5 or newer

Implementation :

```
public class Sample extends Activity {
    private Handler h;
    private GenericApplication app;
    private Protocol protocol;
    private String handlerKey;


    public Sample() {
        super();
        handlerKey="mainActivity";
    }
    private Sample(String s){
        super();
        handlerKey=s;
    }
    public static Handler create(String handlerKey){
        Sample a=new Sample(handlerKey);
        return a.h;
    }
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        app=(GenericApplication)getApplication();
        protocol=app.protocol;
        h = new Handler(protocol, Target.next());
        h.set(this);
        app.ozRegisterHandle(handlerKey, this, Activity.class, h.getTarget());
        super.onCreate(savedInstanceState);
    }
```

Figure 18.11: Minimum of code for the activity class

In Figure 18.11, we see the minimum of code for running an application controled by mozart bridge. A handle is created to allow mozart to know when events happens.

When we want to notify the mozart part, we have to invoke such as in Figure 18.12. This sample code notify that the java method *onStart()* has been invoked in java part. The mozart part might react to this event.

```
@Override
public void onStart() {
    super.onStart();
    try {
        h.invoke(this, GenericActivity.class.getMethod(
                "onStart",
                new Class[] {}),
                new Object[] {});
    } catch (Throwable e) {
        throw new RuntimeException(e);
    }
}
```

Figure 18.12: Sample of code to invoke mozart

In order to get messages, we need to implement a function MainActivity in a Main.oz file such as provided in DeTransDraw project. This function gets a message when invoke in java part.

```
fun{MainActivity Msg}
   case Msg
   of onCreate(!BundleC# _)
      {{{J.c 'java.lang.System'}.get err}.p println(string("Hello World/Activity."#{Label Msg}))}
      void(x)
   else
      void(x)
   end                % End of case
end                   % End of MainActivity
```

Figure 18.13: Main function to receive messages from Java.

18.7 Future work

DeTransDrawid is a prototype of a Mozart application running on Android. The application is separated in two parts which is not so easy to use.

For a real application, it would be better to have weak coupling between the two sides. The idea is that a Java application will be silently started and will be used by mozart applications. This application will only display GUIs. Mozart applications will be independent from the Android application.

QTk for Android has to be completed and fully documented. Creating an android application with Mozart should be as easy as a normal Mozart application.

18.8 Papers and publications

A Toolkit for Peer-to-Peer Distributed User Interfaces: Conceps, Implementation, and Applications

Jérémie Melchior, Donatien Grolaux, Jean Vanderdonckt and Peter Van Roy. ACM SIGCHI symposium on engineering interactive computing systems. July 2009 (see Appendix A.21).

This paper presents a software toolkit for deploying peer-to-peer distributed graphical user interfaces across several dimensions. The toolkit supports several platforms from desktop to mobile in order to develop collaborative applications.

Decentralized transactional collaborative drawing

Jérémie Melchior, Boris Mejías, Yves Jaradin, Peter Van Roy, Jean Vanderdonckt. Submitted to COPS'09 (see Appendix A.22).

This paper proposes a decentralized architecture based on a peer-to-peer network providing decentralized transactional support with replicated storage. As a consequence, there is a gain in fault-tolerance and the transactional protocol eliminates the problem of network delay improving usability and network transparency. The same result can be applied to text edition and other collaborative editing tasks.

Decentralized transactional collaborative drawing - Demo

Boris Mejías, Jérémie Melchior and Yves Jaradin. Demonstrator at Collaboration Meeting for FP6 and FP7 projects. (see Appendix A.23).

This is a description of the demonstration we have presented of DeTransDraw in Internet of Services 2009 Collaboration Meeting for FP6 and FP7 projects.

Chapter 19

D6.5c: Final progress and assessment report with lessons learned

19.1 Executive summary

The final year of SELFMAN has given us many results and insights into how to construct self-managing applications built on a structured overlay network. All this content can be found in the other deliverables. Instead of repeating it here, we explain where to find it. We then conclude by commenting on the general characteristics of SELFMAN with the hindsight of three years of work. This gives us some lessons for the structure of future projects.

19.2 Contractors contributing to the Deliverable

This deliverable was written by UCL(P1) (Peter Van Roy) based on results from the other partners.

19.3 Results

19.3.1 Lessons learned

The progress and assessment of the project can be found in the following deliverables:

- **D4.1b** (chapter 5): Second report on self-configuration support.
- **D4.2b** (chapter 7): Second report on self-healing support.
- **D4.3b** (chapter 9): Second report on self-tuning support.
- **D5.6** (chapter 16): Evaluation of security mechanisms.

These four deliverables focus on specific self-* properties.

- **D5.4a** (chapter 14): Qualitative evaluation of autonomic features of SELFMAN applications.
- **D5.4b** (chapter 15): Quantitative evaluation of autonomic features of SELFMAN applications.

These two deliverables apply a general evaluation methodology for autonomic behavior to the self-* properties.

- **D5.7** (chapter 17): Guidelines for building self-managing applications.

This deliverable summarizes the lessons learned and experience from all the self-* properties into a first approximation to a complete methodology for building self-managing applications.

19.3.2 SELFMAN and the problem of parallel research

Looking back at SELFMAN after three years, we can make some general reflections on the project's structure and its results:

- An interesting characteristic of SELFMAN is the system-level holistic approach, in which we do not concentrate on just one self-* property, but attempt to combine all four of the main ones. This has worked out well for self configuration, self healing, and self tuning, but has proved more difficult for self protection.

- Another characteristic of SELFMAN is that we have done several implementations of our transactional store: one in Erlang, one in Oz, and we are completing one in Java using the Kompics component model. This has allowed us to strengthen our focus on the right kind of system architecture, namely loosely coupled components with asynchronous communication. By using variations of this architecture, we get a better idea of what is essential and what is not.

- SELFMAN has worked on several parallel research tracks, especially the following four: configuration tools (WorkflOz/FructOz/LactOz), component model (Kompics), how to implement and combine self-* properties, and the transactional store and demo applications (Scalaris/Beernet). These tracks have communicated with one another, but because each one was itself a research issue, they could not take full advantage of one another. It would be interesting to do another iteration, where each track starts again but with the insights and results of the other tracks at the beginning.

We conclude that a holistic approach (looking at characteristics together instead of piecemeal) that does several implementations can work well. One problem that cannot be avoided is how to make progress in different areas and at the same time combine this progress together (the problem of parallel research tracks). The parallel research problem is more or less serious depending on the particular system property that is studied. Some system properties are especially sensitive to this problem, in particular security (self protection). Our security work has shown that we need to change an initial assumption in the project (i.e., the topology of the structured overlay network).

But despite the occurrence of the parallel research problem in SELFMAN, we have made good progress in combining different self-* properties. Solving the parallel research problem for security would require another project. In our view this is not a defect of the project but an inherent property of security, since security depends crucially on initial assumptions. We conclude that the initial vision of the project, to build self-managing applications by combining structured overlay networks and component models, is basically sound.

We intend to build on the results of SELFMAN by submitting a successor project that uses the self-managing transactional store as just one part in the development of an architecture for a future Internet.

Appendix A

Publications

A.1 Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions

Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions

Peter Van Roy
Dept. of Computing Science and Engineering
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

Abstract

Programs are fragile for many reasons, including software errors, partial failures, and network problems. One way to make software more robust is to design it from the start as a set of interacting feedback loops. Studying and using feedback loops is an old idea that dates back at least to Norbert Wiener's work on Cybernetics. Up to now almost all work in this area has focused on how to optimize single feedback loops. We show that it is important to design software with multiple interacting feedback loops. We present examples taken from both biology and software to substantiate this. We are realizing these ideas in the SELFMAN project: extending structured overlay networks (a generalization of peer-to-peer networks) for large-scale distributed applications. Structured overlay networks are a good example of systems designed from the start with interacting feedback loops. Using ideas from physics, we postulate that these systems can potentially handle extremely hostile environments. If the system is properly designed, it will perform a reversible phase transition when the failure rate increases beyond a critical point. The structured overlay network will make a transition from a single connected ring to a set of disjoint rings and back again when the failure rate decreases. There is a complete research agenda based on the use of reversible phase transitions for building robust systems. In our current work we are exploring how to expose phase transitions to the application so that it can continue to provide a service. For validation we are building three realistic applications taken from industrial case studies, using a distributed transaction layer built on top of the overlay.

Keywords: software development, self management, feedback, distributed computing, distributed transaction, network partition, Internet, phase transition

1. INTRODUCTION

How can we build software systems that are not fragile? For example, we can exploit concurrency to build systems whose parts are mostly independent. Keeping parts as independent as possible is a necessary first step. But it is not sufficient: as systems become larger, their inherent fragility becomes more and more apparent. Software errors and partial failures become common, even frequent occurrences. Both of these problems can be made less severe by rigorous system design, but for fundamental reasons the problems will always remain. They must be addressed. One way to address them is to build systems as multiple interacting feedback loops. Each feedback loop continuously observes and corrects part of the system. As much as possible of the system should run inside feedback loops, to gain this robustness.

Building a system with feedback loops puts conditions on how it must be programmed. We find that message passing is a satisfactory model: the system is a set of concurrent component instances that communicate through asynchronous messages. Component instances may have internal state but there is no global shared state. Failures are detected at the component level. Using this model lets us reason about the feedback behavior. Similar models have been used by E for

building secure distributed systems [18] and by Erlang for building reliable telecommunications systems [1]. More reasons for justifying this model are given in [24]. For the rest of this paper, we will use this model.

Now that we can program systems with feedback loops, the next question is *how* should these systems be organized. A first rule is that systems should be organized as multiple interacting feedback loops. We find that this gives the simplest structure and makes it easier to reason about the system (see Sections 2 and 3). Single feedback loops can be analyzed using techniques specific to their operation; for example Hellerstein *et al* [10] gives a thorough course on how to use control theory to design and analyze systems with single feedback loops. The problem with systems consisting of multiple feedback loops is their global behavior: how can we understand it, predict it, and design for a desired behavior? We need to understand the issues before we can do a theoretical analysis or a simulation.

In the SELFMAN project [20], we are tackling the problem by starting from an area where there is already some understanding: structured overlay networks (SONs). These networks are an outgrowth of peer-to-peer systems. They provide two basic operations, communication and storage, in a scalable and guaranteed way over a large set of peer nodes (see Section 4). By giving the network a particular topology and by managing this topology well, the SON shows self-organizing properties: it can survive node failures, node leaves, and node joins while maintaining its specification. By using concepts and techniques taken from theoretical physics, we are able to understand in a deep way how SONs work and we can begin to understand how to design them to build robust systems. The concepts of feedback loop and phase transition play an important role in this understanding.

This paper is structured as follows:

- Section 2 defines what we mean by a feedback loop, explains how feedback loops can interact, and motivates why feedback loops are essential parts of any system. We briefly present the mean field approximation of physics and show how it uses feedback to explain the stability of ordinary matter.
- Section 3 gives two nontrivial examples of successful systems that consist of multiple interacting feedback loops: the human respiratory system and the Transmission Control Protocol.
- Section 4 summarizes our own work in this area. We are building a self-management architecture based on a structured overlay network. We conjecture that when designed to support reversible phase transitions, a SON can survive in extremely hostile environments. We support this conjecture by analytical work [14], system design [21], and by analogy from physics [15]. We are currently setting up an experimental framework to explore this conjecture. We target three large-scale distributed applications, built using a transactional service on top of a structured overlay network.

Section 5 concludes by recapitulating how feedback loops can overcome software fragility and why all software should be designed with feedback loops. An important lesson is that systems should be constructed so that they can do reversible phase transitions. Most existing fault-tolerant systems are *not* designed with this goal in mind, so they are broken in a fundamental sense. We explain what this means for structured overlay networks and we show how we have fixed them. We then explain what remains to be done: there is a complete research agenda on how to build robust systems based on the principle of reverse phase transitions.

2. FEEDBACK LOOPS ARE ESSENTIAL

2.1. Definition and history

In its general form, a feedback loop consists of four parts: an observer, a corrector, an actuator, and a subsystem. These parts are concurrent agents that interact by sending and receiving messages. The corrector contains an abstract model of the subsystem and a goal. The feedback loop runs continuously, observing the subsystem and applying corrections in order to approach

the goal. The abstract model should be correct in a formal sense (e.g., according to the semantics of abstract interpretation [5]) but there is no need for it to be complete.

An example of a software system that contains a feedback loop is a transaction manager. It manages system resources according to a goal, which can be optimistic or pessimistic concurrency control. The transaction manager contains a model of the system: it knows at all times which parts of the system have exclusive access to which resources. This model is not complete but it is correct.

In systems with more than one feedback loop, the loops can interact through two mechanisms: *stigmergy* (two loops acting on a shared subsystem) and *management* (one loop directly controlling another). Very little work has been done to explore how to design with interacting feedback loops. In realistic systems, however, interacting feedback loops are the norm.

Feedback loops were studied as a part of Norbert Wiener's cybernetics in the 1940's [29] and Ludwig von Bertalanffy's general system theory in the 1960's [3]. W. Ross Ashby's introductory textbook of 1956 is still worth reading today [2], as is Gerald M. Weinberg's textbook of 1975 explaining how to use system theory to improve general thinking processes [27]. System theory studies the concept of a *system*. We define a system recursively as a set of subsystems (component instances) connected together to form a coherent whole. Subsystems may be primitive or built from other subsystems. The main problem is to understand the relationship between the system and its subsystems, in order to predict a system's behavior and to design a system with a desired behavior.

2.2. Feedback loops in the real world

In the real world, feedback structures are ubiquitous. They are part of our primal experience of the world. For example, bending a plastic ruler has one stable state near equilibrium enforced by negative feedback (the ruler resists with a force that increases with the degree of bending) and a clothes pin has one stable and one unstable state (it can be put temporarily in the unstable state by pinching). Both objects are governed by a single feedback loop. A safety pin has two nested loops with an outer loop managing an inner loop. It has two stable states in the inner loop (open and closed), each of which is adaptive like the ruler's. The outer loop (usually a human being) controls the inner loop by choosing the stable state.

In general, anything with continued existence is managed by one or more feedback loops. Lack of feedback means that there is a runaway reaction (an explosion or implosion). This is true at all size and time scales, from the atomic to the astronomic. For example, the binding of atoms in a molecule is governed by a simple negative feedback loop that maintains equilibrium within given perturbation bounds. At the other extreme, a star at the end of its lifetime collapses until it finds a new stable state. If there is no force to counteract the collapse, then the star collapses indefinitely (at least, until it is beyond our current understanding of how the universe works).

2.2.1. The mean field approximation

The stability of ordinary matter is explained by a feedback loop. An acceptable model for ordinary matter is the mean field approximation, which gives good results outside of critical points (see chapter 1 of [15]). To explain this approximation, we start by the simple assumption that a uniform substance reacts linearly when an external force is applied:

$$\textit{Reaction} = A \times \textit{Force}$$

For example, for a gas we can assume that density n is proportional to pressure p:

$$n = (1/kT) \times p$$

This is the Boyle-Mariotte law for ideal gases, which is valid for small pressures. But this equation gives a bad approximation when the pressure is high. It leads to the conclusion that infinite pressure on a gas will reduce its volume to zero, which is not true.

We can obtain a much better approximation by making the assumption that throughout the substance there exists a force that is a function of the reaction. This force is called the *mean field*. This gives a new equation:

$$\textit{Reaction} = A \times (\textit{Force} + a(\textit{Reaction}))$$

That is, even in the absence of an external force, there is an internal force $a(\textit{Reaction})$ that causes the reaction to maintain itself at a nonzero value. This internal force is the mean field. There is a feedback effect: the mean field itself causes a reaction, which engenders a mean field, and so on. It is this feedback effect that explains, e.g., why a condensed state such as a liquid can exist at low temperatures independent of external pressure. J. Van der Waals applied this reasoning to the ideal gas law, by adding a term:

$$n = 1/kT \times (p + a(n))$$

where n is the density of the gas and p is the pressure. According to this equation, the density n of a fluid can stay at a high value even though the external pressure is low: a condensed state can exist at low temperature independent of the pressure. The internal pressure $a(n)$ replaces the external pressure. Van der Waals chose $a(n) = a \times n^2$ by following the reasoning that internal pressure is proportional to n, the number of molecules per unit of volume, multiplied by the influence of all neighboring molecules on each molecule. This influence is assumed to be proportional to n. This gives a new equation that is a good approximation over a wide range of densities and pressures.

The mean field approach can be applied to a wide range of problems. The limits of the approach are attained near critical points. This is because the correlation distance between molecules diverges. Near a critical point, there is a phase change of the fluid, e.g., a liquid can boil to become a gas. The global behavior of the fluid changes. The behavior of matter near critical points no longer follows the mean field approximation but can be explained using scale invariance laws. We are using this behavior as a guide for the design of software systems (see Section 4).

2.3. Feedback loops in human society

Most products of human civilization need an implicit management feedback loop, called "maintenance," done by a human. Each human is at the center of a large number of these feedback loops. The human brain has a large capacity for creating these loops; some are called "habits" or "chores." If there are too many feedback loops to manage, then the brain can no longer cope: the human complains that "life is too complicated"! We can say that civilization advances by reducing the number of feedback loops that have to be managed explicitly [28]. We postulate that this is also true of software.

2.4. Feedback loops in software

Software is in the same situation as other products of human civilization. Existing software products are very fragile: they require frequent maintenance by a human. To avoid this, we propose that software must be constructed as multiple interacting feedback loops, as an effective way to reduce its fragility. This is already being done in specific domains; here are five example:

- The subsumption architecture of Brooks is a way to implement intelligent systems by decomposing complex behaviors into layers of simple behaviors, each of which controls the layers below it [4].
- IBM's Autonomic Computing initiative aims to reduce management costs of computing systems by removing humans from low-level management loops [11]. The low-level loop is managed by a high-level loop that contains a human.
- Hellerstein *et al* show how to design computing systems with feedback control, to optimize global behavior such as maximizing throughput [10]. Hellerstein shows two examples of adaptive systems with interacting feedback loops: gain scheduling (with dynamic selection among multiple controllers) and self-tuning regulation (where controller gain is continuously adjusted).

- Distributed algorithms for fault tolerance handle a special case of feedback where the observer is a failure detector [16]. The implementation of the failure detector itself requires a feedback loop.
- Structured overlay networks (SONs, closely related to distributed hash tables, DHTs) are inspired by peer-to-peer networks [23]. They use principles of self organization to guarantee scalable and efficient storage, lookup, and routing despite volatile computing nodes and networks. Our own work is in the area of SONs; we explain it further in Section 4.

3. EXAMPLES OF INTERACTING FEEDBACK LOOPS

We give two examples of nontrivial systems that consist of multiple interacting feedback loops (for more examples see [25, 26]). Our first example is taken from biology: the human respiratory system. Our second example is taken from software design: the TCP protocol family.

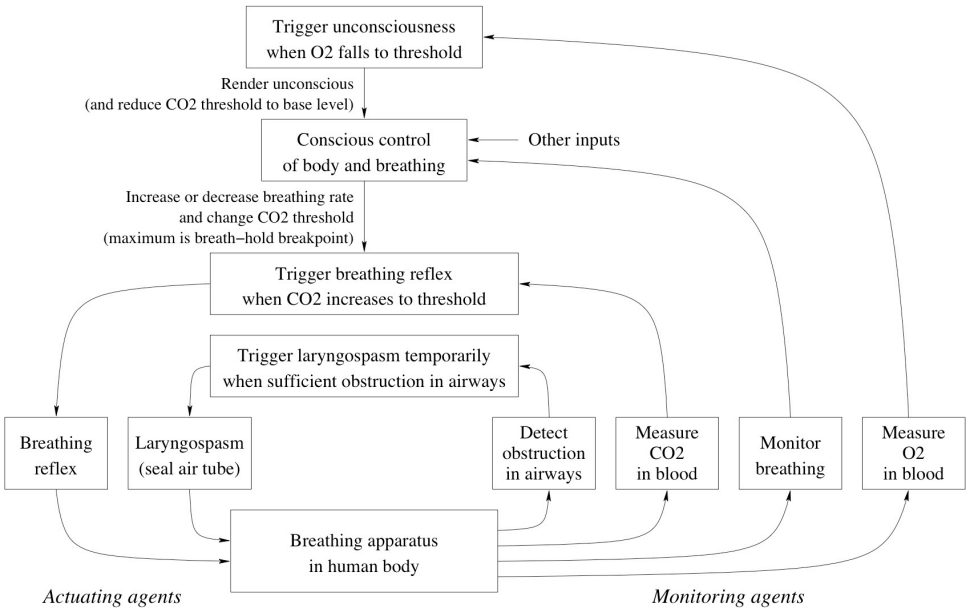


FIGURE 1: The human respiratory system as a feedback loop structure

3.1. The human respiratory system

Successful biological systems survive in natural environments, which can be particularly harsh. Studying them gives us insight in how to design robust software. Figure 1 shows the components of the human respiratory system and how they interact. The rectangles are concurrent component instances and the arrows are message channels. We derived this figure from a precise medical description of the system's behavior [30]. The figure is slightly simplified when compared to reality, but it is complete enough to give many insights. There are four feedback loops: two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious). From the figure we can deduce what happens in many realistic cases. For example, when choking on a liquid or a piece of food, the larynx constricts so we temporarily cannot breathe (this is called laryngospasm). We can hold our breath consciously: this increases the CO_2 threshold so that the breathing reflex is delayed. If you hold your breath as long as possible, then eventually the breath-hold threshold is reached and the breathing reflex happens anyway. A trained person can hold his or her breath long enough so that the O_2 threshold is reached first and they fall unconscious without breathing. When unconscious the breathing reflex is reestablished.

We can infer some plausible design rules from this system. The innermost loops (breathing reflex and laryngospasm) and the outermost loop (falling unconscious) are based on negative feedback using a monotonic parameter. This gives them stability. The middle loop (conscious control) is not

stable: it is highly nonmonotonic and may run with both negative or positive feedback. It is by far the most complex of the four loops. We can justify why it is sandwiched in between two simpler loops. On the inner side, conscious control manages the breathing reflex, but it does not have to understand the details of how this reflex is implemented. This is an example of using nesting to implement abstraction. On the outer side, the outermost loop overrides the conscious control (a fail safe) so that it is less likely to bring the body's survival in danger. Conscious control seems to be the body's all-purpose general problem solver: it appears in many of the body's feedback loop structures. This very power means that it needs a check.

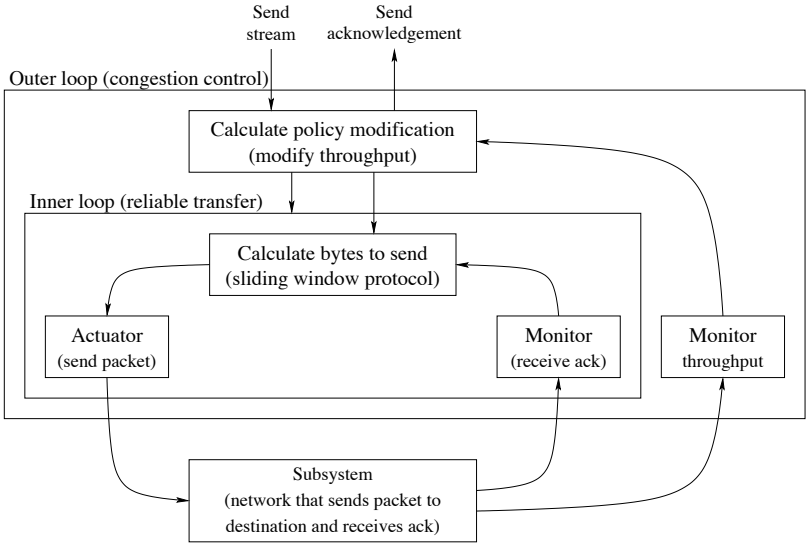


FIGURE 2: TCP as a feedback loop structure

3.2. TCP as a feedback loop structure

The TCP family of network protocols has been carefully tailored over many years to work adequately for the Internet. We consider therefore that its design merits close study. We explain the heart of TCP as two interacting feedback loops that implement a reliable byte stream transfer protocol with congestion control [12]. The protocol sends a byte stream from a source to a destination node. Figure 2 shows the two feedback loops as they appear at the source node. The inner loop does reliable transfer of a stream of packets: it sends packets and monitors the acknowledgements of the packets that have arrived successfully. The inner loop manages a sliding window: the actuator sends packets so that the sliding window can advance. The sliding window can be seen as a case of negative feedback using monotonic control. The outer loop does congestion control: it monitors the throughput of the system and acts either by changing the policy of the inner loop or by changing the inner loop itself. If the rate of acknowledgements decreases, then it modifies the inner loop by reducing the size of the sliding window. If the rate becomes zero then the outer loop may terminate the inner loop and abort the transfer.

4. STRUCTURED OVERLAY NETWORKS AS A FOUNDATION FOR FEEDBACK ARCHITECTURES

Our own work on feedback structures targets large-scale distributed applications. This work is being done in the SELFMAN project [20]. Summarizing briefly, we are building an infrastructure based on a transaction service running over a structured overlay network [19, 26]. We target our design on three application scenarios taken from industrial case studies: a machine-to-machine

messenging application, a distributed knowledge management application (similar to a Wiki), and an on-demand media streaming service [6].

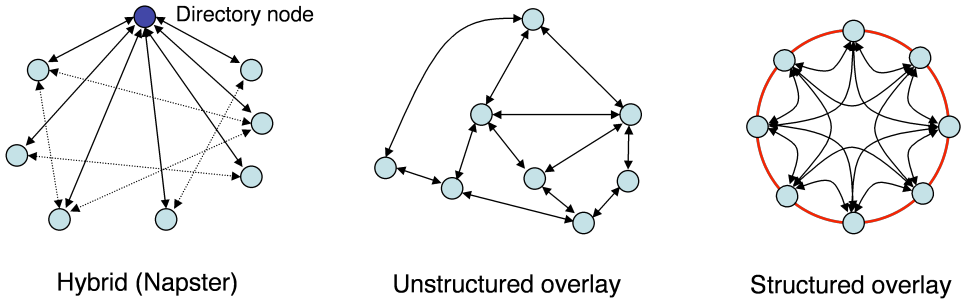


FIGURE 3: Three generations of peer-to-peer networks

4.1. Structured overlay networks

Structured overlay networks are inspired by peer-to-peer networks [23]. In a peer-to-peer network, all nodes play equal roles. There are no specialized client or server nodes. Figure 3 summarizes the history of peer-to-peer networks in three generations. In the first generation (exemplified by Napster), clients are peers but the directory is centralized. In the second generation (exemplified by Gnutella), peer nodes communicate by random neighbor links. The third generation is the structured overlay network. Compared to peer-to-peer systems based on random neighbor graphs, SONs guarantee efficient routing and guarantee lookup of data items. Almost all existing structured overlay networks are organized as two levels, a ring complemented by a set of fingers:

- *Ring structure.* All nodes are connected in a simple ring. The ring is maintained connected despite node joins, leaves, and failures.
- *Finger tables.* For efficient routing, extra links called fingers are added to the ring. The fingers can temporarily be in an inconsistent state. This has an effect only on efficiency, not on correctness. Within each node, the finger table is continuously converging to a consistent state.

Atomic ring maintenance is a crucial part of the overlay. Peer nodes can join and leave at any time. Peers that crash are like peers that leave but without notification. Temporarily broken links create false suspicions of failure.

Structured overlay networks are already designed as feedback structures. They already solve the problem of self management for scalable communication and storage. We are using them as the basis for designing a general architecture for self-managing applications. To achieve this goal, we are extending the SONs in three ways:

- We have devised algorithms for handling imperfect failure detection (false suspicions) [17], which vastly reduces the probability of lookup inconsistency. Imperfect failure detection is handled by relaxing the ring invariant to obtain a so-called “relaxed ring”, which maintains connectivity even with nodes that are suspected (possibly falsely) to be failed. The relaxed ring is always converging to a perfect ring as suspicions are resolved.
- We have devised algorithms for detecting and merging network partitions [21]. This is a crucial operation when the SON crosses a critical point (see Section 4.3).
- We have devised and implemented a transaction algorithm on top of the SON using a symmetric replicated storage and a modified version of the Paxos uniform consensus algorithm to achieve atomic commit with the Internet failure model [19].

4.2. Transactions over a SON

The highest-level service that we are implementing on a SON is a transactional storage. Implementing transactions over a SON is challenging because of churn (the rate of node leaves,

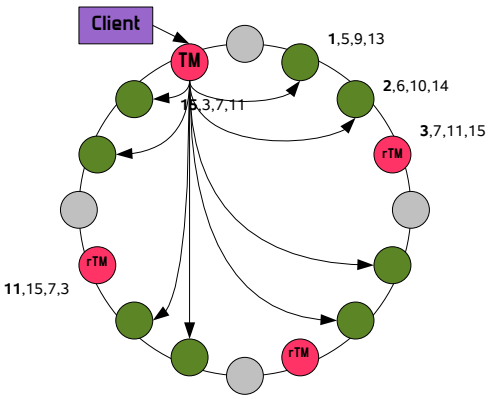


FIGURE 4: Distributed transactions on a structured overlay network

joins, and failures and the subsequent reorganizations of the overlay) and because of the Internet's failure model (crash stop with imperfect failure detection). The transaction algorithm is built on top of a reliable storage service. We implement this using symmetric replication [7].

To avoid the problems of failure detection, we implement atomic commit using a majority algorithm based on a modified version of Paxos [19, 8]. We use an imperfect failure detector to change coordinators in this algorithm. This is implementable on the Internet; because the failure detection is imperfect we may change coordinators too often, but this only affects efficiency, not correctness. We have shown that majority techniques work well for DHTs [22]: the probability of data consistency violation is negligible. If a consistency violation does occur, then this is because of a network partition and we can use the network merge algorithm [21].

We give a simple scenario to show how the algorithm works. A client initiates a transaction by asking its nearest node, which becomes a transaction manager. Other nodes that store data are participants in the transaction. Assuming symmetric replication with degree f, we have f transaction managers and f replicas for each other participating node. Figure 4 shows a situation with $f = 4$ and two nodes participating in addition to the transaction manager. Each transaction manager sends a Prepare message to all replicated participants, which each sends back a Prepared or Abort message to all replicated transaction managers. Each replicated transaction manager collects votes from a majority of participants and locally decides on abort or commit. It sends this to the transaction manager. After having collected a majority, the transaction manager sents its decision to all participants. This algorithm has six communication rounds. It succeeds if more than $f/2$ nodes of each replica group are alive.

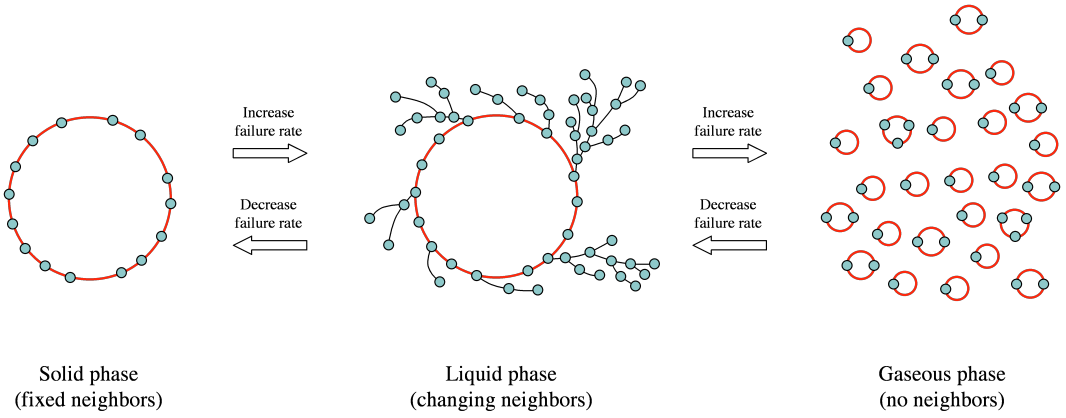


FIGURE 5: Conjectured phase transitions for a relaxed ring SON

4.3. Phase transitions in SONs and their effect on application design

At low node failure rates, a SON is a single ring where each node has fixed neighbors. This corresponds to a solid phase. At high failure rates, a SON will separate into many small rings. At the limit, a SON with n nodes will separate into n single-node SONs. This is the gaseous phase. In between these two extremes we conjecture that there is a liquid phase, the relaxed ring, where the ring is connected but each node does not have a fixed set of neighbors. When a node is subject to a failure suspicion then its set of neighbors changes.

We conjecture that for properly designed SONs phase transitions can occur for changing values of the failure rate. Figure 5 shows the kind of behavior we expect for the relaxed ring. In this figure, we assume that the node failure rate is equal to the node join rate, so that the total number of nodes is stationary. In accord with the Internet's failure model, we also assume that some of the reported failures are not actual failures (they are called failure suspicions [9]). At low failure rates, the ring is connected and does not change (solid phase). At high failure rates, the ring "boils" to become a set of small rings (of size 1, in the extreme case). At intermediate failure rates, the ring may stay connected but because of failure suspicions some nodes get pushed into side branches (relaxed ring).

We support this conjecture by citing [14], which uses the analytical model of [13] to show that phase transitions should occur in the Chord SON [23]. Specifically, [14] shows that three phases are traversed when the average network delay increases, in the following order: a region of efficient lookup, followed by a region where the longest fingers are dead (inefficient lookup), followed by a region where the ring is disconnected. We are setting up simulation experiments to verify this behavior and further explore the phase behavior of SONs.

A SON that behaves in this way will never "fail," it will just change phase. Each phase has well-defined behavior that can be programmed for. These phase transitions should therefore be considered as normal behavior that can be exposed to the application running on top of the SON. An important research question is to determine what the application API should be for phase transitions. At high failure rates, the application will run as many separate parts. When the rate lowers, these parts will combine (they will "condense" using the merge algorithm) and the application should resolve conflicts by an appropriate merge of the information stored in the separate rings. We can see that the application will probably have different consistency models at different failure rates. The transaction algorithm of the previous section will need to be modified to take this into account.

As a final remark, we conclude that the merge algorithm is a necessary part of a SON. Without the merge algorithm, condensation of a gaseous system is not possible. The SON is incomplete without it. With the merge algorithm, the SON and its applications can live indefinitely at any failure rate.

5. CONCLUSIONS

To overcome the fragility of software, we propose to build the software as a set of interacting feedback loops. Each feedback loop monitors and corrects part of the system. No part of the system should exist outside of a feedback loop. We motivate this idea by showing how it exists in real systems taken from biology and software (the human respiratory system and the Internet's TCP protocol family). If the feedback structure is properly designed, then it reacts to a hostile environment by doing a reversible phase transition. For example, when the node failure rate increases, a large overlay network may become a set of disjoint smaller overlay networks. When the failure rate decreases, these smaller networks will coalesce into a large network again. These transitions can be exposed to the application as an API so that it can be written to survive the transition. Important research questions are to determine what this API should be and how it affects application design.

In our own work in the SELFMAN project [20], we have built structured overlay networks that survive in realistically harsh environments (with imperfect failure detection and network

partitioning). We have developed a network merge algorithm that allows structured overlay networks to do reversible phase transitions. We are extending our SON with transaction management to implement three application scenarios derived from our industrial partners. We are currently finishing our implementation and evaluating the behavior of our system. Much remains to be done, e.g., we need to extend the transaction algorithm of Section 4.2 so that it also works correctly during phase transitions.

One important lesson from this work is that all future software systems should be designed so that they can support reversible phase transitions. For example, up to the work reported in [21], SONs could not merge. That means that they could not "condense" (move from a gaseous back to a solid phase) as failure rates decreased. They would "boil" (become disconnected) when failure rates increased and they would stay disconnected when the failure rates decreased. We conclude that network merge is more than just an incremental improvement that helps improve reliability. It is *fundamental* because it allows the system to survive any number of phase transitions. The system is reversible and therefore does not break. Without it, the system breaks after just a single phase transition.[1]

6. ACKNOWLEDGEMENTS

This work is funded by the European Union in the SELFMAN project (contract 34084) and in the CoreGRID network of excellence (contract 004265). Peter Van Roy is the coordinator of SELFMAN. He acknowledges all SELFMAN partners for their insights and research results. In particular, he acknowledges the work on the relaxed ring, network partitioning, symmetric replication, distributed transactions, and the analytic study of SONs, all done by SELFMAN partners. Some of this work was done in the earlier PEPITO and EVERGROW projects.

REFERENCES

[1] Armstrong, Joe. "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, Royal Institute of Technology (KTH), Kista, Sweden, Nov. 2003.

[2] Ashby, W. Ross. "An Introduction to Cybernetics," Chapman & Hall Ltd., London, 1956. Internet (1999): `http://pcp.vub.ac.be/books/IntroCyb.pdf`.

[3] von Bertalanffy, Ludwig. "General System Theory: Foundations, Development, Applications," George Braziller, 1969.

[4] Brooks, Rodney A. *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, RA-2, April 1986, pp. 14-23.

[5] Cousot, Patrick, and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, 4th ACM Symposium on Principles of Programming Languages (POPL 1977), Jan. 1977, pp. 238-252.

[6] France Télécom, Zuse Institut Berlin, and Stakk AB. *User requirements for self managing applications: three application scenarios*, SELFMAN Deliverable D5.1, Nov. 2007, `www.ist-selfman.org`.

[7] Ghodsi, Ali, Luc Onana Alima, and Seif Haridi. *Symmetric Replication for Structured Peer-to-Peer Systems*, Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P 2005), Springer-Verlag LNCS volume 4125, pages 74-85.

[8] Gray, Jim, and Leslie Lamport. *Consensus on transaction commit*. ACM Trans. Database Syst., ACM Press, 2006(31), pages 133-160.

[9] Guerraoui, Rachid, and Luis Rodrigues, "Introduction to Reliable Distributed Programming," Springer-Verlag Berlin, 2006.

[10] Hellerstein, Joseph L., Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. "Feedback Control of Computing Systems," Aug. 2004, Wiley-IEEE Press.

[11] IBM. *Autonomic computing: IBM's perspective on the state of information technology*, 2001, `researchweb.watson.ibm.com/autonomic`.

[12] Information Sciences Institute. "RFC 793: Transmission Control Protocol Darpa Internet Program Protocol Specification," Sept. 1981.

[13] Krishnamurthy, Supriya, Sameh El-Ansary, Erik Aurell, and Seif Haridi. *A statistical theory of Chord under churn*, Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), Ithaca, New York, Feb. 2005.

[1] An interesting question for physicists is to explain why matter behaves in reversible fashion. Software has to be designed for reversibility while simple molecules have this property implicitly.

[14] Krishnamurthy, Supriya, and John Ardelius. *An Analytical Framework for the Performance Evaluation of Proximity-Aware Overlay Networks*, Tech. Report TR-2008-01, Swedish Institute of Computer Science, Feb. 2008 (submitted for publication).

[15] Laguës, Michel and Annick Lesne. "Invariances d'échelle: Des changements d'états à la turbulence" ("Scale invariances: from state changes to turbulence"), Belin éditeur, Sept 2003.

[16] Lynch, Nancy. "Distributed Algorithms," Morgan Kaufmann, San Francisco, CA, 1996.

[17] Mejias, Boris, and Peter Van Roy. *A Relaxed Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks*, XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), Nov. 2007.

[18] Miller, Mark S., Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. *E: Open source distributed capabilities*, 2001, `www.erights.org`.

[19] Moser, Monika, and Seif Haridi. *Atomic Commitment in Transactional DHTs*, Proc. of the CoreGRID Symposium, Rennes, France, Aug. 2007.

[20] SELFMAN: Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components, European Commission 6th Framework Programme three-year project, June 2006 – May 2009, `www.ist-selfman.org`.

[21] Shafaat, Tallat M., Ali Ghodsi, and Seif Haridi. *Dealing with Network Partitions in Structured Overlay Networks*, Journal of Peer-to-Peer Networking and Applications, Springer-Verlag, 2008 (to appear).

[22] Shafaat, Tallat M., Monika Moser, Ali Ghodsi, Thorsten Schütt, Seif Haridi, and Alexander Reinefeld. *On Consistency of Data in Structured Overlay Networks*, CoreGRID Integration Workshop, Heraklion, Greece, Springer LNCS, 2008 (to appear).

[23] Stoica, Ion, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, SIGCOMM 2001, pp. 149-160.

[24] Van Roy, Peter. *Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place*, 8th International Symposium on Functional and Logic Programming (FLOPS 2006), April 2006, Springer LNCS volume 3945, pp. 2-12.

[25] Van Roy, Peter. *Self Management and the Future of Software Design*, Third International Workshop on Formal Aspects of Component Software (FACS 2006), Springer ENTCS volume 182, June 2007, pages 201-217.

[26] Van Roy, Peter, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. *Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project*, Springer LNCS, 2008 (to appear). Revised postproceedings of FMCO 2007, Oct. 2007.

[27] Weinberg, Gerald M. "An Introduction to General Systems Thinking: Silver Anniversary Edition," Dorset House, 2001 (original edition 1975).

[28] Whitehead, Alfred North. Quote: *Civilization advances by extending the number of important operations which we can perform without thinking of them.*

[29] Wiener, Norbert. "Cybernetics, or Control and Communication in the Animal and the Machine," MIT Press, Cambridge, MA, 1948.

[30] Wikipedia, the free encyclopedia. Entry "drowning," August 2006. Internet: `http://en.wikipedia.org/wiki/Drowning`.

A.2 Dealing with Network Partitions and Mergers

Peer-to-Peer Netw Appl
DOI 10.1007/s12083-009-0037-7

Dealing with network partitions in structured overlay networks

Tallat M. Shafaat · Ali Ghodsi · Seif Haridi

Received: 21 October 2007 / Accepted: 17 February 2009

Abstract Structured overlay networks form a major class of peer-to-peer systems, which are touted for their abilities to scale, tolerate failures, and self-manage. Any long-lived Internet-scale distributed system is destined to face network partitions. Although the problem of network partitions and mergers is highly related to fault-tolerance and self-management in large-scale systems, it has hardly been studied in the context of structured peer-to-peer systems. These systems have mainly been studied under churn (frequent joins/failures), which as a side effect solves the problem of network partitions, as it is similar to massive node failures. Yet, the crucial aspect of network mergers has been ignored. In fact, it has been claimed that ring-based structured overlay networks, which constitute the majority of the structured overlays, are intrinsically ill-suited for merging rings. In this paper, we present an algorithm for merging multiple similar ring-based overlays when the underlying network merges. We examine the solution in dynamic conditions, showing how our solution is resilient to churn during the merger, something widely believed to be difficult or impossible. We evaluate the algorithm for various scenarios and show that even when falsely detecting a merger, the algorithm quickly terminates and does not clutter the network with many messages. The algorithm is flexible as the tradeoff between message complexity and time complexity can be adjusted by a parameter.

Keywords DHTs · Network partitions · Network mergers · Structured overlay networks · Loopy rings · Distributed hash tables

1 Introduction

Structured Overlay Networks (SONs)—such as Chord [39], Pastry [34], and SkipNet [17]—are touted for their ability to provide scalability, fault-tolerance, and self-management, making them well-suited for Internet-scale distributed applications. Such Internet-scale systems will always come across network partitions, especially if the system is long-lived. Although the problem of network partitions and mergers is highly related to fault-tolerance and self-management in large-scale systems, it has, with few exceptions, been ignored in the context of structured overlays. This is peculiar, as the importance of the problem has long been known in other problem domains, such as in distributed databases [10] and distributed file systems [40].

Although network partitions are not very common, they do occur. Internet failures, resulting in partitioned networks can occur due to large area link failure, router failure, physical damage to a link/router, router misconfiguration and buggy software updates. Overloaded

T. M. Shafaat (✉) · S. Haridi
KTH - Royal Institute of Technology,
Electrum 229, 164 40 Kista, Sweden
e-mail: tallat@kth.se

S. Haridi
e-mail: haridi@kth.se

A. Ghodsi
Swedish Institute of Computer Science (SICS),
Box 1263, 164 29 Kista, Sweden
e-mail: ali@sics.se

Published online: 14 March 2009

routers, network wide congestion due to denial of service (DoS) attacks and routing loops [32] can also have the same effect as a network partition. Similarly, natural disasters can result in Internet failures. This was observed when an earthquake in Taiwan in December 2006 exposed the issue that global traffic passes through a small number of seismically active "choke points" [30]. Several countries in the region connect to the outside world through these choke points. A number of similar events leading to Internet failures have occurred [6]. On a smaller scale, the aforementioned causes can disconnect an entire organization from the Internet [31], thus partitioning the organization.

It is our firm belief that a crucial requirement for practical SONs is that they should be able to deal with network partitions and mergers. As we show in Section 2, SONs can, by a coincidence, cope with network partitions. Unfortunately, most SONs cannot cope with network mergers.

In fact, it has been claimed that ring-based structured overlays, which constitute the absolute majority of the SONs, are inherently a poor fit for dealing with network mergers. Datta et al. [8] focus on the merging of multiple SONs after a network partition ceases (network merger). They argue that ring-based SONs "cannot function at all until the whole merge process is complete".

The contribution of this paper is an algorithm for merging any number of similar structured overlays. We will limit ourselves to unidirectional ring-based overlays, since they constitute a majority of the SONs. The presented algorithm allows the system designer to adjust, through a *fanout* parameter, the tradeoff between bandwidth consumption (message and bit complexity) and time it takes for the algorithm to complete (time complexity). Through experimental evaluation, we show typical fanout values for which our algorithm completes quickly, while keeping the bandwidth consumption at an acceptable level. We examine the solution in dynamic conditions, showing how our solution is resilient to churn during the merger, something believed to be impossible [8]. We verify that the algorithm works efficiently even if only a single node detects the partition merger. We show that even with large rings with thousands of nodes, our solution is lean as it avoids positive-feedback cycles and, hence, avoids congesting the network. Finally, we show that the algorithm can recover from pathological scenarios, such as loopy rings [25, 38], which might result from network partitions.

The merging of SONs gives rise to problems on two different levels: *routing level* and *data level*. The routing level is concerned with healing of the routing information after a partition merger.

The data level is concerned with the consistency of the data items stored in the SONs. The solutions to this problem might depend on the application and on the semantics of the data operations, e.g. immutable key/value pairs or monotonically increasing values. It is also known that it is impossible to achieve strong (atomic) data consistency, availability,[1] and partition tolerance in SONs [5, 14, 15].

We focus on the problem of dealing with partition mergers at the routing level. Given a solution to the problem at the routing level, it is generally known how to achieve weaker types of data consistency, such as eventual consistency [11, 40].

Outline Section 2 serves as a background by motivating and defining our choice of ring-based SONs. Section 3 introduces the main contributions of this work, *simple ring unification algorithm*, as well as the *gossip-based ring unification algorithm*. Since the latter algorithm builds on the previous, we hope that this has a didactic value. Thereafter, Section 4 evaluates different aspects of the algorithms in various scenarios. Section 5 presents related work. Finally, Section 6 concludes and presents an ambitious agenda for future work.

2 Background

The rest of the paper focuses on ring-based structured overlay networks. Next, we motivate this choice, and thereafter briefly define ring-based SONs. Finally, we show how Chord deals with network partitions and failures.

Motivation for the unidirectional ring geometry We confine ourselves to unidirectional ring-based SONs, such as Chord [39], SkipNet [17], DKS [14], Koorde [20], Mercury [4], Symphony [28], EpiChord [22], and Accordion [23]. But we believe that our algorithms can be adapted easily to other ring-based SONs, such as Pastry [34]. For a more detailed account on directionality and structure in SONs, please refer to Onana et al. [3] and Aberer et al. [1].

The reason for confining ourselves to ring-based SONs is twofold. First, ring-based SONs constitute a majority of the SONs. Second, Gummadi et al. [16] diligently compared the geometries of different SONs,

[1] By availability we mean that a get/put operation should eventually complete.

and showed that the ring geometry is the one most resilient to failures, while it is just as good as the other geometries when it comes to proximity.

To simplify the presentation of our algorithms, we use notation that indicates the use of the Chord [39] SON. But the ideas are directly applicable to all unidirectional ring-based SONs.

A model of a ring-based SON A SON makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some apriori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$.

Every node in the system, has a unique identifier from the identifier space. Node identifiers are typically assumed to be uniformly distributed on the identifier space. Each node keeps a pointer, *succ*, to its *successor* on the ring. The successor of a node with identifier p is the first node found going in clockwise direction on the ring starting at p. Similarly, every node also has a pointer, *pred*, to its *predecessor* on the ring. The predecessor of a node with identifier q is the first node met going in anti-clockwise direction on the ring starting at q. A *successor-list* is also maintained at every node r, which consists of r's c immediate successors, where c is typically set to $\log_2(n)$, where n is the network size.

Ring-based SONs also maintain additional routing pointers on top of the ring to enhance routing. To be concrete, assume that these are placed as in Chord. Hence, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod{\mathcal{N}}$ for $0 \leq i < \log_2(\mathcal{N})$. Our results can easily be adapted to other schemes for placing these additional pointers.

Dealing with partitions and failures in chord Chord handles joins and leaves using a protocol called *periodic stabilization*. Leaves are handled by having each node periodically check whether *pred* is alive, and setting $pred := nil$ if it is found dead. Moreover, each node periodically checks to see if *succ* is alive. If it is found to be dead, it is replaced by the closest alive successor in the successor-list.

Joins are also handled periodically. A joining node makes a lookup to find its successor s on the ring, and sets $succ := s$. Each node periodically asks for its successor's *pred* pointer, and updates *succ* if it finds a closer successor. Thereafter, the node notifies its current *succ* about its own existence, such that the successor can update its *pred* pointer if it finds that the notifying node is a closer predecessor than *pred*. Hence, any joining node is eventually properly incorporated into the ring.

As we mentioned previously, a single node cannot distinguish massive simultaneous node failures from a network partition. As periodic stabilization can handle massive failures [25], it also recovers from network partitions, making each component of the partition eventually form its own ring. We have simulated such scenarios and confirmed these results. The problem that remains unsolved, which is the focus of the rest of the paper, is how several independent rings can efficiently be merged.

3 Ring merging

For two or more rings to be merged, at least one node needs to have knowledge about at least one node in another ring. This is facilitated by the use of *passive lists*. Whenever a node detects that another node has failed, it puts the failed node, with its routing information[2] in its passive list. Every node periodically pings nodes in its passive list to detect if a failed node is alive again. When this occurs, it starts a ring merging algorithm. Hence, a network partition will result in many nodes being placed in passive lists. When the underlying network merges, this will be detected and rectified through the execution of a ring merging algorithm.

A ring merging algorithm can also be invoked in other ways than described above. For example, it could occur that two SONs are created independently of each other, but later their administrators decide to merge them due to overlapping interests. It could also be that a network partition has lasted so long, that all nodes in the rings have been replaced, making the contents of the passive lists useless. In cases such as these, a system administrator can manually insert an alive node from another ring into the passive list of any of the nodes. The ring merger algorithm will take care of the rest.

The detection of an alive node in a passive list does not necessarily indicate the merger of a partition. It might be the case that a single node is incorrectly detected as failed due to a premature timeout of a failure detector. The ring merging algorithm should be able to cope with this by trying to ensure that such false-positives will terminate the algorithm quickly. It might also be the case that a previously failed node rejoins the network, or that a node with the same overlay and network address as a previously failed node joins the ring. Such cases are dealt with by associating with every

[2]By routing information we mean a node's overlay identifier, network address, and nonce value (explained shortly).

node a globally unique random *nonce*, which is generated each time a node joins the network. Hence, if the algorithm detects that a node in its passive list is again alive, it can compare the node's current nonce value with that in the passive list to avoid a false-positive, as that node is likely a different node that coincidentally has the same overlay and network address.

3.1 Simple ring unification

In this section, we present the simple ring unification algorithm (Algorithm 1). As we later show, the algorithm will merge the rings in $O(N)$ time for a network size of N. Later, we show how the algorithm can be improved to make it complete the merger in substantially less time.

Algorithm 1 Simple Ring Unification Algorithm

```
 1: every γ time units and detqueue ≠ ∅ at p
 2:     q := detqueue.dequeue()
 3:     sendto p : MLOOKUP (q)
 4:     sendto q : MLOOKUP (p)
 5: end event

 6: receipt of MLOOKUP (id) from m at n
 7:     if id ≠ n and id ≠ succ then
 8:         if id ∈ (n, succ) then
 9:             sendto id : TRYMERGE (n, succ)
10:         else if id ∈ (pred, n) then
11:             sendto id : TRYMERGE (pred, n)
12:         else
13:             sendto closestprecedingnode(id) : MLOOKUP (id)
14:         end if
15:     end if
16: end event

17: receipt of TRYMERGE (cpred, csucc) from m at n
18:     sendto n : MLOOKUP (csucc)
19:     if csucc ∈ (n, succ) then
20:         succ := csucc
21:     end if
22:     sendto n : MLOOKUP (cpred)
23:     if cpred ∈ (pred, n) then
24:         pred := cpred
25:     end if
26: end event
```

Algorithm 1 makes use of a queue called *detqueue*, which will contain any alive nodes found in the passive list. The queue is periodically checked by every node p, and if it is non-empty, the first node q in the list is picked to start a ring merger. Ideally, p and q will be on two different rings. But even so, the distance between p and q on the identifier space might be very large, as the passive list can contain any previously failed node. Hence, the event MLOOKUP(id) is used to get closer to id through a lookup. Once MLOOKUP(id) gets near its destination id, it triggers the event TRYMERGE(a, b), which tries to do the actual merging by updating *pred* and *succ* pointers to a and b respectively.

The event MLOOKUP(id) is similar to a Chord lookup, which tries to do a greedy search towards the destination id. One difference is that it terminates the lookup if it reaches the destination and locally finds that it cannot merge the rings. More precisely, this happens if MLOOKUP(id) is executed at id itself, or at a node whose successor is id. If an MLOOKUP(id) executed at n finds that id is between n and n's successor, it terminates the MLOOKUP and starts merging the rings by calling TRYMERGE. Another difference between MLOOKUP and an ordinary Chord lookup is that an MLOOKUP(id) executed at n also terminates and starts merging the rings if it finds that id is between n's predecessor and n. Thus, the merge will proceed in both clockwise and anti-clockwise direction.

The event TRYMERGE takes as parameters a candidate predecessor, *cpred*, and a candidate successor *csucc*, and attempts to update the current node's *pred* and *succ* pointers. It also makes two recursive calls to MLOOKUP, one towards *cpred*, and one towards *csucc*. This recursive call attempts to continue the merging in both directions. Figure 1 shows the working of the algorithm.

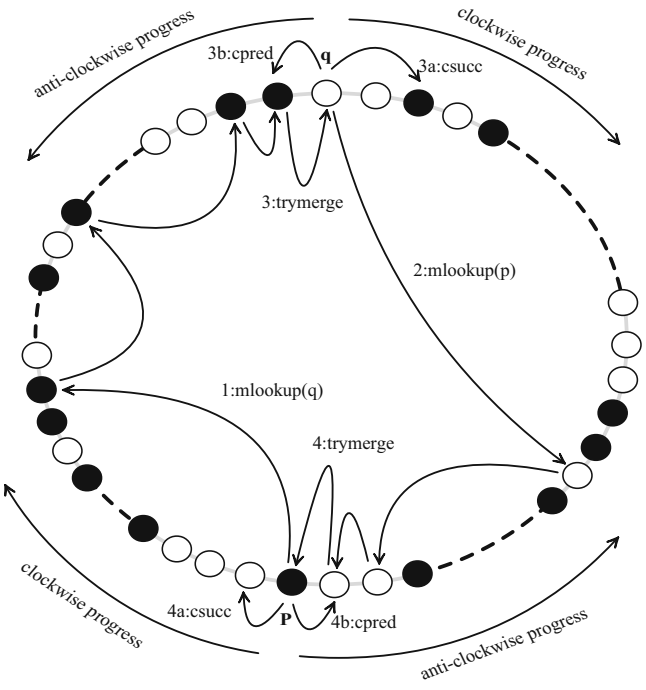


Fig. 1 *Filled circles* belong to SON1 and *empty circles* belong to SON2. The algorithm starts when p detects q, p makes an MLOOKUP to q and asks q to make an MLOOKUP to p

In summary, MLOOKUP closes in on the target area where a potential merger can happen, and TRYMERGE attempts to do local merging and advancing the merge process in both directions by triggering new MLOOKUPs.

3.2 Gossip-based ring unification

The simple ring unification presented in the previous section has two disadvantages. First, it is slow, as it takes $O(N)$ time to complete the ring unification. Second, it cannot recover from certain pathological scenarios. For example, assume two distinct rings in which every node points to its successor and predecessor in its own ring. Assume furthermore that the additional pointers of every node point to nodes in the other ring. In such a case, an *mlookup* will immediately leave the initiating node's ring, and hence may terminate. We do not see how such a pathological scenario could occur due to a partition, but the *gossip-based ring unification algorithm* (Algorithm 2) rectifies both disadvantages of the simple ring unification algorithm. Moreover, the simple ring unification is less robust to churn, as we discuss in the evaluation section.

Algorithm 2 Gossip-based Ring Unification Algorithm

```
1:  every γ time units and detqueue ≠ ∅ at p
2:      ⟨q, f⟩ := detqueue.dequeue()
3:      sendto p : MLOOKUP (q, f)
4:      sendto q : MLOOKUP (p, f)
5:  end event

6:  receipt of MLOOKUP (id, f) from m at n
7:      if id ≠ n and id ≠ succ then
8:          if f > 1 then
9:              f := f − 1
10:             r := randomnodeinRT()
11:             at r : detqueue.enqueue(⟨id, f⟩)
12:         end if
13:         if id ∈ (n, succ) then
14:             sendto id : TRYMERGE (n, succ)
15:         else if id ∈ (pred, n) then
16:             sendto id : TRYMERGE (pred, n)
17:         else
18:             sendto closestprecedingnode(id) : MLOOKUP (id, f)
19:         end if
20:     end if
21: end event

22: receipt of TRYMERGE (cpred, csucc) from m at n
23:     sendto n : MLOOKUP (csucc, F)
24:     if csucc ∈ (n, succ) then
25:         succ := csucc
26:     end if
27:     sendto n : MLOOKUP (cpred, F)
28:     if cpred ∈ (pred, n) then
29:         pred := cpred
30:     end if
31: end event
```

Algorithm 2 is, as its name suggests, gossip-based. The algorithm is essentially the same as the simple ring unification algorithm, with a few additions. The intuition is to have the initiator of the algorithm to immediately start multiple instances of the simple algorithm at random nodes, with uniform distribution. But since the initiator's pointers are not uniformly distributed, the process of picking random nodes is incorporated into MLOOKUP. Thus, MLOOKUP(*id*) is augmented so that the current node randomly picks a node *r* in its current routing table and starts a ring merger between *id* and *r*. This change alone would, however, consume too much resources.

Two mechanisms are employed to prevent the algorithm from consuming too many messages, which could give rise to positive feedback cycles that congest the network. First, instead of immediately triggering an MLOOKUP at a random node, the event is placed in the corresponding node's *detqueue*, which is only checked periodically. Second, a constant number of random MLOOKUPs are created. This is regulated by a fanout parameter called F. Thus, the fanout is decreased each time a random node is picked, and the random process is only started if the fanout is larger than 1. The *detqueue*, therefore, holds tuples, which contain a node identifier and the current fanout parameter. Similarly, MLOOKUP takes the current fanout as a parameter. The rate for periodically checking the *detqueue* can be adjusted to control the rate at which the algorithm generates messages.

4 Evaluation

In this section, we evaluate the two algorithms from various aspects and in different scenarios. There are two measures of interest: *message complexity*, and *time complexity*. We differentiate between the *completion* and *termination* of the algorithm. By completion we mean the time when the rings have merged. By termination we mean the time when the algorithm terminates sending any more messages. Unless said otherwise, message complexity is until termination, while time complexity is until completion.

The evaluations are done in a stochastic discrete event simulator [37] in which we implemented Chord. The simulator uses an exponential distribution for the inter-arrival time between events (joins and failures). To make the simulations scale, the simulator is not packet-level. The time to send a message is an exponentially distributed random variable. The values in the graphs indicate averages of 20 runs with different random seeds.

We first evaluate the message and time complexity of the algorithms in a typical scenario where after merger, many nodes simultaneously detect alive nodes in their passive lists. Next, we evaluate the performance of the algorithm for a worst case scenario when only a single node detects the existence of another ring. The worst case scenario is similar to a case where an administrator wants to merge two SONs and triggers the ring unification algorithm on only a single node. Next, we assess the algorithms for a loopy ring. Thereafter, we evaluate the performance of the algorithms while node joins and failures are taking place during the ring merging process. Next, we compare our algorithm with a self-stabilizing algorithm. Finally, we evaluate the message complexity of the algorithms when a node falsely believes that it has detected another ring.

For the first experiment, the simulation scenario had the following structure. Initially nodes join and fail. After a certain number of nodes are part of the system, we insert a partition event, upon which the simulator divides the set of nodes into as many components as requested by the partition event, dividing the nodes randomly into the partitions but maintaining an approximate ratio specified. For our simulations, we create two partitions. A partition event is implemented using lottery scheduling [42] to define the size of each partition. The simulator then drops all messages sent from nodes in one partition to nodes in another partition, thus simulating a network partition in the underlying network and therefore triggering the failure handling algorithms (see Sections 2 and 3). Furthermore, node join and fail events are triggered in each partitioned component. Thereafter, a network merger event simply allows messages to reach other network components, triggering the detection of alive nodes in the passive lists, and hence starting the ring unification algorithms.

We simulated the simple ring unification algorithm and the gossip-based ring unification algorithm for partitions creating two components, and for fanout values from 1 to 7. For all the simulation graphs to follow, a fanout of 1 represents the simple ring unification algorithm. A time unit was equal to the time it takes for a message to reach its destination node.

Figures 2 and 3 show the time and message complexity for a typical scenario where after a merger, multiple nodes detect the merger and thus start the ring-unification algorithm. The number of nodes detecting the merger depends on the scenario; in our simulations, it was 10–15% of the total nodes. As can be seen in Figs. 2 and 3, the simple ring unification algorithm ($F = 1$) consumes minimum messages but takes maximum time when compared to different variations of

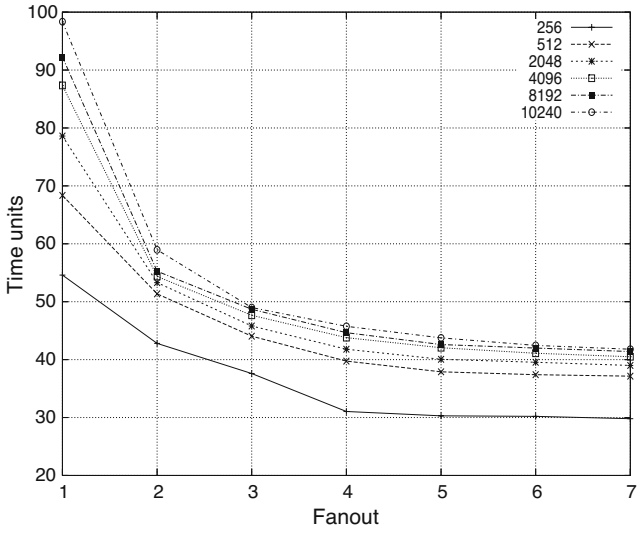


Fig. 2 Evaluation of time complexity for a typical scenario with multiple nodes detecting the merger for various network sizes and fanouts

the gossip-based ring unification algorithm. For higher values of F, the time complexity decreases while the message complexity increases. Increasing the fanout after a threshold value (around 3–4 in this case) will not considerably decrease the time complexity, but will just generate many unnecessary messages.

To proper understand the performance of the proposed algorithm, we generated scenarios where only one node would start the merger of the two rings. We randomly select, with uniform probability, the two

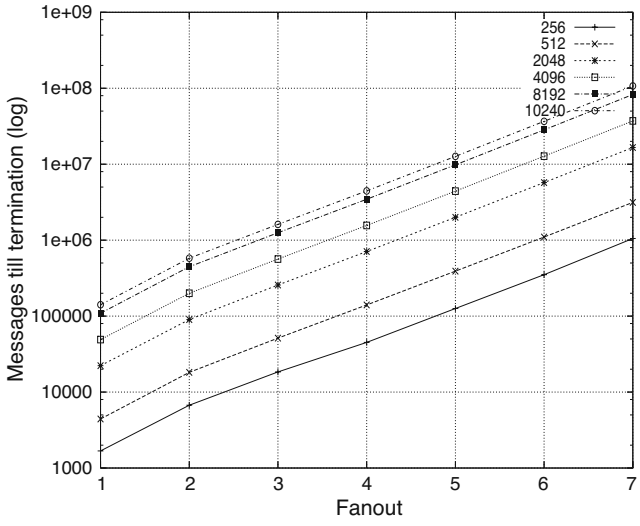


Fig. 3 Evaluation of message complexity for a typical scenario with multiple nodes detecting the merger for various network sizes and fanouts

nodes that are involved in the merger, i.e. the node p that detects the merger and the node that p detects from its passive list. Hence, the distance between them on the ring varies. For our experiments, each of the two rings had approximately half of the total number of nodes in the system before the merger. We choose the rate of checking *detqueue* to be every five time units and the rate of periodic stabilization (PS) to be every ten time units. The motivation for choosing a lower PS rate is to study the performance of the ring unification algorithm with minimum influence from PS.

We simulated ring unification for various network sizes of powers of 2 to study its scalability. Figure 4 shows the time complexity for varying network sizes. The x-axis is on a logarithmic scale, while the y-axis is linear. The graph for the gossip-based algorithms is linear, which suggests a $O(\log n)$ time complexity. In contrast, the simple ring unification graph (F = 1) is exponential, indicating that it does not scale well, i.e. $\omega(\log n)$ time complexity. In Fig. 5, we plot the number of ring unification messages sent by each node during the merger, i.e. the total number of messages induced by the algorithm until termination divided by the number of nodes. The linear graph on a log-log plot indicates a polynomial messages complexity. As expected, the number of messages per node grows slower for simple ring unification compared to gossip-based ring unification.

Figure 6 illustrates the tradeoff between time and message complexity. It shows that the goals of decreasing time and message complexity are conflicting. Thus, to decrease the number of messages, the time for completion will increase. Similarly, opting for convergence in lesser time will generate more messages. A suitable fanout value can be used to adapt the ring unification algorithm according to the requirements and network infrastructure available.

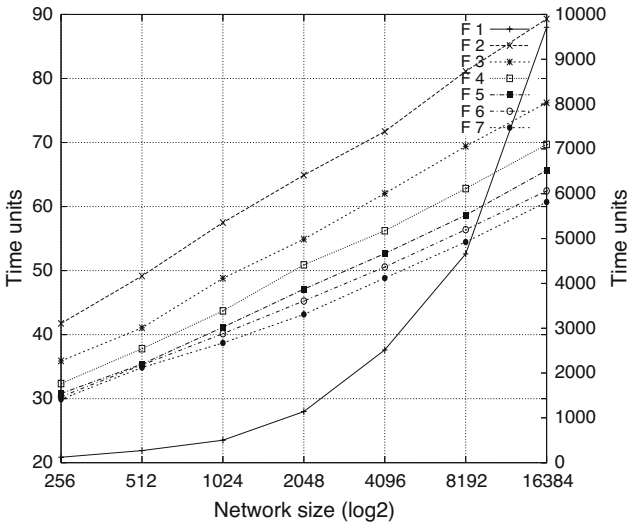


Fig. 4 Evaluation of time complexity when only one node starts the merger. Only F 1 is plotted against the right y-axis

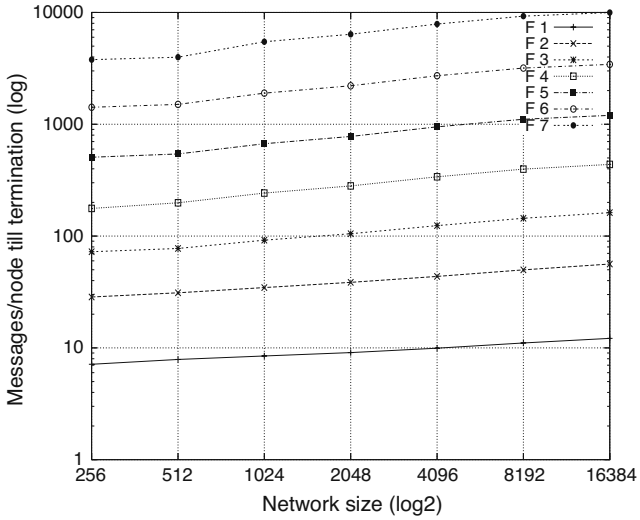


Fig. 5 Evaluation of message complexity per node when only one node starts the merger

For the rest of the evaluations, we use a worst case scenario where only a single node detects the merger.

Next, we evaluated the time and message complexity for a network to converge to a *strongly stable* ring from a *loopy* state of two cycles. As defined by Liben-Nowell et al. [25], a Chord network is *weakly stable* if, for all nodes u, $(u.succ).pred = u$ and *strongly stable* if, in addition, for each node u, there is no node v such that

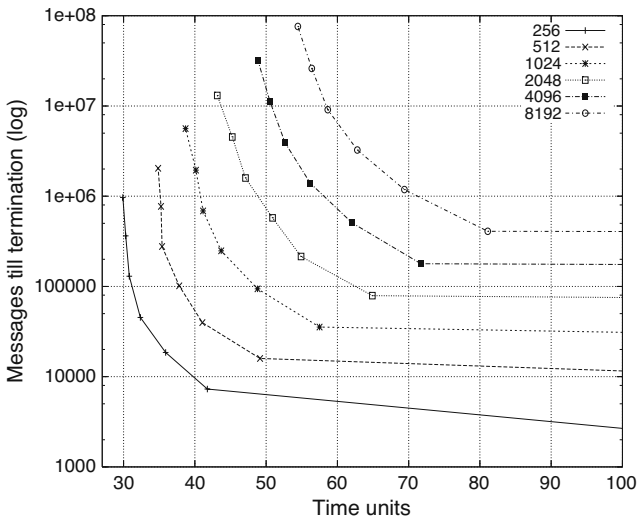


Fig. 6 Tradeoff between time and message complexity

$u < v < u.succ$. A loopy network is one which is weakly but not strongly stable. The scenario for the simulations was to create a loop of two cycles from one-fifth of the total number of nodes. Thereafter, we generated events of node joins for the remaining four-fifth nodes at an exponentially distributed inter-arrival rate. As in all experiments, the identifiers of the joining nodes were generated randomly with uniform probability. Thus, the nodes joined at different points in the loop. We then made one random node detect the loop by discovering a random node from the other cycle, triggering the ring unification algorithm. Figures 7 and 8 show the time and message complexity for the loopy network to converge to a strongly stable ring. The figures depict the effect of fanout on time and message complexity.

We evaluated rings unification under churn, *i.e.* nodes join and fail during the merger. Since we are using a scenario where only one node detects the merger, with low probability, the algorithm may fail to complete and the merged overlay may not converge under churn, especially for simple ring unification. The reason being intuitive: for simple unification, the two MLOOKUPS generated by the node detecting the merger while traveling through the network may fail as the node forwarding the MLOOKUP may fail under churn. With higher values of F and in typical scenarios where multiple nodes detect the merger, the algorithm becomes more robust to churn as it creates multiple MLOOKUPS. The results presented in Figs. 9 and 10 are only when the rings successfully converge. For simulation, after a merge event, we generate events of joins and fails until the unification algorithm terminates. With high churn, we mean that the inter-arrival time between events of joins and fails is less, thus representing highly dynamic conditions. Choosing a high inter-arrival time between events will create less joins and fails and thus churn will be less. For the simulations presented here, we choose inter-arrival time between events of joins and failures to be 30 units for high churn and 45 units for low churn, and an equal probability for a event to be a join or a fail. Figures 9 and 10 show how different values of F affect the convergence of the rings under different levels

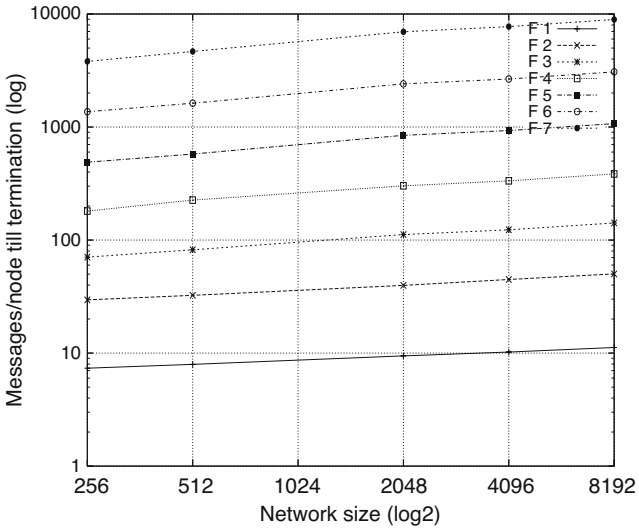


Fig. 8 Evaluation of message complexity for a loopy network

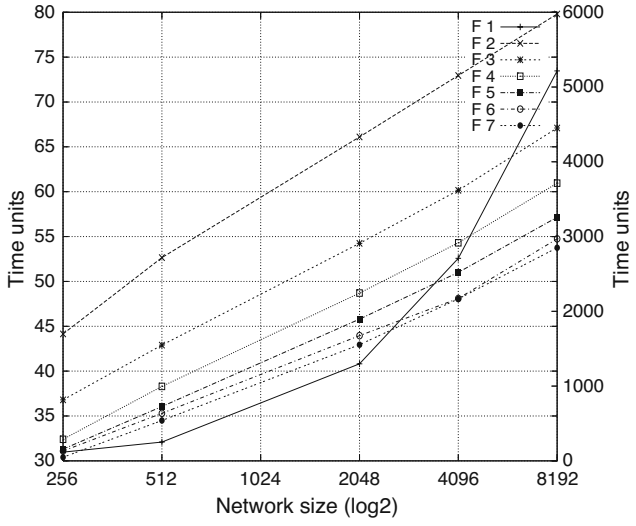


Fig. 7 Evaluation of time complexity for a loopy network. Only F 1 is plotted against the right y-axis

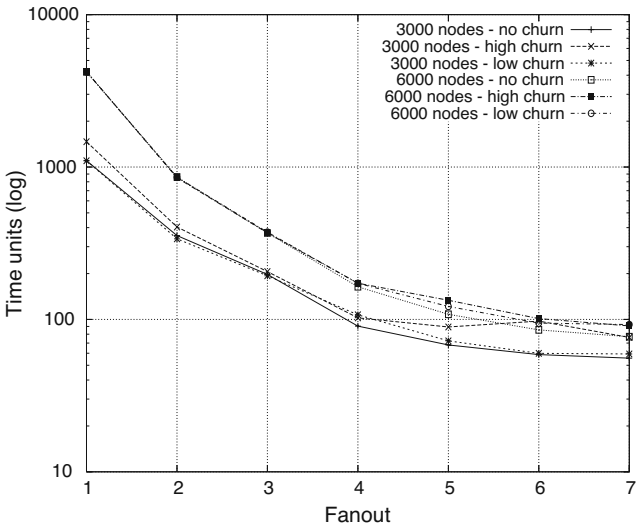


Fig. 9 Evaluation of time complexity under churn

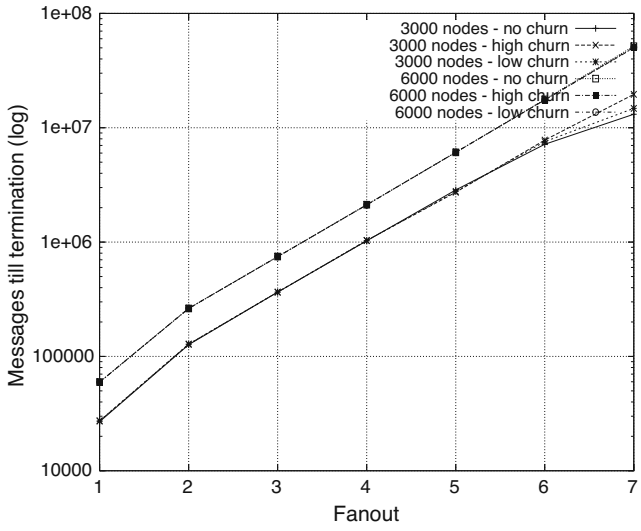


Fig. 10 Evaluation of message complexity under churn

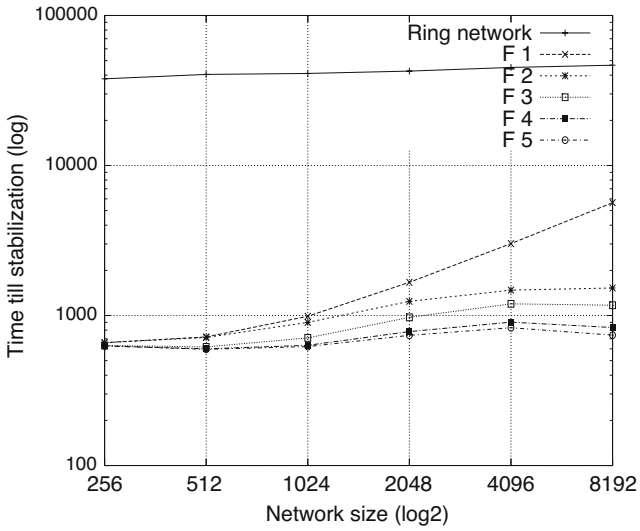


Fig. 11 Comparison of time complexity of ring unification and SSRN

of churn, mainly showing the algorithm works under churn without affecting message and time complexity much.

Further, we simulated the algorithms under churn to see how often they do not converge to a ring. We ran experiments with 200 different seeds for sizes ranging from 256 to 2048 nodes. We considered an execution successful if 95% of the nodes had correct successor pointers, as all successor pointers can not be correct while nodes are joining and failing. Thereafter, the remaining pointers are updated by Chord's periodic stabilization. For the 200 executions, we observed only 1 unsuccessful execution for network size 1024 and 2 unsuccessful executions for network size 2048. The unsuccessful executions happened only for simple ring unification, while executions with gossip based ring unification were always successful. Even for the unsuccessful executions, given enough time, PS updates the successor pointers to correct values.

We compared our algorithm with a Self-Stabilizing Ring Network (SSRN) [36] protocol. The results of our simulations comparing time and message complexity for various network sizes for the two algorithms have been presented in Figs. 11 and 12, depicting that ring unification consumes lesser time and messages compared to SSRN. The main reason for the better performance of our algorithm is that it has been designed specifically for merging rings. On the other hand, SSRN is a non-terminating algorithm that runs in the background like PS to find closer nodes. As evaluated previously, simple ring unification (fanout = 1) does not scale well for time complexity, which can be seen in Fig. 11.

Finally, we evaluate the scenario where a node may falsely detects a merger. Figure 13 shows the message complexity of the algorithm in case of a false detection. As can be seen, for lower fanout values, the message complexity is less. Even for higher fanouts, the number of messages generated are acceptable, thus showing that the algorithm is lean. We believe this to be important as most SONs do not have perfect failure detectors, and hence can give rise to inaccurate suspicions.

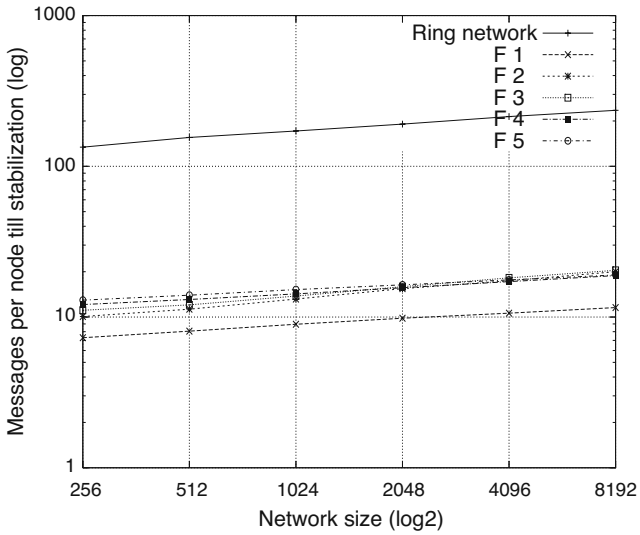


Fig. 12 Comparison of message complexity of ring unification and SSRN

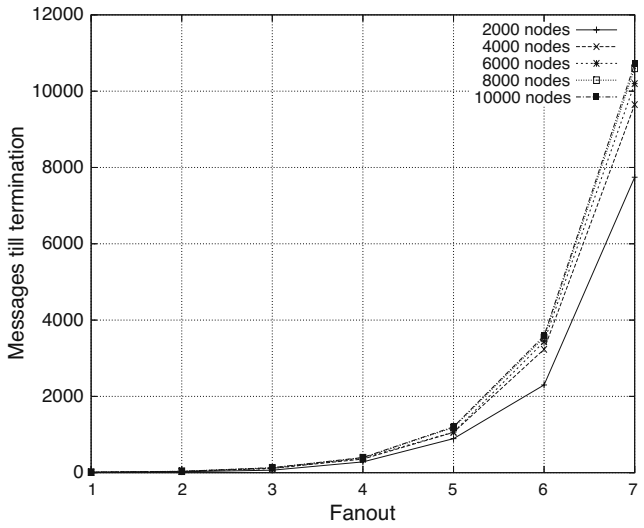


Fig. 13 Evaluation of message complexity in case a node falsely detects a merger for various network sizes and fanouts

Our simulations show that a fanout value of 3–4 is good for a system with several thousand nodes, even with respect to churn and false-positives.

5 Related work

Much work has been done to study the effects of churn on a structured overlay network [27], showing how overlays can cope with massive node joins and failures, thus showing how overlays are resilient to partitions. Datta et al. [8] have presented the challenges of merging two overlays, claiming that ring-based networks cannot operate until the merger operation completes. In contrast, we show how unification can work under churn while the merger operation is not complete. In a followup work, Datta et al. [9] show how to merge two P-Grid [2] SONs. There work differs from ours as P-Grid is a tree-based SON, while we focus on ring-based SONs.

The problem of constructing a SON from a random graph is, in some respects, similar to merging multiple SONs after a network merger, as the nodes may get randomly connected after a partition heals. Shaker et al. [36] have presented a ring-based algorithm for nodes in arbitrary state to converge into a directed ring topology. Their approach is different from ours, in that they provide a non-terminating algorithm which should be used to replace all join, leave, and failure handling of an existing SON. Replacing the topology maintenance algorithms of a SON may not always be feasible, as SONs may have intricate join and leave procedures to guarantee lookup consistency [14, 24, 26]. In contrast, our algorithm is a terminating algorithm that works as a plug-in for an already existing SON.

Kunzmann et al. [21] have proposed methods to improve the robustness of SONs. They propose to use a bootstrapping server to detect a merger by making the peer with the smallest identifier to send periodic messages to the bootstrap server. As soon as the bootstrap server receives messages from different peers, it will detect the existence of multiple rings. Thereafter, all the nodes have to be informed about the merger. While their approach has the advantage of having minimum false detections, it depends on a central bootstrap server. They lack a full algorithm and evaluation of how the merger will happen. Evaluation of the merge detection process and informing all peers about the detection is also missing.

Montresor et al. [29] show how Chord [39] can be created by a gossip-based protocol [18]. However, their algorithm depends on an underlying membership service like Cyclon [41], Scamp [13] or Newscast [19]. Thus the underlying membership service has to first cope with network mergers (a problem worth studying in its own right), where after T-Chord can form a Chord network. We believe one needs to investigate further how these protocols can be combined, and their epochs be synchronized, such that the topology provided by T-Chord is fed back to the SON when it has converged. Though the general performance of T-Chord has been evaluated, it is not known how it performs in the presence of network mergers when combined with various underlying membership services.

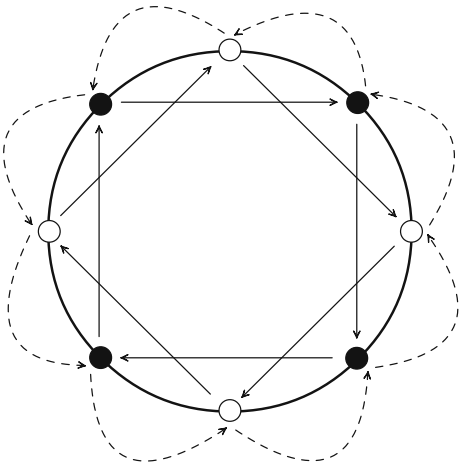


Fig. 14 A case where chord and the ring network protocol would break a connected graph into two components. *Lines* represent successor pointers while *dashed lines* represent a finger

As we show below, it might happen that an initially connected graph can be split into two separate components by the Chord [39] and SSRN [36] protocols. This scenario is a counter-proof of the claim that SSRN is self-stabilizing. Consider a network which consists of two perfect rings, yet the nodes have fingers pointing to nodes in the other ring. This can easily happen in case of unreliable failure detectors [7] or networks partitions. Normally, the PS rate is higher than fixing fingers, thus due to a temporary partition, it might happen that nodes update their successor pointers, yet before they fix their fingers, the partition heals. In such a scenario, SSRN splits the connected graph into two separate partitions, thus creating a partition of the overlay, while the underlay remains connected. An example of such a scenario is shown in Fig. 14, where the filled circles are nodes that are part of one ring and the empty circles are nodes that are part of the other ring. Each node has one finger pointing to a node in the other ring. The fix-finger algorithm in Chord updates the fingers by making lookups. In this case, a lookup will always return a node in the same ring as the one making the lookup. Consequently, the finger pointing to the other ring will be lost. Similarly, the pointer jumping algorithm used by SSRN to update its fingers will also drop the finger pointing to a node in the other ring. On the contrary, the ring-unification algorithm proposed in this paper will fix such a graph and converge it to a single ring.

Some SONs employ the ring based identifier space, which they mix with a prefix-based tree [33]. For example in Pastry [34], a responsible node for an identifier is the node with numerically closest identifier and the lookups are forwarded to nodes sharing the longest prefix with the identifier being looked up. Our algorithm can be modified for use by such SONs by replacing the closestpreceedingnode-procedure with the equivalent for the employed SON. The trymerge-procedure does not have to be changed since updating the predecessor and successor is similar to recording nodes with identifiers closest to a node.

The problem of network partitions and mergers has been studied in other distributed systems, such as in distributed databases [10] and distributed file systems [40]. These studies focus on problems created by the partition and merger on the data level, while our focus is on the routing level. We believe that such ideas, if combined with algorithms such as those we propose, can be used for handling data updates on SONs. That is, nevertheless, outside the scope of this paper.

The results of this paper extend on our previous work [35] by also considering loopy networks, the SSRN protocol [36], and including additional experimental results.

6 Conclusion

We have argued that the problem of partitions and mergers in structured peer-to-peer systems, when the underlying network partitions and recovers, is of crucial importance. We have presented a simple and a gossip-based algorithm for merging similar ring-based structured overlay networks after the underlying network merges.

Though we believe that the problem of dealing with network mergers is crucial, we think that such events happen more rarely. Hence, it might be justifiable in certain application scenarios that a slow paced algorithm runs in the background, consuming little resources, while ensuring that any potential problems with partitions will eventually be rectified. In such scenarios, our simple ring unification is more suitable. If on the other hand, one would prefer to speed up the unification process by consuming more messages, our gossip-based ring unification is more suitable. We have shown how the algorithm can be tuned to achieve a tradeoff between the number of messages consumed and the time before the overlay converges. We have evaluated our solution in realistic dynamic conditions, and showed that with high fanout values, the algorithm can converge quickly under churn. We have also shown that our solution generates few messages even if a node falsely starts the algorithm in an already converged SON. Finally, we have shown that our algorithm recovers from pathological scenarios, such as loopy rings, which might result from network partitions.

We tried many variations of the algorithms before reaching those that are reported in this paper. Initially, we had an algorithm that was not gossip-based, i.e. was not periodic and did not have any randomization. Albeit the algorithm was quite fast, it heavily overconsumed messages, making it infeasible for a large scale network. For that reason, we added the fanout parameter, and made it run periodically. Without randomization, we could construct pathological scenarios, in which that algorithm would not be able to merge the rings.

Future work We believe that dealing with partitions and mergers is a small part of a bigger, and more important, goal: making SONs that can recover from any configuration. We believe that it is desirable to make a self-stabilizing ring algorithm, which can be proved

to recover from all possible states, including loopy and partitioned while consuming minimum time and messages.

We believe that it is interesting to investigate whether gossip-based topology generators, such as T-man [18] and T-chord [29], can be used to handle network mergers. These services, however, make use of an underlying membership service, such as Cyclon [41], Scamp [13], or Newscast [19]. Hence, one has to first investigate how well such membership services recover from network partitions (we believe this to be interesting in itself). Thereafter, one can explore how such topology generators can be incorporated into a SON.

Mathematical analysis of gossip-protocols is often done through simple recurrence relations or by using Markov chains, where the state of the chain can be the number of infected nodes [12]. The algorithms we have proposed mix deterministic DHT algorithms with that of gossip protocols. Consequently, we believe that an analysis of our algorithms will require modelling the routing pointers of every node as part of the chain state. We solicit such an analysis and believe it is an interesting future direction for this research.

Acknowledgements This research has been funded by the European Projects SELFMAN and EVERGROW, VINNOVA 2005-02512 TRUST-DIS, and SICS Center for Networked Systems (CNS).

References

1. Aberer K, Alima LO, Ghodsi A, Girdzijauskas S, Haridi S, Hauswirth M (2005) The essence of P2P: a reference architecture for overlay networks. In: Proceedings of the 5th international conference on peer-to-peer computing (P2P'05). IEEE Computer Society, Los Alamitos, pp 11–20, August
2. Aberer K, Cudré-Mauroux P, Datta A, Despotovic Z, Hauswirth M, Punceva M, Schmidt R (2003) P-grid: a self-organizing structured P2P system. SIGMOD Rec 32(3): 29–33
3. Alima LO, Ghodsi A, Haridi S (2004) A framework for structured peer-to-peer overlay networks. In: Post-proceedings of global computing. Lecture notes in computer science (LNCS), vol 3267. Springer, Berlin Heidelberg New York, pp 223–250
4. Bharambe AR, Agrawal M, Seshan S (2004) Mercury: supporting scalable multi-attribute range queries. In: Proceedings of the ACM SIGCOMM 2004 symposium on communication, architecture, and protocols. ACM, Portland, pp 353–366, March
5. Brewer E (2000) Towards robust distributed systems. Invited talk at the 19th annual ACM symposium on principles of distributed computing (PODC'00)
6. Jahanian F, Labovitz C, Ahuja A (1998) Experimental study of internet stability and wide-area backbone failures. Technical report CSE-TR-382-98, University of Michigan, November
7. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267
8. Datta A, Aberer K (2006) The challenges of merging two similar structured overlays: a tale of two networks. In: Proceedings of the first international workshop on self-organizing systems (IWSOS'06). Lecture notes in computer science (LNCS), vol 4124. Springer, Berlin Heidelberg New York, pp 7–22
9. Datta A (2007) Merging intra-planetary index structures: decentralized bootstrapping of overlays. In: Proceedings of the first international conference on self-adaptive and self-organizing systems (SASO 2007). IEEE Computer Society, Boston, pp 109–118, July
10. Davidson SB, Garcia-Molina H, Skeen D (1985) Consistency in a partitioned network: a survey. ACM Comput Surv 17(3):341–370
11. Demers A, Greene D, Hauser C, Irish W, Larson J, Shenker S, Sturgis H, Swinehart D, Terry D (1987) Epidemic algorithms for replicated database maintenance. In: Proceedings of the 7th annual ACM symposium on principles of distributed computing (PODC'87). ACM, New York, pp 1–12
12. Eugster P Th, Guerraoui R, Handurukande SB, Kouznetsov P, Kermarrec A-M (2003) Lightweight probabilistic broadcast. ACM Trans Comput Syst 21(4):341–374
13. Ganesh AJ, Kermarrec A-M, Massoulié L (2001) SCAMP: peer-to-peer lightweight membership service for large-scale group communication. In: Proceedings of the 3rd international workshop on networked group communication (NGC'01). Lecture notes in computer science (LNCS), vol 2233. Springer, London, pp 44–55
14. Ghodsi A (2006) Distributed *k*-ary system: algorithms for distributed hash tables. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, December
15. Gilbert S, Lynch NA (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM Spec Interest Group Algorithms Comput Theory News 33(2):51–59
16. Gummadi K, Gummadi R, Gribble S, Ratnasamy S, Shenker S, Stoica I (2003) The impact of DHT routing geometry on resilience and proximity. In: Proceedings of the ACM SIGCOMM 2003 symposium on communication, architecture, and protocol. ACM, New York, pp 381–394
17. Harvey N, Jones MB, Saroiu S, Theimer M, Wolman A (2003) Skipnet: a scalable overlay network with practical locality properties. In: Proceedings of the 4th USENIX symposium on internet technologies and systems (USITS'03). USENIX, Seattle, March
18. Jelasity M, Babaoglu Ö (2005) T-man: gossip-based overlay topology management. In: Proceedings of 3rd workshop on engineering self-organising systems (EOSA'05). Lecture notes in computer science (LNCS), vol 3910. Springer, Berlin Heidelberg New York, pp 1–15
19. Jelasity M, Kowalczyk W, van Steen M (2003) Newscast computing. Technical report IR–CS–006, Vrije Universiteit, November
20. Kaashoek MF, Karger, DR (2003) Koorde: a simple degree-optimal distributed hash table. In: Proceedings of the 2nd interational workshop on peer-to-peer systems (IPTPS'03). Lecture notes in computer science (LNCS), vol 2735. Springer, Berkeley, pp 98–107

21. Kunzmann G, Binzenhöfer A (2006) Autonomically improving the security and robustness of structured P2P overlays. In: Proceedings of the international conference on systems and networks communications (ICSNC 2006). IEEE Computer Society, Tahiti, October–November
22. Leong B, Liskov B, Demaine E (2004) EpiChord: parallelizing the chord lookup algorithm with reactive routing state management. In: 12th international conference on networks (ICON'04). IEEE Computer Society, Singapore, November
23. Li J, Stribling J, Morris R, Kaashoek MF (2005) Bandwidth-efficient management of DHT routing tables. In: Proceedings of the 2nd USENIX symposium on networked systems design and implementation (NSDI'05). USENIX, Boston, May
24. Li X, Misra J, Plaxton, CG (2004) Brief announcement: concurrent maintenance of rings. In: Proceedings of the 23rd annual ACM symposium on principles of distributed computing (PODC'04). ACM, New York, p 376
25. Liben-Nowell D, Balakrishnan H, Karger DR (2002) Observations on the dynamic evolution of peer-to-peer networks. In: Proceedings of the first international workshop on peer-to-peer systems (IPTPS'02). Lecture notes in computer science (LNCS), vol 2429. Springer, Berlin Heidelberg New York
26. Lynch NA, Malkhi D, Ratajczak, D (2002) Atomic data access in distributed hash tables. In: Proceedings of the first interational workshop on peer-to-peer systems (IPTPS'02). Lecture notes in computer science (LNCS). Springer, London, pp 295–305
27. Mahajan R, Castro M, Rowstron A (2003) Controlling the cost of reliability in peer-to-peer overlays. In: Proceedings of the 2nd international workshop on peer-to-peer systems (IPTPS'03). Lecture notes in computer science (LNCS), vol 2735. Springer, Berkeley, pp 21–32
28. Manku GS, Bawa M, Raghavan P (2003) Symphony: distributed hashing in a small world. In: Proceedings of the 4th USENIX symposium on internet technologies and systems (USITS'03). USENIX, Seattle, March
29. Montresor A, Jelasity M, Babaoglu Ö (2005) Chord on demand. In: Proceedings of the 5th international conference on peer-to-peer computing (P2P'05). IEEE Computer Society, Los Alamitos, August
30. PINR (2008) Taiwan earthquake on December 2006. http://www.pinr.com/report.php?ac=view_report&report_id=602. Accessd January 2008
31. Oppenheimer D, Ganapathi A, Patterson DA (2003) Why do internet services fail, and what can be done about it? In: USITS'03: proceedings of the 4th conference on USENIX symposium on internet technologies and systems. USENIX Association, Berkeley, pp 1–1
32. Paxson V (1997) End-to-end routing behavior in the internet. IEEE/ACM Trans Netw (TON) 5(5):601–615
33. Plaxton CG, Rajaraman R, Richa, AW (1997) Accessing nearby copies of replicated objects in a distributed environment. In: Proceedings of the 9th annual ACM symposium on parallelism in algorithms and architectures (SPAA'97). ACM, New York, pp 311–320
34. Rowstron A, Druschel P (2001) Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of the 2nd ACM/IFIP international conference on middleware (MIDDLEWARE'01). Lecture notes in computer science (LNCS), vol 2218. Springer, Heidelberg, pp 329–350, November
35. Shafaat TM, Ghodsi A, Haridi S (2007) Handling network partitions and mergers in structured overlay networks. In: Proceedings of the 7th international conference on peer-to-peer computing (P2P'07). IEEE Computer Society, Los Alamitos, pp 132–139, September
36. Shaker A, Reeves DS (2005) Self-stabilizing structured ring topology P2P systems. In: Proceedings of the 5th international conference on peer-to-peer computing (P2P'05). IEEE Computer Society, Los Alamitos, pp 39–46, August
37. SicsSim (2008) http://dks.sics.se/p2p07partition/. Accessed January 2008
38. Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H (2002) Chord: a scalable peer-to-peer lookup service for internet applications. Technical report TR-819, Massachusetts Institute of Technology (MIT), January
39. Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H (2003) Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans Netw (TON) 11(1):17–32
40. Terry DB, Theimer M, Petersen K, Demers AJ, Spreitzer M, Hauser C (1995) Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proceedings of the 15th ACM symposium on operating systems principles (SOSP'95). ACM, New York, pp 172–183, December
41. Voulgaris S, Gavidia D, van Steen M (2005) Cyclon: inexpensive membership management for unstructured p2p overlays. J Netw Syst Manag 13(2):197–217
42. Waldspurger CA, Weihl WE (1994) Lottery scheduling: flexible proportional-share resource management. In: Proceedings of the first symposium on operating systems design and implementation (OSDI'94). USENIX, Seattle, pp 1–11, November

Tallat M. Shafaat is a PhD candidate at KTH - Royal Institute of Technology, Sweden and a member of CSL at SICS - Swedish Institute of Computer Science. Earlier, he completed his B.Sc. at GIK Institute of Engineering Sciences and Technology, Pakistan and an M.Sc. at KTH - Royal Institute of Technology. His research interests include large-scale distributed systems, distributed algorithms and peer-to-peer systems.

Ali Ghodsi is a senior researcher at the Swedish Institute of Computer Science (SICS). He got his PhD in Computer Science from KTH–Royal Institute of Technology in 2006, and an M.B.A. and an M.Sc. from Mid-Sweden University in 2002. His research interest is in distributed computing in general, and in the theory and practice of large-scale dynamic distributed systems in particular.

Seif Haridi received the Ph.D. degree in computer systems from the KTH - Royal Institute of Technology, Sweden, in 1981. He is currently the scientific leader of the Computer Systems Laboratory and the Chief Scientist of the Swedish Institute of Computer Science (SICS) in Kista, Sweden. He is also a Professor of computer systems at the School of Information and Communication Technology, the KTH - Royal Institute of Technology, Sweden.

A.3 Network Size Estimation for Structured Overlays

A Practical Approach to Network Size Estimation for Structured Overlays

Tallat M. Shafaat[1], Ali Ghodsi[2], and Seif Haridi[1]

[1] Royal Institute of Technology (KTH),
School of Information and Communication, Stockholm, Sweden
{tallat,haridi}@kth.se
[2] Computer Systems Laboratory,
Swedish Institute of Computer Science, Stockholm, Sweden
ali@sics.se

Abstract. Structured overlay networks have recently received much attention due to their self-* properties under dynamic and decentralized settings. The number of nodes in an overlay fluctuates all the time due to churn. Since knowledge of the size of the overlay is a core requirement for many systems, estimating the size in a decentralized manner is a challenge taken up by recent research activities. Gossip-based Aggregation has been shown to give accurate estimates for the network size, but previous work done is highly sensitive to node failures. In this paper, we present a gossip-based aggregation-style network size estimation algorithm. We discuss shortcomings of existing aggregation-based size estimation algorithms, and give a solution that is highly robust to node failures and is adaptive to network delays. We examine our solution in various scenarios to demonstrate its effectiveness.

1 Introduction

Structured peer-to-peer systems such as Chord [20] and Pastry [18] have received much attention by the research community recently. These systems are typically very scalable and the number of nodes in the system immensely varies. The network size is, however, a global variable which is not accessible to individual nodes in the system as they only know a subset of the other nodes. This information is, nevertheless, of great importance to many structured p2p systems, as it can be used to tune the rates at which the topology is maintained. Moreover, it can be used in structured overlays for load-balancing purposes [4], deciding successor-lists size for resilience to churn [12], choosing a level to determine outgoing links [14], and for designing algorithms that adapt their actions depending on the system size [1].

Due to the importance of knowing the network size, several algorithms have been proposed for this purpose. Out of these, gossip-based aggregation algorithms [8], though having higher overhead, provide the best accuracy [17]. Consequently, we focus on gossip-based aggregation algorithms in this paper. While

K.A. Hummel and J.P.G. Sterbenz (Eds.): IWSOS 2008, LNCS 5343, pp. 71–83, 2008.

aggregation algorithms can be used to calculate different aggregates, e.g. average, maximum, minimum, variance etc., our focus is on counting the number of nodes in the system.

Although Aggregation [8] provides accurate estimates, it suffers from a few problems. First, Aggregation is highly sensitive to the *overlay topology* that it is used with. Convergence of the estimate to the real network size is slow for non-random topologies. On the contrary, the majority of structured p2p overlays have non-random topologies. Thus, it is not viable to directly use Aggregation in these systems. Second, Aggregation works in rounds, and the estimate is considered converged after a predefined number of rounds. As we discuss in section 4.1, this can be problematic. Finally, Aggregation is highly sensitive to node failures.

In this paper, we suggest a gossip algorithm based on Aggregation to be executed continously on every node to estimate the total number of nodes in the system. The algorithm is aimed to work on structured overlay networks. Furthermore, the algorithm is robust to failures and adaptive to the network delays in the system.

Outline. Section 2 serves as a background for our work. Section 3 describes our solution and discusses how the proposed solution handles the dynamism in the network. Thereafter, section 4 gives a detailed evaluation of our work. Section 5 discusses the related work, and finally, section 6 concludes.

2 Background

In this section, we briefly define a model of a ring-based structured overlay underlying our algorithm. We also describe the original Aggregation algorithm suggested by Jelasity et. al. [8].

2.1 A Model of a Ring-Based Structured Overlay Network

A ring-based structured overlay network consists of nodes which are assigned unique identifiers belonging to a ring of identifiers $\mathcal{I} = \{0, 1, \cdots, N-1\}$ for some large constant N. This is general enough to encompass many existing structured peer-to-peer systems such as Chord[20], Pastry[18] and many others.

Every node has a pointer to its successor, which is the first node met going clockwise on the ring. Every node also has a pointer to its predecessor, which is first node met going anti-clockwise on the ring. For instance, in a ring of size $N = 1024$ containing the nodes $\mathcal{P} = \{10, 235, 903\}$, we have that $succ_{\mathcal{P}}(10) = 235$, $succ_{\mathcal{P}}(903) = 10$, $pred_{\mathcal{P}}(235) = 10$, and $pred_{\mathcal{P}}(10) = 903$.

In this paper, we assume that there exists an out-of-bound mechanism to make all of the predecessor and successor pointers correct. This can, for example, be achieved by using periodic stabilization[20].

Apart from successor and predecessor pointers, each node has additional long pointers in the ring for efficient routing. Different structured overlays use different schemes to place these extra pointers. Our work is independent of how the extra pointers are placed.

While this model looks specific to ring topologies, other structured topologies use similar metrics, for instance, the XOR-metric [16] or butterfly networks [14] Our work can be extended to incorporate metrics other then that employed in ring-based overlays.

2.2 Gossip-Based Aggregation

The Aggregation algorithm suggested by Jelasity et. al. [8] is based on push-pull gossiping, shown in Algorithm 1.

Algorithm 1. Push-pull gossip executed by node p in Aggregation [8]

1: do periodically every δ time units	do forever
2: q ← getneighbour()	s_q ← receive(*)
3: send s_p to q	send s_p to sender(s_q)
4: s_q ← receive(q)	s_q ← update(s_p, s_q)
5: s_q ← update(s_p, s_q)	
(a) Active thread	(b) Passive thread

The method GetNeighbour returns a uniform random sampled node over the entire set of nodes provided by an underlying sampling service like Newscast [7]. The method Update computes a new local state based on the node p's current local state s_p and the remote node's state s_q.

The time interval δ after which the active thread initiates an exchange is called a *cycle*. Given that all nodes use the same value of δ, each node roughly participates in two exchanges in each cycle, one as an initiator and the other as a receipient of an exchange request. Thus, the total number of exchanges in a cycle are roughly equal to $2 \cdot n$, where n is the network size.

For network size estimation, one random node sets its local state to 1 while all other nodes set their local states to 0. The global average is thus $\frac{1}{n}$, where n is the number of nodes. Executing the aggregation algorithm for a number of cycles decreases the variance of local states of nodes but keeps the global average the same. Thus, after convergence, a node p can estimate the network size as $\frac{1}{s_p}$. For network size estimation, Update(s_p, s_q) returns $\frac{s_p+s_q}{2}$.

Aggregation [8] achieves up-to-date estimates by periodically restarting the protocol, i.e. local values are re-initialized and aggregation starts again. This is done after a predefined number of cycles γ, called an epoch.

The main disadvantage of Aggregation is that a failure of a single node early in an epoch can severely effect the estimate. For example, if the node with local state 1 crashes after executing a single exchange, 50% of the value will disappear, giving $2 \cdot n$ as the final size estimate. This issue is further elaborated in section 4.3. Another disadvantage, as we discuss in section 4.1, is predefining the epoch length γ.

3 The Network Size Estimation Algorithm

A naive approach to estimate the network size in a ring-based overlay would be pass a token around the ring, starting from, say node i and containing a variable v initialized to 1. Each node increments v and forwards the token to its successor i.e. the next node on the ring. When the token reaches back at i, v will contain the network size. While this solution seems simple and efficient, it suffers from multiple problems. First, it is not fault-tolerant as the node with the token may fail. This will require complicated modifications for regenerating the token with the current value of v. Second, the naive approach will be quite slow, as it will take $O(n)$ time to complete. Since peer-to-peer systems are highly dynamic, the actual size may have changed completed by the time the algorithm finishes. Lastly, at the end of the naive approach, the estimate will be known only to node i which will have to broadcast it to all nodes in the system. Our solution aims at solving all these problems at the expense of a higher message complexity than the naive approach.

Our goal is to make an algorithm where each node tries to estimate the average inter-node distance, Δ, on the identifier space, i.e. the average distance between two consecutive nodes on the ring. Given a correct value of Δ, the number of nodes in the system can be estimated as $\frac{N}{\Delta}$, N being the size of the identifier space.

Every node p in the system keeps a local estimate of the average inter-node distance in a local variable d_p. Our goal is to compute $\frac{\sum_{i \in \mathcal{P}} d_i}{|\mathcal{P}|}$. The philosophy underlying our algorithm is the observation that at any time the following invariant should always be satisfied: $N = \sum_{i \in \mathcal{P}} d_i$.

We achieve this by letting each node p initialize its estimate d_p to the distance to its successor on the identifier space. In other words, $d_p = succ(p) \ominus p$, where $\ominus$ represents subtraction modulo N. Note that if the system only contains one node, then $d_p = N$. Clearly, a correctly initialized network satisfies the mentioned invariant as the sum of the estimates is equal to N.

To estimate Δ, we employ a modified aggregation algorithm. Since we do not have access to random nodes, we implement the GetNeighbour method in Alg. 1 by returning a node reached by making a random walk of length h. For instance, to perform an exchange, p sends an exchange request to one of its neighbours, selected randomly, with a hop value h. Upon receiving such a request, a node r decrements h and forwards the request to one of its own neighbours, again selected randomly. This is repeated until h reaches 0, after which the exchange takes place between p and the last node getting the request.

Given that GetNeighbour returns random nodes, after a number of exchanges (logarithmic number of steps, to the network size, as show in [8]), every node will have $d_p = \frac{\sum_{i \in \mathcal{P}} d_i}{|\mathcal{P}|}$. On average in each cycle, each node initiates an exchange once, which takes h hops, and replies to one exchange. Consequently, the number of messages for the aggregation algorithm are roughly $hops \times n + n$ per cycle.

3.1 Handling Churn

The protocol described so far does not take into account the dynamicity of large scale peer-to-peer systems. In this section, we present our solution as an extension of the basic algorithm described in section 3 to handle dynamism in the network.

The basic idea of our solution is that each node keeps different levels of estimates, each with a different accuracy. The lowest level estimate is the same as d_n in the basic algorithm. As the value in the lowest level converges, it is moved to the next level. While this helps by having high accuracy in upper levels, it also gives a continuous access to a correct estimated value while the lowest level is re-initialized. Furthermore, we restart the protocol adaptively, instead at a predefined interval.

Our solution is shown in Algorithm 2. Each node n keeps track of the current epoch in *nEpoch* and stores the estimate in a local variable *ndvalue* instead of d_n in the basic algorithm. *ndvalue* is a tuple of size l, i.e.

$$ndvalue = (ndvalue_{l-1}, ndvalue_{l-2}, \cdots, ndvalue_0)$$

The tuple values are called levels. The value at level 0 is the same as d_n in the basic algorithm and has the most recent updated estimate but with high error, while level $l-1$ has the most accurate estimate but incorporates updates slowly.

A node n initializes its estimate, method InitializeEstimate in Alg. 2, by setting level 0 to $succ_{\mathcal{P}}(n) \ominus n$. The method LeftShiftLevels moves the estimate of each level one level up, e.g. left shifting a tuple $e = (e_{l-1}, e_{l-2}, \cdots, e_0)$ gives $(e_{l-2}, e_{l-3}, \cdots, e_0, nil)$. The method Update(a, b) returns an average of each level, i.e. $(\frac{a_{l-1}+b_{l-1}}{2}, \frac{a_{l-2}+b_{l-2}}{2}, \cdots, \frac{a_0+b_0}{2})$.

To incorporate changes in the network size due to churn, we also restart the algorithm, though not after a predefined number of cycles, but adaptively by analyzing the variance. We let the lowest level converge and then restart. This duration may be larger than a predefined γ or less, depending on the system-wide variance in the system of the value being estimated. We achieve adaptivity by using a sliding window at each node. Each node stores values of the lowest level estimate for each cycle in a sliding window W of length w. If the coefficient of variance of the sliding window is less than a desired accuracy e.g. 10^{-2}, the value is considered converged, denoted by the method Converged in Alg. 2.

Once the value is considered to have converged based on the sliding window, there are different methods of deciding which node will restart the protocol, denoted by the method IAmStarter in Alg. 2. One way is as used in [8], each node restarts the protocol with probability $1/\hat{n}$, where $\hat{n}$ is the estimated network size. Given a reasonable estimate in the previous epoch, this will lead to one node restarting the protocol with high probability. It does not matter if more than one node restarts the protocol in our solution. On the contrary, multiple nodes restarting an epoch in [8] is problematic since only one node should set its local estimate to 1 in an epoch. Consequently, an epoch has to be marked with a unique identifier in [8]. Another method is that a node n restarts the protocol which has $0 \in [n, n.succ)$. For our simulations, we use the first method.

Algorithm 2. Network size estimation

```
1: every δ time units at n
2:     if converged() and iamstarter() then
3:         simpleBroadcast(nEpoch)
4:     end if
5:     sendto randomNeighbour() : ReqExchange(hops, nEpoch, ndvalue)
6: end event

7: receipt of ReqExchange(hops, mEpoch, mdvalue) from m at n
8:     if hops > 1 then
9:         hops := hops − 1
10:        sendto randomNeighbour() : ReqExchange(hops, mEpoch, mdvalue)
11:    else
12:        if nEpoch > mEpoch then
13:            sendto m : ResExchange(false, nEpoch, ndvalue)
14:        else
15:            trigger ⟨ MoveToNewEpoch | mEpoch ⟩
16:            ndvalue := update(ndvalue, mdvalue)
17:            updateSlidingWindow(ndvalue)
18:            sendto m : ResExchange(true, nEpoch, ndvalue)
19:        end if
20:    end if
21: end event

22: receipt of ResExchange(updated, mEpoch, mdvalues) from m at n
23:    if updated = false then
24:        trigger ⟨ MoveToNewEpoch | mEpoch ⟩
25:    else
26:        if nEpoch = mEpoch then
27:            dvalue := mdvalues
28:        end if
29:    end if
30: end event

31: receipt of DeliverSimpleBroadcast(mEpoch) from m at n
32:    trigger ⟨ MoveToNewEpoch | mEpoch ⟩
33: end event

34: upon event ⟨ MoveToNewEpoch | epoch ⟩ at n
35:    if nEpoch < mEpoch then
36:        leftShiftLevels()
37:        initializeEstimate()
38:        nEpoch := epoch
39:    end if
40: end event
```

Once a new epoch starts, all nodes should join it quickly. Aggregation [8] achieves this by the logarithmic epidemic spreading property of random networks. Since we do not have access to random nodes, we use a simple broadcast scheme [3] for this purpose, which is both inexpensive ($O(n)$ messages) and fast ($O(\log n)$ steps). The broadcast is best-effort, as even if it fails, the new epoch number is spread through exchanges.

When a new node joins the system, it starts participating in the size estimation protocol when the next epoch starts. This happens either when it receives the broadcast, or its predecessor initializes its estimate. Until then, it keeps forwarding any requests for exchange to a randomly selected neighbour.

Handling churn in our protocol is much simpler and less expensive on bandwidth than other aggregation algorithms. Instead of running multiple epochs as in [8], we rely on the fact that a crash in our system does not effect the end estimate as much as in [8]. This is explored in detail in section 4.3.

4 Evaluation

To evaluate our solution, we implemented the Chord [20] overlay in an event-based simulator [19]. For the first set of experiments, the results are for a network size of 5000 nodes using the King dataset [5] for message latencies. Since we do not have the King dataset for a 5000 node topology, we derive the 5000 node pair-wise latencies from the distance between two random points in a Euclidean space. The mean RTT remains the same as in the King data. This technique is the same as used in [11]. For larger network sizes, the results are for 10^5 nodes using exponentially distributed message latencies with mean 5 simulation time units. For the figures, $\delta = 8 * \textit{mean-com}$ means the cycle length is 8×5.

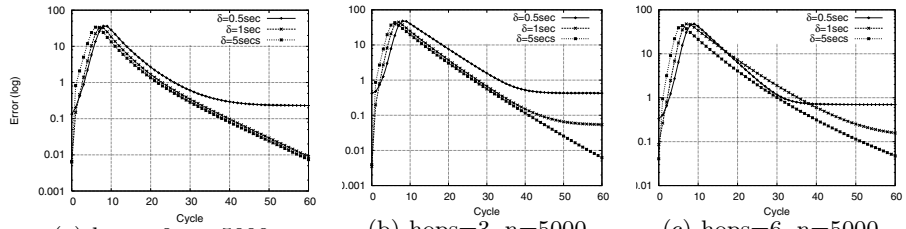
4.1 Epoch Length γ

We investigated the effect of δ on convergence of the algorithm. The results are shown in Figure 1, where error= $\frac{1}{n}\sum_{i=1}^{n}|d_i - \frac{N}{n}|$. It shows that when the ratio between communication delay and δ is significant, e.g. $\delta = 0.5$ *secs* or $8 * \textit{mean-com}$, the aggregate converges slowly and to a value with higher error. For cases where the ratio is insignificant, e.g. $\delta = 5$ *secs* or $24 * \textit{mean-com}$, the convergence is faster and the converged value has lower error. The reason for this behaviour is that when δ is small, the expected number of exchanges per cycle do not occur.

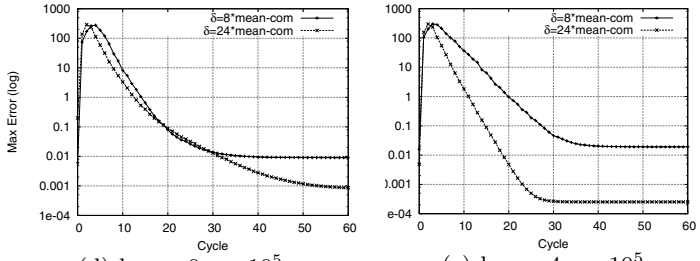
Since δ and γ effect convergence rate and accuracy, our solution aims at having adaptive epoch lengths. Another benefit of using an adaptive approach as ours is that the protocol may converge much before a predefined γ, thus sending messages in vain. If the protocol was restarted, these extra cycles could have been used to get updated aggregate value or include churn effects faster.

4.2 Effect of the Number of Hops

Figure 2 shows convergence of the algorithm for different values of δ and number of hops h to get a random node. For small values of δ, e.g. 0.5 secs and $8 *$

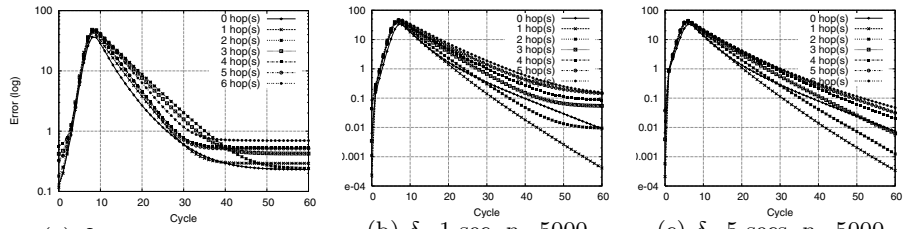


(a) hops=0, n=5000 (b) hops=3, n=5000 (c) hops=6, n=5000

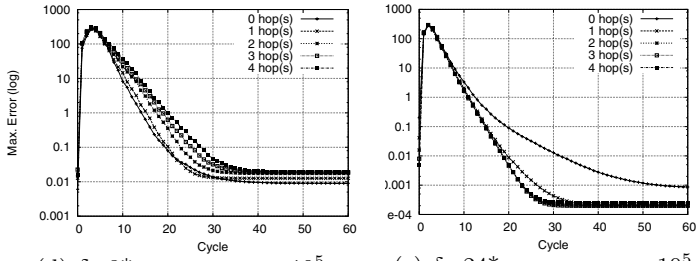


(d) hops=0, n=10^5 (e) hops=4, n=10^5

Fig. 1. Error for the estimate of inter-node distance d in the system



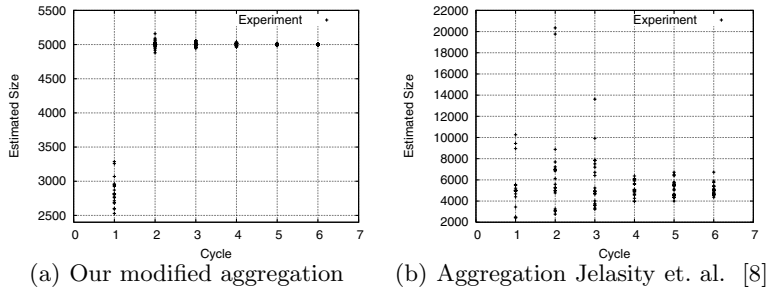
(a) δ=0.5 sec, n=5000 (b) δ=1 sec, n=5000 (c) δ=5 secs, n=5000



(d) δ=8*mean-com, n=10^5 (e) δ=24*mean-com, n=10^5

Fig. 2. Error for the estimate of inter-node distance d in the system

mean-com, $h = 0$ gives best convergence. The reason for this behaviour is that since δ is very small (thus, is comparative to communication delays), having multiple hops will not have enough exchanges in a cycle. Thus, convergence



(a) Our modified aggregation (b) Aggregation Jelasity et. al. [8]

Fig. 3. Estimated size when 50% of the nodes out of 5000 fail suddenly. X-axis gives the cycle at which the sudden death occured in the epoch.

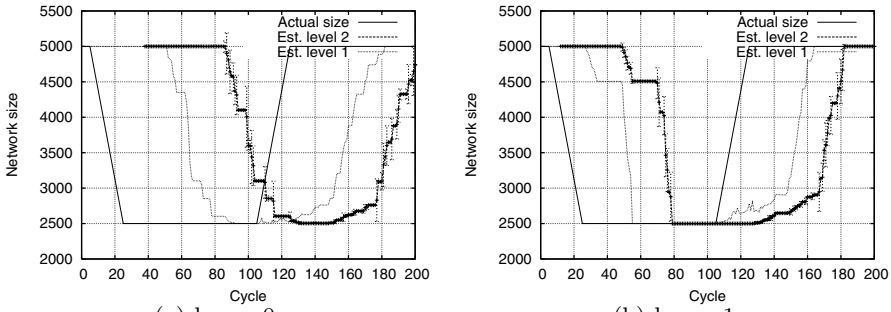
takes longer time and the error is larger for larger values of h. On the contrary, as we increase δ, higher values of h give convergence in lesser time and lower error. These results also advocate to have an adaptive epoch length.

4.3 Churn

Flash Crowds. Next, we evaluated a scenario where a massive node failure occurs. Contrary to [8] where failure of nodes with higher local estimate effects the end estimate more than with lower local estimate, failure of any node is equal in our protocol. The results for a scenario where 50% of the nodes fail at different cycles of an epoch is shown in Figure 3. Our modified aggregation solution, Fig. 3(a), is not as severly affected by the sudden death as the original Aggregation algorithm, fig. 3(b). Infact, in some experiments with Aggregation, the estimate became infinite (not shown in the figure). This happens when all the nodes with non-zero local estimates fail. For our solution, the effect of a sudden death is already negligible if the nodes crash after the third cycle.

Continuous Churn. We ran simulations for a scenario with churn, where nodes join and fail. The results are shown in figure 4, 5 and 7. The results are for extreme churn cases, i.e. 50% nodes fail within a few cycles and afterwards, 50% nodes join within a few cycles. The graphs show how the estimation follows the actual network size. The standard deviation of level 2 is shown as vertical bars, which depicts that all nodes estimate the same size. The standard deviation is high only when a new epoch starts, because while evaluating the mean and standard deviation, some nodes have moved to the new epoch, while others are still in the older epoch. The estimate at level 1 converges to the actual size faster than level 2, but the estimates has higher variance as the standard deviation for level 1 (not shown) is higher than for level 2. Figure 5 also shows that compared to $h = 0$, higher values of h follow the trend of the actual size faster.

Next, we simulated a network of size 4500 and evaluated our algorithm under continous churn. In each cycle, we failed some random nodes and joined new nodes. As explained in section 3, new nodes do not participate in the algorithm



(a) hops=0 (b) hops=1

Fig. 4. Mean estimated size by each node with standard deviation

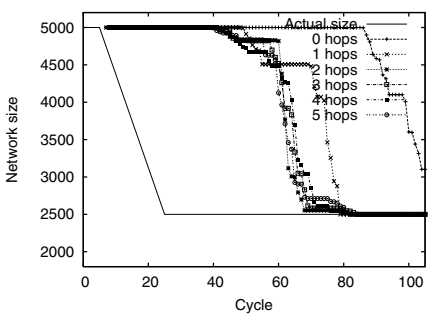


Fig. 5. Mean estimated size for different values of h for level 2. Gives a comparison of how fast the nodes estimation follows the real network size.

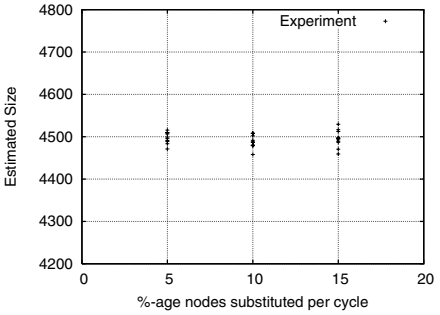


Fig. 6. Estimated network size for 4500 nodes under continuous churn. X-axis gives the percentage of churn events (joins+failures) that occur in each cycle.

till the next epoch starts, yet they can forward requests. Figure 6 shows the results. The plotted dots correspond to the converged mean estimate after 15 cycles for each experiment. The x-axis gives the percentage of churn events, including both failures and joins, that occur in each cycle. Thus, 10% means that $4500 \times \frac{10}{100} \times 15$ churn events occured before the plotted converged value. Figure 6 shows that the algorithm handles continous churn reasonably well.

5 Related Work

Network size estimation in the context of peer-to-peer systems is challenging as these systems are completely decentralized, nodes may fail anytime, the network size varies dynamically over time, and the estimation algorithm needs to continuously update its estimation to reflect the current number of nodes.

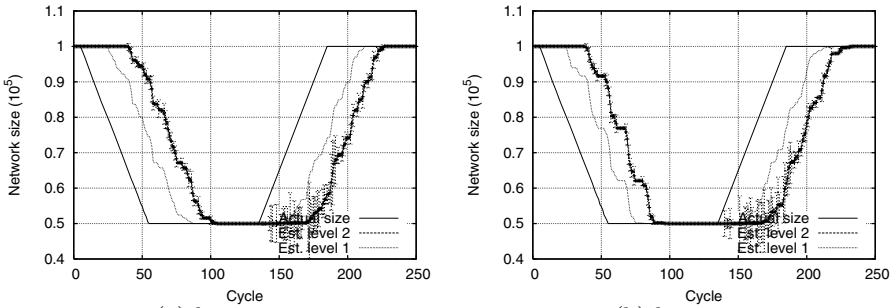


Fig. 7. Mean estimated size by each node with standard deviation

Merrer et. al. [17] compare three existing size estimation algorithms, Sample & Collide [15], Hops Sampling [10] and Aggregation [8], which are representative of three main classes of network size estimation algorithms. Their study yields that although Aggregation is expensive, it produces the most accurate results. Aggregation also has the additional benefit that the estimamte is available on all nodes compared to only at the initiator in the case of Sample & Collide and Hops Sampling. Our work can be seen as an extension of Aggregation, to handle its shortcomings and extend it to non-random topologies, such as structured overlay networks.

The work by Horowitz et. al. [6] is similar to ours in the sense that they also utilize the structure of the system. They use a localized probabilistic technique to estimate the network size by maintaining a structure: a logical ring. Each node estimates the network size locally based on the estimates of its neighbours on the ring. While their technique has less overhead, the estimates are not accurate, the expected accuracy being in the range $n/2 \cdots n$. Their work has been extended by Andreas et. al. [2] specifically for Chord, yet the extended work also suffers similar inaccuracy range for the estimated size. Mahajan et. al. [13] also estimate the network size through the density of node identifiers in Pastry's leafset, yet they neither prove any accuracy range, nor provide any simulation results to show the effectiveness of their technique.

Kempe et. al. [9] have also suggested a gossip-based aggregation scheme, yet their solution focuses only on push-based gossiping. Using push-based gossiping complicates the update and exchange process as a normalization factor needs to be kept track of. On the same, as noted by Jelasity et. al. [8], push-based gossiping suffers from problems when the underlying directed graph used is not strongly connected. Thus, we build our work on push-pull gossip-based aggregation [8]. Similarly, to estimate the network size, Kempe et. al. also propose that one node should initialize its weight to 1, while the other nodes initialize to weight 0, making it highly sensitive to failures early in the algorithm.

The authors of Viceroy [14] and Mercury [1] mention that a nodes distance to its successor can be used to calculate the number of nodes in the system,

but provide no reasoning that the value always converges exactly to the correct value, and thus that their estimate is unbiased.

6 Conclusion

Knowledge of the current network size of a structured p2p system is a prime requirement for many systems, which prompted to finding solutions for size estimation. Previous studies have shown that gossip-based aggregation algorithms, though being expensive, produce accurate estimates of the network size. We have demonstrated the shortcomings in existing aggregation approaches to network size estimation and have presented a solution that overcomes the deficiencies. In this paper, we have argued for an adaptive approach to convergence in gossip-based aggregation algorithms. Our solution is resilient to massive node failures and is aimed to work on non-random topologies such as structured overlay networks.

References

1. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: Supporting Scalable Multi-Attribute Range Queries. In: Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols, OR, USA. ACM Press, New York (2004)
2. Binzenhöfer, A., Staehle, D., Henjes, R.: On the fly estimation of the peer population in a chord-based p2p system. In: 19th International Teletraffic Congress (ITC19), Beijing, China (September 2005)
3. Ghodsi, A.: Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden (December 2006)
4. Godfrey, P.B., Stoica, I.: Heterogeneity and Load Balance in Distributed Hash Tables. In: Proc. of the 24th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM 2005), FL, USA. IEEE Comp. Society, Los Alamitos (2005)
5. Gummadi, K.P., Saroiu, S., Gribble, S.D.: King: estimating latency between arbitrary internet end hosts. In: IMW 2002: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment, pp. 5–18. ACM, New York (2002)
6. Horowitz, K., Malkhi, D.: Estimating network size from local information. Information Processing Letters 88(5), 237–243 (2003)
7. Jelasity, M., Kowalczyk, W., van Steen, M.: Newscast Computing. Technical Report IR–CS–006, Vrije Universiteit (November 2003)
8. Jelasity, M., Montresor, A., Babaoglu, Ö.: Gossip-based Aggregation in Large Dynamic Networks. ACM Trans. on Computer Systems (TOCS) 23(3) (August 2005)
9. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: 44th Symp. on Foundations of Computer Science, FOCS (2003)
10. Kostoulas, D., Psaltoulis, D., Gupta, I., Birman, K., Demers, A.J.: Decentralized schemes for size estimation in large and dynamic groups. In: 4th IEEE International Symp. on Network Computing and Applications (NCA 2005), pp. 41–48 (2005)

11. Li, J., Stribling, J., Morris, R., Kaashoek, M.F.: Bandwidth-efficient management of DHT routing tables. In: Proc. of the 2nd USENIX Symp. on Networked Systems Design and Implementation (NSDI 2005), MA, USA, May 2005, USENIX (2005)
12. Liben-Nowell, D., Balakrishnan, H., Karger, D.R.: Analysis of the Evolution of Peer-to-Peer Systems. In: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC 2002), pp. 233–242. ACM Press, New York (2002)
13. Mahajan, R., Castro, M., Rowstron, A.: Controlling the Cost of Reliability in Peer-to-Peer Overlays. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 21–32. Springer, Heidelberg (2003)
14. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A scalable and dynamic emulation of the butterfly. In: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC 2002). ACM Press, New York (2002)
15. Massoulié, L., Merrer, E.L., Kermarrec, A., Ganesh, A.J.: Peer counting and sampling in overlay networks: random walk methods. In: Proc. of the 25th Annual ACM Symp. on Principles of Distributed Computing (PODC), pp. 123–132 (2006)
16. Maymounkov, P., Mazieres, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR metric. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 53–65. Springer, Heidelberg (2002)
17. Merrer, E.L., Kermarrec, A.-M., Massoulie, L.: Peer to peer size estimation in large and dynamic networks: A comparative study. In: Proc. of the 15th IEEE Symposium on High Performance Distributed Computing, pp. 7–17. IEEE, Los Alamitos (2006)
18. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
19. SicsSim (2008), `http://dks.sics.se/iwsos08sizeest/`
20. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proceedings of the ACM SIGCOMM 2001 Symposium on Communication, Architecture, and Protocols, San Deigo, CA, August 2001, pp. 149–160. ACM Press, New York (2001)

A.4 The Relaxed-Ring: a Fault-Tolerant Topology for Structured Overlay Networks

Parallel Processing Letters

THE RELAXED-RING: A FAULT-TOLERANT TOPOLOGY FOR STRUCTURED OVERLAY NETWORKS

BORIS MEJÍAS AND PETER VAN ROY
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

Received May 2008
Revised July 2008
Communicated by P. Fragopoulou

ABSTRACT

Fault-tolerance and lookup consistency are considered crucial properties for building applications on top of structured overlay networks. Many of these networks use the ring topology for the organization or their peers. The network must handle multiple joins, leaves and failures of peers while keeping the connection between every pair of successor-predecessor correct. This property makes the maintenance of the ring very costly and temporarily impossible to achieve, requiring periodic stabilization for fixing the ring. We introduce the relaxed-ring topology that does not rely on a perfect successor-predecessor relationship and it does not need any periodic maintenance. Leaves and failures are considered as the same type of event providing a fault-tolerant and self-organizing maintenance of the ring. Relaxed-ring's limitations with respect to failure handling are formally identified, providing strong guarantees to develop applications on top of the architecture. Besides permanent failures, the paper analyses temporary failures and false suspicions caused by broken links, which are often ignored.

Keywords: peer-to-peer, network topology, relaxed-ring, self-configuration, fault-tolerance

1. Introduction

Building decentralized applications requires several guarantees from the underlaying peer-to-peer network. Fault-tolerance and consistent lookup of resources are crucial properties that a peer-to-peer system must provide. Structured overlay networks using a Chord-like ring topology [15] are a popular choice when the application needs efficient routing, lookup consistency and accessibility of resources. According to [7], the ring topology is one of the most resilient to failures, and it is competitive with any other structured overlay network with respect to reaching any other node in a small amount of steps.

The ring topology offers many good properties as we just mentioned, but its maintenance presents several challenges in order to provide lookup consistency at any time. Chord itself presents *temporary inconsistencies* with peers massively join-

ing the network, even in fault-free environments, as we will discuss in section 4. A stabilization protocol must be run periodically to fix these inconsistencies increasing the load of the network. One possible solution is presented by DKS [6], offering an atomic join/leave algorithm based on a locking mechanism. Even when this approach offers strong guarantees, we consider locks extremely restrictive for a dynamic network based on asynchronous communication. Every lookup request involving the critical range of keys must be suspended in presence of a join/leave event in order to guarantee consistency. Leaving peers are not allowed to leave the network until they are granted the relevant locks. Given that, crashing peers just leave the network without respecting the protocol of the locking mechanism breaking the guarantees of the system. Another critical problem for performance is presented when a peer crashes while some joining or leaving peer is holding its lock.

The problem with the maintenance of the ring is that routing algorithms and correctly assigning responsibilities over keys, rely on the perfect relationship between predecessor and successor. But, every join, leave or failure, brakes temporary this relationship. In order to solve this problem, existing algorithms require the agreement of three nodes to perform a *join* or *leave* operation. Managing three nodes at the same time can create unexpected problems, mainly because transitivity of communication cannot be assumed. If node a can talk to b, and b can talk to c, it does not mean that a can talk to c. This problem is equivalent to false suspicions of failure. Consider that a, b and c are talking to each other. Suddenly, the connection between a and c is broken. Peer a informs b about this failure, but b sees that c is alive, meaning that a falsely suspected of c. This is why algorithm based on the synchronized agreement of three nodes are not fault-tolerant. A recent work [14] conclude that lookup inconsistencies are mainly caused by false suspicions. Churn does not introduce inconsistencies if periodic stabilization is triggered often enough, but this very costly as it will be shown in Section 4.

The contribution of this work is an algorithm that only needs the agreement of two nodes at each stage of the maintenance of the ring. By working with only two nodes in every step we have arrived to the Relaxed-Ring topology, which allows a ring to be partially open. This approach simplifies the joining algorithm dividing it into two steps involving two peers each. Lookup consistency is guaranteed after every step. The algorithm provides a failure recovery mechanism where only two nodes interact in every step. Graceful leaves are consider a special case of failure, and therefore, they are equivalent events. Because of this, there is no need for a graceful leaving protocol. This is useful for end-users, because they can just shut down their application letting the network to handle their departure. Fault-tolerance is achieved at the level of permanent failures, temporary failures and false suspicions, which results from broken links, which are often ignored.

The Relaxed-Ring adds robustness to the network in presence of churn and failures. It simplifies the arrival and departure of peers by dividing these event into smaller and correct steps. Due to the relaxed topology, the routing performance is shortly degraded. On the other hand, there is no need for periodic maintenance of

the ring, because the Relaxed-Ring remains correct after every step, reducing the cost of maintenance. Part of this contribution has been published in [11], where the Relaxed-Ring was presented from the point of view of its design as a self-managing system. This work is focused on the correctness of the algorithm through analytical results, and an empirical validation with simulations.

The next section gives more details of the related work. Section 3 describes the relaxed-ring architecture and its guarantees. We continue with further analysis of the topology. Evaluation of the relaxed-ring is presented in section 4, ending with conclusions and future work.

2. The problem and related work

Chord is the canonical structured overlay network using ring topology. Its algorithms for ring maintenance handling joins and leaves have been already studied [6] showing problems of temporary inconsistent lookups, where more that one node appears to be the responsible for the same key. Peers need to trigger periodic stabilization in order to fix inconsistencies. Existing analyses conclude that the problem comes from the fact that joins and leaves are not atomic operations, and they always need the synchronization of three peers, which is hard to guarantee with asynchronous communication, which is inherent to distributed programming.

Existing solutions [8, 9] introduce locks in the algorithms in order to provide atomicity of the join and leave operations, removing the need for a periodic stabilization. Unfortunately, locks are also hard to manage in asynchronous systems, and that is why these solutions only work on fault-free environments, which is not realistic. Another problem with these approaches is that they are not starvation-free, and therefore, it is not possible to guarantee liveness. A better solution using locks is provided by Ghodsi [6], using DKS [2] for its results. This approach is better because it gives a simpler design for a locking mechanism and proves that no deadlock occurs. It also guarantees liveness by proving that the algorithm is starvation-free. Unfortunately, the proofs are given in fault-free environments.

The DKS algorithm for ring maintenance goes already in the right direction because it request the locks of only two peers instead of three (as in [8, 9]). It works as follows. Every peer holds a lock that can be exclusively taken by any peer. The lock grants access to update the pointers of the peer. A joining/leaving peer needs to get its own lock and its successor's lock. Let us consider peer q joining in between p and r. Peer q first has to get its own lock and then the lock of r, its successor candidate. This is sufficient for q to update its predecessor and successor, to update r's predecessor, and to update p's successor pointer. Note that p cannot change its pointers because that would require getting r's lock, which is already taken by q. The situation of q leaving is analogous, with the difference that p show acquires q's lock in order to perform any action. This mechanism guarantees that if the relevant locks are acquired, the join/leave can be performed *atomically.*

One of the problem with the algorithm is that even when it only requires the

lock of two peers, it still requires the *atomic* update of the pointers of *three* peers. While this three changes are made, no lookup involving peers p, q or r is allowed. A more complex problem is that the algorithm relies on peers gracefully leaving the ring, which is neither efficient nor fault-tolerant. The algorithm becomes very slow if a peer holding a relevant lock crashes. How can the other peers continue? The same problem occurs if a locked peer stop responding. Another problems is that a joining peer q that acquires its own lock and r's lock, is not guaranteed to establish communication with p in order to change its successor pointer.

We are not aware of other approaches solving the problem of atomic join/leave with failure recovery, or other approaches targeting the elimination of periodic stabilization.

3. The Relaxed-Ring

The relaxed-ring topology has evolved from the Peer-to-Peer System (P2PS) [5], and it is implemented using the Mozart-Oz programming system [13]. As any overlay network built using ring topology, in our system every peer has a successor, predecessor, and fingers to jump to other parts of the ring in order to provide efficient routing. Ring's key-distribution is formed by integers from 0 to $N-1$ growing clockwise. For the description of the algorithms we will use event-driven notation. When a peer receives a message, the message is triggered as an event in the ring maintenance tier.

Range between keys, such as $(p, q]$ follows the key distribution clockwise, so it is possible that $p > q$, and then the range goes from p to q passing through 0. Parentheses '(' and ')' excludes a key from the range and, '[' and ']' includes it.

As we previously mentioned, one of the problem we have observed in existing ring maintenance algorithms is the need for an agreement between three peers to perform a join/leave action. We provide an algorithm where every step only needs the agreement of two peers, which is guaranteed with a point-to-point communication. In the specific case of a join, instead of having one step involving 3 peers, we have two steps involving 2 peers. Lookup consistency is guaranteed after every step, therefore, the network can still answer lookup requests while simultaneous nodes are joining the network. Another relevant difference is that we do not rely on graceful leaving of peers. We treat leaves and failures as the same event. This is because failure handling already includes graceful leaves as a particular case.

Normally the overlay is a ring with predecessor and successor knowing each other. If a new node joins in between these two peers, it introduces two changes. The first one is to contact the successor. This step already allows the new peer to be part of the network through its successor. The second step, contacting the predecessor, will close the ring again. Following this reasoning, our first invariant is that *every peer is in the same ring as its successor*. Therefore, it is enough for a peer to have connection with its successor to be considered inside the network. Secondly, the responsibility of a peer starts with the key of its predecessor, excluding

predecessor's key, and it finishes with its own key. Therefore, a peer does not need to have connection with its predecessor, but it must know its key. These are two crucial properties that allow us to introduce the relaxation of the ring. When a peer cannot connect to its predecessor, it forms a branch from the *"perfect ring"*. Figure 1 shows a fraction of a relaxed ring where peer t is the root of a branch, and where the connection between peers q and p is broken.

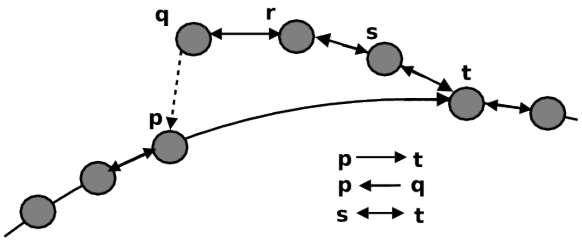


Fig. 1. A branch on the relaxed-ring created because peer q cannot establish communication with p. Peers p and s consider t as successor, but t only considers s as predecessor.

Having the relaxed-ring architecture, we introduce a new principle that modifies the routing mechanism of Chord. The principle is that *a peer p always forwards the lookup request to the possible responsible, even if p is the predecessor of such responsible.* Considering the example in figure 1, p may think that t is the responsible for keys in the interval $(p, t]$, but in fact, there are three other nodes involved in this range. In Chord, p would just reply t as the result of a lookup for key q. In the Relaxed-Ring, the p forwards the message to t. When the message arrives to node t, it is sent backwards to the branch, until it reaches the real responsible. Forwarding the request to the responsible is a conclusion we have already presented in [11], and it has been recently confirmed by Shafaat [14].

Introducing branches into the lookup mechanism modifies the guarantees about proximity offered by Chord. Reaching the root of a branch takes $O(log_k(n))$ hops as in Chord, because the root of the branch belongs to the *core-ring*. Then, the lookup will be delegated a maximum of b hops, where b corresponds to the size of the branch. Then, lookup on the relaxed-ring topology corresponds to $log_k(n) + b$. We will see in section 4 that the average value b is smaller than 1 for large networks.

Before continuing with the description of the algorithms that maintain the relaxed-ring topology, let us define what do we mean by lookup consistency.

Def. *Lookup consistency implies that at any time there is only one responsible for a particular key k, or the responsible is temporary not available.*

Algorithm 1 describes the initial procedure of a node that wants to join the ring. First, it gets its own identifier from a random key-generator. In the implementation, identifiers also represent network references. For simplicity of the description of the algorithms, we will just use the key as identifier and as connection reference. Initially,

the node does not have a successor (*succ*), so it does not belong to any ring, and it does not know its predecessor (*pred*), so obviously, it does not have responsibilities. For resilient purposes, the node uses two sets: a successor list (*succlist*) and an old-predecessor sets (*predlist*). Having an access point, that can be any peer of the ring, the new peer triggers a lookup request for its own key in order to find its best successor candidate. This is quite usual procedure for many Chord-alike systems. When the responsible of the key contacts the new peer, the event *reply_lookup* is triggered in the new peer. This event will generate a joining message that will be discussed in section 3.1.

Algorithm 1 Starting a peer and the lookup algorithm

```
1: procedure init(accesspoint) is
2:     self := getRandomKey()
3:     succ := nil
4:     pred := nil
5:     predlist := ∅
6:     succlist := ∅
7:     send ⟨ lookup | self, self ⟩ to accesspoint
8: end procedure

9: upon event ⟨ lookup | src, key ⟩ do
10:    if (key ∈(pred, self]) then
11:        send ⟨ reply_lookup | self ⟩ to src
12:    else
13:        p := getBetterResponsible(key)
14:        send ⟨ lookup | src, key ⟩ to p
15:    end if
16: end event

17: upon event ⟨ reply_lookup | i ⟩ do
18:    send ⟨ join | self ⟩ to i
19: end event
```

The *lookup* event verifies if the current node is responsible for *key*. If it is not, it picks the best responsible for the key from its routing table, and forwards the request, passing the key and the original source *src*. Choosing the best responsible of a key follows the same mechanism as Chord, with the extra consideration of rooting to the branch when needed, as explained above. One way to decide that a lookup must jump into the branch is by adding a flag to the message called *last*. In the case of Figure 1, when p forwards the messages to t, it sets the flag to *true*. Then, the function *getBetterResponsible* will decide to forward to the predecessor, jumping in to the branch.

3.1. *The join algorithm*

As we have previously mentioned, the relaxed-ring join algorithm is divided in two steps involving two peers each, instead of one step involving three peers as in existing solutions. The whole process is depicted in figure 2, where node q joins in between peers p and r. Following algorithm 1, r replies the lookup to q, and q send the *join* message to r triggering the joining process.

The first step is described in algorithm 2, and following the example, it involves peer q and r. This step consists of two events, *join* and *join_ok*. Since this event may happen simultaneously with other joins or failures, r must verify that it has a successor, respecting the invariant that every peer is in the same ring as its successor. If it is not the case, q will be requested to retry later.

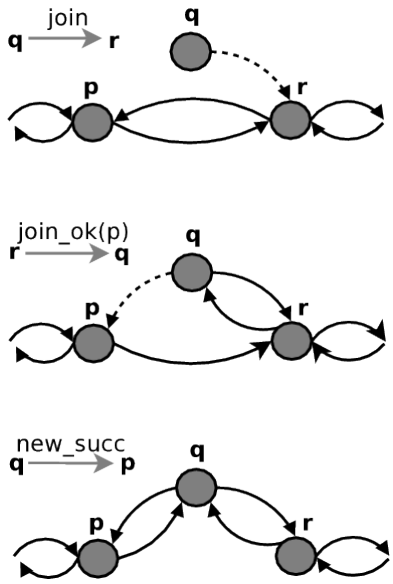


Fig. 2. The join algorithm.

If it is possible to perform the join, peer r verifies that peer q is better predecessor than p. Function *betterPredecessor* checks if the key of the joining peer is in the range of responsibility of the successor candidate. In the example, r verifies that $q \in (p, r]$. If that is the case, p becomes the old predecessor and is added to the *predlist* for resilient purposes. The *pred* pointer is set to the joining peer, and the message *join_ok* is send to it.

It is possible that the responsibility of r has change between the events *reply_lookup* and *join*. In that case, q will be redirected to the corresponding peer with the *goto* message, eventually converging to the responsible of its key.

When the event *join_ok* is triggered in the joining peer q, the *succ* pointer is set to r and *succlist* is initialized. Then, q must set its *pred* pointer to p acquiring its range of responsibility. At this point the joining peer has a valid successor and a range of responsibility, and then, it is considered to be part of the ring, even if p is not yet notified about the existence of q. This is different than all other ring networks we have studied.

Note that before updating the predecessor pointer, peer q must verify that its predecessor pointer is *nil*, or that it belongs to its range of responsibility. This second condition is only used in case of failure recovery and it will be described in section 3.3. In a regular join, *pred* pointer at this stage is always *nil*.

Once q set *pred* to p, it notifies p about its existence with message *new_succ*, triggering the second step of the algorithm.

The second step of the join algorithm basically involves peers p and q, closing the ring as in a regular ring topology. The step is described in algorithm 3. The idea is that when p is notified about the join of q, it updates its successor pointer to q (after verifying that is a correct join), and it updates its successor list with the new information. Functionally, this is enough for closing the ring. An extra event has been added for completeness. Peer p acknowledges its old successor r, about the join of q. When *join_ack* is triggered at peer r, this one can remove p from the resilient *predlist*.

If there is a communication problem between p and q, the event *new_succ* will never be triggered. In that case, the ring ends up having a branch, but it is still able to resolve queries concerning any key in the range $(p, r]$. This is because q has a valid successor and its responsibility is not shared with any other peer. It is important to remark the fact that branches are only introduced in case of communication problems. If q can talk to p and r, the algorithm provides a perfect ring.

No distinction is made concerning the special case of a ring consisting in only one node. In such a case, *succ* and *pred* will point to $self$ and the algorithm works identically. The algorithm works with simultaneous joins, generating temporary or permanent branches, but never introducing inconsistencies. Failures are discussed in section 3.3. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

Theorem 1. The relaxed-ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.

Proof.

Let us assume the contrary. There are two peers p and q responsible for key k. If p and q have the same successor is not relevant, because both peers would forward the lookup to the successor, and the successor can resolve the conflict. The problem is when p and q have the same predecessor j, sharing the same range of responsibility. This means that $k \in (j, p]$ and $k \in (j, q]$ introducing a inconsistency because of the overlapping or ranges. Let us see now that the algorithm prevents two nodes from

Algorithm 2 Join step 1 - adding a new node

```
1: upon event ⟨ join | i ⟩ do
2:     if succ == nil then
3:         send ⟨ try_later | self ⟩ to i
4:     else
5:         if betterPredecessor(i) then
6:             oldp := pred
7:             pred := i
8:             predlist := {oldp} ∪ {predlist}
9:             send ⟨ join_ok | oldp, self, succlist ⟩ to i
10:        else if (i < pred) then
11:            send ⟨ goto | pred ⟩ to i
12:        else
13:            send ⟨ goto | succ ⟩ to i
14:        end if
15:    end if
16: end event

17: upon event ⟨ join_ok | p, s, sl ⟩ do
18:    succ := s
19:    succlist := {s} ∪ sl \ getLast(sl)
20:    if (pred == nil) ∨ (p ∈ (pred, self)) then
21:        pred := p
22:        send ⟨ new_succ | self, succ, succlist ⟩ to pred
23:    end if
24: end event

25: upon event ⟨ goto | j ⟩ do
26:    send ⟨ join | self ⟩ to j
27: end event
```

having the same predecessor. The join algorithm updates the predecessor pointer upon events *join* and *join_ok*. In the event *join*, the predecessor is set to a new joining peer j. This means that no other peer was having j as predecessor because it is a new peer. Therefore, this update does not introduce any inconsistency. Upon event *join_ok*, the joining peer j initiates its responsibility having a member of the ring as predecessor, say i. The only other peer that had i as predecessor before is the successor of j, say p, which is the peer that triggered the *join_ok* event. This message is sent only after p has updated its predecessor pointer to j, and thus, modifying its responsibility from $(i, p]$ to $(j, p]$, which does not overlap with j's responsibility $(i, j]$. Therefore, it is impossible that two peers has the same predecessor. □

Algorithm 3 Join step 2 - Closing the ring

```
1: upon event ⟨ new_succ | s, olds, sl ⟩ do
2:     if (succ == olds) then
3:         oldsucc := succ
4:         succ := s
5:         succlist := {s} ∪ sl \ getLast(sl)
6:         send ⟨ join_ack | self ⟩ to oldsucc
7:         send ⟨ upd_succlist | self, succlist ⟩ to pred
8:     end if
9: end event

10: upon event ⟨ join_ack | op ⟩ do
11:     if (op ∈ predlist) then
12:         predlist := predlist \ {op}
13:     end if
14: end event
```

3.2. *Reducing size of branches*

Let us consider again figure 1. If nodes keeps on joining as predecessors of peer t, the branch will increase its size, even if they could have a good connection with peer p. An improvement on the join algorithm will be that node t sends a *hint* message to node p avoid new joining peer. If p cannot talk to q, it does not mean that it can not talk to r or s. If the p can contact the *hinted* node, it will add it as its successor, making the branch shorter. This hint message will not modify the predecessor pointers of r or s. Peer t uses its *predlist* list for sending hints.

3.3. *Failure Recovery*

In order to provide a robust system that can be used on the Internet, it is unrealistic to assume a fault-free environment or perfect failure detectors, meaning complete and accurate. We assume that every faulty peer will eventually be detected (strongly complete), and that a broken link of communication does not implies that the other peer has crashed (inaccurate). To terminate failure recovery algorithms we assume that eventually any inaccuracy will disappear (eventually strongly accurate). This kind of failure detectors are feasible to implement on the Internet.

When the point-to-point communication layer detects a failure of one of the nodes, the *crash* event is triggered as it is described in algorithm 4. The detected node is removed from the resilient sets *succlist* and *predlist*, and added to a *crashed* set. If the detected peer is the successor, the recovery mechanism is triggered. The *succ* pointer is set to *nil* to avoid other peers joining while recovering from the failure, and the successor candidate is taken from the successors list. The variable

succ_candidate should be initialized to *nil* in the *init* event of Algorithm 1, but it was not included to avoid confusion at that part of the analysis of the algorithm. The real value is initialized at line 7 of the *crashed* event. The function *getFirst* returns the peer with the first key found clockwise, and removes it from the set. It returns *nil* if the set is empty. Function *getLast* is analogous. Note that as every crashed peer is immediately removed from the resilient sets, these two functions always return a peer that appears to be alive at this stage. The successor candidate is contacted using the *join* message, triggering the same algorithm as for joining. If the successor candidate also fails, a new candidate will be chosen. This is verified in the *if* condition.

If a peer p detects that its predecessor *pred* has crashed, it will not trigger the recovery mechanism. It is *pred*'s predecessor who will contact p. In case that no peer contacts p for recovery, p could guess a predecessor candidate from its *predlist*, at the risk of breaking lookup consistency, but closing the ring again. We will not explore this case further in this paper because it does not violate our definition of consistent lookup. To solve it, it is necessary to set up a time-out to replace the faulty predecessor by the predecessor candidate, but it would always take the risk of a reacting to a false suspicion.

When a link recovers from a temporary failure, the *alive* event is triggered. This can be implemented by using watchers or a fault stream per distributed entity [4]. In this case, it is enough to remove the peer from the *crashed* set. This will terminate any pending recovery algorithm. The faulty peer will trigger by itself the corresponding recovery events with the relevant peers.

Algorithm 4 Failure recovery

```
1:  upon event ⟨ crash | p ⟩ do
2:      succlist := succlist \ {p}
3:      predlist := predlist \ {p}
4:      crashed := {p} ∪ crashed
5:      if (p == succ) ∨ (p == succ_candidate) then
6:          succ := nil
7:          succ_candidate := getFirst(succlist)
8:          send ⟨ join | self ⟩ to succ_candidate
9:      end if
10: end event

11: upon event ⟨ alive | p ⟩ do
12:     crashed := crashed \ {p}
13: end event
```

Figure 3 shows the recovery mechanism triggered by a peer when it detects that its successor has a failure. The figure depicts two equivalent situations. The above

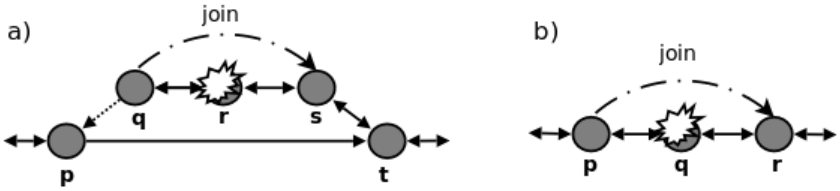


Fig. 3. Failures simple to handle: (a) In a branch, q and s detect that r has crashed. Only q triggers failure recovery. (b) Pers p and r detects q has crashed. Peer p triggers the recovery mechanism.

one corresponds to a regular crash of a node in a perfect ring. The situation below shows that a crash in a branch is equivalent as long as there is a predecessor that detects the failure.

Having now the knowledge of the *crashed* set, algorithm 5 gives complete definition of the function *betterPredecessor* used in algorithm 2. Since the *join* event is used both for a regular join and for failure recovery, the function will decides if a predecessor candidate is better than the current one if it belongs to its range of responsibility, or if the current *pred* is detected as a faulty peer.

Algorithm 5 Verifying predecessor candidate

```
1: function betterPredecessor(i) is
2:     if (i ∈ (pred, self)) then
3:         return (true)
4:     else
5:         return (pred ∈ crashed)
6:     end if
7: end function
```

Knowing the recovery mechanism of the relaxed-ring, let us come back to our joining example and check what happens in cases of failures. If q crashes after the event *join*, peer r still has p in its *predlist* for recovery. If q crashes after sending *new_succ* to p, p still has r in its *succlist* for recovery. If p crashes before event *new_succ*, p's predecessor will contact r for recovery, and r will inform this peer about q. If r crashes before *new_succ*, peers p and q will contact simultaneously r's successor for recovery. If q arrives first, everything is in order with respect to the ranges. If p arrives first, there will be two responsible for the ranges $(p, q]$, but one of them, q, is not known by any other peer in the network, and it fact, it does not have a successor, and then, it does not belong to the ring. Then, no inconsistency is introduced in any case of failure. In case of a network partition, these peers will get divided in two or three groups depending on the partition. In such case, they will continue with the recovery algorithm in their own rings. Global consistency is

impossible to achieve, but every ring will be consistent in itself.

Since failures are not detected by all peers at the same time, redirection during recovery of failures may end up in a faulty node. The correct version of the *goto* event is described in algorithm 6. If a peer is redirected to a faulty node, it must insist with its successor candidate. Since failure detectors are strongly complete, the algorithm will eventually converge to the correct peer.

Algorithm 6 Modified goto

```
1: upon event ⟨ goto | p ⟩ do
2:     if (p ∉ crashed) then
3:         send ⟨ join | self ⟩ to p
4:     else
5:         send ⟨ join | self ⟩ to succ_candidate
6:     end if
7: end event
```

Figure 4 shows two simultaneous crashes together with a new peer joining before the peer used for recovery. If the recovery *join* message arrives first, the ring will be fixed before the new peer joins, resulting in a regular join. If the new peer starts the first step of joining before the recovery, it will introduce a temporary branch because of its impossibility of contacting the faulty predecessor. When the recovery *join* message arrive, the recovering peer will contact the new joining peer, fixing the ring and removing the branch.

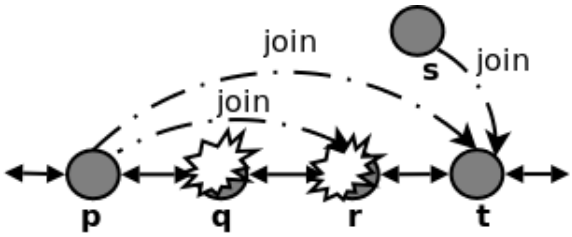


Fig. 4. Multiple failure recovery and simultaneous join. Peer p detects the crash of its successor q. First successor candidate r has also crashed. Peer p contacts t at the same time peer s tries to join the network. Both *join* messages are the same.

There are failures more difficult to handle than the ones we have already analysed. Figure 5 depicts a broken link and the crash of the tail of a branch. In the case of the broken link (inaccuracy), the failure recovery mechanism is triggered, but the successor of the suspected node will not accept the join message. The described algorithm will eventually recover from this situation when the failure detector eventually provides accurate information.

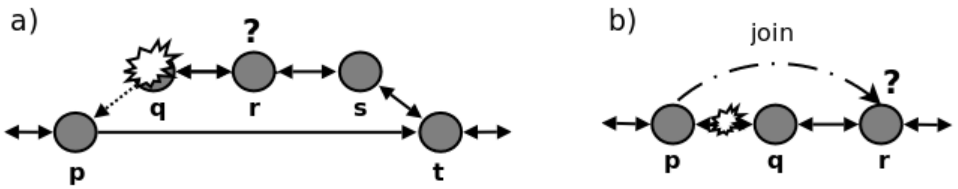


Fig. 5. Failures difficult to handle: (a) failure of the tail of branch, nobody is responsible for range $(p, q]$ (b) broken link generating a false suspicion of p about q.

In the case of the crash of the node at the tail of a branch, there is no predecessor to trigger the recovery mechanism. In this case, the successor could use one of its nodes in the predecessor list to trigger recovery, but that could introduce inconsistencies if the suspected node has not really failed. If the tail of the branch has not really failed but it has a broken link with its successor, then, it becomes temporary isolated and unreachable to the rest of the network. Having unreachable nodes means that we are in presence of network partitioning. The following theorem describes the guarantees of the relaxed-ring in case of temporary failures with no network partitioning.

Theorem 2. Simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition.

Proof.

Every failure of a node is eventually detected by its successor, predecessor and other peers in the ring having a connection with the faulty node. The successor and other peers register the failure in the *crashed* set, and remove the faulty peer from the resilient sets *predlist* and *succlist*, but they do not trigger any recovery mechanism. Only the predecessor triggers failure recovery when the failure of its successor is detected, contacting only one peer from the successor list at the time. Then, there is only one possible candidate to replace each faulty peer, and then, it is impossible to have two responsible for the same range of keys. If a simultaneous join occurs (as in figure 4), there are two possible cases. If the recovery happens first, the join will just be as regular join. If the join happens first, the successor candidate will reject the recovery forwarding to the recovery to the new peer. This means that only one successor candidate for recovery will be contact at the time, preventing inconsistencies. □

The problem with respect to network partition is inherent to any overlay network, where a temporary uncertainty cannot be avoid, and some guarantees must be sacrificed. A deeper analysis is provided by Ghodsi [6], and it is related to the proof given in [3] about limitations of web services in presence of network partitioning.

Figure 6 depicts a network partition that can occur in the relaxed-ring topology. The proof of theorem 2 is based on the fact that per every failure detected, there is

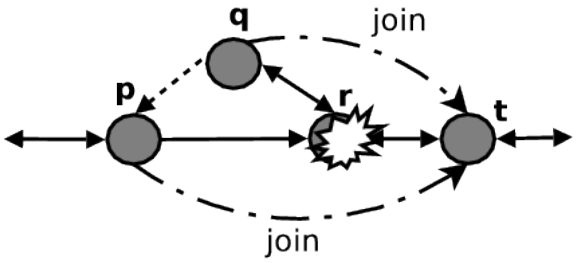


Fig. 6. The failure of the root of a branch triggers two recovery events

only one peer that triggers the recovery mechanism. In the case of the failure of the root of a branch, peer r in the example, there are two recovery messages triggered by peers p and q. If message from peer q arrives first to peer t, the algorithm handle the situation without problems. If message from peer p arrives first, the branch will be temporary isolated, behaving as a network partition introducing a temporary inconsistency. This limitation of the relaxed-ring is well defined in the following theorem.

Theorem 3.

Let r be the root of a branch, *succ* its successor, *pred* its predecessor, and *predlist* the set of peers having r as successor. Let p be any peer in the set, so that $p \in predlist$. Then, the crash of peer r may introduce temporary inconsistent lookup if p contacts *succ* for recovery before *pred*. The inconsistency will involve the range $(p, pred]$, and it will be corrected as soon as *pred* contacts *succ* for recovery.

Proof. There are only two possible cases. First, *pred* contacts *succ* before p does it. In that case, *succ* will consider *pred* as its predecessor. When p contacts *succ*, it will redirect it to *pred* without introducing inconsistency. The second possible case is that p contacts *succ* first. At this stage, the range of responsibility of *succ* is $(p, succ]$, and of *pred* is $(p', pred]$, where $p' \in [p, pred]$. This implies that *succ* and *pred* are responsible for the range $(p', pred]$, where in the worse case $p' = p$. As soon as *pred* contacts *succ* it will become the predecessor because $pred > p$, and the inconsistency will disappear. □

Theorem 3 clearly states the limitation of branches in the systems, helping developers to identify the scenarios needing special failure recovery mechanisms. Since the problem is related to network partitioning, there seems to be no easy solution for it. An advantage of the relaxed-ring topology is that the issue is well defined and easy to detect, improving the guarantees provided by the system in order to build fault-tolerant applications on top of it.

3.4. *Resilient information*

During the starting and join algorithms we have mentioned *predlist* and *succlist* for resilient purposes. The basic failure recovery mechanism is triggered by a peer when it detects the failure of its successor. When this happens, the peer will contact the members of the successor list successively. The objective of the *predlist* is to recover from failures when there is no predecessor that triggers the recovery mechanism. This is expected to happen only when the tail of a branch has crashed.

Algorithm 7 describes how the update of the successor list is propagated while the list contains new information. The predecessor list is updated only during the join algorithm and upon failure recoveries.

Algorithm 7 Update of successor list

```
1: upon event ⟨ upd_succlist | s, sl ⟩ do
2:     newsl := {s} ∪ sl \ getLast(sl)
3:     if (s == succ) ∧ (succlist ≠ newsl) then
4:         succlist := newsl
5:         send ⟨ upd_succlist | self, succlist ⟩ to pred
6:     end if
7: end event
```

4. Evaluation

This section is dedicated to the evaluation of the relaxed-ring. We analyse four aspects: the *amount of branches* that can appear on a network, the *size of branches*, the *number of messages* generated by the ring-maintenance protocol, and the verification of *lookup consistency* on unstable scenarios. The evaluation is done using a simulator implemented in Mozart [13, 10], where every node run autonomously on its own lightweight thread. Nodes communicate with each other by message passing using ports. We consider that these properties make the simulator more realistic. Every network is run several times using different seeds for random number generation. Charts are built using the average values of these executions.

4.1. *Branches and messages*

Figure 7 shows the amount of branches that can appear on networks with 1000 to 10000 nodes. The coefficient c represents the connectivity level of the network, where for instance $c = 0.95$ means that when a node contacts another one, there is only a 95% of probability that they will establish connection. A value of $c = 1.0$ means 100% of connectivity. On that value, no branches are created, meaning that the relaxed-ring behaves as a perfect ring on fault-free scenarios. The worse case corresponds to $c = 0.9$. In that case, we can observe that the amount of branches

is less than 10% of the size of the network, as expected. Consider peers i and k, where i is the current predecessor of k. If they cannot talk to each either, k will form a branch. If another peer j joins in between i and k having good connection with both peers, the branch disappears.

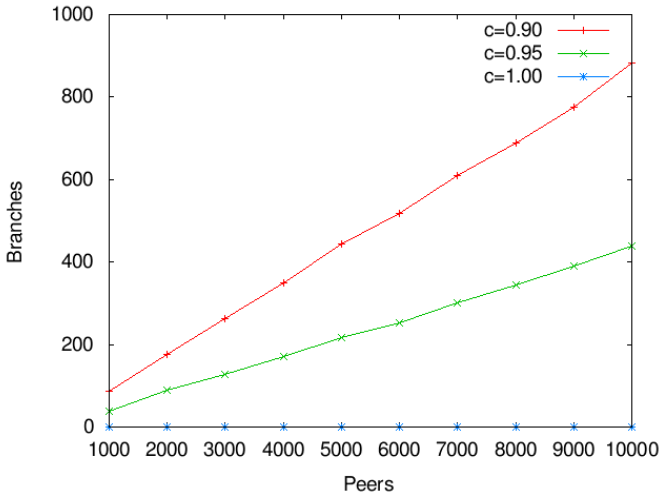


Fig. 7. Average amount of branches generated on networks with connectivity problems. Networks where tested with peers having a connectivity factor c, representing the probability of establishing a connection between peers, where $c \in \{0.9, 0.95, 1\}$.

On the contrary, if a node l joins the network between k and its successor, it will increase the size of the branch, decreasing the routing performance. For that reason, it is important to measure the average size of branches. If message *hint*, explained in section 3.2, works well for peer l, then, the branch will remain on size 1. Having this in mind, let us analyse figure 8. The average size of branches appears to be independent of the size of the network. The value is very similar for both cases where the quality of the connectivity is poor. In none of the cases the average is higher than 2 peers, which is a very reasonable value. If we want to analyse how the size of branches degrades routing performance of the whole network, we have to look at the average considering all nodes that belong to the core ring as providing branches of size 0. This value is represented by the curves *totalavg* on the figure. In both cases the value is smaller that 0.25. Experiments with 100% of connectivity are not shown because there are no branches, so the average size is always 0.

How many messages are exchanged by peers in order to maintain the relaxed-ring structure? How much is the contribution of the *hint* messages to the load in order to keep branches short? These questions are answered in figure 9. We can observe that the amount of messages increases linearly with the size of the network keeping reasonable rates. The fault-free scenario has no *hint* messages as expected, but the

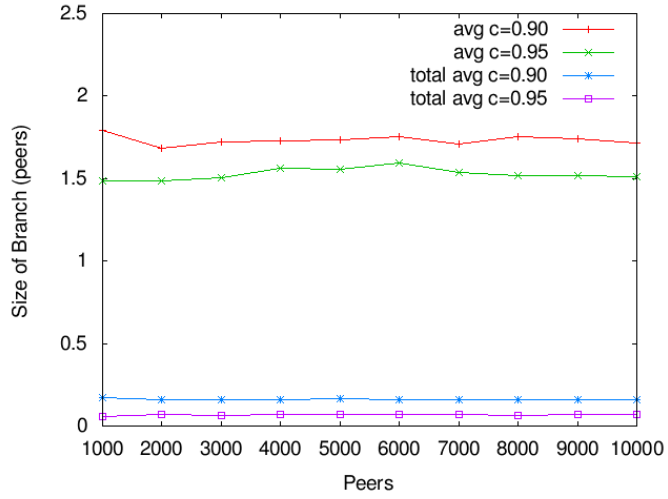


Fig. 8. Average size of branches depending on the quality of connections: *avg* corresponds to existing branches and *totalavg* represents how the whole network is affected.

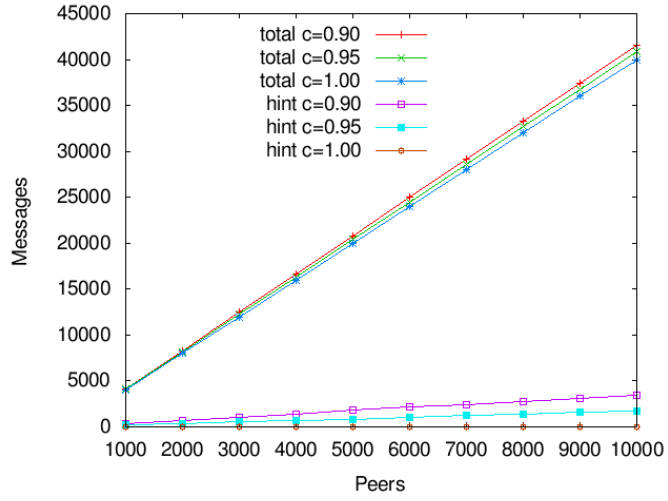


Fig. 9. Number of messages generated by the relaxed-ring maintenance. Three curves labeled *total* represent the total amount of messages exchanged between all peers depending on the connectivity coefficient. Curves labeled *hint* represent the contribution of *hint* messages to the total amount.

total amount of messages is still pretty similar to the cases where connectivity is poor. This is because there are less normal join messages in case of failures, but this amount is compensated by the contribution of *hint* messages. We observe anyway that the contribution of *hint* messages remains low.

4.2. *Comparison with Chord*

We have also implemented Chord in our simulator. Experiments were only run in fault-free scenarios with full connectivity between peers, and thus, in better conditions than our experiments with the relaxed-ring. Even though, we observed many lookup inconsistencies on high churn. To reduce inconsistency, we trigger periodic stabilization on all nodes at different rates. The best results appeared after triggering stabilization after the join of every 4 nodes. We call this value stabilization rate. As seen in figure 10, the largest the network, the less inconsistencies are found. An inconsistency is detected when two *reachable* nodes are signalized as responsible for the same key. We can observe that stabilization rates of 5 converges pretty fast to 0 inconsistencies. Stabilization every 6 new joining peers only converge on networks of 4000 nodes. On the contrary, rate values of 7 and 8 presents immediately a high and non-decreasing amount of inconsistencies. Those networks would only converge if churn is reduced to 0. These values are compared with the worse case of the relaxed-ring (connectivity factor 0.9) where no inconsistencies where found.

We have observed that lookup consistency can be maintained in Chord at very good levels if periodic stabilization is triggered often enough. The problem is that periodic stabilization demands a lot of resources. Figure 11 depicts the load related to every different stabilization rate. Logically, the worse case corresponds to most frequently triggered stabilization. If we only consider networks until 3000 nodes, it seems that the cost of periodic stabilization pays back for the level of lookup consistency that it offers, but this cost seems too expensive with larger networks.

In any case, the comparison with the relaxed-ring is considerable. While the relaxed-ring does not pass 5×10^4 messages for a network of 10000 nodes, a stabilization rate of 7 on a Chord network, starts already at 2×10^5 with the smallest network of 1000 nodes. Figure 11 clearly depicts the difference on the amount of messages sent. The point is that there are too many stabilization messages triggered without modifying the network. On the contrary, every join on the relaxed-ring generate more messages, but they are only triggered when they are needed.

5. Future Work

Apart from the simulator used for the validation, we have tested the Relaxed-Ring using a real implementation running distributed processes on small networks. We are currently testing our implementation on PlanetLab [1] to address more aggressive environments, and where we expect to report more about on failure recovery. The basic layers of P2PS providing point-to-point communication and the relaxed-ring maintenance are very stable. Our future work will be focused on the upper layers in order to deal with network partitioning. Apart from failure recovery, we are interested in building a service oriented architecture that will require a robust naming service and reliable broadcast. We also plan to build a replicated transactional distributed hash table based on a modified Paxos consensus algorithm [12].

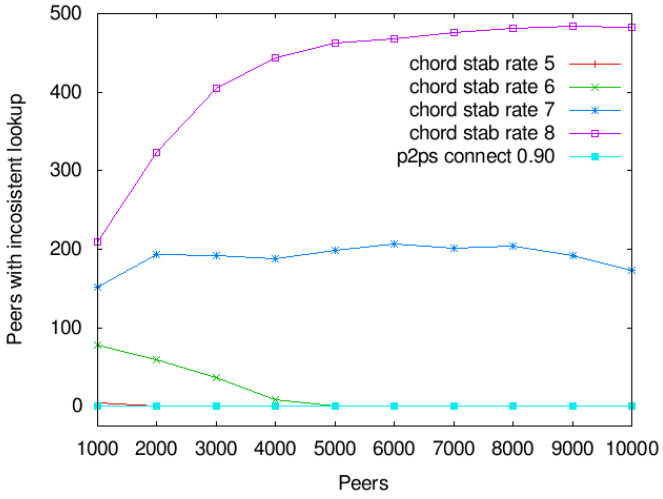


Fig. 10. Amount of peers with overlapping ranges of responsibilities, introducing lookup inconsistencies, on Chord networks under different stabilization rates for different network sizes. Comparison with the Relaxed-Ring (p2ps) with a bad connectivity. The stabilization rate represent the amount of peers joining/leaving the network between every stabilization round. The value of zero in the Y-axis has been raised in order to spot the curve of the Relaxed-Ring and Chord with a very frequent stabilization rate equal to 5.

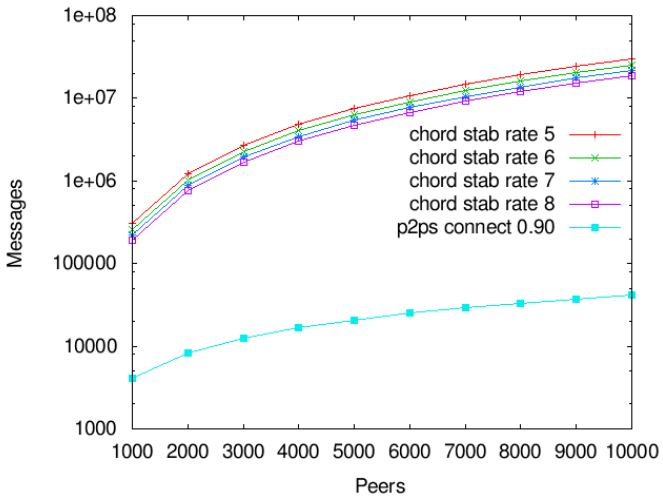


Fig. 11. Load of messages in Chord due to periodic stabilization, compared to the load of the Relaxed-Ring maintenance with bad connectivity. Y-axis presented in logarithmic scale.

6. Conclusion

In this paper we have presented a novel Relaxed-Ring topology for fault-tolerant and self-organizing peer-to-peer networks. The topology is derived from the simpli-

fication of the join algorithm requiring the synchronisation of only two peers at each stage. As a result, the algorithm introduces branches to the ring. These branches can only be observed in presence of connectivity problems between peers, and they help the system to work in realistic scenarios. The topology adds some complexity to the routing algorithm, but it does not degrade the complexity of its performance. We consider this issue a small drawback in comparison to the gain in fault tolerance and cost-efficiency in ring maintenance.

The topology makes feasible the integration of peers with very poor connectivity. Having a connection to a successor is sufficient to be part of the network. Leaving the network can be done instantaneously without having to follow a departure protocol, because the failure-recovery mechanism will deal with the missing node. The guarantees and limitations of the system are clearly identified and formally stated providing helpful indications in order to build fault-tolerant applications on top of this structured overlay network.

Acknowledgements

The authors would like to thank the *distoz* group at Université catholique de Louvain and S. González for comments on this work. This research is mainly funded by SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

References

[1] PlanetLab. *http://www.planet-lab.org*, 2008.

[2] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 344, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM Press.

[4] Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.

[5] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. *http://p2ps.info.ucl.ac.be*, 2008.

[6] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.

[7] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity, 2003.

[8] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Active and concurrent topology maintenance. In *DISC*, pages 320–334, 2004.

[9] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2):126–148, 2006.

[10] Boris Mejías. CiNiSMO: Concurrent Network Simulator in Mozart-Oz, Université catholique de Louvain, Belgium. *http://p2ps.info.ucl.ac.be/cinismo*, 2008.

[11] Boris Mejías and Peter Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *XXVI International Conference of the Chilean Computer Science Society*. IEEE Computer Society, November 2007.

[12] Monika Moser and Seif Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.

[13] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org, 2008.

[14] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08)*. ACM, June 2008.

[15] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.5 Visualizing Transactional Algorithms for DHTs

Visualizing Transactional Algorithms for DHTs

Boris Mejías
Université catholique de Louvain
boris.mejias@uclouvain.be

Mikael Högqvist
Zuse Institute Berlin
hoegqvist@zib.de

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

1. Introduction

Distributed Hash Tables (DHT) provide interesting properties for storing and retrieving data in decentralized systems. They are usually built on top of Structured Overlay Networks (SON) which has self-organizing and fault-tolerant properties. A DHT offers a simple interface to store and lookup elements associated with a key. The operations are basically $put(key, value)$ and $get(key)$. Every peer is responsible for a set of keys and if a peer fails, another peer takes over its responsibility. But what happens with the data of the crashed peer? Either the data must be re-inserted into the system or it can be replicated and recovered on the new responsible node. A replication mechanism must guarantee that the recovered replicated value is the same as the last value stored before the failure.

Data replication becomes more complex when the application running on top of the peer-to-peer network requires the update of several values stored on the DHT at the same time. This is typically done as a *transaction* involving keys belonging to different sets, and hence, involving different peers. How are the different peers coordinated in order to decide if the whole transaction must *commit* or *abort*? How do the replicas of these peers get the last valid data?

The two-phase commit protocol (2PC) is one of the most popular choices for implementing distributed transactions, being used since the 1980s. Unfortunately, its use on peer-to-peer networks is very inefficient because it relies on the survival of the transaction manager, as explained further in section 2. A three-phase commit protocol (3PC) has been designed in order to overcome the limitation of 2PC. However, 3PC introduces an extra round-trip which results in higher latency and increased message load. We advocate the use of an algorithm based on Paxos consensus [4, 2]. This algorithm is especially adapted for the requirements of a DHT and can survive a crash of the coordinator during a transaction. Compared to 3PC, it reduces latency and overall message load by requiring less message round-trips.

Demonstrator

We implement two-phase commit and the Paxos consensus-based algorithm on top of a Chord-like structured overlay network [5], extending the PEPINO network inspector [3] for visualization. By introducing arbitrary failures, the demonstrator shows why two-phase commit does not work on peer-to-peer networks. Then, the robustness of Paxos consensus is tested by injecting failures on a certain amount of transaction managers and participants, showing the failure recovery mechanism of this protocol.

2. Two-phase commit

The pseudo-code below implements a swap operation within a transaction. The objective is that the instructions from the beginning of the transaction (BOT) until its end (EOT) are executed atomically to avoid race conditions with other concurrent operations. The values of $item\_i$ and $item\_j$ are stored on different peers. The operators put and get are replaced by $read$ and $write$ in order to differentiate a regular DHT from a transactional DHT.

```
BOT
    x = read(item_i);
    y = read(item_j);
    write(item_j, x);
    write(item_i, y);
EOT
```

In order to guarantee atomic commit of a transaction on a decentralized storage, two-phase commit uses a *validation* phase and a *write* phase, coordinated by a *transaction manager* (TM). All peers responsible for the items involved in the transaction, as well as their replicas, become *transaction participants* (TP). Initially, the TM sends a request to every TP to *prepare* the transaction. If the item is available, the TP will lock it and acknowledge the *prepare* request. Otherwise, it will reply *abort*. The *write* phase follows *validation* once the replies are collected by the TM. If none of the participants voted *abort*, then the decision will be *commit*. When the participants receive the commit message from the TM, they will make the update permanent and release the lock on the item. An abort message will discard any update and release the item locks.

The problem with the 2PC protocol is that relies too much on the survival of the transaction manager. If the TM fails during the validation phase, it will block all the TPs that acknowledged the prepare message. A very reliable TM is required for this protocol, but it cannot be guaranteed on peer-to-peer networks.

3. Paxos Consensus Algorithm

The 3PC protocol avoids the blocking problem of 2PC at the cost of an extra message round-trip. This solution might be acceptable for cluster-based applications but not for peer-to-peer networks, where it is better to have less rounds with more messages than adding extra rounds to the protocol. This problem lead to the recent introduction of [4] based on Paxos consensus [2].

The idea is to add replicated transaction managers (rTM) that can take over the responsibility of the TM in case of failure. The other advantage is that decisions can be made considering a majority of the participants reaching consensus, and therefore, not all participants needs to be alive or reachable to commit the transaction. This means that as long as the majority of participants survives, the algorithm terminates even in presence of failures of the TM and TPs, without blocking the involved items.

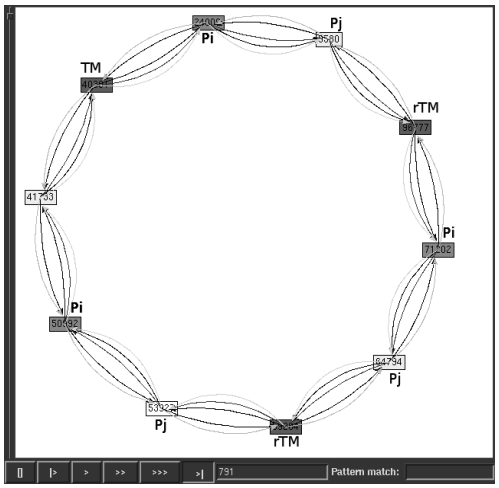


Figure 1. Network during a transaction with replicated manager and participants.

Figure 1 depicts a network visualized by the demonstrator. Different colours are assigned to TMs and TPs depending on the item they are responsible for. The labels in the figure are not in the simulator, but where added here for clarity. The TMs and TPs in the protocol are replicated using symmetric replication as described in [1]. Figure 2 shows the initial effect of introducing an arbitrary failure on the transaction manager, breaking the connection of the ring. The demonstrator continues with the recovery of the ring, and the election of a new TM from the rTMs.

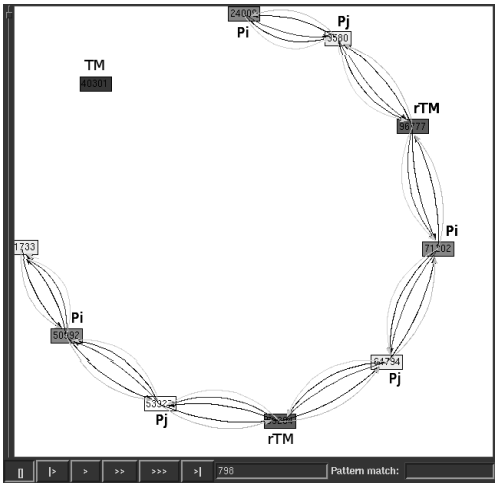


Figure 2. Failure of the transaction manager

4. Summary

The focus of this demonstrator is on the study of algorithms for implementing transactions on peer-to-peer networks. Their visualization contributes to the analysis and test of the protocols, verifying their tolerance to failures. In particular, we show a DHT running two-phase commit and the Paxos consensus algorithm.

5. Acknowledgements

The authors would like to thank Monika Moser and Seif Haridi for their help on the understanding of the Paxos consensus algorithm. This work is mainly funded by project SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

References

[1] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, Dec. 2006.

[2] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[3] D. Grolaux, B. Mejías, and P. Van Roy. PEPINO: PEer-to-Peer network INspectOr. In *The Seventh IEEE International Conference on Peer-to-Peer Computing*, 2007.

[4] M. Moser and S. Haridi. Atomic commitment in transactional DHTs. In *Proceedings of the CoreGRID Symposium*, 2007.

[5] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.6 Transactional DHT Algorithms

Transactional DHT Algorithms

Monika Moser[1], Seif Haridi[2], Tallat M. Shafaat[2], Thorsten Schütt[1], Mikael Högqvist[1], Alexander Reinefeld[1]

[1] Zuse Institute Berlin (ZIB), Germany
[2] Royal Institute of Technology (KTH), Sweden

Abstract. We present a framework for transactional data access on data stored in a DHT. It allows to atomically read and write items and to run distributed transactions consisting of a sequence of read and write operations on the items. Items are symmetrically replicated in order to achieve durability of data stored in the SON. To provide availability of items despite the unavailability of some replicas, operations on items are quorum-based. They make progress as long as a majority of replicas can be accessed. Our framework processes transactions optimistically with an atomic commit protocol that is based on Paxos atomic commit. We present algorithms for the whole framework with an event based notation. Additionally we discuss the problem of lookup inconsistencies and its implications on the one-copy serializability property of the transaction processing in our framework.

Table of Contents

Transactional DHT Algorithms 1
Monika Moser, Seif Haridi, Tallat M. Shafaat, Thorsten Schütt, Mikael Högqvist, Alexander Reinefeld
1 Introduction 3
2 Structured Overlay Network 3
2.1 The Overlay Model 4
2.2 Lookup Consistency and Responsibility 5
2.3 Availability 7
3 Transactions on a SON 7
3.1 1-Copy Serializability 7
3.2 Transaction Processing 8
The Paxos Protocol 9
Atomic Commit with Paxos 10
3.3 Replication 11
3.4 Serializability in Presence of Responsibility Inconsistency 11
4 Transaction Algorithms 12
4.1 System Architecture 12
4.2 Transaction ID and Transaction Item 12
4.3 System Assumptions 13
4.4 Identifiers, Modules and Operations 13
4.5 Algorithms 15
4.6 Read Phase 15
4.7 Commit Phase 20
Initialization 20
Validation 22
Consensus 24
5 Transaction Algorithms: Failure Handling 30
5.1 Failure of the Leader 30
5.2 Failure of a TP 32
6 Transactional Replica Maintenance 33
6.1 Copy Operation 34
6.2 Join and Leave 34
7 Evaluation 36
7.1 Analytical Evaluation of the Commit Protocol 36
Number of messages 36
Upper Timebounds 37
7.2 Experimental Evaluation 38
8 Discussions 40

1 Introduction

DHTs are fully decentralized and highly scalable systems that provide the ability to store and lookup data. They use the lookup service of a Structured Overlay Network for Internet-scale applications. The interface DHTs provide on their data is mostly a simple put/get interface. Often data in DHTs is immutable or consistency guarantees on data are weak. However many distributed systems require stronger guarantees like they are given by atomic data operations. We present a framework with transactional access to data stored in a DHT. It provides high availability of data and one-copy serializability for transactions on that data. A transaction consists of a sequence of one or more read and/or write operations that is executed atomically.

DHTs are dynamic systems where nodes are able to join the system or crash at any time. In order to maintain durability and availability of data, items are replicated. Each item consists of a fixed number of replicas. To tolerate the unavailability of a subset of replicas our transaction mechanisms are majority-based. This means that they are able to make progress if a majority of replicas is accessible. Therefore replication factor has to be chosen in a way that the availability of a majority of replicas is very high.

Read and write operations access at least a majority of replicas and choose the one with the highest timestamp. The atomic commit protocol that is needed to coordinate a distributed transaction also makes use of the majority idea. In order to prevent a single transaction manager from blocking the whole protocol if it fails, the framework uses a Paxos based non-blocking atomic commit protocol[7]. There, the single transaction manager is replaced by a set of nodes that all together act as the transaction manager. The protocol makes progress if the majority of these nodes does not fail until every participant in the transaction receives the outcome of the transaction.

In this paper we present the algorithms for our transactional framework. We use an event-based notation as it is well suited to present an asynchronous message-passing system. The framework builds on various techniques known from distributed database systems. The processing of transactions is done optimistically with a non-blocking atomic commit protocol in the end. Concurrency control is included in the atomic commit phase. The basic idea is to either acquire all necessary locks at the same time or to abort the transaction, thus avoiding distributed deadlock detection. Timestamps are used for each replica to determine whether items read in the read phase of the transaction are still valid in the commit phase. As the transaction processing is optimistic locks are only held during the commit phase. We combine the non-blocking Paxos atomic commit protocol with quorum techniques for access on replicated data.

2 Structured Overlay Network

In this section we introduce the model of the SON which underlies our framework. Thereby we refer to self-stabilization mechanisms that are used in Chord

[13]. However our framework is not restricted to a DHT based on Chord but can be applied to other key-based SONs.

2.1 The Overlay Model

A Model of a Ring-based SON. A DHT makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some a priori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$.

Every node in the system, has an unique identifier from the identifier space. Each node keeps two pointers: *succ*, to its *successor* on the ring, and *pred* to its *predecessor*. The successor of a node with identifier p is the first node found going in a clockwise direction on the ring starting at p. The predecessor of a node with identifier p is the first node met going anti-clockwise on the ring starting at p. For each node p, a *successor-list* is also maintained consisting of p's c immediate successors, where c is typically set to $\log_2(n)$, where n is the network size.

Ring-based DHTs also maintain additional routing pointers, called fingers, on top of the ring to enhance routing [1]. For our analysis, we assume that these pointers are placed as in Chord. Hence, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod{\mathcal{N}}$ for $0 \leq i < \log_2(\mathcal{N})$. Our results are independent of the chosen scheme for placing the fingers.

Dealing with Joins and Failures in Chord. A DHT system is a continuously running system and there is no notion of crash recovery. Whenever a node fails there is another node that becomes responsible for the items of the failed nodes. The protocols are based on a crash-stop model of nodes. This implies that if a node crashes and then reboot to re-join the network, it will be considered as a new node.

Chord handles joins and failures using a protocol called *periodic stabilization*. Figure 1 shows part of the protocol presented in [13]. Failures of predecessors are handled by having each node periodically check whether its *pred* is alive, and setting $pred := nil$ if the predecessor is found dead. Moreover, each node periodically checks to see if *succ* is alive. If it is found to be dead, it is replaced by the closest alive successor in the successor-list.

Each node periodically asks for its successor's *pred* pointer, and updates its *succ* pointer if it gets a closer successor. Thereafter, the node notifies its current *succ* about its own existence, such that the successor can update its *pred* pointer if it finds that the notifying node is a closer predecessor than *pred*.

Joins are also handled by the ring stabilization protocol. Joining nodes lookup their successor s on the ring, and sets $succ := s$. Periodic stabilization will eventually fix its predecessor and successor. Hence, any joining node is eventually properly incorporated into the ring.

Failure Detectors. DHTs provide a platform for Internet-scale systems, aimed at working on an asynchronous network. Informally, a network is asynchronous if there is no bound on message delay. Thus, no timing assumptions can be made

```
n.join(n')
  predecessor = nil;
  sucessor = n'.findsuccessor(n);

n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor])
    successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n' ∈ (predecessor, n])
    predecessor = n';

n.check predecessor()
  if (predecessor has failed)
    predecessor = nil;
```

Fig. 1: Part of the Chord protocol, presented in [13], which is related to successor and predecessor pointers.

in such a system. Due to the absence of timing restrictions in an asynchronous model, it is difficult to determine if a node has actually crashed or is very slow to respond. This gives rise to inaccurate suspicion of node failure.

Failure detectors are modules used by a node to determine if its neighbors are alive or dead. Since we are working in an asynchronous model, a failure detector can only provide probabilistic results about the failure of a node. Thus, we have failure detectors working probabilistically.

Failure detectors are defined based on two properties: *completeness* and *accuracy* [3]. In a crash-stop model, completeness requires the failure detector to eventually detect all crashed nodes. Accuracy relates to the mistake a failure detector can make to decide if a node has crashed or not. A perfect failure detector is accurate all the times, while the accuracy of an unreliable failure detector is defined by its probability of working correctly.

2.2 Lookup Consistency and Responsibility

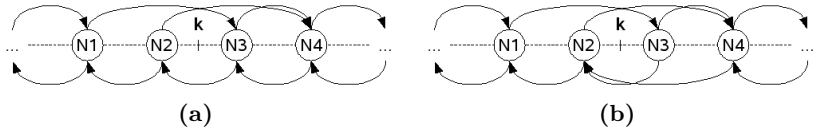
A consequence of imperfect failure detectors are inconsistent lookups and inconsistent responsibilities. We explain these terms in the following. Lookup consistency and responsibility consistency are important concepts when we reason about data consistency in our transactional DHT. Basically responsibility consistency is a requirement for guaranteeing data consistency.

In the following, we define lookup consistency and responsibility consistency, and explain how they can be violated. We use the term **configuration** of a SON to denote the set of all nodes and their pointers to neighboring nodes at a certain

point in time. A SON evolves by either changing a pointer, or adding/removing a node.

Definition 1 *A lookup for a key is consistent, if in a configuration lookups for this key made from different nodes, return the same node.*

Lookup consistency can be violated if some nodes' successor pointers do not reflect the current ring structure. Figure 2a illustrates a scenario, where lookups for key k can return inconsistent results. It shows nodes with their successor and predecessor pointers. This configuration may occur if node $N1$ falsely suspected $N2$ as failed, while at the same time $N2$ falsely suspected $N3$ as failed. A lookup for key k ending at $N2$ will return $N4$ as the responsible node for k, whereas a lookup ending in $N1$ would return $N3$.



(a) **(b)**

Fig. 2: Lookup inconsistency and responsibility inconsistency. Nodes with successor and predecessor pointers: (a) Example with wrong successor pointers. (b) Example with wrong successor pointers and overlapping responsibilities.

Definition 2 *A node n is said to be* locally responsible *for a certain key, if the key is in the range between its predecessor and itself, noted as $(n.pred, n]$. We call a node* globally responsible *for a key, if it is the only node in the system that is* locally responsible *for it.*

The responsibility of a node changes whenever its predecessor is changed. If a node has an incorrect predecessor pointer, the range of keys it is responsible for can overlap with another node's key range. Thus there are several nodes responsible for a part of the key range.

Definition 3 *The responsibility for a key k is consistent if there is a node globally responsible for k.*

A configuration where responsibility consistency for key k is violated is shown in Figure 2b. Here, lookup consistency for k cannot be guaranteed and both nodes, $N3$ and $N4$, are locally responsible for k. However, in Figure 2a, $N3$ is globally responsible despite lookup inconsistency and $N4$ is not responsible. The configuration depicted in Figure 2b can arise from the configuration shown in Figure 2a with an additional wrong suspicion of node $N4$ about its predecessor $N3$.

Lookup consistency and responsibility consistency cannot be guaranteed in a SON. As we will show later responsibility consistency is an assumption of our

system in order to guarantee data consistency. However in [12] we show that the probability for a violation of responsibility consistency is very low. E.g. with a reasonable probability for a failure detector to make false positives with two percent the probability to get consistent responsibility for a replica is more than 99.999%.

2.3 Availability

Another important property in our system is the availability of a key. In order to make progress operations in our system have to be able to access a sufficient number of replicas.

Definition 4 *A key k is available if there exists a reachable node n such that n is locally responsible for k.*

Availability of a key in a SON is both affected by churn and inaccurate failure detectors. Due to churn a key is unavailable when the node that is responsible for it fails until a successor node takes over responsibility and is reachable in the system. This is illustrated in Figure 3b where the key k is unavailable because of the failure of node $N2$. A node n is said to be reachable for a node n', if there exists a path from n to n'. Also during a join process when a node is transferring responsibility for a certain key range to the joining node, keys in that range are unavailable until the joining node is reachable in the system. Figure 3c illustrates a scenario where the joining node $N2$ already took over responsibility for key k but is not yet reachable in the system as node $N1$ has not set its successor pointer to $N2$. In the second case inaccurate failure detectors cause unavailability when a node that falsely suspects its successor will remove the pointer to this node. Thus, keys for which the suspected node is responsible will temporarily become unavailable. In figure 3d node $N1$ suspects node $N2$, thus k becomes unavailable.

3 Transactions on a SON

Usually DHTs provide a simple put/get interface to store and retrieve data. Hardly they provide consistency guarantees on data and often they are restricted to immutable data. Our framework is able to provide a transactional interface on top of a SON. It provides read and write operations that are executed transactionally as well as the ability to execute transactions that consist of a sequence of different operations.

3.1 1-Copy Serializability

Our algorithms provide 1-copy serializability. In our system items are replicated and there might exist replicas with different versions. However transactions produce a serializable history as if there was only one copy available to transactions. A history H of transactions is serializable if all committed transactions in H issue the same operations and receive the same responses as in some sequential history S that consists of the transactions committed in H [8].

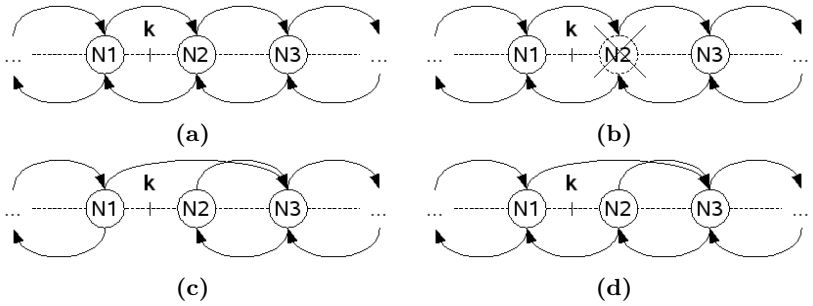


Fig. 3: Availability. Nodes with successor and predecessor pointers: (a) Example where key k is available. (b) Example where k is unavailable due to the failure of $N2$. (c) Example where k is unavailable during the joining process of $N2$. (d) Example where k is unavailable because $N1$ suspected $N2$ to have failed.

3.2 Transaction Processing

Transactions in our system are executed optimistically. They are processed in the following three phases:

- **Read phase (R):** Operations that are part of the transaction are executed within a transaction managers local workspace that is private to the transaction. Changes made by write operations are not visible to other transactions.
- **Validation phase (V):** Once a transaction should be committed, all involved data managers that are responsible for the data that is part of the transaction, check whether the operations are valid. Version numbers are used to determine if another transaction has made changes after the transaction's read phase.
- **Write phase (W):** If all data managers successfully validated the operations on their data, changes can be made permanent.

Atomic Commit. The validation phase and the write phase are executed within an *atomic commit protocol*. An atomic commit protocol coordinates all processes that are involved in a transaction. It ensures that all data managers decide on the same outcome of the transaction. The decision is commit if all data managers are able to validate the operations or abort if there exists at least one data manager that cannot validate an operation. Figure 4 shows a basic commit algorithm called the 2-Phase-Commit Protocol. There is one node called the *transaction manager (TM)* that coordinates the protocol. Data managers are called *transaction participants (TP)*. The TM asks the TPs to validate by sending a *prepare* request. The TPs either reply with *prepared* or *abort*. The TM collects the votes and sends *commit* if all TPs voted to be prepared otherwise it sends *abort*. When a TP receives commit it will make all changes permanent if promised to do when sending the prepared message. If a TP receives abort it won't make any changes permanent. Our framework executes transactions

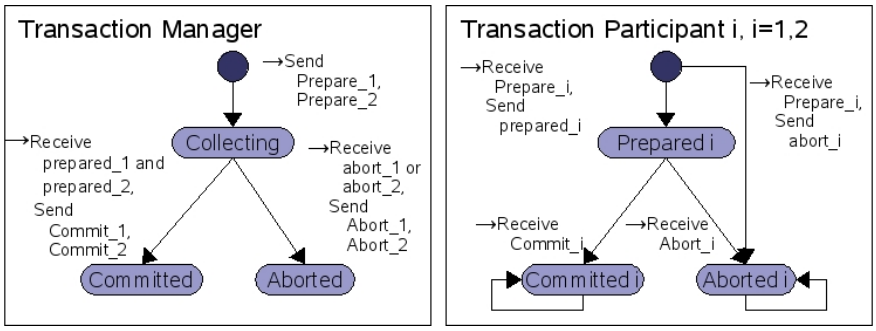


Fig. 4: State-charts for a 2-Phase-Commit Protocol with 2 Participants and 1 Transaction Manager

optimistically in the read phase. Thus the commit protocol will decide on abort if other transactions were committed in between.

A 2-Phase-Commit protocol is blocking if the TM fails in the state collecting and the TPs are not able to retrieve the outcome of the transaction. Therefore we use a non-blocking atomic commit protocol that is based on Paxos [10] in our framework. Gray introduced Paxos Atomic Commit in [7]. It uses replicated transaction managers which all collect votes from the data managers. If the leading transaction manager fails, these replicated transaction managers take over and distribute the decision of the atomic commit protocol to all participants. The Paxos Protocol and the Paxos Atomic Commit protocol are described later in this section.

Concurrency Control. Once a data manger successfully validates an operation on an item it has to ensure that no other transaction gets validated on the same item with a conflicting operation until the atomic commit protocol decides on the outcome. Therefore read and write locks are used during the commit phase that prevent concurrent conflicting validations.

Locks are only held during the atomic commit phase. Instead of letting transactions wait for a lock a TP will vote to abort in the atomic commit phase if it cannot acquire the lock for an operation. In that case the transaction has to be re-executed. This avoids distributed deadlock detection. We assume that read and write operations are less frequent than in traditional database systems. Thus the ratio of aborted transactions should be small.

The Paxos Protocol Paxos is an algorithm which guarantees uniform consensus. Consensus is necessary when a set of nodes has to decide on a common value. Uniform consensus satisfies the following properties: 1. *Uniform agreement*, which means that no two nodes decide differently, regardless of whether they fail after the decision was taken; 2. *Validity* describes the property that the value which is decided can only be a value that has been proposed by some

node; 3. *Integrity*, meaning no node may decide twice and finally 4. *Termination*, every node eventually decides some value [9]. Paxos assumes an eventual leader election to guarantee termination. Eventual leader election can be built by using inaccurate failure detectors.

Paxos defines different roles for the nodes. There are *Proposers*, which propose a value, and *Acceptors*, which either accept a proposal or reject it in a way that guarantees uniform agreement. Paxos as described in [10] assumes that each node may act as both proposer and acceptor. In our solution presented below we use different nodes as proposers and acceptors.

The above mentioned properties of uniform agreement can be guaranteed by Paxos whenever a majority of acceptors is alive. That means, it tolerates the failure of F acceptors out of initially $2F + 1$ acceptors.

Paxos basically consists of two phases called the read and write phase. In the *read phase* a node makes a proposal and tries to get a promise that his value will be accepted by a majority or it gets a value that it must adopt for the write phase. In the *write phase* a node tries to impose the value resulting from the read phase on a majority of nodes. Either the read or write phase may fail. Proposals are ordered by proposal numbers. By using an eventual leader to coordinate different proposals, the algorithm will eventually terminate.

Atomic Commit with Paxos Uniform consensus alone is not enough for solving atomic commit. Atomic commit has additional requirements on the value decided. If some node proposes abort or is perceived to have crashed by other nodes before a decision was taken, then all nodes have to decide on abort. To decide on commit, all nodes have to propose prepared.

In the Paxos Commit protocol [7] we have a set of acceptors, with a distinguished leader, and a set of proposers. The set of acceptors play the role of the coordinator and the set of proposers are those who have to decide in the atomic commit protocol.

Each proposer creates a separate instance of the Paxos algorithm with itself as the only proposer to decide on either prepared or abort. All instances share the same set of acceptors. It can be noted that the Paxos consensus can be optimized, because there is only one proposer for each instance. If a proposer fails, one of the acceptors, normally the leader, acts on behalf of that proposer in the particular Paxos instance and proposes abort.

Acceptors store the decision of all proposers and send the acknowledgment for the vote of a TP's Paxos instance to the leader. Whenever the leader has collected enough acknowledgments for each participant's Paxos instance, it decides on commit if all instances have decided on prepared or it decides on abort if there is at least one Paxos instance of a participant that decides on abort. Thereafter the final abort/commit is sent to the initial proposers. If the leader is suspected by the eventual failure detector, another leader will take over and can extract the decision from a majority of acceptors and complete the protocol.

The state-chart of a proposer is similar to the state-chart of a TP in the original 2PC protocol, as shown in figure 4. Also the state-chart of an acceptor

is similar to that of the TM, referring to the same figure. But instead of sending the decision commit to the participants, the acceptors send the outcome to the leader.

3.3 Replication

To provide higher reliability items are replicated. Each item has a fixed number of replicas. The replication scheme used here is key based. A key based replication essentially means that an item, which is a key-value pair, is stored under r replica keys. Thus, to store an item under key k, the value will be stored in the DHT under keys $Kr = \{k_1, k_2, k_3 \dots k_r\}$. We say that the replication degree is r and for key k, the set Kr to be the set of keys under which k is replicated. Each replica can be accessed symmetrically as the function to determine the replica keys is system-wide known [4].

As SONs usually are highly dynamic systems, operations on an item should make progress despite the unavailability of a number of replicas. Reads and writes thus require that a majority of replicas is accessible. A minority of replicas might be temporarily unavailable without hindering progress. Operations in our SON use majority-based algorithms. Majority-based algorithms are a special case of quorum algorithms. Quorum algorithms were introduced by Gifford [6] in order to maintain replicated data. Each replica is assigned a certain amount of votes. Read operations have to collect rv votes and write operations have to collect wv votes, where $rv + wv$ exceeds the total number of votes assigned to all replicas of an item. This ensures that read operations include at least one replica that was included by the latest write operation. In majority-based algorithms each replica is assigned exactly one vote and read and write operations have to include a majority of $m = \lfloor \frac{r}{2} \rfloor + 1$ votes. Thus they intersect in at least one replica.

3.4 Serializability in Presence of Responsibility Inconsistency

In order to ensure that $rv + wv$ always exceeds the total number of votes assigned to all replicas, the number of replicas in the system has to be constant for our majority-based algorithms. Each operation on an item has to ensure that it includes at least a majority of replicas, while the majority is based on the system's replication factor r. An additional replica that is added to the system would violate the above mentioned condition. However a responsibility inconsistency is equal to adding an additional replica. In that case two conflicting operations might end up with working on two disjoint sets with a majority of replicas, which we call *majority set*. This happens if two operations work on majority sets that both include distinct nodes that are involved in a responsibility inconsistency for one replica, but have no other replica in common. In that case it is not possible to detect a conflict between these operations, which can violate serializability. As it is not possible to ensure responsibility consistency, it is not possible to ensure serializability. However the existence of a responsibility inconsistency does not necessarily implicate disjoint majority sets for two conflicting operations.

In [12] we calculated the probability for two operations in one configuration to work on non-disjoint majority sets. If the probability for a failure detector to make false positives is 2% and therefore the probability to have a consistent responsibility is 99.999%, the probability for non-disjoint majority sets is 99.9999% if $r = 3$.

4 Transaction Algorithms

In this section we present the algorithms for the transactional DHT. We use an event based notation similar to the one used in [9].

4.1 System Architecture

Nodes in the system can take different roles in a transaction. For each transaction there exists the role of a leading Transaction Manager (TM), called *Leader*, which is the node the client is connected to. Additionally, a number of replicated Transaction Managers (rTM) are created according to the set of acceptors in Paxos commit. Nodes that are responsible for a replica of an item that is involved in the transaction have the role of a Transaction Participants (TP) in the protocol. Each node of the SON can have any number of TMs and TPs that are involved in different transactions. If there are multiple TPs on a node, they must share the database that contains the items with information about read and write locks. Each TP maintains a set of records for ongoing transactions, that have not yet been committed. Each record has a transaction ID, the new proposed value for the items the TP maintains and the new proposed version.

4.2 Transaction ID and Transaction Item

The leader of each transaction creates an unique transaction ID (TID). This ID is part of the SON's key space and can be treated like a item key. The leader creates the TID in a way such that it has a replica key of TID in its own key space. According to the replication scheme there are $r-1$ additional replica keys for the TID. The set of rTMs is determined by the nodes that are responsible for these associated replica keys. Thus the number of all TMs (Leader + rTMs) is equal to the replication factor r. At the end of a transaction each TM will store a replica of a so called *transaction item* with $\{TID, Decision\}$ as the {key, value} pair. We assume that a majority of rTMs does not change its responsibility such that the replica of the TID would not be part of its key space any more. Therefore a node that did not recieve the decision of the transaction can retrieve the decision by doing a quorum read on an item with the TID as key. The transaction item is maintained in the same way as normal items are. However it has to be garbage collected after a certain time.

4.3 System Assumptions

We identify the assumptions related to liveness, no nodes are blocked, and safety, no data is corrupted and 1-copy serializability is not violated. For liveness, it is assumed that direct communication between nodes as well as the bulk procedure is reliable. In addition, a majority of TMs must be alive and keep the TID within their range of responsibility until all alive TPs receive the transaction decision. This assumption is an extension of Paxos where all acceptors (TMs) must be alive during the protocol. For safety, we assume that a majority of replicas for an item are alive and that a majority of lookups targeting these replicase are consistent. A violation of the safety requirements may lead to inconsistent state. The probability of this happening is directly related to the replication factor.

4.4 Identifiers, Modules and Operations

Identifiers. Figure 5 lists all identifiers and variables used in the algorithms. The first part contains general identifiers that are used at transaction managers and transaction participants. The second part contains variables that are maintained by a transaction manager. Additionally, we introduce structures for votes that are received by transaction managers and for acknowledgments of votes. The last part contains variables that store information kept by a transaction participant.

External Modules Used in the Algorithms. The following modules are used by the algorithms

- EventuallyPerfectFailureDetector ($\Diamond P$) [9]
- EventualLeaderDetector (Ω) [9]

A leader uses a failure detector on every replica of the involved items. If it does not get a vote for a replica within a certain time threshold, it will start a failure handling procedure. A failure detector raises the event SUSPECT(tp) when it suspects the transaction participant tp to have failed. A leader election mechanism is used to guarantee progress of the atomic commit protocol. The set of replicated transaction manager will elect a new leader if they suspect the leader to have failed. The leader detector module raises the event TRUST($newleader$) to install a new leader.

Bulk Operation. The algorithms make use of a so called bulk operation [5]. This operation sends events to all nodes that are responsible for a key in a specified set of identifiers. E.g. a read operation on an item can be done with a bulk operation on the set of replica keys for that item.

Identifiers	
item	record
item.key	key
item.val	value
item.ts	timestamp/version number
item.op	kind of operation: write or read
tm	transaction manager
tp	transaction participant
r	replication degree of the system
Information maintained by a TM	
tid	ID of the transaction
TPs	set of Transaction Participants
TMs	set of replicated Transaction Managers
I	set of items involved in the transaction
Votes	Votes of the participants
AcksTMs	Acknowledgments sent by the TMs to the Leader
outcome	Overall outcome of the transaction
state	Either collectingNodes/collectingVotes/locallyDecided/decided
Suspected	set of nodes which are suspected to have failed
leader	the address of the leader
client	the address of the client issuing the transaction
vts	timestamp of a vote
rvts	timestamp of a vote acknowledged in a read phase
wvts	timestamp of a vote acknowledged in a write phase
ItemsInTrans	set of items that are currently involved in a transaction
Information contained in a vote	
i.key	key of the item the vote refers to
rkey	the key of the replica
vote	PREPARED/ABORT decision of a tp
vts	timestamp of the proposal - number of the proposal
Votes[i.key][rkey] = (vote, rts, wvts)	
vote	PREPARED/ABORT decision of a tp
rvts	timestamp of the vote that was accepted during the read phase
wvts	timestamp of the vote that was accepted (write phase)
AcksTMs[i.key][rkey]: {(vote, vts)*}	
vote	PREPARED/ABORT decision of a tp
vts	timestamp of the proposal that was accepted (write phase)
Information kept by a TP	
tid	ID of the transaction
TMs	transaction managers
i	the item
decision_of_tp	the decision it made

Fig. 5: Identifiers used in the algorithms

4.5 Algorithms

In the following we present the algorithms for the transaction processing. We first show the algorithm for the fault-free scenario. The algorithms for failure handling are shown separately in Section 5. The whole transaction processing algorithms refers to the execution of exactly one transaction. Thus we commit information that identifies a particular transaction for better readability.

The algorithms can be structured into different phases. Figure 6 identifies two main phases of a transaction. One is the *Read Phase* where the client determines the operations that are part of the transaction. The second one is the *Commit phase* which we further divide into *Initialization*, *Validation* and *Consensus*.

In the initialization phase the leader determines all nodes that act as replicated transaction managers (TMs). It determines the nodes by a key based search (lookup). After initialization these nodes communicate directly with each other without using a key based search. In the validation phase all TPs are sent a prepare request by a key based search. They are asked to validate the operations on the items they are responsible for. After validation the consensus on the outcome of the transaction is started, based on the validation results. The outcome is sent to the TPs directly with out doing a lookup.

4.6 Read Phase

During the read phase the client determines the operations that are part of the transaction. It can be any sequence of read and write operations. A client is connected to a certain node in the system. This node becomes the initial transaction manager which will act as the leader during the protocol. Figure 7 shows the particular communication steps in the read phase. The client instructs the leader to start a transaction (Algorithm 1) and to do operations until the client tells the leader to commit the transaction. The leader keeps track of the operations and keeps updates on items private to its local workspace. For read operations it will retrieve the value and version number of the item the client wants to read, while for write operation it has to retrieve the version number only (Algorithm 2). When the client signals the end of the transaction the leader will start a commit phase. Instead of instructing the leader to commit the transaction the client can also instruct it to abort the transaction before the commit phase. E.g. if the client reads a value that does not meet a certain condition. In that case the leader can simply throw away logged information on that transaction as the TPs have not yet made changes to their state, such that they do not have to be notified about this user triggered abort. Once a client tells the leader to commit a transaction the client cannot abort it any more on its own behalf.

Algorithm 2 includes a function *latest(Items)* which extracts the item with the highest version number from a set of items. It uses a *DB(key, rkey)* function that reads a replica from the local database. The replica is identified by the key of the item and a replica key or replica number. The function *replicakeys(key)* return all keys of replicas for a certain key.

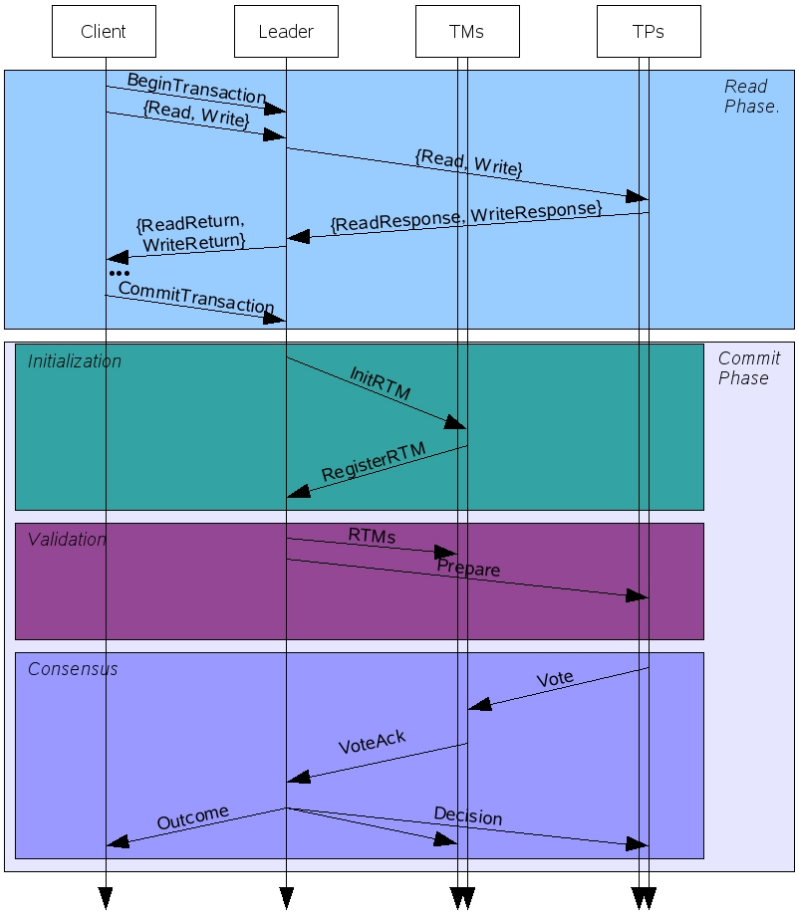


Fig. 6: The figure shows the different phases for a transaction together with the messages sent between the participating nodes

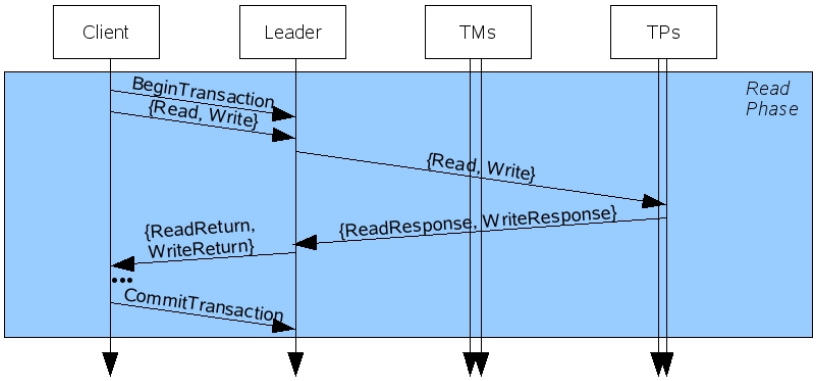


Fig. 7: The figure shows the messages which are part of the read phase. To start a transaction a client issues a BeginTransaction and signals the end of a transaction by a request to commit the transaction. In between it will add several *read* and *write* operations.

Algorithm 1 Interface to the Transaction Manager: Client signals Begin and End of a Transaction

```
1: upon event BEGINTRANSACTION() from client at tm
2:     client := client
3:     I := ∅
4:     readID:= writeID:= ⊥
5:     tid := generateTID()
6: end event

7: upon event COMMITTRANSACTION() from client at tm
8:     trigger STARTCOMMIT()
9: end event

10: upon event ABORTTRANSACTION() from client at tm
11:     delete information on transaction
12: end event
```

Algorithm 2 Processing of a Read Operation due to a Client's Read Request

```
1: function latest(Items) returns item is
2:     tmp_item := item{key:= ⊥, val:=⊥, ts:=−1, op:=⊥}
3:     foreach i in Items do
4:         if i.ts >tmp_item.ts then
5:             tmp_item := i
6:         end if
7:     end foreach
8:     return tmp_item

                                  ▷ A client requests a read operation at the Transaction Manager
9: upon event READ(key) from client at tm
10:    readID := createRID()
11:    Reads := ∅
12:    trigger BULK(replicakeys(key), {READ, key, readID})
13: end event

                                                          ▷ At the Transaction Participant
14: upon event BULK(rkey,{READ, key, readID}) from tm at tp
15:    i := DB(key, rkey)
16:    sendto tm : READRESPONSE(key, i.val, i.ts, readID)
17: end event

                                                          ▷ At the Transaction Manager
18: upon event READRESPONSE(key, val, ts, id) from tp at tm
19:    if readID=id then
20:        Reads := Reads ∪{(key, val, ts)}
21:    end if
22: end event

23: upon |Reads| ≥ (⌊r/2⌋ + 1) ∧ readID ≠⊥ do
24:    (k, val, v):= latest(Reads)
25:    I:= I ∪ item{key:= k, val:=val, ts:=v, op:=r}
26:    sendto client : READRETURN(value)
27:    readID := ⊥
28: end event
```

Algorithm 3 Processing of a Write Operation due to a Client's Write Request

▷ A client requests a write operation at the Transaction Manager

1: **upon event** WRITE($key, value$) **from** *client* **at** *tm*
2: writeID := createWID()
3: Writes := $\emptyset$
4: **trigger** BULK(ReplicaKeys(key), {WRITE, key, writeID})
5: **end event**

▷ At the Transaction Participant

6: **upon event** BULK($rkey$,{WRITE, key, $writeID$}) **from** tm **at** tp
7: i := DB(key, $rkey$)
8: **sendto** tm : WRITERESPONSE($key, i.ts, writeID$)
9: **end event**

▷ At the Transaction Manager

10: **upon event** WRITERESPONSE(key, ts, id) **from** tp **at** tm
11: **if** writeID=id **then**
12: Writes := Writes $\cup\{(key, ts)\}$
13: **end if**
14: **end event**

15: **upon** |Writes| $\geq (\lfloor r/2 \rfloor + 1) \wedge$ writeID $\neq \perp$ **do**
16: (k, v):= latest(Writes)
17: I:= I $\cup$ $item${key:= k, val:=$value$, ts:=v+1, op:=w}
18: **sendto** client : WRITERETURN(success)
19: writeID := $\perp$
20: **end event**

4.7 Commit Phase

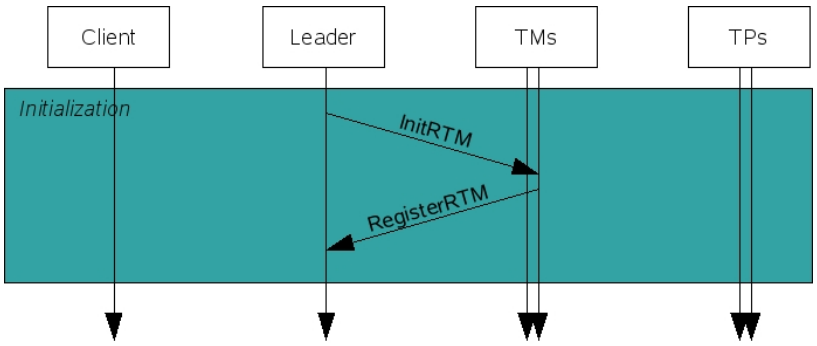


Fig. 8: Before starting the commit protocol the Leader determines the nodes that act as replicated TMs.

Initialization Figure 8 shows the course of events of the initialization phase. Initially nodes that have to act as rTMs are determined by a key based search based on the replica keys of the transaction ID. They have to be known before the commit protocol is started in order to enable leader election among the TMs and make the protocol fault-tolerant and non-blocking. In the next phase the leader will tell the rTMs the addresses of all other rTMs. As long as a majority of rTMs is reachable the protocol can decide on an outcome of the transaction.

Algorithm 4 contains the events and event handlers of the initialization phase. A node that gets a request to initialize a rTM has to initialize its data structures to collect the votes and the acknowledgments. For each item key and its replica keys the vote is initialized with a *rts* (read timestamp) with a value 1. The reason is that a TP will immediately start a write phase in its Paxos instance as it can be sure to be the first one that votes in that particular instance [7]. A TP's proposal number thus is 1.

Once the leader has collected enough TMs it can start the next phase. The leader always tries to collect all TMs. However if some of these nodes do not respond it can start the next phase after a timeout if it has collected at least a majority of TMs. A majority of TMs including the leader is necessary to guarantee progress for Paxos atomic commit.

Algorithm 4 Initialize the involved processes

```
1: upon event startCommit() at leader
                                        ▷ client is the process issuing the transaction
2:     trigger bulk(replicas(tid), {InitRTM, leader, tid, I, client})
3: end event

4: upon event bulk(rtid, {InitRTM, l, id, Items, cl}) from s at rtm
5:                                      ▷ A new transaction manager instance is created
6:     leader := l
7:     tid:= id
8:     I := Items
9:     client := cl
10:    foreach i in I do
11:        foreach rkey in replicaKeys(i.key) do
12:            Votes [i.key][rkey] := (⊥, 1, 0)
13:            AcksTMs [i.key][rkey] := ∅          ▷ Necessary if it becomes a leader
14:        end foreach
15:    end foreach
16:    sendto leader : registerRTM(rtm)
17:    state:= collectingVotes
18: end event

19: upon event registerRTM() from rtm at tm
20:    TMs := TMs ∪ rtm
21: end event

22: upon (|TMs| = r) do
23:    trigger startValidation
24: end event
```

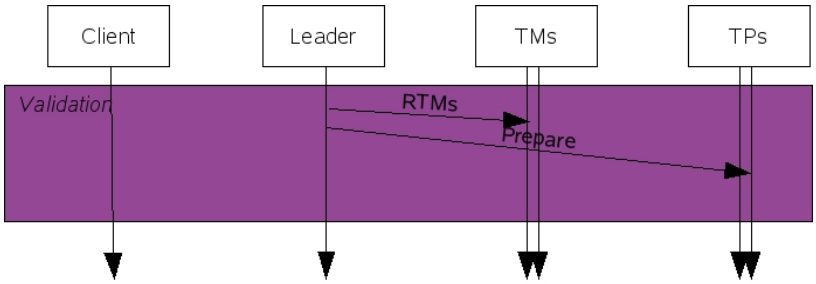


Fig. 9: The Leader sends a Prepare messages to the TPs which will start the validation and tells the TMs about all rTMs.

Validation After the initialization phase the leader tells each rTM the addresses of the other rTMs such that these are able to run a leader election mechanism among them. Additionally the leader sends the prepare request to the TPs together with the addresses of the nodes that act as rTMs. The prepare request is sent with a lookup operation on all replicas of the involved items. Figure 9 shows the course of events and Algorithm 5 the event handlers at the TMs.

The TPs check whether they can validate the operations sent by the prepare request. Each TP therefore locally applies the concurrency control mechanism. This is based on timestamps and locks. A node uses two dictionaries *readLock* and *writeLock* that contain locks on items. While write locks are exclusive, several read locks can be set at the same time. The locks are globally stored in a node. The $storeToLOG(Params)$ is a function that stores any information on a transaction in a TP's LOG. The $getFromLOG()$ function accordingly gets the information from the LOG. Algorithm 6 contains the procedure for validation.

For read operations a TP checks whether there is no lock for a concurrent write and whether the timestamp of the read request is valid, i.e. larger than or equal to the local item. If both checks are successful it will add a read lock and return prepared. For write operations the TP first ensures that there are no read or write locks set for concurrent conflicting operations. Then it compares the timestamp of the proposed item and the local item. This timestamp must to be equal to the currently stored timestamp + 1. If this is not the case, it means that a write operation has changed the item since it was accessed during the read phase. If both of the checks were successful the procedure will return prepared, otherwise abort. After the validation procedure, the TP starts to propose in a Paxos instance for this particular validation result.

Algorithm 5 Transaction Manager: Sending of a Prepare request

```
1: upon event startValidation at tm
2:     sendtoall TMs : RTMs(TMs, I)
3:     foreach i in I do
4:         trigger bulk(replicas(i), {Prepare, leader, tid, i, rkey, TMs})
5:     end foreach
6:     state:= collectingVotes
7: end event

8: upon event  RTMs(rtms, I) from tm at rtm
9:     TMs:= rtms
10:    (Ω).init(rtms)
11:    foreach i in I do
12:        ⋄P.init({replica : (i.key, rkey) ∈ i})
13:    end foreach
14: end event
```

Algorithm 6 Validation Procedure at a Transaction Participant/Concurrency Control

```
1: upon event PREPARE(item, rkey, tid, TMs) from tm at tp
2:     ItemsInTrans:= ItemsInTrans ∪ (rkey, tid)
3:     vote:= validate(item, rkey)
4:     trigger PROPOSE(item.key, rkey, TMs, vote, 1)
5: end event

6: procedure validate(item rkey, tid) returns PREPARED/ABORT is
7:     i:= DB(item.key, rkey)
8:     result := ⊥
9:     if item.op = read then
10:        if writeLock[(i.key, rkey)] = ⊥ & (item.ts ≥i.ts) then
11:            readLock[(i.key, rkey)] := readLock[(i.key, rkey)] + 1
12:            storeToLOG(tid, item.key, item.ts,r, rkey,PREPARED)
13:            result:= PREPARED
14:        else
15:            storeToLOG(tid, item.key, item.ts,r, rkey,ABORT)
16:            result := ABORT
17:        end if
18:    else
19:        if writeLock[(i.key, rkey)] = ⊥ & readLock[(i.key, rkey)] = ⊥ & (item.ts =
       i.ts+1) then
20:            writeLock[(key, rkey)] := 1
21:            storeToLOG(tid, item.key, item.ts, item.val,w, rkey, PREPARED)
22:            result:= PREPARED
23:        else
24:            store (tid, item.key, item.ts, item.val,w, rkey, ABORT)
25:            result := ABORT
26:        end if
27:    end if
28:    return result
```

Consensus This phase uses the Paxos atomic commit protocol [7]. At the end of it each TP has to receive the same decision on the transaction to ensure the uniform consensus properties. The decision will be commit if the decision for all items is prepared, it will be abort if the decision for at least one item is abort.

A Paxos instance is started for each replica. The TP that is responsible for the replica uses this Paxos instance to distribute its vote on the set of replicated transaction managers that act as the acceptors. The replicated transaction managers accept the vote of a TP if they did not get a read request with a higher timestamp for the particular Paxos instance. Instead of sending the acknowledgment to the TP they send it to the leader (Algorithm 8). The reason is that the decision on the outcome of the atomic commit protocol is based on the decisions for all items. Therefore, the leader collects all acknowledgments from where it can derive the outcome of the atomic commit protocol. As soon as the outcome is known the leader sends it to all involved nodes and notifies the client.

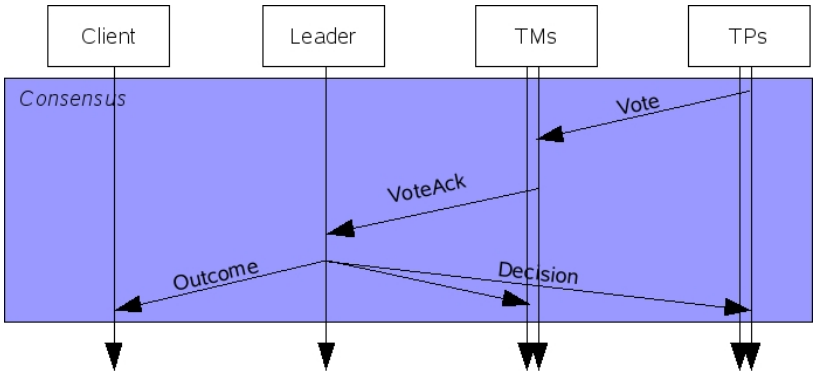


Fig. 10: After validation the TPs start a consensus to distribute their validation result. The leader will collect the decision for each particular consensus instance and conclude on the overall outcome of the transactions based on these instances.

As items in our DHT are replicated and operation on items require at least a majority of replicas to be accessible, the decision for an item also has to be based on a majority of replicas. Therefore the TM collects the votes of the TPs per item. A decision for an item is prepared if a majority of TPs that are responsible for a replica of that item vote to be prepared. The decision for an item is abort if there cannot be a majority of TPs that are responsible for a replica that vote to be prepared (Algorithm 9).

A TP that retrieves the decision for a transaction reads the information about the item it voted for from its LOG (Algorithms 10 and 11). It has to reset locks and write the item to the database if necessary. The learners are related to replica maintenance, which is explained in Section 6.

A TM that retrieves the decision for a transaction stores the decision in a special item, called *transaction item*, by calling $storeTransactionItem(decision)$. This item is replicated like all other items. The key for this transaction item can be deduced from the transaction ID. All TMs are nodes that are responsible for a replica key of the transaction item. If a node does not receive the decision by the normal execution of the protocol it can retrieve the decision by reading the transaction item like every normal item. We assume that a majority of replicated transaction managers may not fail and may not change their responsibility of the key range where the replica key of the TID is a member of until the outcome of the transaction is stored in the DHT. This can be achieved by choosing a proper replication factor.

Algorithm 7 Start Paxos Atomic Commit: Propose in a Paxos Instance

```
1: upon event PROPOSE(key, rkey, TMs, vote, ts) at p
2:     if ts=1 then
3:         sendtoall TMs : VOTE(key, rkey, vote, ts)
4:     else
5:         sendtoall TMs : READVOTE(key, rkey, ts)        ▷ See failure handling in 13
6:     end if
7: end event
```

Algorithm 8 Paxos Atomic Commit - Write Phase

```
1: upon event VOTE(key, rkey, vote, vts) from n at tm   ▷ n is either a tp or a tm
2:     if vts = 1 then
3:         TPs[key]:= TPs[key] ∪ (p, rkey)
4:     end if
5:     (currVote, rvts, wvts) := Votes[key][rkey]
6:     if vts ≥rvts and vts ≥ wvts then
7:         Votes[x.key][rkey] := (vote, rvts, vts)
8:         sendto Leader : VOTEACK(key, rkey, vote, vts)
9:     end if
10: end event

11: upon event VOTEACK(key, rkey, vote, vts) from tm at leader
12:     AcksTMs[key][rkey] := AcksTMs[key][rkey] ∪ {(vote, vts)}
13: end event
```

Algorithm 9 Paxos Atomic Commit - Making the Decision

▷ Ensure that a majority of TMs has stored the decision for a particular replica

function isPreparedReplica($i, rkey$) **returns** boolean **is**
    acks := AcksTMs[$i.key$][$rkey$]
    return $\exists vts : |\{(\text{PREPARED}, vts)\} \in \text{acks}| \geq \lfloor r/2 \rfloor + 1$

function isAbortReplica(i, n) **returns** boolean **is**
    acks := AcksTMs[$i.key$][$rkey$]
    return $\exists vts : |\{(\text{ABORT}, vts)\} \in \text{acks}| \geq \lfloor r/2 \rfloor + 1$

▷ Check whether the votes for a majority of replicas for an item is PREPARED

function isPrepared(i) **returns** boolean **is**
    return $|\{rkey : \text{isPreparedReplica}(i, rkey)\}| \geq \lfloor r/2 \rfloor + 1$

function isAbort(i) **returns** boolean **is**
    return $|\{rkey : \text{isAbortReplica}(i, rkey)\}| \geq \lceil r/2 \rceil$

upon $\forall i \in$ I: isPrepared(i) **at** *leader* **do**
    sendtoall TPs : DECISION(COMMIT)
    sendtoall TMs : DECISION(COMMIT)
    state:= DECIDED
    sendto client : OUTCOME(COMMIT)
end event

upon $\exists i \in$ I: isAbort(i) **at** *leader* **do**
    sendtoall TPs : DECISION(ABORT)
    sendtoall TMs : DECISION(ABORT)
    state:= DECIDED
    sendto client : OUTCOME(ABORT)
end event

upon event DECISION(*decision*) **from** tm **at** tm
    storeTransactionItem(*decision*)
    state:= DECIDED
end event

upon event DECISION(*decision*) **from** tm **at** tp
    trigger DECIDE(*decision*)
end event

Algorithm 10 Paxos Atomic Commit - Decide COMMIT at a TP

```
1: upon event DECIDE(COMMIT) at tp
2:     if not stored(COMMIT) then
3:         item := getFromLOG(tid)
4:         if Item = ⊥ then
5:             sendafterdelay(time, DECISION(COMMIT)) to tp
6:         else
7:             (key, val, ts, op, rkey, vote, tid) = item
8:             if op = r then
9:                 if vote = PREPARED then
10:                    readLock[(key, rkey)] := readLock[(key, rkey)] - 1
11:                end if
12:            else                                                    ▷ op = w
13:                if vote = PREPARED then
14:                    writeLock[(key, rkey)] := 0
15:                    DB(key, rkey) := (val, ts)
16:                else
17:                    DB(key, rkey) := (val, ts)
18:                end if
19:            end if
20:            learners := {l|(l, ltid) ∈ Learners[rkey] ∧ ltid == tid}
21:            foreach l in learners do
22:                remove(learners, Learners[rkey])
23:                ltidrest := {tid|(lr, tid) ∈ Learners[rkey] ∧ lr == l}
24:                if ltidrest = ∅ then
25:                    sendto l : COPYDATARESPONSE({(key, val, ts)})
26:                end if
27:            end foreach
28:            ItemsInTrans := ItemsInTrans \rkey
29:            storeToLOG(COMMIT)
30:        end if
31:    end if
32: end event
```

Algorithm 11 Paxos Atomic Commit - Decide ABORT at a TP

```
1: upon event DECIDE(ABORT) at tp
2:     if not stored(ABORT) then
3:         item := getFromLOG()
4:         if item = ⊥ then
5:             sendafterdelay(time, DECISION(tid, ABORT)) to tp
6:         else
7:             (key, val, ts, op, rkey, vote, tid) = item
8:             if op = r then
9:                 if vote = PREPARED then
10:                    readLock[(key, rkey)] := readLock[(key, rkey)] - 1
11:                end if
12:            else                                                    ▷ op = w
13:                if vote = PREPARED then
14:                    writeLock[(key, rkey)]) := 0
15:                end if
16:            end if
17:            learners := {l|(l, ltid) ∈ Learners[rkey] ∧ ltid == tid}
18:            foreach l in learners do
19:                remove(learners, Learners[rkey])
20:                ltidrest := {tid|(lr, tid) ∈ Learners[rkey] ∧ lr == l}
21:                if ltidrest = ∅ then
22:                    sendto l : COPYDATARESPONSE({(key, val, ts)})
23:                end if
24:            end foreach
25:        end if
26:        storeToLOG(ABORT)
27:    end if
28: end event
```

5 Transaction Algorithms: Failure Handling

This section covers failure handling of the atomic commit protocol during the consensus phase. Critical failures are those that block the transaction participants leaving a replica in a locked state. These can occur after the leader has sent the prepare request to the TPs. If the leader fails before that or if the leader cannot contact enough TPs or rTMs the transaction will simply be aborted. Failures have to be handled in a way that guarantees progress of the transaction processing while guaranteeing the properties of uniform consensus and atomic commit. In the following we distinguish between the failure of a TP and the failure of a leader. The failure handling reflects the one described in the Paxos atomic commit paper[7].

5.1 Failure of the Leader

When the leader fails the leader election mechanism will elect a new leader among all TMs. This new leader has to retrieve enough acknowledgments from the TPs' Paxos instances in order to be able to decide on the outcome of the transaction. Therefore it starts with a read phase in each single Paxos instance of the TPs (Algorithm13). In the write phase it will adopt the votes that have been proposed so far or vote to abort if there hadn't been a vote in the Paxos instance. As soon as the new leader got enough acknowledgments it can distribute the outcome of the commit phase. Figure 11 shows the course of events when a leader fails.

Algorithm 12 Replicated Transaction Manager: Trusting a New Leader

```
1: upon event TRUST(leader) at tm
2:     Leader := leader
3:     newts := nextVts()
4:     if state ≠ DECIDED ∧Leader = self then
5:         foreach i in I do
6:             foreach rkey in replicas(i.key) do
7:                 trigger PROPOSE(i.key, rkey, TMs, ⊥, newts)
8:             end foreach
9:         end foreach
10:    end if
11: end event
```

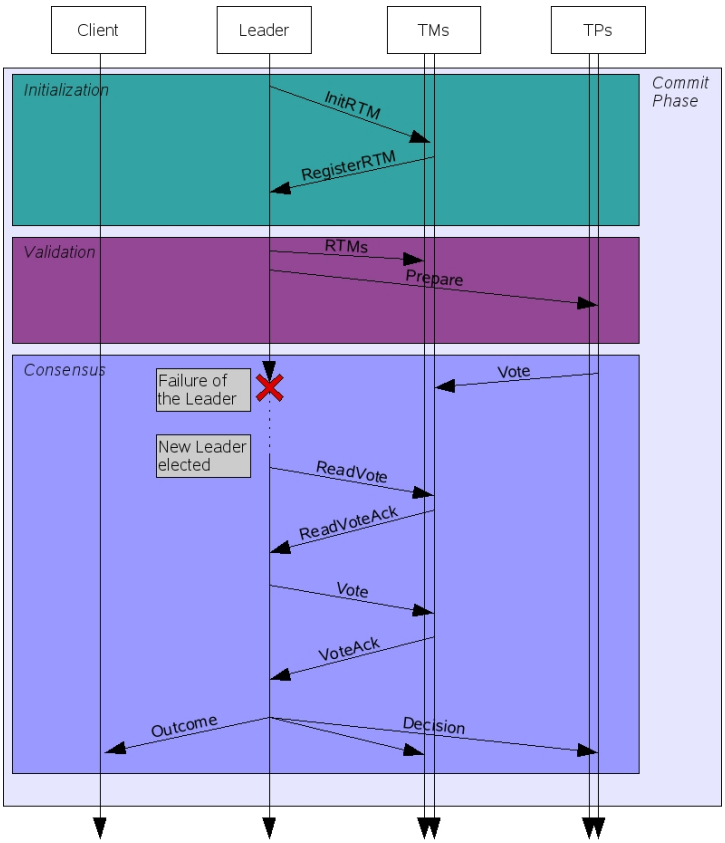


Fig. 11: If a Leader fails, another node will become the new leader and start a read phase in each TP's Paxos instance.

Algorithm 13 Paxos Atomic Commit - Read phase

```
 1: upon event readVote(i.key, rkey, vts) from leader at tm
 2:     (currVote, rvts, wvts) := Votes[i.key][rkey]
 3:     if vts >rvts and vts >wvts then
 4:         Votes[i.key][rkey] := (currVote, vts, wvts)
 5:         sendto Leader : readVoteAck(i.key, rkey, vts, (currVote, wvts))
 6:     end if
 7: end event

 8: upon event readVoteAck(i.key, rkey, vts, (vote, wvts)) from tm at leader
 9:     ReadVotes[i.key][rkey][vts] := ReadVotes[i.key][rkey][vts] ∪ {(vote, wvts)}
10: end event

11: upon ∃i.key, rkey, vts : | ReadVotes[i.key][rkey][vts]| ≥ ⌊r/2⌋ + 1 at Leader do
12:     vote := highest(ReadVotes[i.key][rkey][vts])
13:     if vote = ⊥ then
14:         vote := ABORT
15:     end if
16:     sendtoall TMs : vote(i.key, rkey, vote, vts)
17: end event
```

5.2 Failure of a TP

It must be noted that the protocol makes progress as long as foreach item a majority of TPs that are responsible for a replica is alive. In that case it is not necessary to start a failure handling for a TP. If there exists one item for which a majority of TPs cannot be reached, a leader would have to start voting in as many Paxos instances it needs to get a majority of votes for the item. However this situation occurs only if the system is broken, as we assume that for each item a majority of nodes being responsible for a replica is always available.

When the leader does not get a vote for a replica it suspects the corresponding TP to have failed. It will start to propose in the Paxos instance for that replica. As it does not know whether its suspicion is correct or the TP is alive and has already started to vote, the leader starts with a read phase. It will learn from the TMs whether there has been a vote made by the TP. If this is not the case the leader is free in its decision and will start a write phase with the value abort. Thus the leader decides to abort on behalf of the suspected TP.

Algorithm 14 Leader: Suspecting a TP during Consensus - Start a Read Phase

```
1: function nextVts() returns vts is
2:     select the next vts depending on the nodes key
3:                                   ▷ TM_r will come with a proposal numbered r+1
4:                                   ▷ Set of TMs: TM_1, TM_2, .., TM_r
5:
6: upon event SUSPECT((key, rkey)) at leader
7:     foreach i in I do
8:         trigger PROPOSE(key, rkey, TMs, nextVts())
9:     end foreach
10: end event
```

6 Transactional Replica Maintenance

When a node joins the structured overlay network it will take over the responsibility for a certain key range from an existing node in the system. Similarly, when a node fails, its successor in the structured overlay network has to take over responsibility for the failed node's key range. A node knows that its responsibility has changed whenever it has to update its predecessor pointer.

The framework has to handle joins and failures of a node on the data level in a way that maintains data consistency. Handling these events on the data level is done after they are handled on the overlay level. E.g. a joining node n first sets its successor and predecessor pointers and notifies the corresponding nodes about itself. Thereafter the new node n has to retrieve the data it is responsible for. Data retrieval mechanisms may not decrease the number of up-to-date copies for an item and they may not violate serializability of transactions.

Whenever the predecessor of a node is changed the event is handled on the data-level. There are two possible situations. Either the range the node is responsible for was increased or it was decreased. In the first case the node has to fetch the data it has not stored yet, in the second case the node has to drop data it is no longer responsible for.

A new copy of an item can be initialized by any transaction that writes that item or it is initialized explicitly[2]. As the first possibility may potentially take a long time and other copies of the item might fail during that time, it is better to do an explicit initialization of the copy to decrease the probability of a system failure. The system fails if an operation that is performed on a majority of nodes holding a replica cannot return an up-to-date item. Explicit initializing of a new copy can be done by a copy operation that reads the existing copies of the item and stores the current value in the new copy. Additionally all transactions that update the item must know of the new copy. Otherwise a non-serial history could arise by the following three events:

- Transaction $T1$ updates x and y
- Node $N1$ joins and will become responsible for replicas x_1 and y_1

– Transaction $T3$ reads x and y

The non-serial sequential history of events and transaction phases that can occur:

1. $T1$ - Initialization phase: Get addresses of involved nodes
2. $T1$ - Validation phase: Send prepare request with lookup
3. $N1$ joins the ring (updates its successor, predecessor and the others)
4. $N1$ copies x_1 from an existing copy: Gets the old value
5. $T1$ - Consensus phase: Outcome is received by all TPs
6. $N1$ copies y_1 from an existing copy: Gets the new value
7. x_1 at $N1$ is out of date

In this situation the number of replicas that are out of date would be increased. This happens if the initialization of an item does not take into account ongoing transactions on the item. The copy operation to initialize a replica has to be done either with a transaction or by adding the new node as learner to ongoing transactions, as we will explain in the following. A node that has to initialize the replicas in its range has to know which items exist that have replicas in that range and it has to copy the data for these replicas. If the initialization would have been done by one transaction this transaction would fail if there is even one single write operation on one of the items. Another possibility is that all ongoing transactions have the new node as a so called *learner* for the outcome. This concept is used in our framework and introduced in the following.

6.1 Copy Operation

The operation that initializes copies in a certain key range will be called *copy operation* in the following. Within this operation a node asks the nodes that are responsible for the corresponding remaining replicas of the items in its new range to send their replicas to it. It has to read from at least a majority of replicas. Once the nodes being responsible for the remaining replicas get a copy request, they have to remember the new node as a learner for all ongoing transactions. They send to the new node all replicas that are not involved in a transaction. Items that are currently involved in the transaction are sent as soon as there is no transaction run on them any more for which the new node is registered as learner. This is a way to let all ongoing transactions know the new copy. The new node will not be able to answer requests for items in its range until it has retrieved the data for it.

6.2 Join and Leave

When a node joins or leaves the system the successor of the joining or leaving node changes its predecessor. This means that the key range of the successor node is changed. In the following we assume that the routing layer triggers an event called NewPredecessor. Upon such an event a node checks whether the

range it is responsible for has increased or has decreased. In the first case it has to copy data, in the second case it has to drop data. Similarly, a new node will fetch data once it knows the range it is responsible for.

Algorithm 15 Data Level: New Predecessor Event from the Routing Level

```
        ▷ The NewPredecessor event is triggered on the routing level, after a new
    predecessor is set due to join or periodic stabilization
 1: upon event NewPredecessor(prevpred, newpred) at n
 2:     if newpred ≠ nil then
 3:         if newpred ∈ (prevpred, n) then
 4:             trigger dropData(prevpred, newpred)
 5:         else
 6:             trigger fetchData(newpred, prevpred)
 7:         end if
 8:     else
 9:         oldpred=prevpred
10:     end if
11:     if prevpred = nil then
12:         trigger dropData(n, newpred)
13:         if oldpred ∈ (newpred, n) then
14:             trigger fetchData(newpred, oldpred)
15:         end if
16:     end if
17: end event
```

Algorithm 16 shows the copy operation. A node that has to fetch data first calculates all the other replica keys that correspond to its key range. It starts a bulk operation with these replica keys to read the data it has to store. Each node that gets a copydata request will send the values or add the requesting node as a learner to the transactions on items that are in the requested range.

Algorithm 16 Data level: Drop and fetch data

```
                                   ▷ After setting successor and predecessor and notifying the others
 1: upon event FETCHDATA(start, end) at n
 2:     ReplicaKeys:= getCorrespondingReplicaKeys(start, end),
 3:     trigger BULK(ReplicaKeys, {COPYDATA})
 4: end event

                                   ▷ After a new predecessor is set on the overlay level
 5: upon event DROPDATA(start, end) at n
 6:     DB:= DB \ DB ([start, end])
 7: end event

 8: upon event BULK([x, y], {COPYDATA})) from n' at n
 9:     foreach (r.key, tid) in ItemsInTrans do
10:         if r.key ∈ [x, y] then
11:             Learners([r.key]) = Learners([r.key]) ∪ (n', tid)
12:         end if
13:     end foreach
14:     Items := DB([x, y])
15:     Items := Items \ ItemsInTrans
16:     sendto n' : COPYDATARESPONSE(Items)
17: end event

18: upon event COPYDATARESPONSE(Replicas) from n at newnode
19:     foreach i in Replicas do
20:         myrkey := getOwnReplicaKey(i.key)
21:         MyData[(myrkey, i.key)]:= MyData([(myrkey, i.key)]) ∪ (i.val, item.ts)
22:         if | MyData [(myrkey, i.key)]| ≥ ⌊r/2⌋ + 1 then
23:             if latest(MyData[(myrkey, i.key)]) > DB(i.key, myrkey) then
24:                 DB(i.key, myrkey):=(i.val, i.ts)
25:             end if
26:         end if
27:     end foreach
28: end event
```

7 Evaluation

7.1 Analytical Evaluation of the Commit Protocol

Number of messages Figure 12 illustrates the different communication steps that are necessary for a failure free execution of the protocol. Including the initialization phase six steps are required to make a decision on the outcome of the transaction. Note that in our algorithms the leader is at the same time a transaction manager. The number of messages depends on the replication factor r in the system and the number n of items involved in the transaction. This are the number of messages sent in each step:

1. r Lookup message for TMs

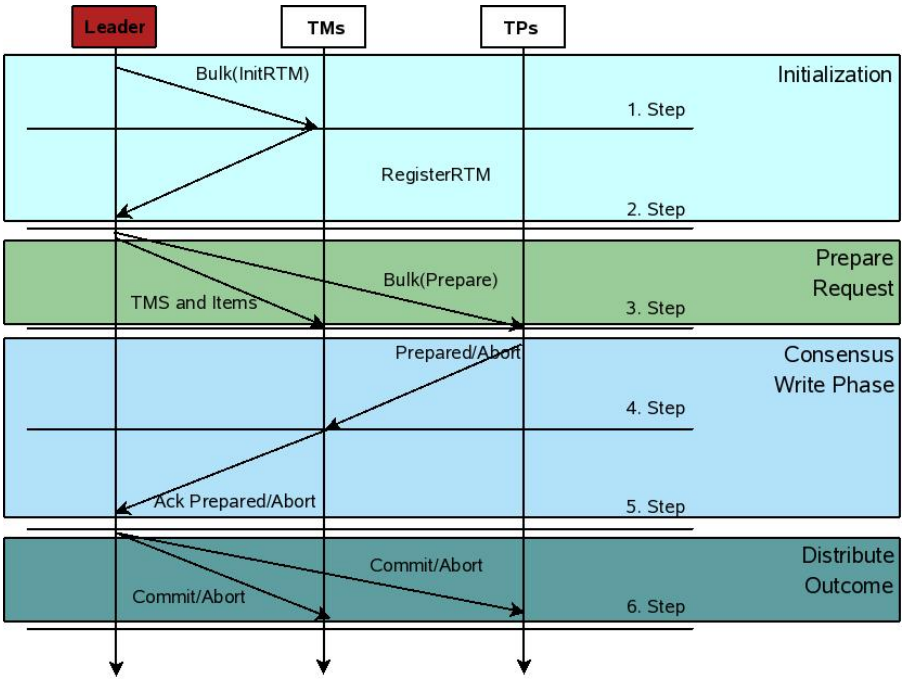


Fig. 12: The figure shows the communication steps required for the whole atomic commit protocol

2. r Registration messages from TMs
3. $n * r + r$ Prepare request to TPs and information message to TMs
4. $n * r * r$ Vote of TPs to all TMs
5. $n * r * r$ Acknowledgment for each vote
6. $n * r + r$ Decision sent to the TPs and TMs

Overall we need $(1+r)2nr+4r$ messages. The r messages from the first step use a bulk operation that is based on lookups. Similarly the $n * r$ prepare requests in the third step also use lookups. Note that the number of messages can be optimized. A TM can send all the acknowledgments for the votes it got within one message instead of sending a separate message for each vote.

Upper Timebounds

1. $O(logN)$ Lookup request to TMs
2. $O(1)$ Direct communication: Latency of slowest TM
3. $O(logN)$ Lookup request to TPs
4. $O(1)$ Direct communication: Latency of slowest conncection from a TP to a TM in a majority for an item

5. $O(1)$ Direct communication: Latency of slowest TM to leader connection
6. $O(1)$ Direct communication: Latency of slowest leader to TMs and TPs connection

7.2 Experimental Evaluation

The transaction algorithm is implemented as part of Scalaris [11], a P2P-based key/value store. We tested the performance of Scalaris and the transaction algorithm on an Intel cluster up to 16 nodes. Each node has two Quad-Core E5420s (8 cores in total) running at 2.5 GHz and 16 GB of main memory. The nodes are connected via GigE and Infiniband; we used the GigE network for our evaluation.

On each physical node we were running one multi-core Erlang virtual machine. Each virtual machine hosted 16 Scalaris nodes. We used a replication degree of four, that is, there exist four copies of each key-value pair.

We tested two operations: a read and a modify operation. The read operation reads a key-value pair. The modify operation reads a key-value pair, increments the value and writes the result back to the distributed Scalaris store. To guarantee consistency, the read-increment-write is executed within a transaction. The read operation, in contrast, simply reads from a majority of the keys. The benchmarks involved the following steps:

- Start watch.
- Start n Erlang client processes in each VM.
- Execute the read or modify operation i times in each client.
- Wait for all clients to finish.
- Stop watch.

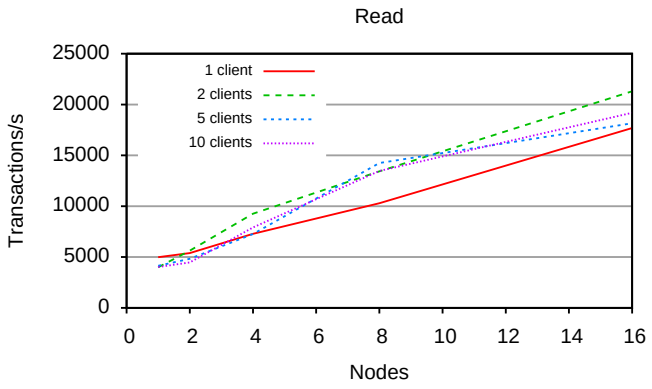


Fig. 13: Read performance of the transaction algorithm. The read operation is executed with increasing number of local threads and cluster sizes.

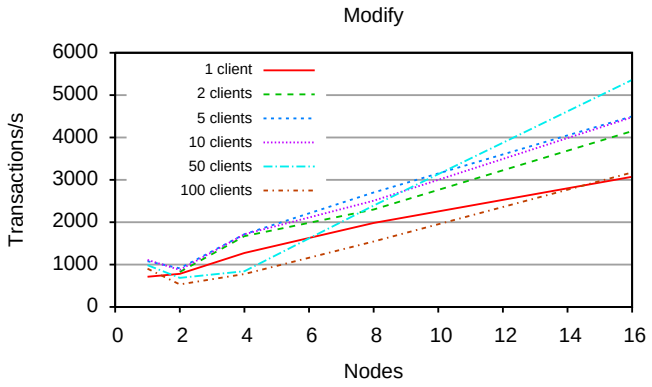


Fig. 14: Write performance of the transaction algorithm. The write operation is executed with increasing number of local threads and cluster sizes.

Figure 13 and 14 shows the results for various numbers of clients per VM (see the colored graphs). In the read benchmarks depicted in Fig. 13, each thread reads a key 2000 times while the modify benchmarks in Fig. 14 modify each key 100 time in each thread.

As can be seen, the system scales about linearly over a wide range of system sizes. In the read benchmarks (Fig. 13), two clients per VM produce an optimal load for the system, resulting in more than 20,000 read operations per second on a 16 node (=128 core) cluster. Using only one client (red graph) does not produce enough operations to saturate the system, while five clients (blue graph) cause too much contention. Note that each read operation involves accessing a majority (3 out of 4) replicas.

The performance of the modify operation (Fig. 14) is of course lower, but still scales nicely with increasing system sizes. Here, the best performance of 5,500 transactions per second is reached with fifty load generators per VM, each of them generating approximately seven transactions per second. This results in 344 transactions per second on each server.

Note that each modify transaction requires Scalaris to execute the adapted Paxos algorithm, which involves finding a majority (i.e. 3 out of 4) of transaction participants and transaction managers, plus the communication between them. The performance graphs illustrate that a single client per VM does not produce enough transaction load, while fifty clients are optimal to hide the communication latency between the transaction rounds. Increasing the concurrency further to 100 clients does not improve the performance, because this causes too much contention. Note that for the 100-clients-case, there are actually 16*100 clients issuing increment transactions. Overall, both graphs illustrate the linear scalability of Scalaris.

8 Discussions

The algorithms do not include garbage collection issues. A transaction manager has to keep the information for a transaction long enough to be sure that each transaction participant knows the outcome. Transaction participants could acknowledge that they got the decision of the atomic commit protocol.

References

1. L. O. Alima, A. Ghodsi, and S. Haridi. A Framework for Structured Peer-to-Peer Overlay Networks. In *Post-proceedings of Global Computing*, Lecture Notes in Computer Science (LNCS), pages 223–250. Springer Verlag, 2004.
2. Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.
3. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 1996.
4. A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems . In *Proceedings of the 3rd International VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'05)*, volume 4125 of *Lecture Notes in Computer Science (LNCS)*, pages 74–85. Springer-Verlag, 2005.
5. Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD thesis, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.
6. David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.
7. Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.
8. Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.
9. Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
10. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
11. Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable transactional p2p key/value store. In *ACM SIGPLAN Erlang Workshop*, September 2008.
12. Tallat M. Shafaat, Monika Moser, Ali Ghodsi, Thorsten Schütt, and Alexander Reinefeld. On consistency of data in structured overlay networks. In *CoreGRID Integration Workshop 2008*, 2008.
13. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.7 The Design of a Transactional Key-Value Store Using the Kompics Component Model

The Design of a Transactional Key-Value Store Using the Kompics Component Model

Cosmin Arad and Seif Haridi

SICS Technical Report T2009:09, ISSN: 1100-3154, ISRN: SICS-T2009/09-SE
Revision 1: July 4, 2009

Abstract

We document the architectural design of a transactional key-value store, based on a distributed hash-table (DHT). The DHT is built on top of the Chord# overlay network. Data items are replicated within the DHT using a key-based replication scheme, namely symmetric replication. Replica agreement for committing transactions is achieved using the Paxos Commit protocol. The system assumes a trusted deployment infrastructure, like a data-center or a set of connected data-centers. We model the system entities as Kompics components.

1 Introduction

The transactional key-value store presented here [6] offers a strongly consistent and fault-tolerant storage service for applications running in data-centers. The servers providing the storage are organized in a distributed hash-table (DHT) running on top of the Chord# overlay network [7]. Clients access the storage service by using a connector library. Data is replicated for fault-tolerance and availability. Transactions on stored data items are committed atomically at all replicas storing the involved data items, using the Paxos Commit protocol [4]. Data item availability depends on the presence of at least a majority of the replicas in the system. When a majority of the nodes storing replicas for one data item are absent, availability is traded off for strong consistency.

The system assumes a partially-synchronous network environment [5]. Node failures and additions occur rarely relatively to the transaction rates and false failure suspicions are assumed to be rare [8]. These assumptions are realistic for a data-center environment.

This document presents the design of the transactional key-value store using the Kompics component model [1, 2]. Entities in the system are modeled as Kompics components. Components local to one system node communicate by passing events, through well-defined ports. Ports specify the types of events communicated by components. Components residing on different system nodes communicate by messages. Messages are events that a network component on the source node marshals and sends to the network component on the destination node which unmarshals and delivers them to other components residing on the destination node.

We present the component architecture of the system nodes, the components with the protocols they implement and the messages they exchange.

2 System architecture

The transactional key-value store is provided by a set of *servers*. The storage service is accessed by (trusted) *clients*. We assume that each client can contact all servers. (This is needed in the *read phase* when the client needs to contact a majority of servers.) Since the servers in a typical data-center need not be publicly accessible, we further assume that both servers and client run inside the data-center. The storage service can be made available to (untrusted) clients outside the data-center through *proxies*. A proxy is a machine which runs inside the data-center and is publicly accessible from outside the data-center. The system architecture is illustrated in Figure 1. The nodes marked with S represent servers. Servers are organized in a DHT based on

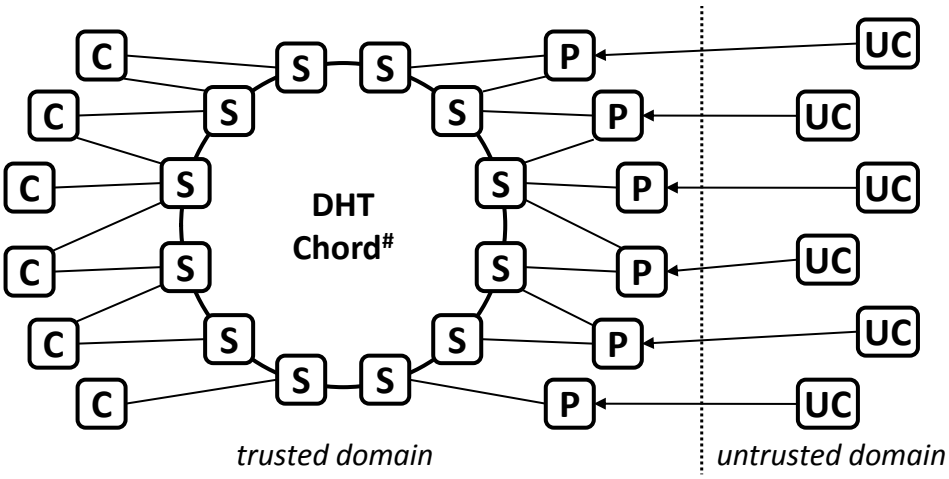


Figure 1: The overall system architecture.

the Chord# structured overlay which maintains a logical ring structure between the servers. The nodes marked with C represent trusted clients and the nodes marked with P represent proxies. Servers, trusted clients, and proxies are machines running inside the trusted domain, i.e., a data-center or a set of connected data-centers. Untrusted clients (marked UC) running outside the trusted domain access the storage service through proxies.

3 Component architecture

We now model the software architecture of the clients, the proxies, and the servers, as Kompics components.

3.1 Client

Clients are machines that execute data-center applications which use the transactional key-value store. We depict the software architecture of a client process in Figure 2. The top-level component is ClientMain. ClientMain contains an Application component. This can be any application that uses the transactional key-value store. (A client can contain multiple applications that concurrently use the key-value store.) The transactional key-value storage service is offered to the Application by the TdhtClient component through the TDHT port. To carry out a transaction, the TdhtClient communicates with a number of servers over the network. Network communication is provided by the MinaNetwork component [9] through the Network port. The JavaTimer component provides timeouts through the Timer port.

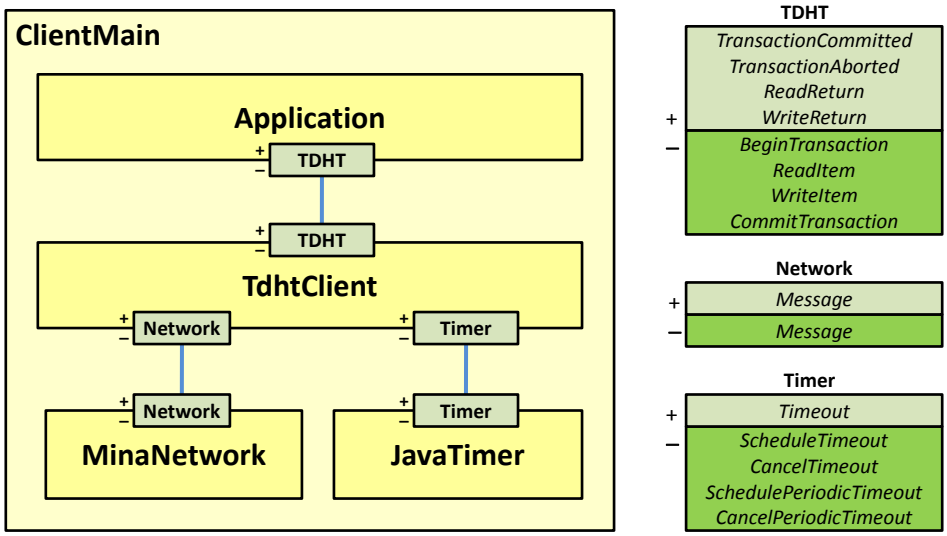


Figure 2: The client component architecture.

A typical transaction proceeds as follows: the Application triggers a *BeginTransaction* event, followed by a number of *ReadItem* and *WriteItem* events. On receiving a *ReadItem* event, the TdhtClient performs a quorum read operation, by contacting the servers which are responsible for storing the replicas of the read item. Upon receiving a majority of responses or on timeout, the TdhtClient component sends a *ReadReturn* event to the Application. Write operations proceed similarly. Read and write operations are recorded by the TdhtClient in a *transaction log*. The Application attempts to commit a transaction by sending a *CommitTransaction* event. Upon receiving it, the TdhtClient sends the transaction log to one of the servers that will attempt to commit the transaction and inform the client of the result. If the transaction is committed successfully, the TdhtClient sends a *TransactionCommitted* event to the Application. Otherwise, it sends a TransactionAborted event and the Application can retry the transaction.

The Application can issue multiple transactions in parallel. The events pertaining to one transaction are correlated by a transaction identifier randomly picked by the application and specified in every event. When multiple applications use the TdhtClient and there is a transaction identifier clash, the latter transaction is aborted. This should happen rarely.

The TdhtClient communicates with server nodes by sending messages over a network. The network communication service is offered to the TdhtClient by the MinaNetwork component, through the Network port. *Message* events sent by the TdhtClient on the Network port are received by the MinaNetwork which marshals them and sends them to their destination, where another MinaNetwork component unmarshals them and delivers them on its Network port. The actual messages sent by the TdhtClient are subtypes of *Message* and their type is used at the destination node to dispatch them to the right component.

The JavaTimer offers timeout services to the TdhtClient through the Timer port. The TdhtClient creates a subtype of the *Timeout* event and embeds it into a *ScheduleTimeout* event which it sends out on the Timer port. When it receives it, the JavaTimer schedules a timer task and when this expires, it sends back the embedded *Timeout* event on the Timer port. The TdhtClient receives its specific *Timeout* event, on the Timer port, after an amount of milliseconds previously specified in the *ScheduleTimeout* event. Every *Timeout* event automatically gets a random unique identifier. The TdhtClient can save the identifier of a *Timeout* event that it previously scheduled and use it to create and send a *CancelTimeout* event on the Timer port. This cancels the previously scheduled timeout. *Timeout* events that occur periodically can also be scheduled with the JavaTimer.

A client initially has to know the address of at least one server. The client uses the server to find out the addresses of the clients which store replicas of a data item for which the client needs to issue a majority read operation. If data items are replicated r times, the clients needs to issue r Chord$^{\#}$ lookups to find the addresses of the r servers responsible for storing the replicas. Since the client in not part of the Chord$^{\#}$ ring, the initial server known by the client issues these lookup in the ring, on behalf of the client. From the results of these lookups the client builds knowledge of other servers in the system, and over time, the client can probe these servers and use servers with the lowest latency in its subsequent requests.

3.2 Proxy

The proxy architecture is similar to the client architecture. A proxy is a client whose application acts as a server for untrusted clients. The interface that the proxy exposes to clients outside the data-center is intentionally left unspecified. Different proxy Application components can be designed to offer different interfaces (using different protocols) to the clients in the untrusted domain.

3.3 Server

We depict the software architecture of a server process in Figure 3. The top-level component is ServerMain. The logic of the server is encapsulated in the TdhtServer component which is detailed in Figure 4. ServerMain contains a MinaNetwork and a JavaTimer component to provide communication and timer services to the TdhtServer.

ServerMain also contains a JettyWebServer component. This embeds a web server which allows inspecting the state of the server using a web browser. Web requests coming from a web browser to the embedded web server generate *WebRequest* events on the Web port. These requests are handled inside the TdhtServer by the TdhtWebApplication component. This inspects the state of the various server components, prints it to an HTML

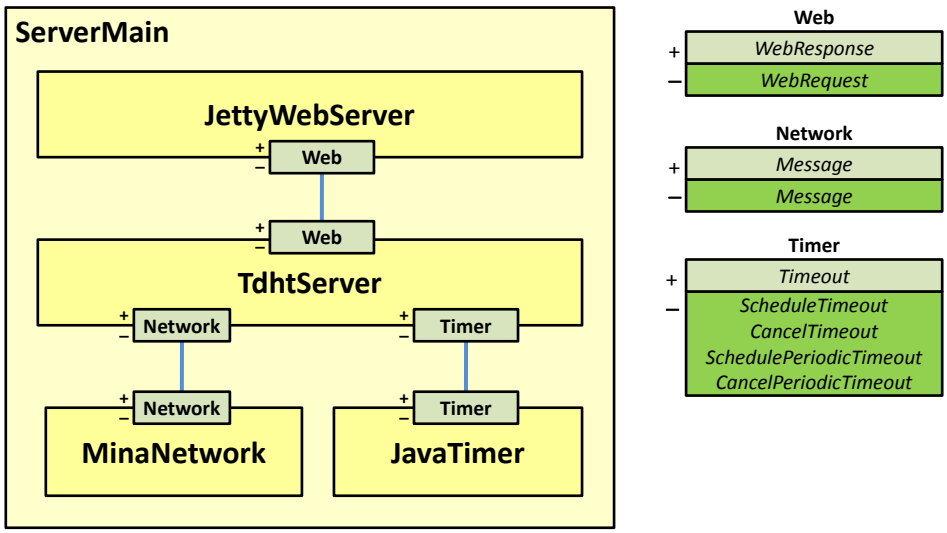


Figure 3: The server component architecture.

page, and sends this page as a *WebResponse* event back to the JettyWebServer through the Web port. This mechanism offers valuable insight into the state of the system and can be used for debugging or troubleshooting.

The main subcomponents of TdhtServer are TransactionManager (TM), TransactionParticipant (TP), and Chord#. The Chord# component implements the Chord# protocols for maintaining the overlay structure and carry out lookup operations. The Chord# component offers the lookup service through the StructuredOverlayNetwork (SON) port. This service is used by the TM and TP components to find the addresses of other servers responsible for various data items, and to be informed by the current responsibility of the local server.

To maintain the Chord# overlay structure when a close neighbor crashes, failure detection is performed on the predecessor node and on the nodes in the successor list of the current node. The failure detection service is provided by the FailureDetector component through the FD port. As a result of receiving a *StartMonitoringPeer* event, the FailureDetector periodically probes the remote peer and when it ceases to receive responses, the FailureDetector triggers a *PeerFailureSuspicion* event. The Chord# component reacts to the failure suspicion by replacing the failed neighbor with another one.

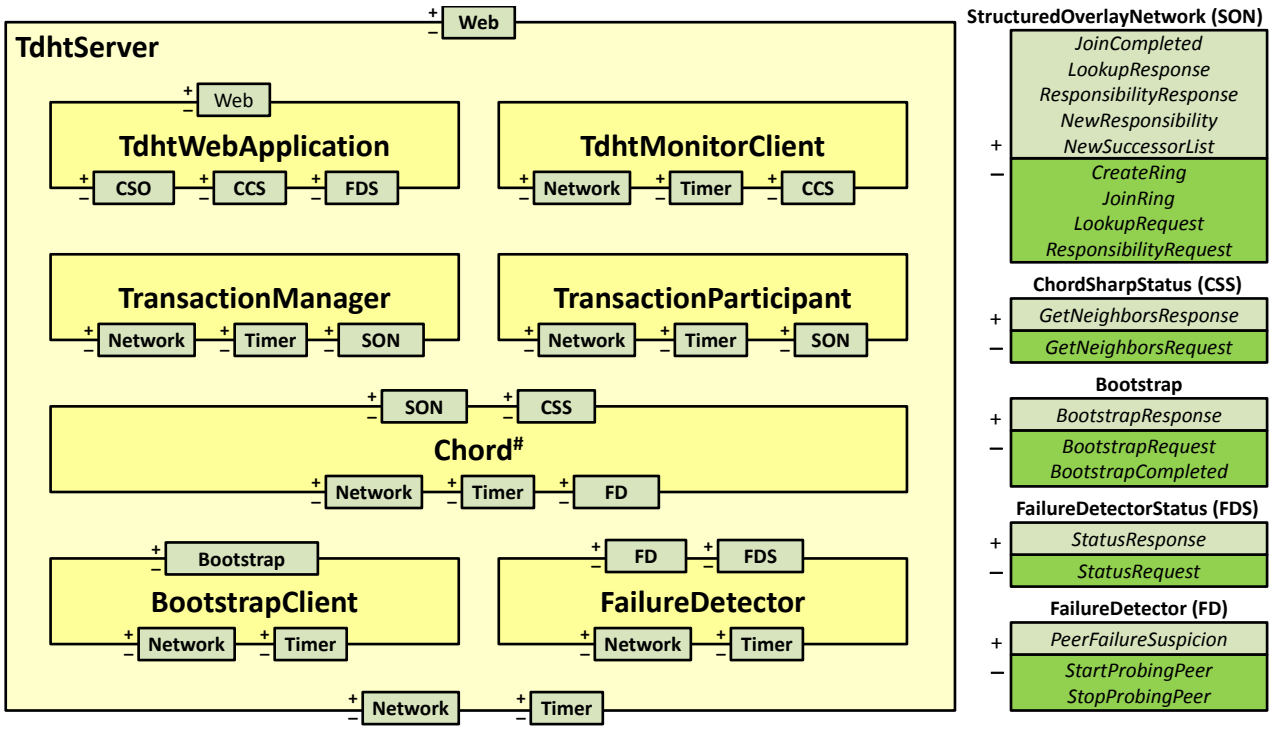


Figure 4: The TdhtServer component.

To join the Chord# overlay, a freshly started TdhtServer uses the Bootstrap service offered by the BootstrapClient component. When the TdhtServer is started, it issues a *BootstrapRequest* to the BootstrapClient. This talks to a well-known bootstrap server or set of servers to get the addresses of some overlay nodes already present in the system and returns these addresses to the TdhtServer in a *BootstrapResponse*. Using the inside server addresses, the TdhtServer issues a *JoinRing* event on the SON port of the Chord# component. After joining the overlay, Chord# triggers a *JoinCompleted*. When the TdhtServer received this event, it triggers a *BootstrapCompleted* event on the Bootstrap port of the BootstrapClient, which can now register the address of the local server with the bootstrap servers.

The TM and TP components implement a transactional key-value store on top of the Chord# overlay network. The TM component handles the coordination of transactions processing and the TP component handles the data availability maintenance in the face of server membership dynamism. We give the details of the TM and TP components in the following sections.

The TdhtMonitorClient periodically inspects the state of the various subcomponents of TdhtServer and pushes it to a monitoring server which aggregates the global view of the system, and makes it available for web-based inspection. This mechanism aids in troubleshooting and gives an overview of the system.

3.4 Transaction manager

The TM component implements the role of a transaction manager [6]. When a client enters the commit phase for a transaction, it contacts one server, which will act as the primary transaction manager for that transaction. The primary transaction manager with the help of $m_r - 1$ replicated transaction managers decides the outcome of the transaction. A TM component implements the behavior of both a primary and a replicated transaction manager, depending on the current transaction. A TM will be the primary transaction manager in some transactions, and it will be a replicated transaction manager for other transactions.

When a TM is chosen by a client to be the primary transaction manager for a transaction, the TM receives from the client the transaction log for that transaction. The TM contacts all current replicated transaction managers and all transaction participants listed in the transaction log. The participants validate the transaction locally and send their votes (to commit or abort the transaction) to all managers for the current transaction. One Paxos consensus instance is executed for every transaction participant. The transaction participant acts as the sole proposer, the managers act as learners, and the primary transaction manager acts as the learner.

Let us consider an example where data items are replicated d_r times, and a primary transaction manager has $m_r - 1$ replicated transaction managers. For a transaction that touches k distinct data items, there will be $k \times d_r$ transaction participants which will act as the sole proposers in $k \times d_r$ Paxos instances. In each such Paxos instance, the m_r managers (including the primary) act as acceptors and the primary transaction manager acts as learner.

3.4.1 Transaction manager replication

A background activity of the TM component is maintaining a set of replicated transaction managers. The first m_r nodes in the Chord# successor list of a transaction manager, will be used by the transaction manager as replicated transaction managers (RTMs). Whenever the Chord# successor list changes, the Chord# component triggers a *NewSuccessorList* event, which informs the TM about the new successor list. The TM uses the new successors as RTMs in subsequent transactions. Since Chord# already performs failure detection on the successor list, this scheme alleviates the need for the TM to perform failure detection on the RTMs.

3.5 Transaction participant

The TP component implements the role of a transaction participant [6]. There are two main activities carried out by the TP component: data maintenance and transaction validation. The TP implements substantial parts of the functionality of a DHT. The TP is constantly informed by the SON service offered by Chord# component of its responsibility range on the Chord# ring. The TP stores the data item with keys falling in its responsibility range. When its responsibility range shrinks due to a new predecessor having joined, the TP hands-over some of its data items to the new predecessor. When its responsibility range expands, due to the failure of its current

predecessor, the TP retrieves the items previously stored by the failed predecessor, from the nodes storing their replicas, to maintain their replication degree.

The TP component also participates in transaction validation and acts as a proposer in its own consensus instance during Paxos Commit.

3.5.1 Data replication

Each data item stored in the key-value store is replicated for persistence and availability. Data items are replicated symmetrically on the DHT ring [3]. A data item consists of a key and a value, where the key is a string and the value is an opaque object. The key-space is the space of strings allowed by some alphabet used as a system parameter. The Chord# ring identifiers are also strings from the same alphabet. For a replication degree of d_r, a key 'k' is mapped to d_r ring identifiers: r_1, r_2, ..., r_{d_r}, where r_i is 'ik'. For example, with a replication degree of 4, key "abc" is mapped onto ring identifiers "1abc", "2abc", "3abc", and "4abc". When we describe the attributes of the messages, we denote the type of a ring identifier like "2abc" with `ringKey` and the type of a data item key like "abc" with `dataKey`.

4 Messages

We now present the messages exchanged by the different components described in the previous section.

4.1 Messages exchanged between a TdhtClient and Chord#

During the read phase of a transaction, the client needs to do majority read operations for all items involved in the transaction, i.e., read the value and the version of the data item from a majority of replicas. The client only knows the Chord# ring identifiers of the replicas, but not the addresses of the servers responsible for storing the replicas. The client finds the addresses of the replicas by issuing lookups on the Chord# ring. For this a TdhtClient sends a *RemoteLookupRequest* message to the Chord# component of a known server. The Chord# resolves the lookup and sends back a *RemoteLookupResponse* to the TdhtClient.

RemoteLookupRequest	
`long`	lookupId
`ringKey`	key

RemoteLookupResponse	
`long`	lookupId
`ringKey`	key
`address`	responsible
`ok|fail`	status

Table 1: Messages exchanged between TdhtClient and Chord#.

4.2 Messages exchanged between a TdhtClient and a TransactionParticipant

Once the client knows the addresses of the transaction participants storing the replicas of one data item to be read in a transaction, the TdhtClient sends one *ReadItem* message to each participant. The TP replies to a *ReadItem* message with a *ReadResponse* message containing the ring identifier, the value, and the version number of the requested data item. For an item to be written during a transaction, the TdhtClient sends a *WriteItem* message to the TP, which replies with a *WriteResponse* message.

ReadItem	
`long`	readId
`ringKey`	key

ReadResponse	
`long`	readId
`ringKey`	key
`object`	value
`long`	version

WriteItem	
`long`	writeId
`ringKey`	key

WriteResponse	
`long`	writeId
`ringKey`	key
`long`	version

Table 2: Messages exchanged between TdhtClient and TP.

For a data replication degree of d_r, the client sends out d_r *ReadItem* or *WriteItem* messages, but only waits to receive $\lceil d_r/2 \rceil$, i.e., a majority of *ReadResponse* or *WriteResponse* messages. Once it receives the responses, the client records in the transaction log the following: (1) the operation performed, (2) the data item key, (3) the highest version number among the received majority, (4) the value associated with that version number (for reads), (5) the addresses of the servers responsible for the item replicas, and (6) the considered majority.

4.3 Messages exchanged between a TdhtClient and a TransactionManager

Once a client has completed the read phase of a transaction, and it has built a transaction log containing all items that need to be read and written, and the addresses of all transaction participants that need to validate the transaction, the client initiates the commit phase of the transaction by sending a *BeginCommit* message to a known server that will act as a primary transaction manager for this transaction.

When a primary TM receives a *BeginCommit* message, it begins the commit phase for the transaction specified in the transaction log received in the message. The first step is to create a *transaction descriptor*. The transaction descriptor contains an ordered list of the addresses of the TMs acting as acceptors in the consensus instances occurring during the commit phase. This list is built from the primary TM's current list of replicated TMs. The first TM on the list is the primary transaction manager and the learner in all those consensus instances. Should the primary TM fail during the commit phase, the second TM on the list becomes the primary TM.

The primary TM executes the commit phase together with its replicated TMs and the TPs involved in the transaction. When the commit phase is completed, the primary TM informs the client about the decision, by sending it a *CommitOutcome* message.

Once the primary TM builds a transaction descriptor it sends it back to the TdhtClient in a *TransactionDescriptor* message. If the client does not receive a *CommitOutcome* message within a predefined period of time, it assumes that the primary TM has failed and sends a new *BeginCommit* message to the second TM listed in the transaction descriptor. The secondary TM will now propose in all consensus instances in the transaction descriptor and it will try to decide the transaction, as if it were the primary TM.

BeginCommit	
`bigint`	xactionId
`xLog`	xactionLog

TransactionDescriptor	
`bigint`	xactionId
`xDesc`	xactionDescriptor

CommitOutcome	
`bigint`	xactionId
`committed|aborted`	outcome

Table 3: Messages exchanged between TdhtClient and TM.

4.4 Messages exchanged between TransactionManagers

Once the primary TM has started the commit phase and built the transaction descriptor it sends this descriptor to all replicated TMs in an *InitRTM* message. As they collect *Phase2A* messages from the TPs, the replicated TMs relay them as *Phase2B* messages to the primary TM who is the learner. As soon as the primary TM receives a majority (w.r.t. m_r) of *Phase2B* messages for one consensus instance, it can decide the outcome of that instance. If the outcome of at least a majority (w.r.t. d_r) of the consensus instances for every data item involved in the transaction is `commit`, then the primary TM decides to commit the transaction and sends a *CommitDecision* message to all other TMs and to all TPs.

InitRTM	
`bigint`	xactionId
`xLog`	xactionLog
`xDesc`	xactionDescriptor

Phase2B	
`bigint`	xactionId
`bigint`	consensusId
`prepared|abort`	acceptedVal

CommitDecision	
`bigint`	xactionId
`committed|aborted`	outcome

Phase1A	
`bigint`	xactionId
`bigint`	consensusId
`int`	ballot

Phase1B	
`bigint`	xactionId
`bigint`	consensusId
`int`	acceptedBallot
`prepared|abort`	acceptedVal

Phase2A	
`bigint`	xactionId
`bigint`	consensusId
`int`	ballot
`prepared|abort`	proposedValue

Table 4: Messages exchanged between TMs.

When the primary TM fails and one of the replicated TMs takes over as primary TM, it has to decide whether to commit or abort a transaction by proposing in phase 1 of all consensus instances of that transaction. It does so by sending *Phase1A* messages to all TMs listed in the transaction descriptor. All alive TMs will reply with *Phase1B* messages containing the highest ballot number they accepted in a phase 2 and the value they accepted if any. Upon receiving a majority of *Phase1B* messages, the new primary TM will send *Phase2A* messages to all TMs, using a ballot number higher than the highest accepted ballot number seen in the majority of received *Phase1B* messages, and a proposed value of `prepared` if the accepted value received in the *Phase1B* message with the highest accepted ballot number was `prepared`, or `abort` otherwise.

The primary TM will also go through the above phase 1 proposal, for any consensus instance for which it times out on receiving a *Phase2A* message from the TP associated with that instance.

4.5 Messages exchanged between a TM and a TP

At the same time when the primary TM sends *InitRTM* messages to the secondary TMs, it also sends *InitTP* messages to all the TPs listed in the transaction log. The *InitTP* message contains the transaction log which is needed by the TP to validate the transaction locally, as well as the transaction descriptor, telling the TP the addresses of all TMs participating in the transaction as acceptors.

Upon receiving the *InitTP* message, the TP tries to validate the transaction locally. This entails checking that the local data items are not currently locked by another transaction (read items are not locked fro writing and write items are not locked for either reading or writing) and that the version numbers are valid (same version locally and in the log for read items and the version in the log is one greater than the local version for write items). If the can TP validate the transaction successfully, it sends a *Phase2A* message with a ballot number 0 and a `prepared` proposed value, to all TMs. If the transaction cannot be validated, the TP proposes `abort` in the *Phase2A* message.

When the primary TM has reached a decision for the entire transaction it sends *CommitDecision* messages to all other TMs and to all TPs. Upon receiving a *CommitDecision* message, the TP unlocks all items locked after validation (for the current transaction). If the outcome of the transaction is `committed`, then the TP will also update the local value and version number for all local data items written in the transaction, according to the transaction log.

If the TP times out before receiving a *CommitDecision* message is behaves as described in failure scenario 2 in Section 6.

InitTP	
`bigint`	xactionId
`xLog`	xactionLog
`xDesc`	xactionDescriptor

Phase2A	
`bigint`	xactionId
`bigint`	consensusId
`int`	ballot (0)
`prepared|abort`	proposedValue

CommitDecision	
`bigint`	xactionId
`committed|aborted`	outcome

Table 5: Messages exchanged between TM and TP.

4.6 Messages exchanged between TransactionParticipants

The TP component is responsible for the management of data items. Data items need to be stored on the nodes which are responsible for the keys of the items, according to the Chord$^{\#}$ overlay. Therefore, whenever the Chord$^{\#}$ component signals the TP that the responsibility range of the current node shrunk, due to a new predecessor joining the ring, the TP needs to move some data items to its new predecessor. The TP sends the data items over to its new predecessor in one or more *TransferItems* messages. When the receiving TP has received all *TransferItems* messages for the transfered range, it replies with a *TrnasferComplete* message. During the transfer the source TP does not delete the items. It keeps all items until it receives the *TransferComplete* event, but it marks them as unavailable. Items are available at the destination TP as soon as they arrive in a *TransferItems* message. Items that are locked (by a transaction in progress) at the source TP are marked as such in the *TransferItems* message, and they do not immediately become available at the destination TP. The source TP transfers the locked items as soon as they become unlocked. If the destination TP times out on receiving the locked items, it performs a majority read to get the latest version and value of the items from replicas other

TransferItems	
`ringKey`	rangeBegin
`ringKey`	rangeEnd
`int`	part
`int`	partsTotal
`item[]`	dataItems

TransferComplete	
`ringKey`	rangeBegin
`ringKey`	rangeEnd

RecoverItems	
`ringKey`	rangeBegin
`ringKey`	rangeEnd
`address`	requester

ItemsRecovered	
`ringKey`	rangeBegin
`ringKey`	rangeEnd
`int`	part
`int`	partsTotal
`item[]`	dataItems
`address[]`	successorList

Table 6: Messages exchanged between TPs.

than the source TP. The destination TP also performs a majority read for all the items that it is responsible for but are not yet available (still in transit) and were requested by a client in a read or write operation.

Whenever the current predecessor of a TP fails, the TP becomes responsible for the items of its failed predecessor and needs to recover them from their replicas. This situation is illustrated in Figure 5.

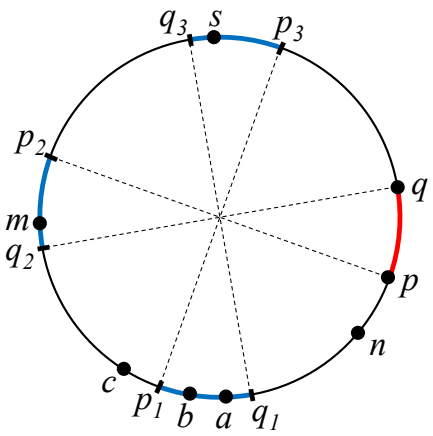


Figure 5: Node n's predecessor, p, failed, so node n becomes responsible for all p's items. Since node q was p's predecessor, node n needs to recover all items in the range $(q, p]$ (marked red in the figure) from their replicas. Here keys are replicated symmetrically and the replication degree is 4. The replicas of the items in the range $(q, p]$ reside on the blue ranges. Node n needs to ask all nodes responsible for items in the blue ranges for their items. Node n issues lookups for keys q_1, q_2, and q_3 (the symmetric keys of the beginning of the range, q) and finds the addresses of nodes a, m, and s. Node n asks node a for the items in range $(q_1, p_1]$, it asks node m for the items in range $(q_2, p_2]$, and it asks node s for the items in range $(q_3, p_3]$. Node a forwards n's request to b, who also forwards it to c. Nodes m and s react similarly to n's request.

Node n sends a *RecoverItems* message to nodes a, m, and s. Node a forwards the same message to node b, since b's responsibility range, $(a,b]$, overlaps with the requested range $(q_1, p_1]$. Node b also forwards the *RecoverItems* message to node c, since c's responsibility range, $(b,c]$, overlaps with the requested range $(q_1, p_1]$. Node a sends one or more *ItemsRecovered* messages to node n, containing the data items in the rage $(q_1, a]$. Similarly, nodes b and c send *ItemsRecovered* messages to node n containing their data items in ranges $(a,b]$ and $(b, p_1]$ respectively. In a similar fashion node n will receive the items in the ranges $(q_2, p_2]$ and $(q_3, p_3]$. Node n recovers the items in the range $(q, p]$ using the received item replicas. For every recovered item it uses the lastest version and value among the 3 replicas. Node n uses the successor lists received in *ItemsRecovered* messages to make sure it has recovered items from all nodes being responsible for a range.

5 Garbage collection

Periodically TMs check whether they have data associated with transactions that have completed (either committed or aborted) and are now older than some predefined age (for example a couple of hours or a day). Such transaction data is discarded. If a TM finds local transaction data for uncompleted transactions older than some predefined age (for example one hour), the TM attempts to complete the transaction by acting as the primary TM and proposing in all consensus instances of that transaction.

6 Failure scenarios

We try to cover all the possible failure cases that may occur during a transaction.

1. *The client fails during read phase.* This poses no problem. Since the commit phase was not yet initiated there is no data associated with the transaction at any server and no TP has locked any data items. Since

the client is crashed, there is no interest in the transaction and everything looks as if the transaction never happened.

2. *The client fails during commit phase.* If no server failure occurs the transaction commits successfully. If the primary TM also fails, there is no client to retry the commit with the secondary TM. The TPs that have locked data items and are waiting for *CommitDecision* messages from the primary TM, will time out. At this time the locked TPs will start acting as the client and try to recover the transaction status. The locked TPs will send *BeginCommit* messages to the secondary TM listed in the transaction descriptor. If the locked TPs time out again they repeat the procedure with the next TM. The alternative is to have the TMs execute a leader election algorithm among themselves.
3. *A transaction manager fails during read phase.* This is transparent if the client chooses a different TM for the commit phase. If the client chooses the failed TM for the commit phase, we have the same scenario as when the TM fails during the commit phase (see below).
4. *A transaction participant fails during read phase.* This poses no problem as long as a majority of the replicas of a data item are present. If a majority is present, the client can complete its majority read and move to the commit phase. If a majority of TPs are failed during the read phase, the majority read operation will fail and the transaction is aborted by the client.
5. *A transaction manager fails during commit phase.* If this is the primary TM, then the client will retry to commit the transaction using the secondary TM. If a replicated TM fails there is no problem as long as a majority of TMs are alive during the commit phase.
6. *A transaction participant fails during the commit phase.* If the TP fails after sending its proposal in its consensus instance, the transaction proceeds as if the TP never failed. If the TP fails before sending its proposal, the primary TM will time out on this proposal and propose abort in the consensus instance associated with the failed TP. The transaction can still commit as long as for every item, a majority of the consensus instances associated with item replicas decide commit.

Summary of timeouts:

- the client times out on a *CommitOutcome* message.
- a TP times out on a *Decision* message from the primary TM.
- the primary TM times out on an *Accept* message from a TP.
- a replicated TM times out on an *Accept* message from a TP.
- a TP times out on a *TransferItems* message from another TP.

References

[1] Kompics: Reactive Component Model for Distributed Computing. http://kompics.sics.se, 2009.

[2] Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, simulating, and deploying peer-to-peer systems using the Kompics component model. In *COMmunication System softWAre and middlewaRE (COMSWARE)*, Dublin, Ireland, 2009.

[3] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH–Royal Institute of Technology, Stockholm, Sweden, October 2006.

[4] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[5] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[6] Monika Moser and Seif Haridi. Atomic commitment in transactional DHTs. In *CoreGRID*, pages 151–161, 2007.

[7] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Structured overlay without consistent hashing: Empirical results. *Cluster Computing and the Grid, IEEE International Symposium on*, 2:8, 2006.

[8] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-Based Consistency and Availability in Structured Overlay Networks. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08)*. ACM, June 2008.

[9] The Apache Software Foundation. Apache MINA project. http://mina.apache.org, 2008.

Changelog

Revision 2 completed in the future

- description of the various failure handling procedures.
- description of the various recovery procedures.

Revision 1 completed on 4-Jul-2009

- description of the overall system architecture and of the component architectures for the client, the proxy, and the server;
- description of the DHT and of the data replication mechanisms;
- description of the TM replication components;
- description of the various objects: tables, data items, keys.

A.8 Scalaris: Users and Developers Guide

Scalaris

Users and Developers Guide

Version 0.1

Florian Schintke, Thorsten Schütt

May 15, 2009

Contents

I Users Guide 5

1 Introduction 7

2 Download and Installation 9
- 2.1 Requirements . . . 9
- 2.2 Download . . . 9
  - 2.2.1 Development Branch . . . 9
  - 2.2.2 Releases . . . 9
- 2.3 Configuration . . . 9
- 2.4 Build . . . 10
  - 2.4.1 Linux . . . 10
  - 2.4.2 Windows . . . 10
  - 2.4.3 Java-API . . . 10
- 2.5 Running Scalaris . . . 11
  - 2.5.1 Running on a local machine . . . 11
  - 2.5.2 Running distributed . . . 11
- 2.6 Installation . . . 12
- 2.7 Logging . . . 12

3 Using the system 13
- 3.1 JSON API . . . 13
  - 3.1.1 Deleting a key . . . 15
- 3.2 Java command line interface . . . 16
- 3.3 Java API . . . 16

4 Testing the system 17
- 4.1 Running the unit tests . . . 17

II Developers Guide 19

5 How a node joins the system 21
- 5.1 General Erlang server loop . . . 21
- 5.2 Starting additional local nodes after boot . . . 21
  - 5.2.1 Supervisor-tree of a Scalaris node . . . 22
  - 5.2.2 Starting the or-supervisor and general processes of a node . . . 22
  - 5.2.3 Starting the and-supervisor with a peer and its local database . . . 24
  - 5.2.4 Initializing a `cs_node`-process . . . 24
  - 5.2.5 Actually joining the ring . . . 25
  - 5.2.6 Beginning to serve requests . . . 27

6 Routing and routing tables in the Overlay 29

6.1 Simple routing table 30
6.1.1 Data types 31
6.1.2 A simple routingtable behaviour 31
6.2 Chord routing table 32
6.2.1 Data types 32
6.2.2 The routingtable behaviour for Chord 33

7 Directory Structure of the Source Code **35**

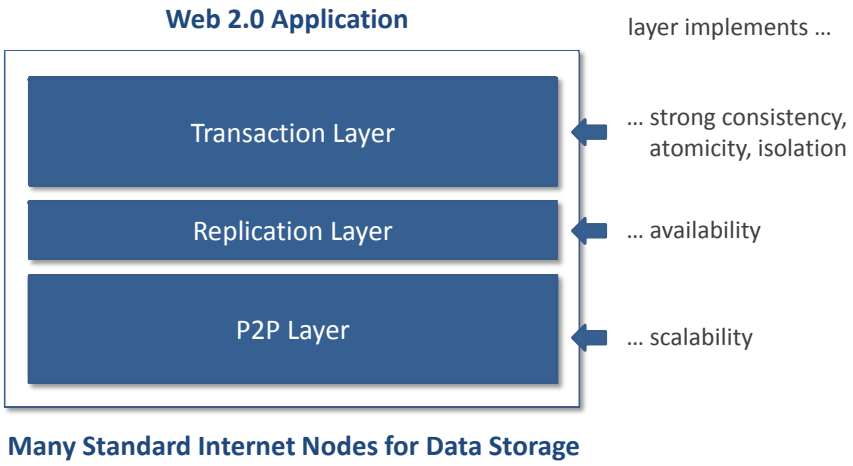
8 Java API **37**

Part I

Users Guide

1 Introduction

Scalaris is a scalable, transactional, distributed key-value store based on the peer-to-peer principle. It can be used to build scalable Web 2.0 services. The concept of Scalaris is quite simple: Its architecture consists of three layers.
It provides self-management and scalability by replicating services and data among peers. Without system interruption it scales from a few PCs to thousands of servers. Servers can be added or removed on the fly without any service downtime.



Scalaris takes care of:

- Fail-over
- Data distribution
- Replication
- Strong consistency
- Transactions

The Scalaris project was initiated by Zuse Institute Berlin and onScale solutions and is partly funded by the EU projects Selfman and XtreemOS. Additional information (papers, videos) can be found at http://www.zib.de/CSR/Projects/scalaris and http://www.onscale.de/scalaris.html.

2 Download and Installation

2.1 Requirements

For building and running Scalaris, some third-party modules are required which are not included in the Scalaris sources:

- Erlang R12
- Erlang OTP (included in Erlang R12)
- GNU Make

Note, the Version 12 of Erlang is required. Scalaris will not work with older versions.
To build the Java API the following modules are required additionally:

- Java Development Kit 1.6
- Apache Ant

Before building the Java API, make sure that **JAVA_HOME** and **ANT_HOME** are set. **JAVA_HOME** has to point to a JDK 1.6 installation, and **ANT_HOME** has to point to an Ant installation.

2.2 Download

The sources can be obtained from http://code.google.com/p/scalaris.

2.2.1 Development Branch

You find the latest development version in the svn repository:

```
# Non-members may check out a read-only working copy anonymously over HTTP.
svn checkout http://scalaris.googlecode.com/svn/trunk/ scalaris-read-only
```

2.2.2 Releases

Releases can be found under the 'Download' tab on the web-page.

2.3 Configuration

Scalaris reads two configuration files from the working directory: **bin/scalaris.cfg** (mandatory) and **bin/scalaris.local.cfg** (optional). The former defines default settings and is included in the release. The latter can be created by the user to alter settings. A sample file is **bin/scalaris.local.cfg.example**. A local configuration file is necessary to run Scalaris on distributed nodes:

File **scalaris.local.cfg**:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Settings for distributed Erlang
```

```
% (see cs_send.erl to switch)

% {boot_host, {boot,'boot@foo.bar.com'}}.
% {log_host, {boot_logger, 'boot@foo.bar.com'}}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Settings for TCP mode.
% (see cs_send.erl to switch)

% Insert the appropriate IP-addresses for your setup
% as comma separated integers:
% IP Address, Port, and label of the boot server
{boot_host, {{127,0,0,1},14195,boot}}.

% IP Address, Port, and label of the log server
{log_host, {{127,0,0,1},14195,boot_logger}}.
```

boot_host defines the node where the boot server is running, which is contacted to join the system.

2.4 Build

2.4.1 Linux

Scalaris uses autoconf for configuring the build environment and GNU Make for building the code.

```
%> ./configure
%> make
%> make docs
```

For more details read **README** in the main Scalaris checkout directory.

2.4.2 Windows

We are currently not supporting Scalaris on Windows. However, we have two small `bat` files for building and running a boot server. It seems to work but we make no guarantees.

- Install Erlang
- Install OpenSSL (for crypto module)
- Checkout scalaris code from SVN
- copy an appropriate EMakefile_for from **contrib/win32** to the trunk-directory
- Adapt the path to your Erlang installation in build.bat
- run build.bat
- Go to the bin sub-directory
- Adapt the path to your Erlang installation in boot.bat
- run boot.bat

2.4.3 Java-API

The following commands will build the Java API for Scalaris:

```
%> make java
```

This will build scalaris.jar, which is the library for accessing the overlay network. Optionally, the documentation can be build:

```
%> cd java-api
%> ant doc
```

2.5 Running Scalaris

In Scalaris there are two kinds of processes:

- boot servers
- regular servers

In every Scalaris, at least one boot server is required. It will maintain a list of nodes taken part in the system and allows other nodes to join the ring. For redundancy, it is also possible to have several boot servers.

2.5.1 Running on a local machine

Open at least two shells. In the first, go into the bin directory:

```
%> cd bin
%> ./boot.sh
```

This will start the boot server. On success http://localhost:8000 should point to the management interface page of the boot server. The main page will show you the number of nodes currently in the system. After a couple of seconds a first Scalaris should have started in the boot server and the number should increase to one. The main page will also allow you to store and retrieve key-value pairs.
In a second shell, you can now start a second Scalaris node. This will be a 'regular server'. Go in the bin directory:

```
%> cd bin
%> ./cs_local.sh
```

The second node will read the configuration file and use this information to contact the boot server and will join the ring. The number of nodes on the web page should have increased to two by now. Optionally, a third and fourth node can be started on the same machine. In a third shell:

```
%> cd bin
%> ./cs_local2.sh
```

In a fourth shell:

```
%> cd bin
%> ./cs_local3.sh
```

This will add 3 nodes to the network. The web pages at http://localhost:8000 should show the additional nodes.

2.5.2 Running distributed

Scalaris can be installed on other machines in the same way as described in Sect. 2.6. In the default configuration, nodes will look for the boot server on localhost on port 14195. You should create a **`scalaris.local.cfg`** pointing to the node running the boot server.

```
% Insert the appropriate IP-addresses for your setup
% as comma separated integers:
% IP Address, Port, and label of the boot server
{boot_host, {{127,0,0,1},14195,boot}}.
```

If you are using the default configuration on the boot server it will listen on port 14195 and you only have to change the IP address in the configuration file. Otherwise the other nodes will not find the boot server. On the remote nodes, you only need to call **./cs_local.sh** and they will automatically contact the configured boot server.

2.6 Installation

For simple tests, you do not need to install Scalaris. You can run it directly from the source directory. Note: **make install** will install scalaris into **/usr/local**. But is more convenient to build RPMs and install those.

```
svn checkout http://scalaris.googlecode.com/svn/trunk/ scalaris-0.0.1
tar -cvjf scalaris-0.0.1.tar.bz2 scalaris-0.0.1 --exclude-vcs
cp scalaris-0.0.1.tar.bz2 /usr/src/packages/SOURCES/
rpmbuild -ba scalaris-0.0.1/contrib/scalaris.spec
```

Your source and binary rpm will be generated in /usr/src/packages/SRPMS and RPMS. We also build rpms using checkouts from svn and provide them using the openSUSE BuildService at http://download.opensuse.org/repositories/home:/tschuett/. RPM packages are available for

- Fedora 9, 10,
- Mandriva 2008, 2009,
- openSUSE 11.0, 11.1,
- SLE 10, 11,
- CentOS 5 and
- RHEL 5.

Inside those repositories you will also find an erlang rpm - you don't need this if you already have a recent enough erlang version!

2.7 Logging

Scalaris uses the log4erl library (see **contrib/log4erl** for logging status information and error messages. The log level can be configured in **bin/scalaris.cfg**. The default value is `error`; only errors and severe problems are logged.

```
%% @doc Loglevel: debug < info < warn < error < fatal < none
{log_level, error}.
```

In some cases, it might be necessary to get more complete logging information, e.g. for debugging. In 5.2 on page 21, we are explaining the startup process of Scalarisnodes in more detail, here the `info` level provides more detailed information.

```
%% @doc Loglevel: debug < info < warn < error < fatal < none
{log_level, info}.
```

3 Using the system

3.1 JSON API

Scalaris supports a JSON API for transactions. To minimize the necessary round trips between a client and Scalaris, it uses request lists, which contain all requests that can be done in parallel. The request list is then send over to a Scalaris node with a POST message. The result is an opaque TransLog and a list containing the results of the requests. To add further requests to the transaction, the TransLog and another list of requests may be send to Scalaris. This process may be repeated as necessary. To finish the transaction, the request list can contain a 'commit' request as last element, which triggers the validation phase of the transaction processing.
The JSON-API can be accessed via the Scalaris-Web-Server running on port 8000 by default and the page **`jsonrpc.yaws`** (For example at: http://localhost:8000/jsonrpc.yaws). The following example illustrates the message flow:

Client → **Scalaris node**

Make a transaction, that sets two keys:

```
{
  "method":"req_list",
  "version":"1.1",
  "params":
    [
      [
        { "write":{"keyA":"valueA"} },
        { "write":{"keyB":"valueB"} },
        { "commit":"commit" }
      ]
    ],
  "id":0
}
```

← Scalaris sends results back

```
{ "result":
  { "results":
      [
        { "op":"commit",
          "value":"ok",
          "key":"ok" },
        { "op":"write",
          "value":"valueB",
          "key":"keyB" },
        { "op":"write",
          "value":"valueA",
          "key":"keyA" }
      ],
    "translog":
      [...]
  },
  "id" : 0
}
```

In a second transaction: Read the two keys →

```
{
  "method":"req_list",
  "version":"1.1",
  "params":
    [
      [
        { "read":"keyA" },
        { "read":"keyB" }
      ]
    ]
  "id":0
}
```

← Scalaris sends results back

```
{ "result":
  {"results":
    [
      { "op":"read",
        "value":"valueB",
        "key":"keyB" },
      { "op":"read",
        "value":"valueA",
        "key":"keyA" }
    ],
   "translog":
    [...] // this list is the translog
          // for further operations!
          // We name it TLOG here.
  },
  "id" : 0
}
```

Calculate something with the read values and make further requests, here a write and the commit for the whole transaction. Include also the latest translog we got from Scalaris (named **TLOG** here). →

```
{
  "method":"req_list",
  "version":"1.1",
  "params":
    [
      TLOG, // translog from prev. result.
      [
        { "write":{"keyA":"valueA2"} },
        { "commit":"commit" }
      ]
    ],
  "id" : 0
}
```

← Scalaris sends results back

```
{ "result":
  { "results":
    [ { "op":"commit",
        "value":"ok",
        "key":"ok" },
      { "op":"write",
        "value":"valueA2",
        "key":"keyA" }
    ],
    "translog":
    [...]
  },
  "id" : 0
}
```

A sample usage of the JSON API using Ruby can be found in **contrib/jsonrpc.rb**.
A single request list must not contain a key more than once!
The allowed requests are:

```
{ "read":"any_key" }

{ "write":{"any_key":"any_value"} }

{ "commit":"commit" }
```

The possible results are:

```
{ "op":"read", "key":"any_key", "value":"any_value" }
{ "op":"read", "key":"any_value", "fail":"reason" } // 'not_found' or 'timeout'

{ "op":"write",  "key":"any_key", "value":"any_value" }
{ "op":"read", "key":"any_key", "fail":"reason" }

{ "op":"commit", "value":"ok", "key":"ok" }
{ "op":"commit", "value":"fail", "fail":"reason" }
```

3.1.1 Deleting a key

Outside transactions keys can also be deleted, but it has to be done with care, as explained in the following thread on the mailing list: http://groups.google.com/group/scalaris/browse_thread/thread/ff1d9237e218799.

```
{
  "method":"delete",
  "version":"1.1",
  "params":
    [
      { "key":"any_key" }
    ],
  "id" : 0
}
```

Two sample results

```
{ "result":
  { "ok":2, // how many replicas were deleted successsfully
    "results": [ "ok", "ok", "locks_set", "undef" ]
  }
}
```

```
{ "result":
  { "failure":"reason" }
}
```

3.2 Java command line interface

The jar file contains a small command line interface client. For convenience, we provide a wrapper script called **scalaris** which setups the Java environment:

```
%> cd java-api
%> ./scalaris -help
usage: scalaris
 -g,--getsubscribers <topic>   get subscribers of a topic
 -help                         print this message
 -minibench                    run mini benchmark
 -p,--publish <params>         publish a new message for a topic: <topic>
                               <message>
 -r,--read <key>               read an item
 -s,--subscribe <params>       subscribe to a topic: <topic> <url>
 -u,--unsubscribe <params>     unsubscribe from a topic: <topic> <url>
 -w,--write <params>           write an item: <key> <value>
```

Read and write can be used to read resp. write from/to the overlay. getsubscribers, publish, and subscribe are the PubSub functions.

```
%> ./scalaris -write foo bar
write(foo, bar)
%> ./scalaris -read foo
read(foo) == bar
```

The scalaris library requires that you are running a 'regular server' on the same node. Having a boot server running on the same node is not sufficient.

3.3 Java API

The **scalaris.jar** provides the command line client as well as a library for Java programs to access Scalaris. The library provides two classes:

- **Scalaris** provides a high-level API similar to the command line client.
- **Transaction** provides a low-level API to the transaction mechanism.

For details we refer the reader to the Javadoc:

```
%> cd java-api
%> ant doc
%> firefox doc/index.html
```

4 Testing the system

4.1 Running the unit tests

There are some unit tests in the **test** directory. You can call them by running **make test** in the main directory. The results are stored in a local **index.html** file.
The tests are implemented with the **common-test** package from the Erlang system. For running the tests we rely on **run_test**, which is part of the **common-test** package, but is not installed by default. **configure** will check whether **run_test** is available. If it is not installed, it will show a warning and a short description of how to install the missing file.
Note: for the unit tests, we are setting up and shutting down several overlay networks. During the shut down phase, the runtime environment will print extensive error messages. These error messages do not indicate that tests failed! Running the complete test suite takes about 5 minutes. Only when the complete suite finished, it will present statistics on failed and successful tests.

Part II

Developers Guide

5 How a node joins the system

5.1 General Erlang server loop

Servers in Erlang often use the following structure to maintain a state while processing received messages:

```
receive
  Message ->
    State1 = f(State),
    loop(State1)
end.
```

The server runs an endless loop, that waits for a message, processes it and calls itself using tail-recursion in each branch. The loop works on a **State**, which can be modified when a message is handled.

5.2 Starting additional local nodes after boot

After booting a new Scalaris-System as described in Section 2.5.1 on page 11, ten additional local nodes can be started by typing **admin:add_nodes**(10) in the Erlang-Shell that the boot process opened [1].

```
scalaris/bin> ./boot.sh
[...]
=INFO REPORT==== 12-May-2009::16:24:18 ===
Yaws: Listening to 0.0.0.0:8000 for servers
 - http://localhost:8000 under ../docroot
[info] [ CC ] this() == {{127,0,0,1},14195}
[info] [ DNC <0.96.0> ] starting DeadNodeCache
[info] [ DNC <0.96.0> ] starting Dead Node Cache
[info] [ RM <0.97.0> ] starting ring maintainer

[info] [ RT <0.99.0> ] starting routingtable
[info] [ Node <0.101.0> ] joining 315238232250031455306327244779560426902
[info] [ Node <0.101.0> ] join as first 315238232250031455306327244779560426902
[info] [ FD <0.74.0> ] starting pinger for {{127,0,0,1},14195,<0.101.0>}
[info] [ Node <0.101.0> ] joined
[info] [ CY ] Cyclon spawn: {{127,0,0,1},14195,<0.102.0>}
(boot@csr-pc9)1> admin:add_nodes(10)
```

In the following we will trace, what this function does to join additional nodes to the system. The function **admin:add_nodes**(**int**) is defined as follows.

File **admin.erl**:

```
37 %%--------------------------------------------------------------------
38 %% Function: add_nodes(int()) -> ok
39 %% Description: add new Scalaris nodes
40 %%--------------------------------------------------------------------
41 % @doc add new Scalaris nodes on the local node
42 % @spec add_nodes(int()) -> ok
43
44 add_nodes(Count) ->
```

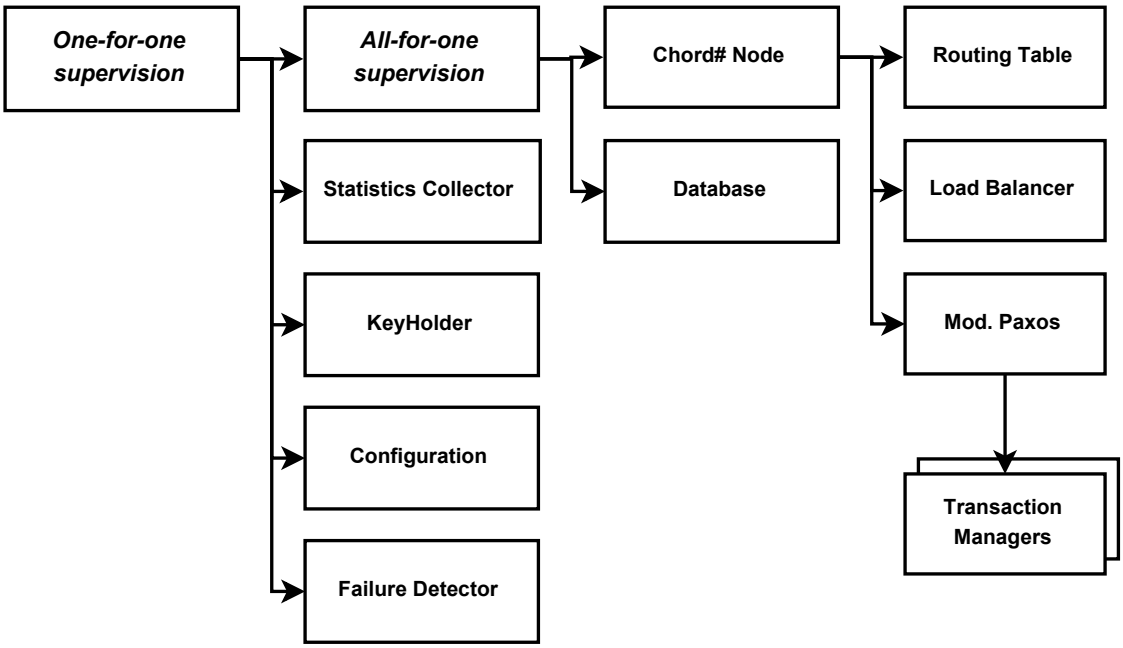
[1] Increase the log level to `info` to get the detailed startup logs. See Sect. 2.7 on page 12

```
45      add_nodes(Count, 0).
46
47  % @spec add_nodes(int(), int()) -> ok
48  add_nodes(Count, Delay) ->
49      add_nodes_loop(Count, Delay).
50
51  add_nodes_loop(0, _) ->
52      ok;
53  add_nodes_loop(Count, Delay) ->
54      supervisor:start_child(main_sup, {randoms:getRandomId(),
55                                        {cs_sup_or, start_link, []},
56                                        permanent,
57                                        brutal_kill,
58                                        worker,
59                                        []}),
60      timer:sleep(Delay),
61      add_nodes_loop(Count - 1, Delay).
```

It calls **`add_nodes_loop(Count, Delay)`** with a delay of 0. This function starts a new child for the main supervisor **`main_sup`**. As defined by the parameters, to actually perform the start, the function **`cs_sup_or:start_link`** is called by the Erlang supervisor mechanism. For more details on the OTP supervisor mechanism see Chapter 18 of the Erlang book [1] or the online documentation at http://www.erlang.org/doc/man/supervisor.html.

5.2.1 Supervisor-tree of a Scalaris node

When starting a new node in the system, the following supervisor tree is created:



5.2.2 Starting the or-supervisor and general processes of a node

Starting supervisors is a two step process: the supervisor mechanism first calls the **`init()`** function of the defined module (**`cs_sup_or:init()`** in this case) and then calls the start function (**`start_link`** here.
So, lets have a look at **`cs_sup_or:init`**, the 'Scalaris *or* supervisor'.

File **`cs_sup_or.erl`:**

```
61  init([Options]) ->
```

```
62      InstanceId = string:concat("cs_node_", randoms:getRandomId()),
63      boot_server:connect(),
64      KeyHolder =
65          {cs_keyholder,
66           {cs_keyholder, start_link, [InstanceId]},
67           permanent,
68           brutal_kill,
69           worker,
70           []},
71      RSE =
72          {rse_chord,
73           {rse_chord, start_link, [InstanceId]},
74           permanent,
75           brutal_kill,
76           worker,
77           []},
78      Supervisor_AND =
79          {cs_supervisor_and,
80           {cs_sup_and, start_link, [InstanceId, Options]},
81           permanent,
82           brutal_kill,
83           supervisor,
84           []},
85      RingMaintenance =
86          {?RM,
87           {?RM, start_link, [InstanceId]},
88           permanent,
89           brutal_kill,
90           worker,
91           []},
92      RoutingTable =
93          {routingtable,
94           {rt_loop, start_link, [InstanceId]},
95           permanent,
96           brutal_kill,
97           worker,
98           []},
99      DeadNodeCache =
100         {deadnodecache,
101          {dn_cache, start_link, [InstanceId]},
102          permanent,
103          brutal_kill,
104          worker,
105          []},
106     {ok, {{one_for_one, 10, 1},
107           [
108            KeyHolder,
109       DeadNodeCache,
110            RingMaintenance,
111            RoutingTable,
112            Supervisor_AND
113            %RSE
114           ]}}.
```

The return value of the **init()** function specifies the child processes of the supervisor and how to start them. Here, we define a list of processes to be observed by a **one_for_one** supervisor. The processes are: **KeyHolder**, **DeadNodeCache**, **RingMaintenance**, **RoutingTable**, and a **Supervisor_AND** process.

The term **{one_for_one, 10, 1}** specifies that the supervisor should try 10 times to restart each process before giving up. **one_for_one** supervision means, that if a single process stops, only that process is restarted. The other processes run independently.

The **cs_sup_or:init()** is finished and the supervisor module, starts all the defined processes by calling the functions that were defined in the list of the **cs_sup_or:init()**.

For a join of a new node, we are only interested in the starting of the **Supervisor_AND** process here. At that point in time, all other defined processes are already started and running.

5.2.3 Starting the and-supervisor with a peer and its local database

Again, the OTP will first call the **init()** function of the corresponding module:

File **cs_sup_and.erl:**

```
58 init([InstanceId, Options]) ->
59     Node =
60         {cs_node,
61          {cs_node, start_link, [InstanceId, Options]},
62          permanent,
63          brutal_kill,
64          worker,
65          []},
66     DB =
67         {?DB,
68          {?DB, start_link, [InstanceId]},
69          permanent,
70          brutal_kill,
71          worker,
72          []},
73     Cyclon =
74         {cyclon,
75          {cyclon.cyclon, start_link, [InstanceId]},
76          permanent,
77          brutal_kill,
78          worker,
79          []},
80     {ok, {{one_for_all, 10, 1},
81           [
82            DB,
83            Node,
84            Cyclon
85           ]}}.
```

It defines three processes, that have to be observed using an **one_for_all**-supervisor, which means, that if one fails, all have to be restarted. Passed to the **init** function is the **InstanceId**, a random number to make nodes unique. It was calculated a bit earlier in the code. Exercise: Try to find where.
As you can see from the list, the **DB** is started before the **Node**. This is intended and important, because **cs_node** uses the database, but not vice versa. The supervisor first completely initializes the DB process and afterwards calls **cs_node:start_link**. We only go into details here, for the latter.

File **cs_node.erl:**

```
378 %% @doc spawns a scalaris node, called by the scalaris supervisor process
379 %% @spec start_link(term()) -> {ok, pid()}
380 start_link(InstanceId) ->
381     start_link(InstanceId, []).
382
383 start_link(InstanceId, Options) ->
384     gen_component:start_link(?MODULE, [InstanceId, Options], [{register, InstanceId, cs_node}]).
```

cs_node implements the **gen_component** behaviour. This component was developed by us to enable us to write code which is similar in syntax and semantics to the examples in [2]. Similar to the **supervisor** behaviour, the component has to provide an **init** function, but here it is used to initialize the state of the component. This function is described in the next section.
Note: ?**MODULE** is a predefined Erlang macro, which expands to the module name, the code belongs to (here: **cs_node**).

5.2.4 Initializing a cs_node-process

File **cs_node.erl**:

```
356 %% @doc joins this node in the ring and calls the main loop
357 -spec(init/1 :: ([any()]) -> cs_state:state()).
358 init([_InstanceId, Options]) ->
359     case lists:member(first, Options) of
360         true ->
361             ok;
362         false ->
363             timer:sleep(crypto:rand_uniform(1, 100) * 100)
364     end,
365     Id = cs_keyholder:get_key(),
366     {First, State} = cs_join:join(Id),
367     if
368         not First ->
369             cs_replica_stabilization:recreate_replicas(cs_state:get_my_range(State));
370         true ->
371             ok
372     end,
373     log:log(info,"[ Node ~w ] joined",[self()]),
374     State.
```

The **gen_component** behaviour registers the **cs_node** in the process dictionary. Formerly, the process had to do this himself, but we moved this code into the behaviour. If the **cs_node** is the first node, he will start immediately. Otherwise, the process sleeps for a random amount of time. If you would start 1000 processes with **admin:add_nodes**(1000), the boot-server would receive many join requests at the same time, which is not intended. It will also make the ring stabilization process more complicated. Adding 100s of nodes within a short period of time induces more churn into the system, than the ring maintenance can handle.
Then, the node retrieves its **Id** from the keyholder: **Id** = **cs_keyholder:get_key**(). In the first call, a random identifier is returned, otherwise the latest set value. If the **cs_node**-process failed and is restarted by its supervisor, this call to the keyholder ensures, that the node still keeps its **Id**, assuming that the keyholder process is not failing. This is important for the load-balancing and for consistent responsibility of nodes to ensure consistent lookup in the structured overlay. Note: the name **Key-holder** actually is an id-holder.
If a node changes its position in the ring for load-balancing, the key-holder will be informed and the **cs_node** finishes itself. This triggers a restart of the corresponding database process via the and-supervisor. When the supervisor restarts both processes, they will retrieve the new position in the ring from the key-holder and join the ring there.
The supervisor was configured to restart a node at most 10 times. Does that mean, that a node can only change its position in the ring 10 times (caused by load-balancing)?

5.2.5 Actually joining the ring

After retrieving its identifier, the node starts the join process (**cs_join:join**).

File **cs_join.erl**:

```
87 %% @doc join a ring and return initial state
88 %%      the boolean indicates whether it was the first
89 %%      node in the ring or not
90 %% @spec join(Id) -> {true|false, state:state()}
91 %%   Id = term()
92 join(Id) ->
93     log:log(info,"[ Node ~w ] joining ~p",[self(), Id]),
94     Ringsize = boot_server:number_of_nodes(),
95     if
96         Ringsize == 0 ->
97             State = join_first(Id),
98             cs_reregister:reregister(),
```

```
99              {true, State};
100         true ->
101             case cs_lookup:reliable_get_node(erlang:get(instance_id),
102                                              Id, 60000) of
103                 error ->
104                     join(Id);
105                 {ok, Succ} ->
106                     State = join_ring(Id, Succ),
107                     cs_reregister:reregister(),
108                     {false, State}
109             end
110     end.
```

The boot-server is contacted to retrieve the known number of nodes in the ring. If the ring is empty, **join_first** is called. Otherwise, **join_ring** is called.
If the ring is empty, the joining node is the only node in the ring and will be responsible for the whole key space. **join_first** just creates a new state for a Scalaris node consisting of an empty routing table, a successorlist containing itself, itself as its predecessor, a reference to itself, its responsibility area from **Id** to **Id** (the full ring), and a load balancing schema.

File **cs_join.erl:**

```
50  %% @doc join an empty ring
51  join_first(Id) ->
52          log:log(info,"[ Node ~w ] join as first ~w",[self(), Id]),
53      Me = node:make(cs_send:this(), Id),
54      ?RM:initialize(Id, Me, Me, Me),
55      routingtable:initialize(Id, Me, Me),
56      cs_state:new(?RT:empty(Me), Me, Me, Me, {Id, Id}, cs_lb:new(), ?DB:new()).
```

The macro **?RT** maps to the configured routing algorithm and **?RM** to the configured ring maintenance algorithm. It is defined in **chordsharp.hrl**. For further details on the routing see Chapter 6 on page 29.
The state is defined in

File **cs_state.erl:**

```
57  new(RT, Successor, Predecessor, Me, MyRange, LB, DB) ->
58      #state{
59       routingtable = RT,
60       successor = Successor,
61       predecessor = Predecessor,
62       me = Me,
63       my_range = MyRange,
64       lb=LB,
65       join_time=now(),
66       deadnodes = gb_sets:new(),
67       trans_log = #translog{
68         tid_tm_mapping = dict:new(),
69         decided = gb_trees:empty(),
70         undecided = gb_trees:empty()
71        },
72       db = DB
73      }.
```

If a node joins an existing ring, **reliable_get_node** is called for the own **Id** in **cs_join:join()**. This lookup delivers the node who is currently responsible for the new node's identifier – the successor for the joining node. If this lookup fails for some reason, it is tried again, by recursivly calling the **join()**.
What, if the **Id** is exactly the same as that of the existing node? This could lead to lookup and responsibility inconsistency? Can this be triggered by the load-balancing? This is a bug, that should be fixed!!!
Then, **cs_join:join_ring** is called:

File **cs_join.erl**:

```
61 join_ring(Id, Succ) ->
62     log:log(info,"[ Node ~w ] join_ring ~w",[self(), Id]),
63     Me = node:make(cs_send:this(), Id),
64     UniqueId = node:uniqueId(Me),
65     cs_send:send(node:pidX(Succ), {join, cs_send:this(), Id, UniqueId}),
66     receive
67         {join_response, Pred, Data} ->
68             log:log(info,"[ Node ~w ] got pred ~w",[self(), Pred]),
69             case node:is_null(Pred) of
70                 true ->
71                     DB = ?DB:add_data(?DB:new(), Data),
72                     ?RM:initialize(Id, Me, Pred, Succ),
73                     routingtable:initialize(Id, Pred, Succ),
74                     cs_state:new(?RT:empty(Succ), Succ, Pred, Me, {Id, Id}, cs_lb:new(), DB);
75                 false ->
76                     cs_send:send(node:pidX(Pred), {update_succ, Me}),
77                     DB = ?DB:add_data(?DB:new(), Data),
78                     ?RM:initialize(Id, Me, Pred, Succ),
79                     routingtable:initialize(Id, Pred, Succ),
80                     cs_state:new(?RT:empty(Succ), Succ, Pred, Me, {node:id(Pred), Id},
81                                  cs_lb:new(), DB)
82             end
83     end.
```

First the node is initialized. Then it sends a **join** message to the successor including a reference to itself and the chosen **Id**.
The message is received by the old node in **cs_node.erl**. There exists a **{join, X}** handler.

File **cs_node.erl**:

```
302 on({join, Source_PID, Id, UniqueId}, State) ->
303     cs_join:join_request(State, Source_PID, Id, UniqueId);
```

This triggers a call to **join_request** on the old node.

File **cs_join.erl**:

```
39 join_request(State, Source_PID, Id, UniqueId) ->
40     Pred = node:new(Source_PID, Id, UniqueId),
41     {DB, HisData} = ?DB:split_data(cs_state:get_db(State), cs_state:id(State), Id),
42     cs_send:send(Source_PID, {join_response, cs_state:pred(State), HisData}),
43     ?RM:update_pred(Pred),
44     cs_state:set_db(State, DB).
```

The **cs_node** notifies the ring maintenance, that he has a new predecessor. Then he removes the key-value pairs from his database which are now in the responsibility of the joining node. Then it sends a **join_response** to the new node with its former predecessor, the data, it has to host, and its successorlist.
Back on the joining node: it waits for the **join_response** message in **cs_join:join_ring()**. The next steps after the message was received from the old node are to initialize the maintenance components for the ring and routing table, the database and the state of the **cs_node**.

5.2.6 Beginning to serve requests

cs_join:join() was called from **cs_node:start()**, which now continues

File **cs_node.erl**:

```
356 %% @doc joins this node in the ring and calls the main loop
357 -spec(init/1 :: ([any()]) -> cs_state:state()).
358 init([_InstanceId, Options]) ->
359     case lists:member(first, Options) of
360         true ->
```

```
361             ok;
362         false ->
363             timer:sleep(crypto:rand_uniform(1, 100) * 100)
364     end,
365     Id = cs_keyholder:get_key(),
366     {First, State} = cs_join:join(Id),
367     if
368         not First ->
369             cs_replica_stabilization:recreate_replicas(cs_state:get_my_range(State));
370         true ->
371             ok
372     end,
373     log:log(info,"[ Node ~w ] joined",[self()]),
374     State.
```

The **cs_replica_stabilization:recreate_replicas()** function is called, which is not yet implemented. It would recreated necessary replicas that were lost due to load-balancing and node failures.
Finally, the loop for request handling is started.

6 Routing and routing tables in the Overlay

Each node of the ring can perform searches in the overlay.
A search is done by a lookup in the overlay, but there are several other demands for communication between peers, so Scalaris provides a general interface to route a message to another peer, that is currently responsible for a given **key**.

File **cs_lookup.erl:**

```
[...]
unreliable_lookup(Key, Msg) ->
    get_pid(cs_node) ! {lookup_aux, Key, Msg}.

unreliable_get_key(Key) ->
    unreliable_lookup(Key, {get_key, cs_send:this(), Key}).
[...]
```

The message **Msg** could be a **get** which retrieves content from the responsible node or a **get_node** message, which returns a pointer to the node.
All currently supported messages are listed in the file **cs_node.erl**.
The message routing is implemented in lookup.erl

File **lookup.erl:**

```
[...]
lookup_fin(Msg) ->
    self() ! Msg.

lookup_aux(State, Key, Msg) ->
    Terminate = util:is_between(cs_state:id(State), Key, cs_state:succ_id(State)),
    P = ?RT:next_hop(State, Key),
    ?LOG("[ ~w | I | Node    | ~w ] lookup_aux ~w ~w ~s~n",
      [calendar:universal_time(), self(), Terminate, P, Key]),
    if
        Terminate ->
            cs_send:send(P, {lookup_fin, Msg});
        true ->
            cs_send:send(P, {lookup_aux, Key, Msg})
    end.
[...]
```

Each node is responsible for a certain key interval. The function **util:is_between** is used to decide, whether the key is between the current node and its successor. If that is the case, final step is done using **lookup_fin()**, which delivers the message to the local node. Otherwise, the message is forwarded to the next nearest known peer (listed in the routing table) determined by **?RT:next_hop**.
routingtable.erl is a generic interface for routing tables. It can be compared to interfaces in Java. In Erlang interfaces can be defined using a so called 'behaviour'. The files **rt_simple** and **rt_chord** implement the behaviour 'routingtable'.
The macro ?RT is used to select the current implementation of routing tables. It is defined in **chordsharp.hrl**.

File **chordsharp.hrl:**

```
26 %%This file determines which kind of routingtable is used. Uncomment the
27 %%one that is desired.
```

```
28
29 %%Standard Chord routingtable
30 -define(RT, rt_chord).
31
32 %%Simple routingtable
33 %-define(RT, rt_simple).
```

The functions, that have to be implemented for a routing mechanism are defined in the following file:

File **routingtable.erl**:

```
42 behaviour_info(callbacks) ->
43     [
44      % create a default routing table
45      {empty, 1},
46      % mapping: key space -> identifier space
47      {hash_key, 1}, {getRandomNodeId, 0},
48      % routing
49      {next_hop, 2},
50      % trigger for new stabilization round
51      {init_stabilize, 3},
52      % dead nodes filtering
53      {filterDeadNode, 2},
54      % statistics
55      {to_pid_list, 1}, {get_size, 1},
56      % for symmetric replication
57      {get_keys_for_replicas, 1},
58      % for debugging
59      {dump, 1},
60      % for bulkowner
61      {to_dict, 1}
62     ];
```

empty/1 gets a successor passed and generates an empty routing table. The data structure of the routing table is undefined. It can be a list, a tree, a matrix ...

hash_key/1 gets a key and maps it into the overlay's identifier space.

getRandomNodeId/0 returns a random node id from the overlay's identifier space. This is used for example when a new node joins the system.

next_hop/2 gets a routing table and a key and returns the node, that should be contacted next (is nearest to the id).

init_stabilize/3 is called periodically to rebuild the routing table. The parameters are the identifier of the node, the successor and the old routing table state.

filterDeadNode/2 is called by the failuredetector and tells the routing table about dead nodes to be eliminated from the routing table. This function cleans the routing table.

to_pid_list/1 get all PIDs of the routing table entries.

get_size/1 get the routing table's size.

get_keys_for_replicas/1 Returns for a given **Key** the keys of its replicas. This used for implementing symmetric replication.

dump/1 dump the state. Not mandatory, may just return **ok**.

to_dict/1 returns the routing tables entries in an array-like structure. This is used by bulk-operations to create a broadcast tree.

6.1 Simple routing table

One implementation of a routing table is the **rt_simple**, which routes via the successor, which is inefficient, as it needs a linear number of hops to reach its goal. A more robust implementation, would use a successor list. This implementation is not very efficient on churn.

6.1.1 Data types

First, the data structure of the routing table is defined:

File **rt_simple.erl**:

```
39 % @type key(). Identifier.
40 -type(key()::pos_integer()).
41 % @type rt(). Routing Table.
42 -ifdef(types_are_builtin).
43 -type(rt()::{node:node_type(), gb_tree()}).
44 -else.
45 -type(rt()::{node:node_type(), gb_trees:gb_tree()}).
46 -endif.
```

A routing table is a pair of a node (the successor) and an (unused) **gb_tree**. Keys in the overlay are identified by integers.

6.1.2 A simple routingtable behaviour

File **rt_simple.erl**:

```
50 %% @doc creates an empty routing table.
51 %%      per default the empty routing should already include
52 %%      the successor
53 -spec(empty/1 :: (node:node_type()) -> rt()).
54 empty(Succ) ->
55     {Succ, gb_trees:empty()}.
```

The empty routing table consists of the successor and an empty **gb_tree**.

File **rt_simple.erl**:

```
59 %% @doc hashes the key to the identifier space.
60 -spec(hash_key/1 :: (any()) -> key()).
61 hash_key(Key) ->
62     BitString = binary_to_list(crypto:md5(Key)),
63     % binary to integer
64     lists:foldl(fun(El, Total) -> (Total bsl 8) bor El end, 0, BitString).
```

Keys are hashed using MD5 and have a length of 128 bits.

File **rt_simple.erl**:

```
75 %% @doc returns the next hop to contact for a lookup
76 %% @spec next_hop(cs_state:state(), key()) -> pid()
77 next_hop(State, _Key) ->
78     cs_state:succ_pid(State).
```

Next hop is always the successor.

File **rt_simple.erl**:

```
82 %% @doc triggered by a new stabilization round
83 -spec(init_stabilize/3 :: (key(), node:node_type(), rt()) -> rt()).
84 init_stabilize(_Id, Succ, _RT) ->
85     % renew routing table
86     empty(Succ).
```

init_stabilize/3 resets its routing table with the current successor.

File **rt_simple.erl**:

```
90 %% @doc removes dead nodes from the routing table
91 -spec(filterDeadNode/2 :: (rt(), cs_send:mypid()) -> rt()).
92 filterDeadNode(RT, _DeadPid) ->
93     RT.
```

filterDeadNodes/2 does nothing, as only the successor is listed in the routing table and that is reset periodically in **init_stabilize**/3.

File **rt_simple.erl**:

```
97  %% @doc returns the pids of the routing table entries .
98  -spec(to_pid_list/1 :: (rt()) -> [cs_send:mypid()]).
99  to_pid_list({Succ, _RoutingTable} = _RT) ->
100     [node:pidX(Succ)].
```

to_pid_list/1 returns the pids of the routing tables, as defined in **node.erl**.

File **rt_simple.erl**:

```
109 normalize(Key) ->
110     Key band 16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.
111
112 %% @doc returns the replicas of the given key
113 -spec(get_keys_for_replicas/1 :: (key() | string()) -> [key()]).
114 get_keys_for_replicas(Key) when is_integer(Key) ->
115     [Key,
116      normalize(Key + 16#40000000000000000000000000000000),
117      normalize(Key + 16#80000000000000000000000000000000),
118      normalize(Key + 16#C0000000000000000000000000000000)
119     ];
120 get_keys_for_replicas(Key) when is_list(Key) ->
121     get_keys_for_replicas(hash_key(Key)).
```

The **get_keys_for_replicas**/1 implements symmetric replication, here. The call to **normalize** implements the modulo by throwing high bits away.

File **rt_simple.erl**:

```
126 %% @doc
127 -spec(dump/1 :: (rt()) -> ok).
128 dump(_State) ->
129     ok.
```

dump/1 is not implemented.

6.2 Chord routing table

The file **rt_chord.erl** implements Chord's routing.

6.2.1 Data types

File **rt_chord.erl**:

```
40  -type(key()::pos_integer()).
41  -ifdef(types_are_builtin).
42  -type(rt()::gb_tree()).
43  -else.
44  -type(rt()::gb_trees:gb_tree()).
45  -endif.
```

The routing table is a **gb_tree**. Identifiers in the ring are integers. Note, that in Erlang integer can be of arbitrary precision. For Chord, the identifiers are in $[0, 2^{128})$, i.e. 128-bit strings.

6.2.2 The routingtable behaviour for Chord

File **rt_chord.erl**:

```
49 %% @doc creates an empty routing table.
50 -spec(empty/1 :: (node:node_type()) -> rt()).
51 empty(_Succ) ->
52     gb_trees:empty().
```

empty/1 returns an empty **gb_tree**.
hash_key(**Key**) and **getRandomNodeId** call their counterparts from **rt_simple.erl**

File **rt_chord.erl**:

```
67 %% @doc returns the next hop to contact for a lookup
68 -spec(next_hop/2 :: (cs_state:state(), key()) -> cs_send:mypid()).
69 next_hop(State, Id) ->
70     case util:is_between(cs_state:id(State), Id, cs_state:succ_id(State)) of
71         %succ is responsible for the key
72         true ->
73             cs_state:succ_pid(State);
74         % check routing table
75         false ->
76             RT = cs_state:rt(State),
77             next_hop(cs_state:id(State), RT, Id, 127, cs_state:succ_pid(State))
78     end.
```

next_hop traverses the routing table beginning with the longest finger (2^{127}) by calling the helper function **next_hop**/5.

File **rt_chord.erl**:

```
82 % @private
83 -spec(next_hop/5 :: (key(), rt(), key(), pos_integer(), cs_send:mypid()) -> cs_send:mypid()).
84 next_hop(_N, _RT, _Id, 0, Candidate) -> Candidate;
85 next_hop(N, RT, Id, Index, Candidate) ->
86     case gb_trees:lookup(Index, RT) of
87         {value, Entry} ->
88             case util:is_between_closed(N, node:id(Entry), Id) of
89                 true ->
90                     node:pidX(Entry);
91                 false ->
92                     next_hop(N, RT, Id, Index - 1, Candidate)
93             end;
94         none ->
95             next_hop(N, RT, Id, Index - 1, Candidate)
96     end.
```

If the entry exists, it is retrieved from the **gb_tree**. If the id of the routing table entry is between ourselves and the searched id, the finger is chosen. If anything fails, **Candidate** (the successor) is chosen.
Why could a routing table entry be **null**? **filterDeadNodes** changes entries to **null**.
BUG: Instead of directly returning **Candidate** one should further traverse the routing table for shorter appropriate fingers. If doing so, a check whether **Index** is zero, would become necessary.
If the finger is to long, recursively try the next shorter finger.

File **rt_chord.erl**:

```
100 %% @doc starts the stabilization routine
101 -spec(init_stabilize/3 :: (key(), node:node_type(), rt()) -> rt()).
102 init_stabilize(Id, Succ, RT) ->
103     % calculate the longest finger
104     Key = calculateKey(Id, 127),
```

```
105     % trigger a lookup for Key
106     cs_lookup:unreliable_lookup(Key, {rt_get_node, cs_send:this(), 127}),
107     cleanup(gb_trees:iterator(RT), RT, Succ).
```

The routing table stabilization is triggered with the index 127 and then runs asynchronously, as we do not want to block the **rt_loop** to perform other request while recalculating the routing table. We have to find the node responsible for the calculated finger and therefore perform a lookup for the node with a **rt_get_node** message, including a reference to ourselves as the reply-to address and the index to be set.
The lookup performs an overlay routing by passing the massage until the responsible node is found. There, the message is delivered to the **cs_node**. At the destination the message is handled in **cs_node.erl**:

File **cs_node.erl**:

```
193 on({rt_get_node, Source_PID, Cookie}, State) ->
194     cs_send:send(Source_PID, {rt_get_node_response, Cookie, cs_state:me(State)}),
195     State;
```

The remote node just sends the requested information back directly in a **rt_get_node_response** message including a reference to itself. When receiving the routing table entry, we call **stabilize**/5.

File **rt_chord.erl**:

```
142 %% @doc updates one entry in the routing table
143 %%      and triggers the next update
144 -spec(stabilize/5 :: (key(), node:node_type(), rt(), pos_integer(), node:node_type()) -> rt()).
145 stabilize(Id, Succ, RT, Index, Node) ->
146     case node:is_null(Node) of
147         true ->
148             RT;
149         false ->
150             case (node:id(Succ) == node:id(Node)) or (Id == node:id(Node)) or (Index == -1) of
151                 true ->
152                     % delete lower entries
153                     prune_table(RT, Index);
154                 false ->
155                     NewRT = gb_trees:enter(Index, Node, RT),
156                     Key = calculateKey(Id, Index - 1),
157                     cs_lookup:unreliable_lookup(Key, {rt_get_node, cs_send:this(), Index - 1}),
158                     NewRT
159             end
160     end.
```

stabilize/5 assigns the received routing table entry and triggers to fill the next shorter one using the same mechanisms as described.
When the shortest finger is the successor, then filling the routing table is stopped, as no further new entries would occur. It is not necessary, that **Index** reaches 1 to make that happen. If less than 2^{128} nodes participate in the system, it may happen earlier.
filterDeadNode removes dead entries from the **gb_tree**.

File **rt_chord.erl**:

```
111 %% @doc remove all entries
112 -spec(filterDeadNode/2 :: (rt(), cs_send:mypid()) -> rt()).
113 filterDeadNode(RT, DeadPid) ->
114     DeadIndices = [Index|| {Index, Node}  <- gb_trees:to_list(RT),
115                            node:pidX(Node) == DeadPid],
116     lists:foldl(fun (Index, Tree) -> gb_trees:delete(Index, Tree) end,
117                 RT, DeadIndices).
```

7 Directory Structure of the Source Code

The directory tree of Scalaris is structured as follows:

bin	contains shell scripts needed to work with Scalaris (e.g. start the boot services, start a node, ...)
contrib	necessary third party packages (yaws and log4erl)
doc	generated erlang documentation
docroot	root directory of the bootserver's webserver
docroot_node	root directory of the normal node's webserver
ebin	the compiled Erlang code (beam files)
java-api	a java api to Scalaris
log	log files
src	contains the Scalaris source code
test	unit tests for Scalaris
user-dev-guide	contains the sources for this document

8 Java API

For the Java API documentation, we refer the reader to Javadoc resp. doxygen. The following commands create the documentation:

```
%> cd java-api
%> ant doc
%> doxygen
```

The Javadoc can be found in **java-api/doc/index.html**. The doxygen files are in **doc-doxygen/html/index.html**.
We provide two kinds of APIs:

- high-level access with **de.zib.scalaris.Scalaris**
- low-level access with **de.zib.scalaris.Transaction**

The former provides general functions for reading and writing single key-value pairs and an API for the built-in PubSub-service. The latter allows the user to write custom transactions which can modify an arbitrary number of key-value pairs within one transaction.

Bibliography

[1] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007

[2] Rachid Guerraoui and Luis Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag, 2006.

A.9 Scalaris: Reliable Transactional P2P Key/-Value Store

Scalaris: Reliable Transactional P2P Key/Value Store

Web 2.0 Hosting with Erlang and Java

Thorsten Schütt Florian Schintke Alexander Reinefeld

Zuse Institute Berlin and onScale solutions

schuett@zib.de, schintke@zib.de, reinefeld@zib.de

Abstract

We present *Scalaris*, an Erlang implementation of a distributed key/value store. It uses, on top of a structured overlay network, replication for data availability and majority based distributed transactions for data consistency. In combination, this implements the ACID properties on a scalable structured overlay.

By directly mapping the keys to the overlay without hashing, arbitrary key-ranges can be assigned to nodes, thereby allowing a better load-balancing than would be possible with traditional DHTs. Consequently, Scalaris can be tuned for fast data access by taking, e.g. the nodes' geographic location or the regional popularity of certain keys into account. This improves Scalaris' lookup speed in datacenter or cloud computing environments.

Scalaris is implemented in Erlang. We describe the Erlang software architecture, including the transactional Java interface to access Scalaris.

Additionally, we present a generic design pattern to implement a responsive server in Erlang that serializes update operations on a common state, while concurrently performing fast asynchronous read requests on the same state.

As a proof-of-concept we implemented a simplified Wikipedia frontend and attached it to the Scalaris data store backend. Wikipedia is a challenging application. It requires—besides thousands of concurrent read requests per seconds—serialized, consistent write operations. For Wikipedia's category and backlink pages, keys must be consistently changed within transactions. We discuss how these features are implemented in Scalaris and show its performance.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed databases; C.2.4 [*Distributed Systems*]: Distributed applications; D.2.11 [*Software architectures*]: Patterns; E.1 [*Data structures*]: Distributed data structures

General Terms Algorithms, Design, Languages, Management, Reliability

Keywords Wikipedia, Peer-to-Peer, transactions, key/value store

Erlang'08, September 27, 2008, Victoria, BC, Canada.

1. Introduction

Global e-commerce platforms require highly concurrent access to distributed data. Millions of read operations must be served within milliseconds even though there are concurrent write accesses. Enterprises like Amazon, eBay, Myspace, YouTube, or Google solve this problems by operating tens or hundreds of thousands of servers in distributed datacenters. At this scale, components fail continuously and it is difficult to maintain a consistent state while hiding failures from the application.

Peer-to-peer protocols provide self-management among peers, but they are mostly limited to write-once/read-many data sharing. To extend them beyond the typical file sharing, the support of consistent replication and fast transactions is an important yet missing feature.

We present *Scalaris*, a scalable, distributed key/value store. Scalaris is built on a structured overlay network and uses a distributed transaction protocol, both of them implemented in Erlang with an application interface to Java. To prove our concept, we implemented a simple Wikipedia clone on Scalaris which performs several thousand transactions per second on just a few servers.

In this paper, we give details on the design and implementation of Scalaris. We highlight Erlang specific topics and illustrate algorithm details with code samples. Talks on Scalaris were given at the IEEE International Scalable Computing Challenge 2008[1], the Google Scalability Conference 2008 [15] and the Erlang eXchange 2008.

The paper is organized as follows. After a brief review of related work we describe the overall system architecture and then discuss implementation aspects in Section 4. In Section 5, we present a generic design pattern of a responsive, stateful server, which is used in Scalaris. We then present our example application, a distributed Wikipedia clone in Section 6 and we end with a conclusion.

2. Related Work

Scalable, transactional data stores are of key interest to the community and hence there exists a wide variety of related work. Amazon's key/value store Dynamo [3] and its commercial counterpart SimpleDB which is used in the S3 service, are similar to our work, because they are also based on a scalable P2P substrate. But in contrast to Scalaris, they implement only eventual consistency rather than strong consistency. Moreover, Dynamo does not support transactions over multiple items.

The work of Baldoni *et al.* [2] focuses on algorithms for the creation of dynamic quorums in P2P overlays—an issue that is of particular relevance for the transaction layer in Scalaris. They show that in P2P systems the quorum acquisition time and the message latency are more important than the quorum size, which has been

[1] Scalaris won the 1^{st} price at SCALE 2008, www.ieeetcsc.org/scale2008

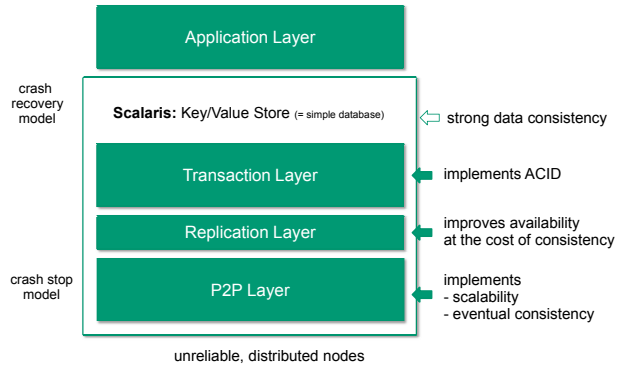


Figure 1. Scalaris system architecture.

traditionally used as a performance metric in distributed systems. This is in line with our results showing that an increasing replication degree r only marginally affects the access time, because the replicas residing in the $\lceil (r+1)/2 \rceil$ fastest nodes take part in the consensus process.

Masud *et al.* [10] also discuss database transactions on structured overlays, but with a focus on the consistent execution of transactions in the presence of failing nodes. They argue that executing transactions over the acquaintances of peers speeds up the transaction time and success rate. Scalaris has a similar concept, but here the peer 'acquaintances' are realized by the load balancer.

With Cassandra [8] and Megastore [4], Facebook and Google recently presented two databases based on the P2P paradigm. Megastore extends Bigtable with support of transactions and multiple indices. Cassandra is more similar to Dynamo as it also provides eventual consistency.

3. System Architecture

Scalaris is a distributed key/value store based on a structured P2P overlay that supports consistent writes. The system comprises three layers (Fig. 1):

- At the bottom, a structured overlay network with logarithmic routing performance builds the basis for the key/value store. In contrast to many other DHTs, our overlay stores the keys in lexicographical order, hence efficient range queries are possible.
- The middle layer implements replication and ACID properties (atomicity, concurrency, isolation, durability) for concurrent write operations. It uses a Paxos consensus protocol [9] which is integrated into the overlay protocol to ensure low communication overhead.
- The top layer hosts the application, a distributed key/value store. This layer can be used as a scalable, fault-tolerant backend for online services for shopping, banking, data sharing, online gaming, or social networks.

Fig. 1 illustrates the three layers. The following sections describe them in more detail.

3.1 P2P Overlay

At the bottom layer, the structured overlay protocol Chord$^{\#}$ [13, 14] is used for storing and retrieving key/value pairs in nodes (peers) that are arranged in a virtual ring. In each of the N nodes, Chord$^{\#}$ maintains a routing table with $O(\log N)$ entries (fingers). In contrast to Chord [17], Chord$^{\#}$ stores the keys in lexicographical order, thereby allowing range queries. To ensure logarithmic

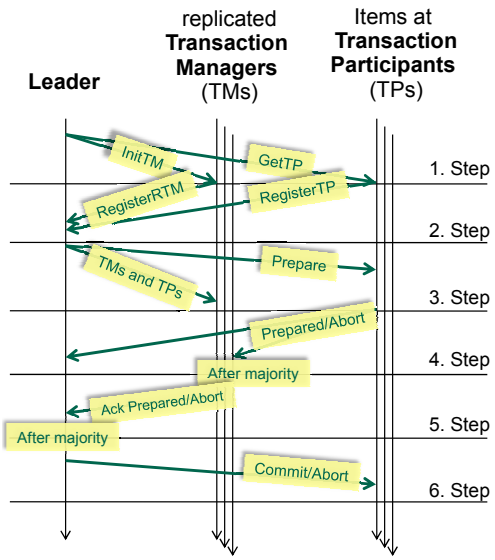


Figure 2. Adapted Paxos used in Scalaris.

routing performance, the fingers in the routing table are computed in such a way that successive fingers in the routing table cross an exponentially increasing number of nodes in the ring.

Chord$^{\#}$ uses the following algorithm for computing the fingers in the routing table (the infix operator $x \ . \ y$ retrieves y from the routing table of a node x):

$$finger_i = \begin{cases} successor & : i = 0 \\ finger_{i-1} \ . \ finger_{i-1} & : i \neq 0 \end{cases}$$

Thus, to calculate the i^{th} finger, a node asks the remote node listed in its $(i-1)^{th}$ finger to which node his $(i-1)^{th}$ finger refers to. In general, the fingers in level i are set to the fingers' neighbors in the next lower level $i-1$. At the lowest level, the fingers point to the direct successors. The resulting structure is similar to a skiplist, but the fingers are computed deterministically without any probabilistic component.

Compared to Chord, Chord$^{\#}$ does the routing in the *node space* rather than the *key space*. This finger placement has two advantages over that of Chord: First, it works with any type of keys as long as a total order over the keys is defined, and second, finger updates are cheaper, because they require just one hop instead of a full search (as in Chord). A proof of Chord$^{\#}$'s logarithmic routing performance can be found in [13].

3.2 Replication and Transaction Layer

The scheme described so far provides scalable access to distributed key/value pairs. To additionally tolerate node failures, we replicate all key/value pairs over r nodes using symmetric replication [5]. Read and write operations are performed on a majority of the replicas, thereby tolerating the unavailability of up to $\lfloor (r-1)/2 \rfloor$ nodes.

Each item is assigned a version number. Read operations select the item with the highest version number from a majority of the replicas. Thus a single read operation accesses $\lceil (r+1)/2 \rceil$ nodes, which is done in parallel.

Write operations are done with an adapted Paxos atomic commit protocol [11]. In contrast to the 3-Phase-Commit protocol (3PC) used in distributed database systems, the adapted Paxos is non-blocking, because it employs a group of *acceptors* rather than a

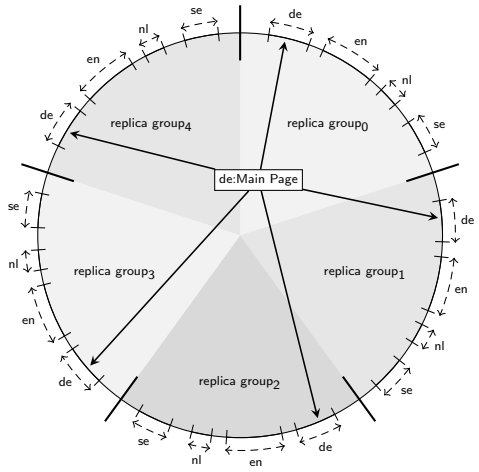


Figure 3. Symmetric replication and multi-datacenter scenario. By assigning the majority of the 'de'-, 'nl'-, and 'se'-replicas to nodes in Europe, latencies can be reduced.

single transaction manager. We select those nodes as acceptors that are responsible for symmetric replication of the transaction manager. The group of acceptors is determined by the transaction manager just before the prepare request is sent to the transaction participants (Fig. 2). This gives a pseudo static group of transaction participants at validation time, which is contacted in parallel.

Write operations and transactions need three phases, including the phase to determine the nodes that participate in the atomic commit. For details see [11, 16].

In Scalaris, the adapted Paxos protocol serves two purposes: First it ensures that all replicas of a *single* key are updated consistently, and second it is used for implementing transactions over *multiple* keys, thereby realizing the ACID properties (atomicity, concurrency, isolation, durability).

3.3 Deployment in Global Datacenters

While we also tested Scalaris on globally distributed servers using PlanetLab[2], its deployment in globally distributed datacenters is more relevant for international service providers. In such scenarios, the latency between the peers is roughly the same and the peers are in general more reliable.

When deploying Scalaris in multi-datacenter environments, a single structured overlay will span over all datacenters. The location of replicas will influence the access latency and thereby the response time perceived by the user. As Chord$^{\#}$ supports explicit load-balancing, it can—besides adapting to e.g. heterogeneous hardware and item popularity—place the replicas in specific centers. A majority of replicas of German Wiki pages, for example, should be placed in European datacenters to reduce the access latency for German users.

Scalaris uses symmetric replication [5]. Here, a key 'de:Main Page' is stored in five different locations in the ring (see Fig. 3). The locations are determined by prefixing the key with '0', '1', ..., '5'. So the key of the third replica is '2de:Main Page' and the third replicas of all German articles will populate a consecutive part of the ring. By influencing the load-balancing strategy we can guarantee this segment to be always hosted in a particular datacenter.

[2] http://www.planet-lab.org

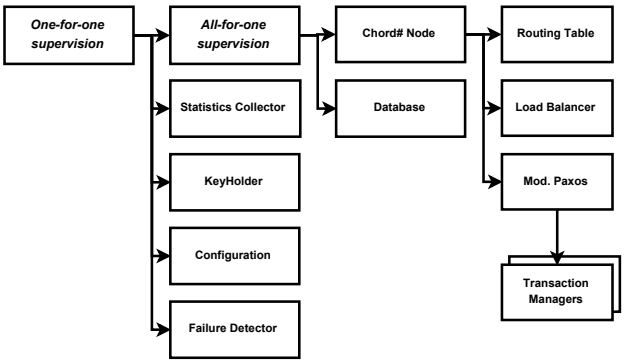


Figure 4. Supervisor tree of a Scalaris node. Each box represents one process.

4. Erlang Implementation

The *actor model* [7] is a popular model for designing and implementing parallel or distributed algorithms. It is often used in the literature [6] to describe and to reason about distributed algorithms. Chord$^{\#}$ and the transaction algorithms described above were also developed according to this model. The basic primitives in this model are actors and messages. Every actor has a state, can send messages, act upon messages and spawn new actors.

These primitives can be easily mapped to Erlang processes and messages. The close relationship between the theoretical model and the programming language allows a smooth transition from the theoretical model to prototypes and eventually to a complete system.

Our Erlang implementation of Scalaris comprises many components. It has a total of 11,000 lines of code: 7,000 for the P2P layer with replication and basic system infrastructure, 2,700 lines for the transaction layer, and 1,300 lines for the Wikipedia infrastructure.

4.1 Components and Supervisor Tree

Scalaris is a distributed algorithm. Each peer runs a number of processes as shown in Fig. 4:

Failure Detector supervises other peers and sends a `crash` message when a node failure is detected.

Configuration provides access to the configuration file and maintains parameter changes made at runtime.

Key Holder stores the identifier of the node in the overlay.

Statistics Collector collects statistics and forwards them to central statistic servers.

Chord$^{\#}$ Node performs all important functions of the node. It maintains, among other things, the successor list and the routing table.

Database stores the key/value pairs of this node. The current implementation uses an in-memory dictionary, but disk store based on DETS or Mnesia could also be used.

The processes are organized in a supervisor tree as illustrated in Fig. 4. The first four processes are supervised by a *one-for-one supervisor* [1]: When a slave crashes, it is restarted by the supervisor. The right-most processes (Chord$^{\#}$ Node and Database) are supervised by an *all-for-one supervisor* which restarts *all* slaves when a single slave crashed. In Scalaris, when either of the Chord$^{\#}$ Node or the Database process fails, the other is explicitly killed and both are restarted to ensure consistency.

4.2 Naming Processes

In Erlang, there are two ways of sending messages to processes: by process id or by addressing the name registered as an atom. This scheme provides a flat name space. We implemented a hierarchical name space for processes.

As described in Sec. 4.1, each Chord$^{\#}$ node comprises a group of processes. Within this group, we address processes by name. For example, the failure detector can be addressed as failure_detector.

Running several Chord$^{\#}$ nodes within one Erlang Virtual Machine (VM) would lead to name clashes. Hence, we implemented a hierarchical process name space where each Chord$^{\#}$ node forms a 'process group'. As a side-effect, we can traverse the naming hierarchy to provide monitoring information grouped by Chord$^{\#}$ nodes.

For this naming scheme, every process stores its group id in its own process dictionary. At startup time, processes announce their name and process identifier to a dictionary inside the VM, which is handled by a separate process in the VM. It can be queried to find processes by name or by traversing the process hierarchy. Additionally, most Chord$^{\#}$ processes support the {'$gen_cast', {debug_info, Requestor}} message, which allows processes to provide custom monitoring information to the web interface.

4.3 WAN Deployment

Erlang provides the 'distributed mode' for small and medium deployments with limited security requirements. This makes it easy to port the application from an Erlang VM to a cluster. In large deployments, however, the network traffic caused by the management tasks within the VM dominates the overall traffic.

In our code, we replaced the '!' operator and the self() function by cs_send:send() resp. cs_send:this(). At compile time we can configure the cs_send module to use the Erlang distributed mode or our own transport layer using TCP/IP, which will be based on the Erlang SSL library in the future.

This approach also allows us to separate the application logic from the transport layer. Hence, NAT traversal schemes and firewall-aware communication can be implemented without the need to change Chord$^{\#}$ code.

4.4 Transaction Interface

Transactions are executed in two phases, the read phase and the commit phase. The read phase goes through all operations of the transaction and keeps the result of each operation in the transaction log. During this phase, the state of the system remains unchanged. In the commit phase, the recorded effects are applied to the database when the ACID properties are not violated.

Read phase. For the read phase, we use a lambda expression which describes the individual operations to be performed in the transaction (see Alg. 4.1). The mentioned transaction log is passed through all calls to the transaction API and updated accordingly. Passing a function to the transaction framework allows us to easily re-execute a transaction after a failure due to concurrency.

Commit phase. The commit phase is started by calling do_transaction (see last line in Alg. 4.1). The transaction is executed asynchronously. The function spawns a new process and returns immediately. The ProcessId which is passed will be notified of the outcome of the transaction. The SuccessFun resp. FailureFun are applied to the result of the transaction before the result is sent back. For the Scalaris implementation, we use the two functions to include transaction numbers into the status messages when a process has several outstanding transactions.

We use the Jinterface package to enable Java programs to perform transactions. The transaction log is managed by the Java program. On a commit the complete log is passed to Erlang and the do_transaction function. Note that transaction descriptions in Java are usually more compact because error handling is done using exceptions (see Alg. 4.2) while in Erlang, the error handling is done in the actual code.

Algorithm 4.1 Incrementing the key *Increment* inside a transaction

```
run_test_increment(State, Source_PID)->
   % the transaction
   TFun = fun(TransLog) ->
      Key = "Increment",
      {Result, TransLog1} = transaction_api:read(Key, TransLog),
      {Result2, TransLog2} =
         if Result == fail ->
               Value = 1,                    % new key
               transaction_api:write(Key, Value, TransLog);
            true ->
               {value, Val} = Result,        % existing key
               Value = Val + 1,
               transaction_api:write(Key, Value, TransLog1)
         end,
      % error handling
      if Result2 == ok ->
         {{ok, Value}, TransLog2};
         true -> {{fail, abort}, TransLog2}
      end
   end,
   SuccessFun = fun(X) -> {success, X} end,
   FailureFun =
     fun(Reason)-> {failure, "test increment failed", Reason} end,

   % trigger transaction
   transaction:do_transaction(State, TFun, SuccessFun,
                              FailureFun, Source_PID).
```

Algorithm 4.2 Java Transactions

```
// new Transaction object
Transaction transaction = new Transaction();
// start new transaction
transaction.start();

//read account A
int accountA =
   new Integer(transaction.read("accountA")).intValue();
//read account B
int accountB =
   new Integer(transaction.read("accountB")).intValue();

//remove 100$ from accountA
transaction.write("accountA",
   new Integer(accountA - 100).toString());
//add 100$ to account B
transaction.write("accountB",
   new Integer(accountB + 100).toString());

transaction.commit();
```

5. Responsive, Stateful Server in Erlang

In distributed server software, slow write operations often block faster reads. Alg. 5.1 shows a generic server architecture (design pattern) that manages reads and writes on a shared state separately. This is done in such a way that read requests can be immediately answered even though a concurrent write operation still blocks the process. Two processes manage the shared state: a public asyn-

Algorithm 5.1 Responsive, stateful server

```
-module(account).
-export([start/0,syncloop/2,slowbalance/2]).

newAccount() -> 0.
start() -> spawn(fun() ->
      Account = newAccount(),
      SyncLoopPid = spawn(account, syncloop, [self(), Account]),
      asyncloop(SyncLoopPid, Account)
    end).

% all requests have to be send to the asyncloop
% read from State via spawns, if its a slow read
% forward writes to the syncloop
asyncloop(SyncLoopPid, State) ->
  receive
  {updatestate, StateNew} ->
    % for better consistency make a join for all spawned
    %   slow reads here
    % for better security, only allow the syncloop
    %   process to update the state
    asyncloop(SyncLoopPid, StateNew);
  {balance, Pid} ->
    Pid ! State,
    asyncloop(SyncLoopPid, State);
  {slowbalance, Pid} ->
    spawn(account, slowbalance, [State, Pid]),
    asyncloop(SyncLoopPid, State);
  % all other messages go to the synchronous loop
  Message ->
      SyncLoopPid ! Message,
      asyncloop(SyncLoopPid, State)
  end.

% internally use a syncloop to serialize all State changes
syncloop(AsyncLoopPid, State) ->
  receive
  {credit, Amount} ->
    NewState = State + Amount,
    AsyncLoopPid ! {updatestate, NewState},
    syncloop(AsyncLoopPid, NewState);
  {draw, Amount} ->
     NewState = State - draw(Amount),
      AsyncLoopPid ! {updatestate, NewState},
     syncloop(AsyncLoopPid, NewState);
    _ ->
      syncloop(AsyncLoopPid, State)
  end.

% functions, that take some time to be executed
slowbalance(State, Pid) ->
  receive
  after 60000 ->
    Pid ! State
  end.

draw(Amount) ->
  receive
  % the bank still works with your money for 10 seconds
  after 10000 ->
    Amount
  end.
```

chronous receive loop asyncloop that performs the reads and forwards the write requests to a private synchronous receive loop syncloop. By this means, write requests are serialized and there is a local atomic point in time when the state changes.

Slow reads may still deliver outdated state. This can be overcome by waiting for all outstanding reads to be completed before changing the state in the asyncloop (not depicted in the algorithm).

Example. Alg. 5.1 shows the processing of states for a bank account. The server provides two read requests (balance and slowbalance) and two write requests (credit and draw) for managing an account. Clients send all their requests to the asyncloop. The server is started by calling account:start(). This spawns a process, which first initializes the account with zero, spawns the syncloop with a reference to itself, and finally executes the asyncloop.

On a balance or slowbalance request to the asyncloop, the account balance is returned to the requesting process from the current state. In case of slowbalance the state is given to a spawned process, which is then executed concurrently in the background. In practice, this spawning should be used when some calculations or other time consuming tasks must be executed on the state before the request can be answered. This way, other requests can be performed by the server concurrently. Here, the corresponding function slowbalance just waits 60 seconds before delivering the result.

In addition, the asyncloop handles updatestate requests as discussed below. All other messages are forwarded to the syncloop.

The syncloop handles the write requests credit and draw. All other messages are ignored and dropped. The syncloop must not spawn processes to calculate state changes, as all state manipulation must be serial to ensure consistency. Here, the draw takes 10 seconds to be performed (the bank uses this time to work with your money). This time has to be consumed synchronously. In practice this could be a time consuming calculation which is necessary to determine the new state. After having calculated the new state, syncloop sends the state with an updatestate request to the asyncloop and works on the new state by itself.

When the asyncloop receives an updatestate message from the syncloop it takes over the new state from the message. This is the atomic point in time when the write request becomes active, as all future requests will operate on this new state.

This leads to a relaxed consistency in the server that is sufficient for updating the routing tables and successor lists. Here, relaxed consistency does not harm, because these tables are subject to churn and will be periodically updated with unreliable link information anyway. If a stronger consistency model is needed, the transaction mechanism of the Erlang Mnesia database package could be used.

6. Use Case: Wikipedia

To demonstrate Scalaris' performance, we chose Wikipedia, the 'free encyclopedia, that anyone can edit', as a challenging test application. In contrast to the public Wikipedia, which is operated on three clusters in Tampa, Amsterdam, and Seoul, our Erlang implementation can be deployed on worldwide distributed servers. We ran it in two installations, one on PlanetLab and one on a local cluster.

The public Wikipedia uses PHP to render the Wikitext to HTML and stores the content and page history in MySQL databases. Instead of using a relational database, we map the Wikipedia content to our Scalaris key/value store [12]. We use the following mappings, using prefixes in the keys to avoid name clashes:

	key	value
page content	title	list of Wikitext for all versions
backlinks	title	list of titles
categories	category name	list of titles

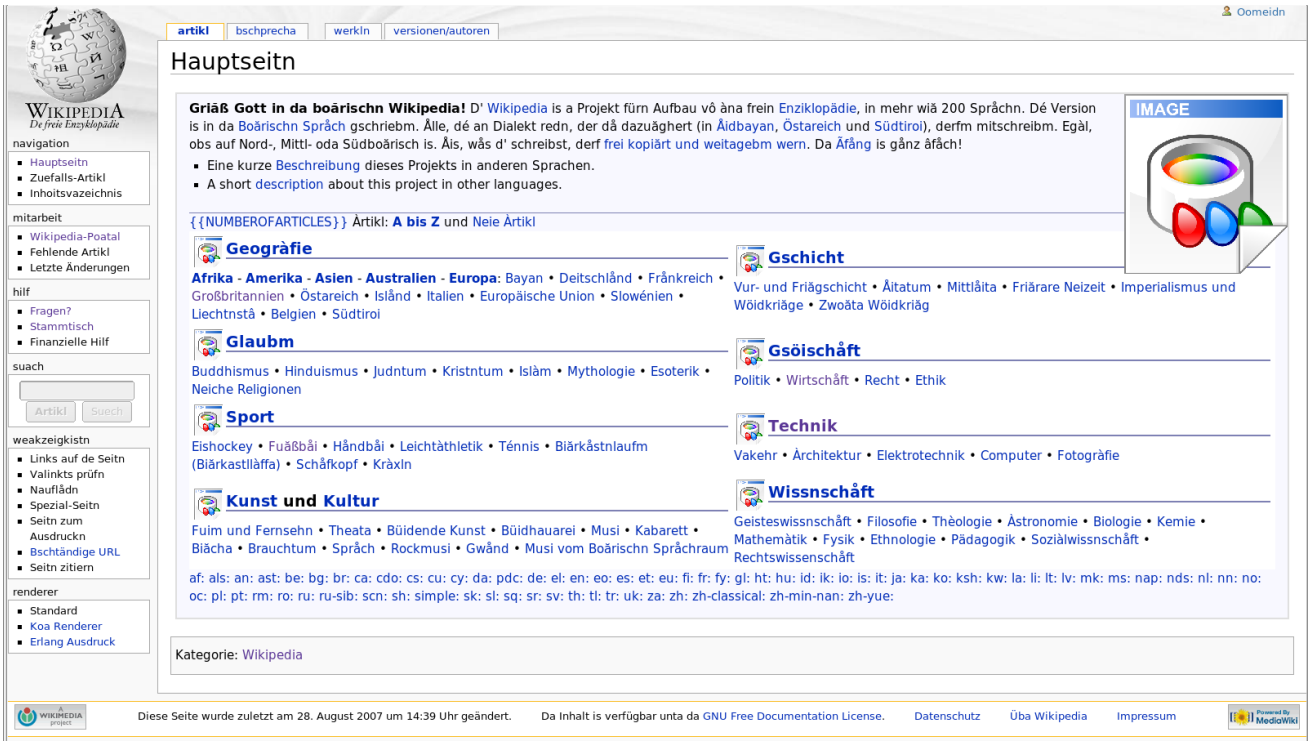


Figure 6. Screenshot of the Bavarian Wikipedia on Scalaris. Images are not included in the dump.

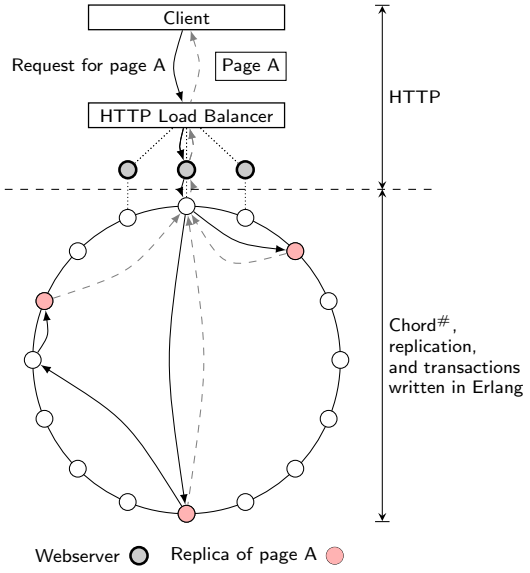


Figure 5. Wikipedia on Scalaris.

The page rendering of the Wikitext is done in Java in the web servers (see Fig. 5) running jetty. Here, we modified the Wikitext renderer of the *plog4u* project for our purposes.

Using this data layout, users may view pages by typing the URL, they can navigate to other pages via hyperlinks, they can edit pages and view the history of changes, and create new pages (see the screenshot in Fig. 7). Since the Wikipedia dumps do not include images, we render a proxy image at the corresponding positions instead. Moreover, we do not maintain a full text index and therefore full text search is not supported by our implementation. This could easily be performed by external crawling and search indexing mechanisms.

When modifying a page, a transaction over all replicas of the responsible keys is created and executed. The transaction includes the page itself, all backlink pages for inserted and deleted links, and all category pages for inserted and deleted categories.

Performance. Our Erlang implementation serves 2,500 transactions per second with just 16 servers. This is better than the public Wikipedia, which serves a total of 45,000 requests per second, of which only 2,000 hit the backend of approx. 200 servers. For the experiments, we used a HTTP load balancer (haproxy) to distribute the requests over all participating servers. The load generator (siege) requested randomly selected pages from the load balancer.

7. Conclusion

We presented *Scalaris*, a distributed key/value store based on the Chord$^{\#}$ structured overlay with symmetric data replication and a transaction layer implementing ACID properties. With Wikipedia as a demonstrator application we showed that Scalaris provides the desired scalability and efficiency.

Our implementation greatly benefited from the use of Erlang/OTP. It provides a set of useful libraries and operating procedures for building reliable distributed applications. As a result, the code is more concise than C or Java code.

Additionally, we presented an Erlang pattern that implements responsive, stateful services by overlapping fast reads with concur-

rent synchronous (slower) write operations. This framework did not only prove useful in our key/value store, but it can be used in many other Erlang implementations.

We believe that Scalaris could be of great value for suppliers of online services such as Amazon, eBay, Myspace, YouTube, or Google. Today, global service providers face the challenge of ensuring consistent data access for millions of customers in a 24/7 mode. In such environments, system crashes, software faults and heavy load imbalances are the norm rather than exceptions. Here, it is a challenging task to maintain a consistent view on data and services while hiding failures from the application.

Our P2P approach with replication and ACID provides a dependable and scalable alternative to standard database technology, albeit with a reduced data model. Each additional peer contributes additional main memory to the system, hence the combined memory capacity resembles that of current (large) SAN storage systems. If this is not sufficient, Scalaris can be easily modified to write its data onto disk. For backup purposes, our ACID implementation allows to take consistent snapshots of all data items during runtime.

Apart from distributed transactional data management, Scalaris can also be used for building scalable, hierarchical pub/sub services, reliable resource selection in dynamic systems, or internet chat services.

Acknowledgments

Many thanks to Joe Armstrong for commenting on our responsive server code and to Nico Kruber for implementing the Java transaction interface and adapting the Wiki renderer. This work was partly funded by the EU project Selfman under grant IST-34084 and the EU project XtreemOS under grant IST-33576.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007

[2] R. Baldoni, L. Querzoni, A. Virgillito, R. Jiménez-Peris, and M. Patiño-Martínez. *Dynamic Quorums for DHT-based P2P Networks.* NCA, pp. 91–100, 2005.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[4] JJ Furman, J. S. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A Scalable Data System for User Facing Applications. *SIGMOD 2008*, Jun. 2008.

[5] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *3rd Intl. Workshop on Databases, Information Systems and P2P Computing*, 2005.

[6] R. Guerraoui and L. Rodrigues. Introduction to Reliable Distributed Programming. Springer-Verlag 2006.

[7] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI, 1973.

[8] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. *SIGMOD 2008*, Jun. 2008.

[9] L. Lamport. Fast Paxos. *Distributed Computing* 19(2):79–103, 2006.

[10] M. M. Masud and I. Kiringa. *Maintaining consistency in a failure-prone P2P database network during transaction processing.* Proceedings of the 2008 International Workshop on Data management in peer-to-peer systems, pp. 27–34, 2008.

[11] M. Moser and S. Haridi. Atomic Commitment in Transactional DHTs. *1st CoreGRID Symposium*, Aug. 2007.

[12] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for Distributed Wikis on Structured Overlays. *18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007)*, Oct. 2007.

[13] T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.

[14] T. Schütt, F. Schintke, and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.

[15] T. Schütt, F. Schintke, and A. Reinefeld. Scalable Wikipedia with Erlang. *Google Scalability Conference*, Jun. 2008.

[16] T.M. Shafaat, M. Moser, A. Ghodsi, S. Haridi, T. Schütt, and A. Reinefeld. Key-Based Consistency and Availability in Structured Overlay Networks. Third Intl. ICST Conference on Scalable Information Systems, June 2008.

[17] I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. *ACM SIGCOMM 2001*, Aug. 2001.

A.10 A Scalable, Transactional Data Store for Future Internet Services

A Scalable, Transactional Data Store for Future Internet Services*

Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Seif Haridi

{reinefeld,schintke,schuett}@zib.de
Zuse Institute Berlin and onScale solutions
and
haridi@kth.se
Royal Institute of Technology, Sweden

Abstract. Future Internet services require access to large volumes of dynamically changing data records that are spread across different locations. With thousands or millions of distributed nodes storing the data, node crashes or temporary network failures are normal rather than exceptions and it is therefore important to hide failures from the application.

We suggest to use peer-to-peer (P2P) protocols to provide self-management among peers. However, today's P2P protocols are mostly limited to write-once/read-many data sharing. To extend them beyond the typical file sharing, the support of consistent replication and fast transactions is an important yet missing feature.

We present *Scalaris*, a scalable, distributed key-value store. Scalaris is built on a structured overlay network and uses a distributed transaction protocol. As a proof of concept, we implemented a simple Wikipedia clone with Scalaris which outperforms the public Wikipedia with just a few servers.

1 Introduction

Web 2.0, that is, the Internet as an information society platform supporting business, recreation and knowledge exchange, initiated a business revolution. Service providers offer Internet services for shopping (Amazon, eBay), online banking, information (Google, Flickr, Wikipedia), social networking (MySpace, Facebook), and recreation (Second Life, online games). In our information society, Web 2.0 services are no longer just nice to have, but customers depend on their continuous availability, regardless of time and space. A typical trend is illustrated by Wikipedia where users are also providers of information. This implies that its underlying data store is updated continously from multiple sources.

* This work was partly funded by the EU projects SELFMAN under grant IST-34084 and the EU project XtreemOS under grant IST-33576.

How to cope with such strong demands, especially in case of interactive community services that cannot be simply replicated? All users access the same Wikipedia, meet in the same Second Life environment and want to discuss with others via Twitter. Even the shortest interruption, caused by system downtime or network partitioning may cause huge losses in reputation and revenue. Web 2.0 services are not just an added value, but they must be dependable. Apart from 24/7 availability, providers face another challenge: they must, for a good user experience, be able to respond within milliseconds to incoming requests, regardless whether thousands or millions of concurrent requests are currently being served. Indeed, scalability is a key challenge. In addition to scalability and availability any global service to be affordable, somehow requires the system to be self managing (see sidebar).

Availability is the proportion of time a system is in a functioning condition. More formally, availability is the ratio of the expected value of the uptime of a system to the aggregate of the expected values of up and down time. Availability is often specified in a logarithmic unit called "nines" which corresponds roughly to a number of nines following the decimal point. "Six nines", for example, denote an availability of 0.999999, allowing a maximum downtime of 31 seconds per year.

Scalability refers to the capability of a system to increase the total throughput under an increased load when resources are added. A scalable database management system is one that can be upgraded to process more transactions by adding new processors, devices and storage, and which can be upgraded easily and transparently without service interrupt.

Self Management refers to the ability of a system to adjust to changing operating conditions and requirements without human intervention at runtime. Self Management includes self configuration, self healing and self tuning.

Our Scalaris system, described below, provides a comprehensive solution for self managing and scalable data management. Scalaris is a transactional key-value store that runs over multiple data centers as well as on peer-to-peer nodes. We expect Scalaris and similar systems to become an important core service of future Cloud Computing environments.

As a common key aspect, all Web 2.0 services have to deal with concurrent data updates. Typical examples are checking the availability of products and their prices, purchasing items and putting them into virtual shopping carts, and updating the state in multi-player online games. Clearly, many of these data operations have to be atomic, consistent, isolated and durable (so-called ACID properties). Traditional centralized database systems are ill-suited for this task, sooner or later they become a bottleneck for business workflow. Rather, a scalable, transactional data store like Scalaris is what is needed.

In this paper, we present the overall system architecture of Scalaris. We have implemented the core data service of Wikipedia using Scalaris. Its scalability and self-* capabilities were demonstrated in the IEEE Scalable Computing Challenge

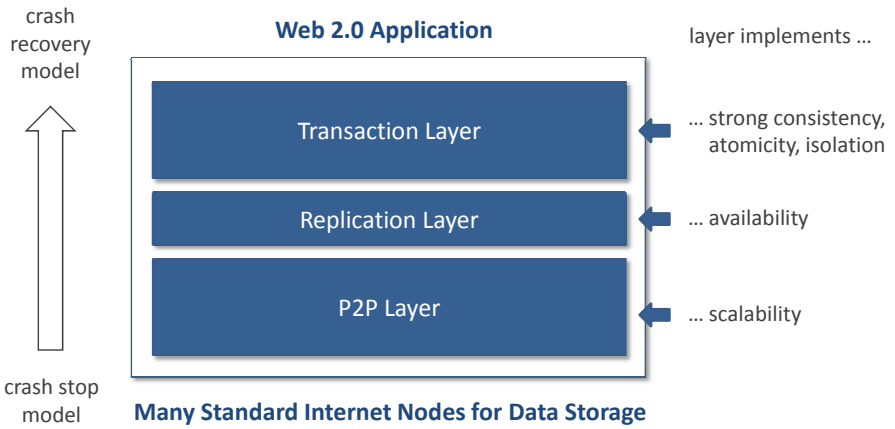


Fig. 1: Scalaris system architecture.

2008, where Scalaris won the 1^{st} price (www.ieeetcsc.org/scale2008). Talks on Scalaris were given at the the Google Scalability Conference 2008 [19] and the Erlang eXchange 2008.

The paper is organized as follows. The following Section provides an overview on Scalaris' system architecture, Section 3 describes its self-management features and Section 4 gives further details on the implementation. In Section 5 we demonstrate how Scalaris can be used for implementing Web 2.0 services. As a proof-of-concept, we have chosen a simple Wikipedia clone; performance results are given in Section 6.

2 Scalaris

As part of the EU funded SELFMAN project we set out to build a distributed key/value store capable of serving thousands or even millions of concurrent data accesses per second. Providing strong data consistency in the face of node crashes and hefty concurrent data updates was one of our major goals.

With Scalaris, we do not attempt to replace current database management systems with their general, full-fledged SQL interfaces. Instead our target is to support transactional Web 2.0 services like those needed for Internet shopping, banking, or multi-player online games. Our system consists of three layers:

– At the bottom, an enhanced structured overlay network, with logarithmic routing performance, provides the basis for storing and retrieving keys and their corresponding values. In contrast to many other overlays, our implementation stores the keys in lexicographical order. Lexicographic ordering instead of random hashing enables control of data placement which is necessary for low latency access in multi-datacenter environments.

– The middle layer implements data replication. It enhances the availability of data even under harsh conditions such as node crashes and physical network failures.
– The top layer provides transactional support for strong data consistency in the face of concurrent data operations. It uses an optimistic concurrency control strategy and a fast non-blocking commit protocol with low communication overhead. This protocol has been optimally embedded in the overlay network.

As illustrated in Fig. 1, these three layers together provide a scalable and highly available distributed key/value store which serves as a core building block for many Web 2.0 applications as well as other global services. The following sections describe the layers in more detail.

2.1 P2P Overlay

At the bottom layer, we use the structured overlay protocol Chord$^{\#}$ [17,18] for storing and retrieving key-value pairs in nodes (peers) that are arranged in a virtual ring. This ring defines a key space where all data items can be stored according to the associated key. In our case we assume that any key is an arbitrarily long string of characters, therefore the key space is infinite. Nodes are placed at arbitrary places on the ring and are responsible for all data between their predecessor and themselves. The placement policy ensures even distribution of load over the nodes. In each of the N nodes, Chord$^{\#}$ maintains a routing table with $O(\log N)$ entries (fingers). In contrast to traditional Distributed Hash Tables (DHTs) like Chord [21], Kademlia [12] and Pastry [15], Chord$^{\#}$ stores the keys in lexicographical order, thereby allowing range queries, and control over the placement of data on the ring structure. To ensure logarithmic routing performance, the fingers in the routing table are computed in such a way that successive fingers in the routing table jump over an exponentially increasing number of nodes in the ring. This finger placement will yield uniform in-/out-degree of the overlay network and thus avoids hotspots.

Chord$^{\#}$ uses the following algorithm for computing the fingers in the routing table (the infix operator $x\ .\ y$ retrieves y from the routing table of a node x):

$$\mathit{finger}_i = \begin{cases} \mathit{successor} & : i = 0 \\ \mathit{finger}_{i-1}\ .\ \mathit{finger}_{i-1} & : i \neq 0 \end{cases}$$

Thus, to calculate the i^{th} finger, a node asks the remote node, listed in its $(i-1)^{th}$ finger, for the node at which its $(i-1)^{th}$ finger refers to. In general, at any node, the fingers at level i are set to the neighbor's finger at the preceding level $i-1$. At the lowest level, the fingers point to the direct successor. The resulting structure is similar to a skiplist, but the fingers are computed deterministically without any probabilistic component and each node has its individual exponentially spaced fingers. The fingers are maintained by a periodic stabilization algorithm according to the above formula.

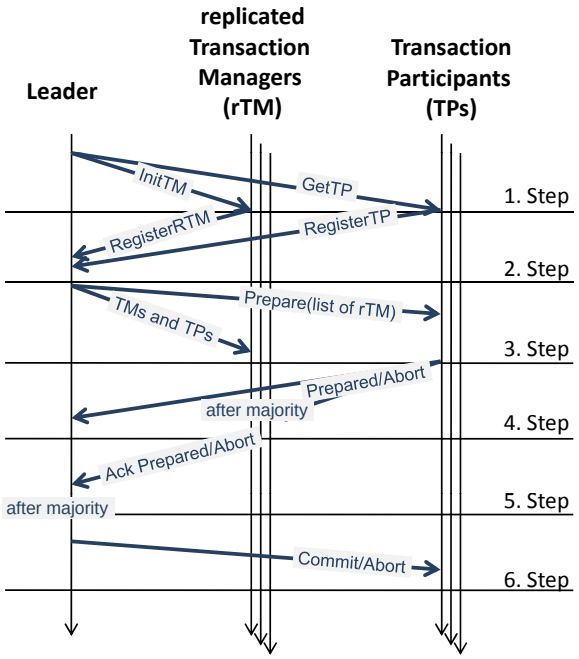


Fig. 2: Adapted Paxos used in Scalaris.

Compared to Chord [21], Chord$^{\#}$ does the routing in the *node space* rather than in the *key space*. This finger placement has three advantages over that of Chord: First, it naturally works with any type of keys as long as a total order over the keys is defined, and second, finger maintenance is cheaper [17], requiring just one hop instead of a full logarithmic search (as in Chord). To support logarithmic routing performance in skewed key distributions while nodes are arbitrarily placed in the key space—which we have to in our scenario—the third and probably most important difference becomes our trump card: the incoming routing links (fingers) will still be evenly distributed across all nodes. This prevents nodes from becoming hot spots and ensures continuous progress when routing.

2.2 Replication and Transaction Layer

The scheme described so far provides scalable access to distributed key/value pairs. To additionally tolerate node failures, we replicate all key/value pairs over r nodes using symmetric replication [5]. Basically each key is mapped by a globally known function to a set of keys $\{k_1, \ldots, k_r\}$ and the item is replicated according to those keys. Read and write operations are performed on a majority

of the replicas, thereby tolerating the unavailability of up to $\lfloor (r-1)/2 \rfloor$ nodes. This scheme is shown to provide key consistency for data lookups under realistic networking conditions [20]. For repairing the replication degree of items, nodes have to read the missing data from a majority of replicas. This is necessary to guarantee strong data consistency.

The system supports transactional semantics. A client connected to the system can issue a sequence of operations including reads and writes within a transactional context, i.e. *begin trans* ... *end trans*. This sequence of operations are executed by a local transaction manager TM associated with the overlay node to which the client is connected. The transaction will appear to be executed atomically if successful, or not executed at all if the transaction aborts.

Transactions in Scalaris are executed optimistically. This implies that each transaction is executed completely locally at the client in a read-phase. If the read phase is successful the TM tries to commit the transaction permanently in a commit phase, and permanently stores the modified data at the responsible overlay nodes. Concurrency control is performed as part of this latter phase. A transaction t will abort only if: (1) other transactions try to commit changes on some overlapping data items simultaneously; or (2) other successful transactions have already modified data that is accessed in transaction t.

Each item is assigned a version number. Read/write operation works on a majority of replicas to obtain the highest version number. A Read operation selects the data value with highest version number, and a write operation increments the highest version number of the item.

The commit phase employs an adapted version of Paxos atomic commit protocol [9], which is non-blocking. In contrast to the 3-Phase-Commit protocol used in distributed database systems, the Paxos commit protocol still works in the majority part of a network that became partitioned due to some network failure. It employs a group of replicated transaction managers (rTM) rather than a single transaction manager. Together they form a set of acceptors with the TM acting as the leader.

The commit is basically divided into two phases, the validation phase and the consensus phase. During the validation phase the replicated transaction managers rTM are initialized, and the updated data items together with references to the rTM are sent to the nodes responsible for the data items in a Prepare message. These latter nodes are called transaction participants TPs.

Each TP proposes 'prepared' or 'abort' in a fast Paxos consensus round with the acceptor set. As each acceptor collects votes from a majority of replicas for each item, it will be able to decide on a commit/abort for the whole transaction. For details see [13,20]. This scheme favors atomicity over availability. It always requires a majority of nodes to be available for the read and commit phase. This policy distinguishes Scalaris from other distributed key-value stores, like e.g. Dynamo [3].

3 Self-Management

For many Web 2.0 services, the total cost-of-ownership is dominated by the costs needed for personnel to maintain and optimize the service. Scalaris greatly reduces the operation cost with its built-in self* properties:

- *Self healing:* Scalaris continuously monitors the hosts it is running on. When it detects a node crash, it immediately repairs the overlay network and the database. Management tasks such as adding or removing hosts require minimal human intervention.
- *Self tuning:* Scalaris monitors the nodes' workload and autonomously moves items to distribute the load evenly over the system in order to improve the response time of the system. When deploying Scalaris over multiple datacenters, these algorithms are used to place frequently accessed items nearby the users.

These protection schemes do not only help in stress situations, but they also monitor and pro-actively repair the system before any service interruption might occur. With traditional database systems these operations require human interference which is error prone and costly. When using Scalaris, fewer system administrators can operate much larger installations compared to legacy databases.

4 Implementation

Implementing distributed algorithms correctly is a difficult and tedious task, especially when using imperative programming languages and multi-threading with a shared state concurrency model. The resulting code is often lengthy and error-prone, because large parts of the code deal with shared objects [22] and with exception handing such as node or network failures.

For this reason, message passing as in the *actor model* [7] is becoming the accepted paradigm for describing and reasoning about distributed algorithms [6]. Scalaris was also developed according to this model. The basic primitives in this model are actors and messages. Every actor has a state, can send messages, act upon messages and spawn new actors.

These primitives are easily mapped to Erlang processes and messages [1]. The close relationship between the specification and the programming language allows a smooth transition from the theoretical model to prototypes and eventually to a complete system.

Our Erlang implementation of Scalaris comprises eight major components with a total of 11,000 lines of code: 7,000 for the P2P layer with replication and basic system infrastructure, 2,700 lines for the transaction layer, and 1,300 lines for the Wikipedia infrastructure. Each Scalaris node is organized into the following components:

- The *Failure Detector* supervises other peers and notifies subscribers of remote node failures.

– The *Configuration Store* provides access to the current configuration and allows modifications of various system parameters.
– The *Key Holder* stores the identifier of the node in the overlay.
– The *Statistics Collector* collects statistics and forwards them to central statistic servers.
– The *Chord#* *Node* component is composed of subcomponents for overlay maintenance and overlay routing. It maintains, among other things, the successor list and the routing table. It provides the functionality of the structured overlay layer.
– The *Database* stores the key-value pairs of this node. The current implementation uses an in-memory dictionary, but disk store based on DETS or Mnesia could also be used.
– The *Transaction Manager* runs the transaction protocols.
– The *Replica Repair* maintains the replication degree of items.

The processes are organized in an Erlang OTP supervisor tree. When any of the slaves crashes, it is restarted by the Erlang supervisor. When either of the Chord# Node or the Database component fails, the other is explicitly killed and both are restarted to ensure consistency. This is equivalent to a new node joining the system.

5 Deployment: Wikipedia on Scalaris

As a challenging benchmark for Scalaris, we implemented the core of Wikipedia, the "free encyclopedia, that anyone can edit". Wikipedia runs on three sites. The main one in Tampa is organized in three layers, the proxy server layer, the web server layer, and the MySQL database layer. The proxy layer serves as a cache for recent requests, and the web server layer runs the application logic and issues requests to the database layer. Wikipedia handles about 50,000 requests per second, from which 48,000 are cache hits in the proxy server layer and 2,000 are processed by the database layer. The proxy and the web server layers are embarrassingly parallel and therefore trivial to scale. From a scalability point of view, only the database layer is challenging.

Our implementation uses Scalaris to replace the database layer. This enables us to run Wikipedia on geographically distributed sites and to scale to almost any number of hosts, as shown in the evaluation section. Our Wikipedia implementation inherits all the favorable properties of Scalaris, such as scalability and self management.

Instead of using a relational database, we map the Wikipedia content to our Scalaris key/value store [14]. We use the following mappings, using prefixes in the keys to avoid name clashes.

	key	**value**
page content	title	list of Wikitext for all versions
backlinks	title	list of titles
categories	category name	list of titles

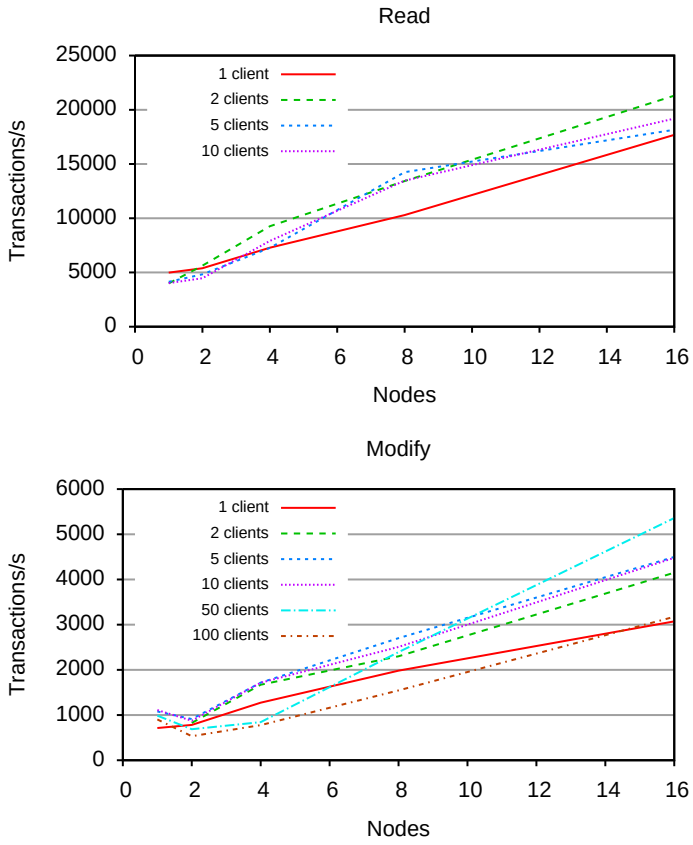


Fig. 3: Performance of Scalaris: (a) Read operation, (b) Modify operation for different numbers of local threads and cluster sizes.

On a page update a transaction across all affected keys (content, backlinks, and categories) and their replicas is triggered.

6 Evaluation

We tested the performance of Scalaris on an Intel cluster up to 16 nodes. Each node has two Quad-Core E5420s (8 cores in total) running at 2.5 GHz and 16 GB of main memory. The nodes are connected via GigE and Infiniband; we used the GigE network for our evaluation.

On each physical node we were running one multi-core Erlang virtual machine. Each virtual machine hosted 16 Scalaris nodes. We used a replication degree of four, that is, there exist four copies of each key-value pair.

We tested two operations: a *read* and a *modify* operation. The *read* operation reads a key-value pair. The *modify* operation reads a key-value pair, increments the value and writes the result back to the distributed Scalaris store. To guarantee consistency, the read-increment-write is executed within a transaction. The read operation, in contrast, simply reads from a majority of the keys.

The benchmarks involved the following steps:

- Start watch.
- Start n Erlang client processes in each VM.
- Execute the read or modify operation i times in each client.
- Wait for all clients to finish.
- Stop watch.

Figure 3 shows the results for various numbers of clients per VM (see the colored graphs). In the read benchmarks depicted in Fig. 3.a, each thread reads a key 2000 times while the modify benchmarks in Fig. 3.b modify each key 100 time in each thread.

As can be seen, the system scales about linearly over a wide range of system sizes. In the read benchmarks (Fig. 3.a), two clients per VM produce an optimal load for the system, resulting in more than 20,000 read operations per second on a 16 node (=128 core) cluster. Using only one client (red graph) does not produce enough operations to saturate the system, while five clients (blue graph) cause too much contention. Note that each read operation involves accessing a majority (3 out of 4) replicas.

The performance of the modify operation (Fig. 3.b) is of course lower, but still scales nicely with increasing system sizes. Here, the best performance of 5,500 transactions per second is reached with fifty load generators per VM, each of them generating approximately seven transactions per second. This results in 344 transactions per second on each server.

Note that each modify transaction requires Scalaris to execute the adapted Paxos algorithm, which involves finding a majority (i.e. 3 out of 4) of transaction participants and transaction managers, plus the communication between them. The performance graphs illustrate that a single client per VM does not produce enough transaction load, while fifty clients are optimal to hide the communication latency between the transaction rounds. Increasing the concurrency further to 100 clients does not improve the performance, because this causes too much contention. Note that for the 100-clients-case, there are actually 16*100 clients issuing increment transactions.

Overall, both graphs illustrate the linear scalability of Scalaris.

7 Summary

Scalaris provides a scalable and self managing transactional key-value store. We have implemented Wikipedia using Scalaris. Its scalability and self* capabilities were demonstrated in the IEEE Scalable Computing Challenge 2008, where Scalaris won the 1st prize.

Compared to other data services, Scalaris has significantly lower operating costs and is self-managing. Scalaris and similar systems will be an important building block for Web 2.0 services and future Cloud Computing environments.

While Wikipedia served here as a first demonstrator to show the potential of Scalaris, we envisage a large variety of commercial Web 2.0 applications ranging from e-commerce and social networks to infrastructure services for maintaining server farms. The Scalaris code is open source (scalaris.googlecode.com).

Acknowledgements

Many thanks to Nico Kruber, Monika Moser, and Stefan Plantikow who implemented parts of Scalaris. Also thanks to Ali Ghodsi, Thallat Shafaat, and Joe Armstrong for their support and many discussions.

References

1. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007
2. R. Baldoni, L. Querzoni, A. Virgillito, R. Jiménez-Peris, and M. Patiño-Martínez. *Dynamic Quorums for DHT-based P2P Networks.* NCA, pp. 91–100, 2005.
3. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.
4. JJ Furman, J. S. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A Scalable Data System for User Facing Applications. *SIGMOD 2008*, Jun. 2008.
5. A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *3rd Intl. Workshop on Databases, Information Systems and P2P Computing*, 2005.
6. R. Guerraoui and L. Rodrigues. Introduction to Reliable Distributed Programming. Springer-Verlag 2006.
7. C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI, 1973.
8. A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. *SIGMOD 2008*, Jun. 2008.
9. L. Lamport. The Part-Time Parliament *ACM Transactions on Computer Systems* 16(2): 133–169, 1998.
10. L. Lamport. Fast Paxos. *Distributed Computing* 19(2):79–103, 2006.
11. M. M. Masud and I. Kiringa. *Maintaining consistency in a failure-prone P2P database network during transaction processing.* Proceedings of the 2008 International Workshop on Data management in peer-to-peer systems, pp. 27–34, 2008.
12. P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. *IPTPS 2002*, Mar. 2002.
13. M. Moser and S. Haridi. Atomic Commitment in Transactional DHTs. *1st CoreGRID Symposium*, Aug. 2007.

14. S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for Distributed Wikis on Structured Overlays. *18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007)*, Oct. 2007.
15. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Middleware 2001*, Nov. 2001.
16. Scalaris code: http://code.google.com/p/scalaris/.
17. T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.
18. T. Schütt, F. Schintke, and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.
19. T. Schütt, F. Schintke, and A. Reinefeld. Scalable Wikipedia with Erlang. *Google Scalability Conference*, Jun. 2008.
20. T.M. Shafaat, M. Moser, A. Ghodsi, S. Haridi, T. Schütt, and A. Reinefeld. Key-Based Consistency and Availability in Structured Overlay Networks. Third Intl. ICST Conference on Scalable Information Systems, June 2008.
21. I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. *ACM SIGCOMM 2001*, Aug. 2001. Concepts, Techniques, and Models of Computer Programming
22. P. Van Roy and S. Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, March 2004.

A.11 Scalaris Java Interface

Scalaris Java Interface

0.2.0

Contents

1 Namespace Index **1**

1.1 Package List 1

2 Class Index **2**

2.1 Class List 2

3 Namespace Documentation **2**

3.1 Package de.zib.scalaris 2

3.1.1 Detailed Description 2

4 Class Documentation **4**

4.1 de.zib.scalaris.ConnectionFactory Class Reference 4

4.1.1 Detailed Description 4

4.1.2 Constructor & Destructor Documentation 5

4.1.3 Member Function Documentation 5

4.2 de.zib.scalaris.DeleteResult Class Reference 8

4.2.1 Detailed Description 8

4.2.2 Constructor & Destructor Documentation 9

4.2.3 Member Data Documentation 9

4.3 de.zib.scalaris.Main Class Reference 9

4.3.1 Detailed Description 9

4.3.2 Member Function Documentation 10

4.4 de.zib.scalaris.Scalaris Class Reference 10

4.4.1 Detailed Description 11

4.4.2 Constructor & Destructor Documentation 14

4.4.3 Member Function Documentation 14

4.5 de.zib.scalaris.Transaction Class Reference 21

4.5.1 Detailed Description 22

4.5.2 Constructor & Destructor Documentation 23

4.5.3 Member Function Documentation 23

1 Namespace Index

1.1 Package List

Here are the packages with brief descriptions (if available):

de.zib.scalaris **2**

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

de.zib.scalaris.ConnectionFactory 4

de.zib.scalaris.DeleteResult 8

de.zib.scalaris.Main 9

de.zib.scalaris.Scalaris 10

de.zib.scalaris.Transaction 21

3 Namespace Documentation

3.1 Package de.zib.scalaris

Classes

- class Scalaris
- class Transaction
- class ConnectionFactory
- class DeleteResult
- class Main

3.1.1 Detailed Description

 This package contains means to communicate with the erlang scalaris ring from Java.

The Scalaris class The de.zib.scalaris.Scalaris class provides methods for reading and writing values, publishing topics, subscribing to urls and getting a list of subscribers with both erlang objects (com.ericsson.otp.erlang.OtpErlangObject) and Java java.lang.String objects.

Example:

```
try {
  Scalaris sc = new Scalaris();
  String value = sc.read("key");
```

```
} catch (ConnectionException e) {
  System.err.println("read failed: " + e.getMessage());
} catch (TimeoutException e) {
  System.err.println("read failed with timeout: " + e.getMessage());
} catch (UnknownException e) {
  System.err.println("read failed with unknown: " + e.getMessage());
} catch (NotFoundException e) {
  System.err.println("read failed with not found: " + e.getMessage());
}
```

See the de.zib.scalaris.Scalaris class documentation for more details.

The Transaction class The de.zib.scalaris.Transaction class provides means to realise a scalaris transaction from Java. After starting a transaction, there are methods to read and write values with both erlang objects (com.ericsson.otp.erlang.OtpErlangObject) and Java java.lang.String objects. The transaction can then be committed, aborted or reset.

Example:

```
try {
  Transaction transaction = new Transaction();
  transaction.start();
  String value = transaction.read("key");
  transaction.write("key", "value");
  transaction.commit();
} catch (ConnectionException e) {
  System.err.println("read failed: " + e.getMessage());
} catch (TimeoutException e) {
  System.err.println("read failed with timeout: " + e.getMessage());
} catch (UnknownException e) {
  System.err.println("read failed with unknown: " + e.getMessage());
} catch (NotFoundException e) {
  System.err.println("read failed with not found: " + e.getMessage());
} catch (TransactionNotFinishedException e) {
  System.out.println("failed: " + e.getMessage());
  return;
}
```

See the de.zib.scalaris.Transaction class documentation for more details.

Author:

Nico Kruber, kruber@zib.de

Version:

2.0

Since:

2.0

4 Class Documentation

4.1 de.zib.scalaris.ConnectionFactory Class Reference

Public Member Functions

- ConnectionFactory ()
- ConnectionFactory (Properties properties)
- void setProperties (Properties properties)
- OtpConnection createConnection (String clientName, boolean clientNameAppendUUID) throws ConnectionException
- OtpConnection createConnection (String clientName) throws ConnectionException
- OtpConnection createConnection () throws ConnectionException
- String getNode ()
- void setNode (String node)
- String getCookie ()
- void setCookie (String cookie)
- String getClientName ()
- void setClientName (String clientName)
- boolean isClientNameAppendUUID ()
- void setClientNameAppendUUID (boolean clientNameAppendUUID)

Static Public Member Functions

- static ConnectionFactory getInstance ()

4.1.1 Detailed Description

Provides means to create connections to scalaris nodes.

This class uses a singleton-alike pattern providing a global (static) instance through its getInstance() method but also allowing for object construction which might be useful when using multiple threads each creating its own connections.

The location of the default configuration file used by ConnectionFactory() can be overridden by specifying the `scalaris.java.config` system property - otherwise the class tries to load `scalaris.properties`.

A user-defined Properties object can also be used by creating objects with ConnectionFactory(Properties) or setting the new values with setProperties(Properties) but must provide the following values (default values as shown)

- `scalaris.node = "boot@localhost"`
- `scalaris.cookie = "chocolate chip cookie"`
- `scalaris.client.name = "java_client"`
- `scalaris.client.appendUUID = "true"`

Author:

Nico Kruber, `kruber@zib.de`

Version:

2.1

Since:

2.0

4.1.2 Constructor & Destructor Documentation

4.1.2.1 de.zib.scalaris.ConnectionFactory.ConnectionFactory ()

Constructor, sets the parameters to use for connections according to values given in a `scalaris.properties` file and falls back to default values if values don't exist.

By default the config file is assumed to be in the same directory as the classes. Specify the `scalaris.java.config` system property to set a different location.

Default values are:

- `scalaris.node = "boot@localhost"`
- `scalaris.cookie = "chocolate chip cookie"`
- `scalaris.client.name = "java_client"`
- `scalaris.client.appendUUID = "true"`

4.1.2.2 de.zib.scalaris.ConnectionFactory.ConnectionFactory (Properties *properties*)

Constructor, sets the parameters to use for connections according to values given in the given Properties object and falls back to default values if values don't exist.

The Properties object should provide the following values (default values as shown):

- `scalaris.node = "boot@localhost"`
- `scalaris.cookie = "chocolate chip cookie"`
- `scalaris.client.name = "java_client"`
- `scalaris.client.appendUUID = "true"`

Parameters:

properties

4.1.3 Member Function Documentation

4.1.3.1 OtpConnection de.zib.scalaris.ConnectionFactory.createConnection () throws ConnectionException

Creates a connection to a scalaris erlang node specified by the given parameters.

Returns:

the created connection

Exceptions:

ConnectionException if the connection fails

4.1.3.2 OtpConnection de.zib.scalaris.ConnectionFactory.createConnection (String *clientName*) throws ConnectionException

Creates a connection to a scalaris erlang node specified by the given parameters. Uses the given client name.

If clientNameAppendUUID is specified a pseudo UUID is appended to the given name. BEWARE that scalaris nodes accept only one connection per client name!

Parameters:

clientName the name that identifies the java client

Returns:

the created connection

Exceptions:

ConnectionException if the connection fails

4.1.3.3 OtpConnection de.zib.scalaris.ConnectionFactory.createConnection (String *clientName*, boolean *clientNameAppendUUID*) throws ConnectionException

Creates a connection to a scalaris erlang node specified by the given parameters. Uses the given client name.

If `clientNameAppendUUID` is specified a pseudo UUID is appended to the given name. BEWARE that scalaris nodes accept only one connection per client name!

Parameters:

clientName the name that identifies the java client

clientNameAppendUUID override the object's setting for clientNameAppendUUID

Returns:

the created connection

Exceptions:

ConnectionException if the connection fails

4.1.3.4 String de.zib.scalaris.ConnectionFactory.getClientName ()

Returns the name of the (Java) client to use when establishing a connection with erlang.

Returns:

the clientName

4.1.3.5 String de.zib.scalaris.ConnectionFactory.getCookie ()

Returns the cookie name to use for connections.

Returns:

the cookie

4.1.3.6 static ConnectionFactory de.zib.scalaris.ConnectionFactory.getInstance () [static]

Returns the static instance of a connection factory.

Returns:

a connection factory

4.1.3.7 String de.zib.scalaris.ConnectionFactory.getNode ()

Returns the name of the node to connect to.

Returns:

the name of the node

4.1.3.8 boolean de.zib.scalaris.ConnectionFactory.isClientNameAppendUUID ()

Returns whether an UUID is appended to client names or not.

Returns:

`true` if an UUID is appended, `false` otherwise

4.1.3.9 void de.zib.scalaris.ConnectionFactory.setClientName (String *clientName*)

Sets the name of the (Java) client to use when establishing a connection with erlang.

Parameters:

clientName the clientName to set

4.1.3.10 void de.zib.scalaris.ConnectionFactory.setClientNameAppendUUID (boolean *clientNameAppendUUID*)

Sets whether to append an UUID to client names or not.

Parameters:

clientNameAppendUUID `true` if an UUID is appended, `false` otherwise

4.1.3.11 void de.zib.scalaris.ConnectionFactory.setCookie (String *cookie*)

Sets the cookie name to use for connections.

Parameters:

cookie the cookie to set

4.1.3.12 void de.zib.scalaris.ConnectionFactory.setNode (String *node*)

Sets the name of the node to connect to.

Parameters:

node the node to set

4.1.3.13 void de.zib.scalaris.ConnectionFactory.setProperties (Properties *properties*)

Sets the object's members used for creating connections to erlang to values provided by the given Properties object.

The Properties object should provide the following values (default values as shown):

- `scalaris.node = "boot@localhost"`
- `scalaris.cookie = "chocolate chip cookie"`
- `scalaris.client.name = "java_client"`
- `scalaris.client.appendUUID = "true"`

NOTE: Existing connections are not changed!

Parameters:

properties the object to get the connection parameters from

The documentation for this class was generated from the following file:

- src/de/zib/scalaris/ConnectionFactory.java

4.2 de.zib.scalaris.DeleteResult Class Reference

Public Member Functions

- DeleteResult (OtpErlangList list) throws UnknownException

Public Attributes

- int ok = 0
- int locks_set = 0
- int undef = 0

4.2.1 Detailed Description

Stores the result of a delete operation.

Author:

Nico Kruber, `kruber@zib.de`

Version:

2.2

Since:

2.2

See also:

Scalaris.delete(String)

4.2.2 Constructor & Destructor Documentation

4.2.2.1 de.zib.scalaris.DeleteResult.DeleteResult (OtpErlangList *list*) throws UnknownException

Creates a delete state object by converting the result list returned from erlang.

Parameters:

list the list to convert

Exceptions:

UnknownException is thrown if an unknown reason was encountered

4.2.3 Member Data Documentation

4.2.3.1 int de.zib.scalaris.DeleteResult.locks_set = 0

Skipped replicas because locks were set.

4.2.3.2 int de.zib.scalaris.DeleteResult.ok = 0

Number of successfully deleted replicas.

4.2.3.3 int de.zib.scalaris.DeleteResult.undef = 0

Skipped replicas because they did not exist.

The documentation for this class was generated from the following file:

- src/de/zib/scalaris/DeleteResult.java

4.3 de.zib.scalaris.Main Class Reference

Static Public Member Functions

- static void main (String[ ] args)

4.3.1 Detailed Description

Class to test basic functionality of the package and to use scalaris from command line.

Author:

Nico Kruber, kruber@zib.de

Version:

2.0

Since:

2.0

4.3.2 Member Function Documentation

4.3.2.1 static void de.zib.scalaris.Main.main (String[ ] *args*) [static]

Queries the command line options for an action to perform.

```
> java -jar scalaris.jar -help
usage: scalaris
 -getsubscribers <topic>   get subscribers of a topic
 -help                     print this message
 -publish <params>         publish a new message for a topic: <topic> <message>
 -read <key>               read an item
 -subscribe <params>       subscribe to a topic: <topic> <url>
 -unsubscribe <params>     unsubscribe from a topic: <topic> <url>
 -write <params>           write an item: <key>

 -minibench                run mini benchmark
```

Parameters:

args command line arguments

The documentation for this class was generated from the following file:

- src/de/zib/scalaris/Main.java

4.4 de.zib.scalaris.Scalaris Class Reference

Public Member Functions

- Scalaris () throws ConnectionException
- Scalaris (OtpConnection conn) throws ConnectionException
- OtpErlangObject readObject (OtpErlangString key) throws ConnectionException, TimeoutException, UnknownException, NotFoundException
- String read (String key) throws ConnectionException, TimeoutException, UnknownException, NotFoundException
- void readCustom (String key, CustomOtpObject<?> value) throws ConnectionException, TimeoutException, UnknownException, NotFoundException
- void writeObject (OtpErlangString key, OtpErlangObject value) throws ConnectionException, TimeoutException, UnknownException

- void write (String key, String value) throws ConnectionException, TimeoutException, UnknownException
- void writeCustom (String key, CustomOtpObject<?> value) throws ConnectionException, TimeoutException, UnknownException
- void publish (OtpErlangString topic, OtpErlangString content) throws ConnectionException
- void publish (String topic, String content) throws ConnectionException
- void subscribe (OtpErlangString topic, OtpErlangString url) throws ConnectionException, TimeoutException, UnknownException
- void subscribe (String topic, String url) throws ConnectionException, TimeoutException, UnknownException
- void unsubscribe (OtpErlangString topic, OtpErlangString url) throws ConnectionException, TimeoutException, NotFoundException, UnknownException
- void unsubscribe (String topic, String url) throws ConnectionException, TimeoutException, NotFoundException, UnknownException
- OtpErlangList getSubscribers (OtpErlangString topic) throws ConnectionException, UnknownException
- ArrayList< String > getSubscribers (String topic) throws ConnectionException, UnknownException
- long delete (String key) throws ConnectionException, TimeoutException, UnknownException, NodeNotFoundException
- long delete (String key, int timeout) throws ConnectionException, TimeoutException, UnknownException, NodeNotFoundException
- DeleteResult getLastDeleteResult () throws UnknownException
- void closeConnection ()

4.4.1 Detailed Description

Provides methods to read and write key/value pairs to a scalaris ring.

Each operation is a single transaction. If you are looking for more transactions, use the Transaction class instead.

Instances of this class can be generated using a given connection to a scalaris node using Scalaris(OtpConnection) or without a connection (Scalaris()) in which case a new connection is created using ConnectionFactory#createConnection().

There are two paradigms for reading and writing values:

- using Java Strings: read(String), write(String, String)

  This is the safe way of accessing scalaris where type conversions are handled by the API and the user doesn't have to worry about anything else.

  Be aware though that this is not the most efficient way of handling strings!

- using custom OtpErlangObjects: readObject(OtpErlangString), writeObject(OtpErlangString, OtpErlangObject)

  Here the user can specify custom behaviour and increase performance. Handling the stored types correctly is at the user's hand.

  An example using erlang objects to improve performance for inserting strings is provided by de.zib.scalaris.examples.CustomOtpFastStringObject and can be tested by de.zib.scalaris.examples.FastStringBenchmark.

Reading values

```
String key;
OtpErlangString otpKey;

 Scalaris sc = new Scalaris();
String value             = sc.read(key);          // read(String)
OtpErlangObject optValue = sc.readObject(otpKey); // readObject(OtpErlangString)
```

For the full example, see de.zib.scalaris.examples.ScalarisReadExample

Writing values

```
String key;
String value;
OtpErlangString otpKey;
OtpErlangString otpValue;

 Scalaris sc = new Scalaris();
sc.write(key, value);             // write(String, String)
sc.writeObject(otpKey, otpValue); // writeObject(OtpErlangString, OtpErlangObject)
```

For the full example, see de.zib.scalaris.examples.ScalarisWriteExample

Deleting values

```
String key;
int timeout;
DeleteResult result;

 Scalaris sc = new Scalaris();
sc.delete(key);                    // delete(String)
sc.delete(key, timeout);           // delete(String, int)
result = sc.getLastDeleteResult(); // getLastDeleteResult()
```

Publishing topics

```
String topic;
String content;
OtpErlangString otpTopic;
OtpErlangString otpContent;
```

```
 Scalaris sc = new Scalaris();
sc.publish(topic, content);       // publish(String, String)
sc.publish(otpTopic, otpContent); // publish(OtpErlangString, OtpErlangString)
```

For the full example, see de.zib.scalaris.examples.ScalarisPublishExample

Subscribing to topics

```
String topic;
String URL;
OtpErlangString otpTopic;
OtpErlangString otpURL;

 Scalaris sc = new Scalaris();
sc.subscribe(topic, URL);       // subscribe(String, String)
sc.subscribe(otpTopic, otpURL); // subscribe(OtpErlangString, OtpErlangString)
```

For the full example, see de.zib.scalaris.examples.ScalarisSubscribeExample

Unsubscribing from topics Unsubscribing from topics works like subscribing to topics with the exception of a NotFoundException being thrown if either the topic does not exist or the URL is not subscribed to the topic.

```
String topic;
String URL;
OtpErlangString otpTopic;
OtpErlangString otpURL;

 Scalaris sc = new Scalaris();
sc.unsubscribe(topic, URL);       // unsubscribe(String, String)
sc.unsubscribe(otpTopic, otpURL); // unsubscribe(OtpErlangString, OtpErlangString)
```

Getting a list of subscribers to a topic

```
String topic;
OtpErlangString otpTopic;

 Vector<String> subscribers;
OtpErlangList otpSubscribers;

 // non-static:
Scalaris sc = new Scalaris();
subscribers = sc.getSubscribers(topic);             // getSubscribers(String)
otpSubscribers = sc.singleGetSubscribers(otpTopic); // getSubscribers(OtpErlangString)
```

For the full example, see de.zib.scalaris.examples.ScalarisGetSubscribersExample

Author:

Nico Kruber, kruber@zib.de

Version:

2.2

Since:

2.0

4.4.2 Constructor & Destructor Documentation

4.4.2.1 de.zib.scalaris.Scalaris.Scalaris () throws ConnectionException

Constructor, uses the default connection returned by ConnectionFactory#createConnection().

Exceptions:

ConnectionException if the connection fails

4.4.2.2 de.zib.scalaris.Scalaris.Scalaris (OtpConnection *conn*) throws ConnectionException

Constructor, uses the given connection to an erlang node.

Parameters:

conn connection to use for the transaction

Exceptions:

ConnectionException if the connection fails

4.4.3 Member Function Documentation

4.4.3.1 void de.zib.scalaris.Scalaris.closeConnection ()

Closes the transaction's connection to a scalaris node.

Note: Subsequent calls to the other methods will throw ConnectionExceptions!

4.4.3.2 long de.zib.scalaris.Scalaris.delete (String *key*, int *timeout*) throws ConnectionException, TimeoutException, UnknownException, NodeNotFoundException

Tries to delete all replicas of the given `key`.

WARNING: This function can lead to inconsistent data (e.g. deleted items can re-appear). Also when re-creating an item the version before the delete can re-appear.

Parameters:

key the key to delete

timeout the time (in milliseconds) to wait for results

Returns:

the number of successfully deleted replicas

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to delete the value

NodeNotFoundException if no scalaris node was found

UnknownException if any other error occurs

Since:

2.2

See also:

delete(String)

4.4.3.3 long de.zib.scalaris.Scalaris.delete (String *key*) throws ConnectionException, TimeoutException, UnknownException, NodeNotFoundException

Tries to delete all replicas of the given `key` in 2000ms.

Parameters:

key the key to delete

Returns:

the number of successfully deleted replicas

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to delete the value

NodeNotFoundException if no scalaris node was found

UnknownException if any other error occurs

Since:

2.2

See also:

delete(String, int)

4.4.3.4 DeleteResult de.zib.scalaris.Scalaris.getLastDeleteResult () throws UnknownException

Returns the result of the last call to delete(String).

NOTE: This function traverses the result list returned by erlang and therefore takes some time to process. It is advised to store the returned result object once generated.

Returns:

the delete result

Exceptions:

UnknownException is thrown if an unknown reason was encountered

See also:

delete(String)

4.4.3.5 ArrayList<String> de.zib.scalaris.Scalaris.getSubscribers (String *topic*) throws ConnectionException, UnknownException

Gets a list of subscribers to a `topic`.

Parameters:

topic the topic to get the subscribers for

Returns:

the subscriber URLs

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

UnknownException is thrown if the return type of the erlang method does not match the expected one

4.4.3.6 OtpErlangList de.zib.scalaris.Scalaris.getSubscribers (OtpErlangString *topic*) throws ConnectionException, UnknownException

Gets a list of subscribers to a `topic`.

Parameters:

topic the topic to get the subscribers for

Returns:

the subscriber URLs

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

UnknownException is thrown if the return type of the erlang method does not match the expected one

4.4.3.7 void de.zib.scalaris.Scalaris.publish (String *topic*, String *content*) throws ConnectionException

Publishes an event under a given `topic`.

Parameters:

topic the topic to publish the content under

content the content to publish

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

4.4.3.8 void de.zib.scalaris.Scalaris.publish (OtpErlangString *topic*, OtpErlangString *content*) throws ConnectionException

Publishes an event under a given `topic`.

Parameters:

topic the topic to publish the content under

content the content to publish

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

The specification of `pubsub.pubsub_api:publish/2` states that the only returned value is `ok`, so no further evaluation is necessary.

4.4.3.9 String de.zib.scalaris.Scalaris.read (String *key*) throws ConnectionException, TimeoutException, UnknownException, NotFoundException

Gets the value stored under the given `key`.

Parameters:

key the key to look up

Returns:

the (string) value stored under the given `key`

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to fetch the value

NotFoundException if the requested key does not exist

UnknownException if any other error occurs

See also:

readObject(OtpErlangString)

4.4.3.10 void de.zib.scalaris.Scalaris.readCustom (String *key*, CustomOtpObject<?> *value*) throws ConnectionException, TimeoutException, UnknownException, NotFoundException

Gets the value stored under the given `key`.

Parameters:

key the key to look up

value container that stores the value returned by scalaris

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to fetch the value

NotFoundException if the requested key does not exist

UnknownException if any other error occurs

See also:

readObject(OtpErlangString)

Since:

2.1

4.4.3.11 OtpErlangObject de.zib.scalaris.Scalaris.readObject (OtpErlangString *key*) throws ConnectionException, TimeoutException, UnknownException, NotFoundException

Gets the value stored under the given `key`.

Parameters:

key the key to look up

Returns:

the value stored under the given `key`

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to fetch the value

NotFoundException if the requested key does not exist

UnknownException if any other error occurs

4.4.3.12 void de.zib.scalaris.Scalaris.subscribe (String *topic*, String *url*) throws ConnectionException, TimeoutException, UnknownException

Subscribes a url to a `topic`.

Parameters:

topic the topic to subscribe the url to

url the url of the subscriber (this is where the events are send to)

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

4.4.3.13 void de.zib.scalaris.Scalaris.subscribe (OtpErlangString *topic*, OtpErlangString *url*) throws ConnectionException, TimeoutException, UnknownException

Subscribes a url to a `topic`.

Parameters:

topic the topic to subscribe the url to

url the url of the subscriber (this is where the events are send to)

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

4.4.3.14 void de.zib.scalaris.Scalaris.unsubscribe (String *topic*, String *url*) throws ConnectionException, TimeoutException, NotFoundException, UnknownException

Unsubscribes a url from a `topic`.

Parameters:

topic the topic to unsubscribe the url from

url the url of the subscriber

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

NotFoundException if the topic does not exist or the given subscriber is not subscribed to the given topic

UnknownException if any other error occurs

4.4.3.15 void de.zib.scalaris.Scalaris.unsubscribe (OtpErlangString *topic*, OtpErlangString *url*) throws ConnectionException, TimeoutException, NotFoundException, UnknownException

Unsubscribes a url from a `topic`.

Parameters:

topic the topic to unsubscribe the url from

url the url of the subscriber

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

NotFoundException if the topic does not exist or the given subscriber is not subscribed to the given topic

UnknownException if any other error occurs

4.4.3.16 void de.zib.scalaris.Scalaris.write (String *key*, String *value*) throws ConnectionException, TimeoutException, UnknownException

Stores the given `key`/`value` pair.

Parameters:

key the key to store the value for

value the value to store

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

See also:

writeObject(OtpErlangString, OtpErlangObject)

4.4.3.17 void de.zib.scalaris.Scalaris.writeCustom (String *key*, CustomOtpObject<?> *value*) throws ConnectionException, TimeoutException, UnknownException

Stores the given `key`/`value` pair.

Parameters:

key the key to store the value for

value the value to store

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

See also:

writeObject(OtpErlangString, OtpErlangObject)

Since:

2.1

4.4.3.18 void de.zib.scalaris.Scalaris.writeObject (OtpErlangString *key*, OtpErlangObject *value*) throws ConnectionException, TimeoutException, UnknownException

Stores the given `key`/`value` pair.

Parameters:

key the key to store the value for

value the value to store

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

The documentation for this class was generated from the following file:

- src/de/zib/scalaris/Scalaris.java

4.5 de.zib.scalaris.Transaction Class Reference

Public Member Functions

- Transaction () throws ConnectionException
- Transaction (OtpConnection conn) throws ConnectionException
- void reset ()
- void start () throws ConnectionException, TransactionNotFinishedException, UnknownException
- void commit () throws UnknownException, ConnectionException
- void abort ()
- OtpErlangObject readObject (OtpErlangString key) throws ConnectionException, TimeoutException, UnknownException, NotFoundException
- String read (String key) throws ConnectionException, TimeoutException, UnknownException, NotFoundException
- void readCustom (String key, CustomOtpObject<?> value) throws ConnectionException, TimeoutException, UnknownException, NotFoundException
- void writeObject (OtpErlangString key, OtpErlangObject value) throws ConnectionException, TimeoutException, UnknownException
- void write (String key, String value) throws ConnectionException, TimeoutException, UnknownException

- void writeCustom (String key, CustomOtpObject<?> value) throws ConnectionException, TimeoutException, UnknownException
- void revertLastOp ()
- void closeConnection ()

4.5.1 Detailed Description

Provides means to realise a transaction with the scalaris ring using Java.

Instances of this class can be generated using a given connection to a scalaris node using Transaction(OtpConnection) or without a connection (Transaction()) in which case a new connection is created using ConnectionFactory#createConnection().

There are two paradigms for reading and writing values:

- using Java Strings: read(String), write(String, String)

  This is the safe way of accessing scalaris where type conversions are handled by the API and the user doesn't have to worry about anything else.

  Be aware though that this is not the most efficient way of handling strings!

- using custom OtpErlangObjects: readObject(OtpErlangString), writeObject(OtpErlangString, OtpErlangObject)

  Here the user can specify custom behaviour and increase performance. Handling the stored types correctly is at the user's hand.

  An example using erlang objects to improve performance for inserting strings is provided by de.zib.scalaris.examples.CustomOtpFastStringObject and can be tested by de.zib.scalaris.examples.FastStringBenchmark.

Example:

```
OtpErlangString otpKey;
OtpErlangString otpValue;
OtpErlangObject otpResult;

 String key;
String value;
String result;

 Transaction t1 = new Transaction(); // Transaction()
t1.start();                          // start()

 t1.write(key, value);              // write(String, String)
t1.writeObject(otpKey, otpValue); // writeObject(OtpErlangString, OtpErlangObject)

 result = t1.read(key);              //read(String)
otpResult = t1.readObject(otpKey); //readObject(OtpErlangString)

 transaction.commit(); // commit()
```

For more examples, have a look at de.zib.scalaris.examples.TransactionReadExample, de.zib.scalaris.examples.TransactionWriteExample and de.zib.scalaris.examples.TransactionReadWriteExample.

Attention: If a read or write operation fails within a transaction all subsequent operations on that key will fail as well. This behaviour may particularly be undesirable if a read operation just checks whether a value already exists or not. To overcome this situation call revertLastOp() immediately after the failed operation which restores the state as it was before that operation.

The de.zib.scalaris.examples.TransactionReadWriteExample example shows such a use case.

Author:

Nico Kruber, kruber@zib.de

Version:

2.2

Since:

2.0

4.5.2 Constructor & Destructor Documentation

4.5.2.1 de.zib.scalaris.Transaction.Transaction () throws ConnectionException

Constructor, uses the default connection returned by ConnectionFactory#createConnection().

Exceptions:

ConnectionException if the connection fails

4.5.2.2 de.zib.scalaris.Transaction.Transaction (OtpConnection *conn*) throws ConnectionException

Constructor, uses the given connection to an erlang node.

Parameters:

conn connection to use for the transaction

Exceptions:

ConnectionException if the connection fails

4.5.3 Member Function Documentation

4.5.3.1 void de.zib.scalaris.Transaction.abort ()

Cancels the current transaction.

For a transaction to be cancelled, only the transLog needs to be reset. Nothing else needs to be done since the data was not modified until the transaction was committed.

See also:

commit()
reset()

4.5.3.2 void de.zib.scalaris.Transaction.closeConnection ()

Closes the transaction's connection to a scalaris node.

Note: Subsequent calls to the other methods will throw ConnectionExceptions!

4.5.3.3 void de.zib.scalaris.Transaction.commit () throws UnknownException, ConnectionException

Commits the current transaction.

The transaction's log is reset if the commit was successful, otherwise it still retains in the transaction which must be successfully committed, aborted or reset in order to be restarted.

Exceptions:

UnknownException If the commit fails or the returned value from erlang is of an unknown type/structure, this exception is thrown. Neither the transaction log nor the local operations buffer is emptied, so that the commit can be tried again.

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

See also:

abort()
reset()

4.5.3.4 String de.zib.scalaris.Transaction.read (String *key*) throws ConnectionException, TimeoutException, UnknownException, NotFoundException

Gets the value stored under the given `key`.

Parameters:

key the key to look up

Returns:

the (string) value stored under the given `key`

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to fetch the value

NotFoundException if the requested key does not exist

UnknownException if any other error occurs

See also:

readObject(OtpErlangString)

4.5.3.5 void de.zib.scalaris.Transaction.readCustom (String *key*, CustomOtpObject<?> *value*) throws ConnectionException, TimeoutException, UnknownException, NotFoundException

Gets the value stored under the given `key`.

Parameters:

key the key to look up

value container that stores the value returned by scalaris

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to fetch the value

NotFoundException if the requested key does not exist

UnknownException if any other error occurs

See also:

readObject(OtpErlangString)

Since:

2.1

4.5.3.6 OtpErlangObject de.zib.scalaris.Transaction.readObject (OtpErlangString *key*) throws ConnectionException, TimeoutException, UnknownException, NotFoundException

Gets the value stored under the given `key`.

Parameters:

key the key to look up

Returns:

the value stored under the given `key` as a raw erlang type

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to fetch the value

NotFoundException if the requested key does not exist

UnknownException if any other error occurs

4.5.3.7 void de.zib.scalaris.Transaction.reset ()

Resets the transaction to its initial state.

This action is not reversible.

4.5.3.8 void de.zib.scalaris.Transaction.revertLastOp ()

Reverts the last (read, parallelRead or write) operation by restoring the last state.

If there was no operation or the last operation was already reverted, this method does nothing.

This method is especially useful if after an unsuccessful read a value with the same key should be written which is not possible if the failed read is still in the transaction's log.

NOTE: This method works only ONCE! Subsequent calls will do nothing.

4.5.3.9 void de.zib.scalaris.Transaction.start () throws ConnectionException, TransactionNotFinishedException, UnknownException

Starts a new transaction by generating a new transaction log.

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TransactionNotFinishedException if an old transaction is not finished (via commit() or abort()) yet

UnknownException if the returned value from erlang does not have the expected type/structure

4.5.3.10 void de.zib.scalaris.Transaction.write (String *key*, String *value*) throws ConnectionException, TimeoutException, UnknownException

Stores the given `key`/`value` pair.

Parameters:

key the key to store the value for

value the value to store

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

See also:

writeObject(OtpErlangString, OtpErlangObject)

4.5.3.11 void de.zib.scalaris.Transaction.writeCustom (String *key*, CustomOtpObject<?> *value*) throws ConnectionException, TimeoutException, UnknownException

Stores the given `key`/`value` pair.

Parameters:

key the key to store the value for

value the value to store

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

See also:

writeObject(OtpErlangString, OtpErlangObject)

Since:

2.1

4.5.3.12 void de.zib.scalaris.Transaction.writeObject (OtpErlangString *key*, OtpErlangObject *value*) throws ConnectionException, TimeoutException, UnknownException

Stores the given `key`/`value` pair.

Parameters:

key the key to store the value for

value the value to store

Exceptions:

ConnectionException if the connection is not active or a communication error occurs or an exit signal was received or the remote node sends a message containing an invalid cookie

TimeoutException if a timeout occurred while trying to write the value

UnknownException if any other error occurs

The documentation for this class was generated from the following file:

- src/de/zib/scalaris/Transaction.java

Index

abort
    de::zib::scalaris::Transaction, 23

closeConnection
    de::zib::scalaris::Scalaris, 13
    de::zib::scalaris::Transaction, 23
commit
    de::zib::scalaris::Transaction, 23
ConnectionFactory
    de::zib::scalaris::ConnectionFactory, 4
createConnection
    de::zib::scalaris::ConnectionFactory, 5

de.zib.scalaris, 1
de::zib::scalaris::ConnectionFactory, 3
    ConnectionFactory, 4
    createConnection, 5
    getClientName, 5
    getCookie, 6
    getInstance, 6
    getNode, 6
    isClientNameAppendUUID, 6
    setClientName, 6
    setClientNameAppendUUID, 6
    setCookie, 7
    setNode, 7
    setProperties, 7
de::zib::scalaris::DeleteResult, 7
    DeleteResult, 8
    locks_set, 8
    ok, 8
    undef, 8
de::zib::scalaris::Main, 9
    main, 9
de::zib::scalaris::Scalaris, 9
    closeConnection, 13
    delete, 13, 14
    getLastDeleteResult, 15
    getSubscribers, 15
    publish, 16
    read, 16
    readCustom, 17
    readObject, 17
    Scalaris, 13
    subscribe, 18
    unsubscribe, 18, 19
    write, 19
    writeCustom, 19
    writeObject, 20
de::zib::scalaris::Transaction, 20
    abort, 23
    closeConnection, 23
    commit, 23
    read, 23
    readCustom, 24
    readObject, 24
    reset, 24
    revertLastOp, 25
    start, 25
    Transaction, 22
    write, 25
    writeCustom, 25
    writeObject, 26
delete
    de::zib::scalaris::Scalaris, 13, 14
DeleteResult
    de::zib::scalaris::DeleteResult, 8

getClientName
    de::zib::scalaris::ConnectionFactory, 5
getCookie
    de::zib::scalaris::ConnectionFactory, 6
getInstance
    de::zib::scalaris::ConnectionFactory, 6
getLastDeleteResult
    de::zib::scalaris::Scalaris, 15
getNode
    de::zib::scalaris::ConnectionFactory, 6
getSubscribers
    de::zib::scalaris::Scalaris, 15

isClientNameAppendUUID
    de::zib::scalaris::ConnectionFactory, 6

locks_set
    de::zib::scalaris::DeleteResult, 8

main
    de::zib::scalaris::Main, 9

ok
    de::zib::scalaris::DeleteResult, 8

publish
    de::zib::scalaris::Scalaris, 16

read
    de::zib::scalaris::Scalaris, 16
    de::zib::scalaris::Transaction, 23
readCustom
    de::zib::scalaris::Scalaris, 17
    de::zib::scalaris::Transaction, 24
readObject

de::zib::scalaris::Scalaris, 17
de::zib::scalaris::Transaction, 24
reset
de::zib::scalaris::Transaction, 24
revertLastOp
de::zib::scalaris::Transaction, 25

Scalaris
de::zib::scalaris::Scalaris, 13
setClientName
de::zib::scalaris::ConnectionFactory, 6
setClientNameAppendUUID
de::zib::scalaris::ConnectionFactory, 6
setCookie
de::zib::scalaris::ConnectionFactory, 7
setNode
de::zib::scalaris::ConnectionFactory, 7
setProperties
de::zib::scalaris::ConnectionFactory, 7
start
de::zib::scalaris::Transaction, 25
subscribe
de::zib::scalaris::Scalaris, 18

Transaction
de::zib::scalaris::Transaction, 22

undef
de::zib::scalaris::DeleteResult, 8
unsubscribe
de::zib::scalaris::Scalaris, 18, 19

write
de::zib::scalaris::Scalaris, 19
de::zib::scalaris::Transaction, 25
writeCustom
de::zib::scalaris::Scalaris, 19
de::zib::scalaris::Transaction, 25
writeObject
de::zib::scalaris::Scalaris, 20
de::zib::scalaris::Transaction, 26

A.12 Developing, Simulating, and Deploying Peer-to-Peer Systems using the Kompics Component Model

Developing, Simulating, and Deploying Peer-to-Peer Systems using the Kompics Component Model

Cosmin Arad
Royal Institute of Technology (KTH) Forum 120
SE-164 40, Kista, Sweden
icarad@kth.se

Jim Dowling
Swedish Institute of Computer Science (SICS) Box 1263
SE-164 29, Kista, Sweden
jdowling@sics.se

Seif Haridi
Royal Institute of Technology (KTH), Swedish Institute of Computer Science (SICS)
seif@sics.se

ABSTRACT

Currently, the development of overlay network systems typically produces two software artifacts: a simulator to model key protocols and a production system for a WAN environment. However, this methodology requires the maintenance of two implementations, as well as adding both development overhead and the potential for errors, through divergence in the different code bases. This paper describes how our message-passing component model, called Kompics, is used to build overlay network systems using a P2P component framework, where the same implementation can be simulated or deployed in a production environment. Kompics enables two different modes of simulation: deterministic simulation for reproducible debugging, and emulation mode for stress-testing systems. We used our P2P component framework to build and evaluate overlay systems, and we show how our model lowers the programming barrier for simulating and deploying overlay network systems.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distribu-ted Systems—*distributed applications*; D.1.3 [**Programming techniques**]: Concurrent programming—*distributed programming*; D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures*; I.6 [**Simulation and Modeling**]: Types of Simulation—*Discrete event.*

General Terms

Design, Experimentation

Keywords

component model, software architecture, event-based systems, peer-to-peer, discrete-event simulation

This research has been partially funded by the European Commission IST Project SELFMAN (contract 34084) and by a Marie Curie Intra-European Fellowship within the 6th European Community Framework Programme.

COMSWARE'09, June 16-19, Dublin, Ireland

1. INTRODUCTION

The development of an overlay network system often starts with the development of a simulation model to help verify a system's expected behaviour, aid in its understanding, and allow for the quicker iterative improvement of its protocols. Later, a production system is built using the simulation model, and it is typically further developed in a test environment, such as PlanetLab [9] for WANs, or Modelnet [22], for LANs. However, this process adds overhead and introduces scope for development errors.

In this paper, we describe how our component model, Kompics, is used to build P2P overlay systems that can be both simulated and deployed in production with the same code base. Kompics is a programming model for building distributed systems consisting of message-passing components with no shared state, decoupled dependencies through publish-subscribe ports and channels, and typed events for inter-component communication. We used Kompics to build a P2P component framework that provides fundamental services such as network messaging, timers, bootstrapping, failure detection, application monitoring and administration. We show how Kompics enables the composition of overlay network systems such as Chord [21] and Cyclon [23], and how the same code for these systems can be run in both deployment and simulation environments. This contrasts with existing P2P frameworks that provide specialised APIs for particular classes of P2P system, such as a *key-based routing* API for structured overlay networks (SON) [11, 3] or a *peer sampling* API for gossip-generated overlay networks [15]. The framework provides fundamental network and timer services as components, while Kompics enables its systematic extension through component composition, giving us the ability to selectively expose complexity to higher layers in the framework.

The Kompics P2P component framework provides three different modes of operation: reproducible simulation, stress-test emulation, and production deployment. Simulation mode is used primarily for debugging and studying properties of the system at large. In this mode, network and time services are replaced by simulated components and Kompics is used with a single-thread deterministic scheduler, thus enabling execution runs to be reproducible. Emulation mode is a novel feature, where the system is run in physical-time, using network services, and multiple workers execute components in parallel. Parallel execution, on multi-core machines, and the use of the network enable us to scale up experiments, which is useful for stress-testing systems. We can scale the number of workers to match the number of cores on our hosts, as well as distributing experiments over multiple hosts (in a LAN). Finally, in production deployment mode, the same source code can be deployed and run in a network environment using components, such as Apache MINA [1], to provide network communication.

This paper is structured as follows. Section 2 introduces the Kompics component model. Section 3 describes our P2P component framework. In Section 4, we evaluate our P2P component framework and the scalability of the Kompics emulator on multi-core hardware. Section 5 covers related work in the areas of P2P frameworks, component-based systems software, and simulators. Finally, Section 6 concludes the paper.

2. KOMPICS COMPONENT MODEL

In Kompics, distributed protocols and services are encapsulated into components that can be composed into hierarchical architectures of composite components. Kompics components interact by passing asynchronous data-carrying typed events. They are decoupled by ports that are connected using typed channels, together providing publish-subscribe communication semantics. The Kompics model presented here has been implemented in Java [4, 2]. The following are the main concepts of the Kompics component model:

Event: Events are passive objects that contain a set of immutable attributes. The type of each attribute is a value type of the underlying type system. Events are typed and they can form type hierarchies.

Port: A port represents a bidirectional event interface of a component and it specifies the types of events that flow into or out of the component. A component *provides* a service through a port and in implementing that service it may *require* services provided by other components. A port type is defined as a set of typed events and the direction in which each event flows - into or out of a component. Every port has an *event queue* that stores events sent to a component over that port until they are handled by the component. Ports are connected together using typed *channels*, where connection cardinality may be one-to-one or one-to-many. As such, ports provide publish-subscribe semantics for component interactions.

Event handler: An event handler is a procedure executed by a component as a reaction to receiving a certain event. An event handler accepts one event parameter of a specific event type. An event handler comprises a sequence of statements in the underlying programming language and Kompics specific statements (e.g., triggering events).

Subscription: A subscription represents the binding of an event handler to a port. A subscription specifies both the event type and a reference to an event handler that will be executed when events of that type (or subtypes of the specified type) are received on that port. A component can subscribe event handlers to more than one event type for a given port.

Channel: A channel is a bidirectional typed connection between two ports, forwarding events between the two ports. A port of a component may be unconnected or connected to many components, using one channel per connection. Channels only carry events of the same type as those specified in the connected ports. Channels have first-in, first-out (FIFO) event delivery semantics, that is, events triggered by a publisher component are executed by all subscriber components in the same order in which they are triggered. A channel can optionally contain event *attribute filters*, which ensure that only those events with specified attribute values are forwarded by the channel. Together, type-based subscriptions and channel attribute filters ensure that a component only receives those events published on a particular port if the component has (1) subscribed for the event's type (or the event is a subtype of the subscription type) and, (2) the event has attribute values that match all supplied attribute filters.

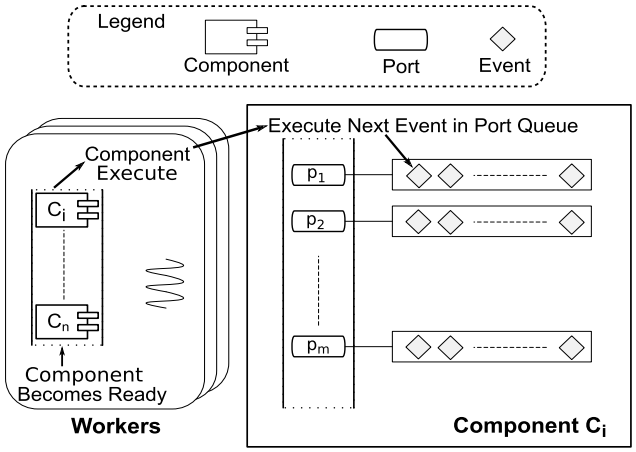


Figure 1: Kompics execution model.

Component: Components are units of functionality in a Kompics application. A component has local state variables, a set of ports, and a set of event handlers. Components do not share state. Components are active entities that interact with each other by triggering (sending) and handling (receiving) events. Components react to events by executing event handlers, and they are decoupled by ports and channels. Components can be composed into composite components that are also themselves components, enabling the construction of a component hierarchy. We say that a parent component is at a higher level in the component hierarchy than its children components.

2.1 Execution Model

Kompics components are reactive. They do not have execution threads of their own. The default Kompics scheduler spawns a number of worker threads that execute event handlers on behalf of components. Typically, the number of workers is equal to the number of processing cores or processors available on the machine. Workers manage the execution of components by transitioning a component to one of three states:

busy: a worker is currently executing one of its event handlers

ready: one or more of its port event queues are not empty and the component is not *busy*

idle: all its port event queues are empty and it is not *busy*

Kompics's execution model is illustrated in Figure 1. Each worker maintains a private work queue of ready components, and each component maintains an event queue for each of its ports. The algorithm for how workers pick components from the work queue and execute their events is described in pseudo-code in Figure 2. The algorithm proceeds as follows. If a worker has no components in its ready queue, it steals work from another worker. Details for this are described in the next subsection. When work becomes available at a worker, it picks the first ready component *C* from its work queue. The worker then transitions *C* to the *busy* state. The component *C* now selects a port with waiting event(s) in a round-robin fashion,

```
while true do
  if isEmpty workQ then
    steal work from another worker
  end if
  var comp : dequeue workQ
  comp state : busy
  var port : selectPort
  var event : port eventQ dequeue
  comp executeHandler port event
  if comp hasEvents then
    comp state : ready
    workQ enqueue comp
  else
    comp state : idle
  end if
end while
```

Figure 2: Worker execution

and then takes the first event from the event queue for that port. (Round-robin selection of ports with waiting events ensures fairness in event execution for the different ports.) The event handler in *C* for that event on that port is then executed. After the handler execution terminates, if all the event queues for *C* are empty, the worker transitions *C* to the *idle* state, otherwise *C* is transitioned to the *ready* state, and it places *C* on the tail of the work queue. When a component is in the *idle* state and an event is published on one of its ports by a worker, the worker transitions it from the *idle* state to the *ready* state and places it on its work queue; there is no scanning of *idle* components for waiting events.

As each worker has a private queue with ready components, different workers can execute event handlers for different component instances in parallel. This improves concurrency, since there is no atomicity of event handlers with respect to handler execution of other components. However, event handlers of one component instance are still guaranteed to be executed sequentially and non-preemptively by workers. This eliminates the need for programmers to synchronise access to local component state variables between different event handlers, which reduces programming complexity.

Batched work stealing

Workers may run out of ready components to execute, in which case they engage in *work stealing* [7]. Work stealing involves a *thief*, a worker with no ready components, contacting the *victim*, the worker with the highest number of ready components, and stealing a batch of half of its ready components. Stolen components are moved from the victim's work queue to the thief's work queue. From our experiments, batching shows a considerable performance improvement over stealing small numbers of ready components. To improve concurrency, the work queue is a lock-free queue, meaning that the victims and thieves can concurrently consume ready components from the queue.

3. P2P COMPONENT FRAMEWORK

Kompics was used to implement a set of generic, reusable components for building overlay network systems, called the Kompics P2P component framework.

3.1 Basic components: Timer and Network

The fundamental services common to all P2P protocols are timer services and network services.

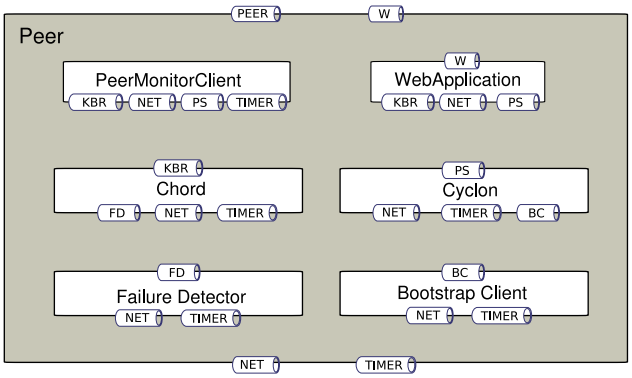


Figure 3: Peer component.

Timer

The *Timer* component is used by other components to schedule alarms and timeouts. To schedule a timeout event, a component *a* triggers a *ScheduleTimeout* event on its *TIMER* port. Through a channel connection the *ScheduleTimeout* event arrives at the *TIMER* port of the *Timer* component. The *ScheduleTimeout* event contains a timeout value and a *Timeout* event. A *Timer* component triggers a *Timeout* event on the *TIMER* port, when a period of time equal to the timeout value has passed. The *Timeout* event arrives at the *TIMER* port of component *a* on the same channel as the *ScheduleTimeout* event. To cancel a scheduled *Timeout* event, component *a* will trigger a *CancelTimeout* event on its *TIMER* port. Each *Timeout* event has a uniquely generated id. To cancel a particular *Timeout*, its id needs to be included as an attribute in the *CancelTimeout* event.

Network

Protocol components use a *Network* component to send messages to remote peers. When a component *a* wants to send a message to a remote peer, it first defines its message as a subtype of *Message*, where the *Message* type includes a source and a destination *Address* attribute. The component *a* triggers its *Message* on its *NET* port. Through a channel connection the *Message* event arrives at the *NET* port of the *Network* component. The *Network* component marshalls the *Message* event and sends it to its destination, where another *Network* component is responsible for receiving the event and publishing it on its local *NET* port.

3.2 Peer component

Figure 3 outlines the architecture of a *Peer* component. The *Peer* component is a container for the components representing the protocols executing at one peer. The *Peer* component has a *NET* port, through which its children components send and receive messages from the network, a *TIMER* port, through which the *Peer*'s children components can schedule timeouts, and a *WEB* port, through which web requests coming from a web server are handled by a *WebApplication* within the peer.

The *Peer* component in Figure 3 contains two overlay network components, *Chord* and *Cyclon*, and some P2P framework components: a *BootstrapClient*, a *FailureDetector*, and a *PeerMonitorClient*. We describe the role of these components in the following sections.

A *Peer* has both a *NET* port and a *TIMER* port that need to be bound to concrete implementations. In simulation mode, these ports are bound to loopback network and simulated timer compo-

nents, while in production mode, they are bound to network messaging middleware and Java timer components. A *Peer* contains a globally unique *Address*, a triplet of *IP address port id* . *id* is an abstract peer identifier that is generally specialized for a specific overlay network, such as a key-based routing (KBR) addressing scheme like Chord [21]. In a production system, the peer component typically has an embedded *Application*, see Figure 6.

To elaborate on how the *Network* component is used in a P2P system, we now describe how our implementation of the *Cyclon* overlay network uses the network while constructing the random overlay network topology. Firstly, *Cyclon* components send shuffle messages to each other to update their set of random neighbours. The *Cyclon* component is subscribed to the *NET* channel for both *ShuffleRequest* and *ShuffleReply* types (both subtypes of *Message*). A *Network* component is subscribed to the same *NET* channel for *Message* types, see Figure 6. When the *Cyclon* component starts a gossiping round, it sends a *ShuffleRequest* to its *NET* port, and since *ShuffleRequest* is a subtype of *Message*, it is delivered over a channel to the *NET* port on a *Network* component. A handler in the *Network* component receives the event, as it is subscribed to receive messages of type *Message* arriving on the *NET* port. The handler sees that the event is destined for a remote peer, marshalls the *ShuffleRequest* and sends it over the network using the IP and port from the destination *Address* attribute of the *Message* event. The destination node receives the *ShuffleRequest* at its *Network* component and publishes the *ShuffleRequest* on its *NET* port. Type-based subscription ensures that the *Cyclon* component receives the *ShuffleRequest* from the peer's *NET* port. Similarly, a *ShuffleReply* is sent back to the source peer.

3.3 Peer sub-components

In Figure 3, we can see the set of components that make up a *Peer* component, including two overlay network components (*Chord* and *Cyclon*), components providing services used by the overlay components (*FailureDetector* and *BootstrapClient*), and components that use an overlay network component (*PeerMonitorClient* and *WebApplication*). The *Chord* component shown is simplified, hiding its internal *Ring* and *Router* components. The *key-based routing* (*KBR*) port enables the replacement of the *Chord* component with another *KBR* implementation. The *Cyclon* component implements a gossip-generated random overlay network. As with the *KBR* channel, the peer-sampling channel (*PS*) port allow the replacement of *Cyclon* with any other implementation of a random overlay network supporting the *PS* port. *Cyclon* uses a local *BootstrapClient* component that communicates with a well-known remote *Bootstrap* server (not shown in Figure 3), to allow peers to join an existing overlay network. *Chord*, by contrast, does not require a bootstrap service, as it can be bootstrapped using a reference to any peer in the overlay. *Chord* uses a *FailureDetector* component to receive a notification when one of its neighbour (predecessor or successor) nodes is suspected to have crashed.

Peer monitoring and administration.

The framework also provides support for monitoring the state of the P2P overlay using web browsers. To support this feature, a *PeerMonitorClient* component periodically requests the state of *Peer* components. The *PeerMonitorClient* component provided in the framework can be configured to monitor *Chord* or *Cyclon*. It is reusable for overlays that handle *KBR* or *PS* events, and can also be customized to collect application-specific data. The data collected by the *PeerMonitorClient* component is sent to a *PeerMonitorServer* (shown in Figures 4 and 6), which aggregates a global view of the overlay from the *PeerMonitorClient* component inside

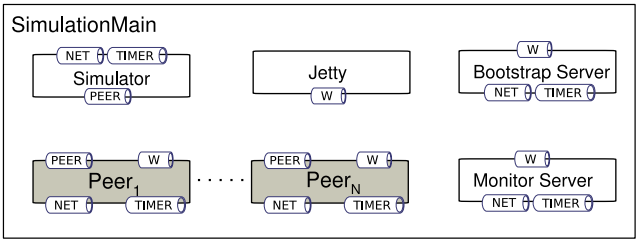


Figure 4: Reproducible simulation.

every peer. The *PeerMonitorServer* can be accessed via a web interface, allowing users to inspect the global state of the system in real-time. The *PeerMonitorServer*'s web interface contains hyperlinks to the individual peers in the system. These hyperlinks point to the peer's *WebApplication* component, see Figure 3. Users can send HTTP requests to the Peer's *WebApplication* component to inspect peers and administer them. For example, using the *WebApplication* interface to a *Chord* component, a user can issue *insert* or *lookup* operations for that peer, using a web browser. From the *PeerMonitorServer*, the web interface can be used to inspect the Chord ring, from which a user can click down to individual peers. This whole infrastructure is reusable for any overlay network component that implements the *KBR* port. The same applies for random overlay components that provide the *PS* port.

3.4 Simulation mode

Programmers can switch between the different simulation, emulation, or production modes by simply replacing the *Main* (or root) component; this is typically done using a command-line switch or configuration file update. In the simulation scenario of Figure 4, the *SimulationMain* component is responsible for creating and starting a *Simulation* component, as well as the *Peer*, *Jetty*, *PeerMonitorServer*, and *Bootstrap* components. The *Simulation* component's *Peer* port is connected to all the *Peer* components using one channel per *Peer*. The *Simulation* component provides an interface to start, stop and fail individual *Peer* components.

Simulation mode is characterised in Kompics by the support for only a single worker with a single work queue. A single worker ensures the reproducibility of simulations, as multiple workers introduce non-determinism, due to operating system control over the scheduling of workers.

In simulation mode, a simulated *Timer* component implementation is required to alter simulation time. This simulated *Timer* component typically slows down simulated time (compared to physical time) to allow the simulation of larger systems. Other references in the application to physical time, such as through `System.currentTimeMillis()` need to be replaced to refer to simulation time.

In simulation mode, each peer subscribes to receive messages from the *NetworkSimulation* component (not shown in Figure 4) over the *NET* port. Peer subscriptions are parameterized by the peer's *Address*, so that the channel connecting the peer and the *NetworkSimulation* component uses attribute filtering to ensure that only those events of type *Message* with the peer's destination address are delivered to the peer. We also model the network at the message-passing level in the *NetworkSimulation* component, instead of the more computationally expensive network packet and/or physical level. We have implemented different models of message loss and latency, including uniform delay distribution models and a latency map based on the King data-set [13].

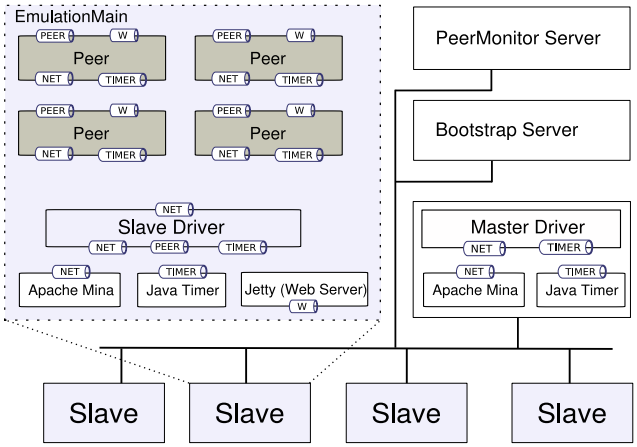


Figure 5: Distributed emulation in Kompics.

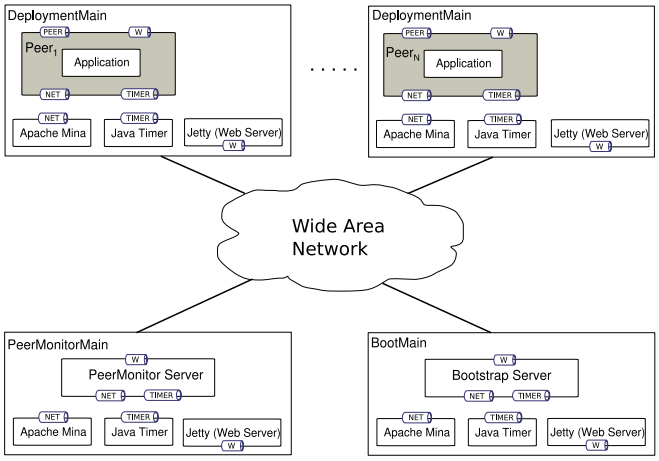


Figure 6: Deployment architecture.

3.5 Emulation mode

Emulation mode contrasts with simulation mode in that production *Timer* and *Network* components are used, instead of simulated implementations of them. Emulation mode supports multiple workers, each with their own work queue, enabling emulation mode to scale as the number of available processing cores is increased.

In both the emulation and deployment modes, there is currently one supported implementation of the *Network* component based on Apache MINA [1], see Figure 6. MINA is a network communication library that provides an event-driven asynchronous API over transports such as TCP/IP and UDP/IP using the Java NIO package. MINA supports pluggable marshallers, including a built-in object serializer, which is useful during prototyping, while message-specific marshallers, for compact binary or text protocols, can be written for production systems. The *Network* component embeds MINA and automatically manages connections to other nodes.

Distributed emulation

When the amount of available memory becomes a bottleneck to scaling the size of the emulated systems, the way to further scale the simulation is to distribute it across a number of machines, building a *distributed simulation*. Given our message-passing architecture, distributing the simulation over a cluster is straightforward. In Figure 6 it can be seen how we split the simulation component into a master and a set of slaves, distributed across the machines in the cluster. The master drives the simulation by sending messages to the slaves. Each slave drives the simulation on one machine, managing the peers local to that machine. The master is hosted on a separate machine. In the simulator's component architecture we replace the simulator component with the *SlaveDriver* component, and we move the bootstrap and peer monitoring servers to other hosts (to improve system scalability).

Analysis of physical-time emulation

In Kompics's emulation mode, experiments are run in physical-time, that is, in the special case where simulation time is equivalent to physical system execution time. This allows us to have multiple workers executing the experiment without the need to synchronize the workers on the passage of simulation time. It also enables us to run production code directly in simulation. However, the use of physical-time means that events may not execute at the expected simulation time due to delays. However, most distributed systems, and all P2P systems, are tolerant to messaging delays within some application-specific bounds.

Lin et al. showed [18] that this approach is valid to the extent that the delay of events in queues does not affect application invariants. Application invariants are properties of the application that must be maintained over all execution runs. Lin et al. showed [18] that for P2P systems, application invariants can be specified as conditions on the logic of timers. So, for example, an application-level timeout event cannot be delayed for an amount of time longer than its expiration time, otherwise it would timeout before it could be handled, breaking the application invariant. In Kompics simulations run on a single multi-core machine, events will encounter increasing processing delays with increasing system load. Processing delays happen if the system generates more events than it can process over a period of time. In our evaluation of the Cyclon overlay, we investigate the extent to how large the system can grow for different numbers of processing cores, while maintaining application invariants in Cyclon. That is, we have to keep the highest event processing delays considerably below the time for the minimum timeout expiration in the Cyclon application.

A drawback of physical-time emulation is that we cannot take advantage of the time-compression effect of time-stepped simulators and simulation runs are not reproducible (although, production distributed systems do not have reproducibility, either). However, the benefit of physical-time is improved scalability, since it avoids the cost of simulation controllers agreeing on the passage of simulation time.

3.6 Production mode

In production mode, the same *Network* and *Timer* components that were used in emulation mode can be used. In the production scenario of Figure 6, we can see that the *DeploymentMain* component is responsible for creating and starting a single *Peer* component, parameterized by its *Address* and a *NET* port connected to Apache Mina *Network* component implementation.

4. EVALUATION

In this paper we have argued that P2P systems built from Kompics components can be executed both in a production and in a simulation environment, and that by introducing a new emulation mode, simulation experiments can be scaled to larger sizes with additional processing cores while maintaining experimental integrity. In this section we give a qualitative evaluation of the Kompics pro-

gramming methodology and a quantitative evaluation of the multi-core scalability of our emulation approach.

4.1 Analysis of the P2P component framework

We have implemented the Kompics component model and the P2P component framework in Java. Components and events are specified as Java classes, while channels are generic objects that do not need to be specialized. Firstly, switching from deployment mode to simulation mode is minimally intrusive, requiring the replacement of the *Main* component of the system. This integrated simulation and production code-base should reduce development overhead and the potential for programmer errors.

Our component framework also enables the re-use of the P2P components for different overlay systems. Chord and Cyclon use the same implementations of *Peer*, *Network*, *Timer*, *BootstrapClient*, and *Simulation* components, as well as the same *BootstrapServer* and *PeerMonitorServer*. The main abstraction in helping reuse components is the use of ports to decouple sender and receiver components. Neither the sender nor the receiver of an event has a reference to the other component, only a reference to the port connecting them. Ports, together with typed channels connecting them, make components oblivious to the number and type of the components they communicate with. Component decoupling enables the easier reuse of components, as components are not dependent on the type of components they are connected to, the number of components they are connected to, and components do not block waiting on replies from other (potentially unreliable) components.

Another benefit of the framework when programming P2P systems, are the event filtering functions. Type-subscription filtering and attribute filtering reduce the potential for programmer error by freeing the programmer from having to explicitly write code that rejects events not that are destined for a component. Our filtering implementation also improves system scalability by removing unnecessary testing if components have subscribed for particular events. Without filtering support, the testing of N components connected to a channel for subscriptions would require $O\ N$ operations. In Kompics, both type-based and attribute filtering are implemented using a constant-time lookup on a hash table, enabling events to be filtered in $O\ 1$ operations. This filtering implementation significantly improves performance for large-scale P2P simulations, for example, when delivering an event to a single peer on the shared network channel with N subscribers, where N can be on the order of thousands.

4.2 Scalability of emulation mode

We evaluated the scalability of our emulation mode for multi-core hardware by performing an experiment that tests the scalability of experiments to an increasing the number of available processing cores. Our hardware setup was a Mac Pro machine with 2 quad-core 2.8GHz Intel Xeon E5462 CPUs, Windows XP 32bit, and the Sun Java server VM version 1.6.0 update 7 with a heap size of 1426 Mb and a parallel garbage collector. We executed this experiment using 1, 2, 4, and 8 Kompics workers.

Our first experiment is to test the scalability of our implementation of Cyclon for an increasing number of processing cores. Our expectations for the experiment are as follows. As the size of the simulated P2P system is increased, more components are created in our emulator. A larger number of components leads to an increased flow of events passed between the components. With bounded processing power and an increased number of events in the system, we expect that each event will experience a larger delay due to event queuing time before being processed. If this event queuing time exceeds a certain application-specific bound, timing-related invariants

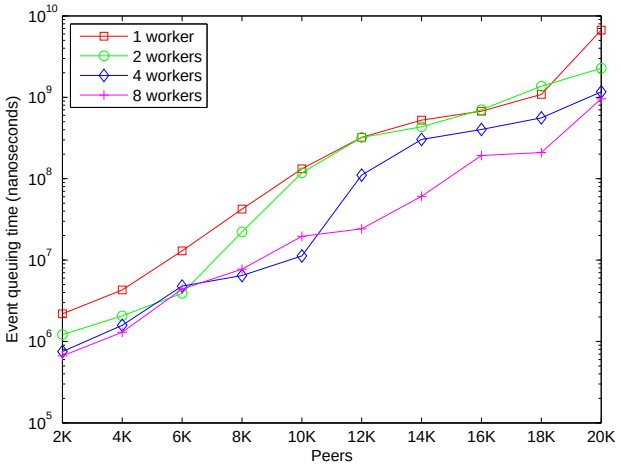


Figure 7: The 99th percentile of event queuing time as a function of simulated system size.

of the simulated system may be broken. Thus, to gain confidence in the integrity of our simulation studies we must make sure that event queuing time is bounded.

We implemented the Cyclon overlay from Kompics components and used it to evaluate the multi-core scalability of the Kompics emulator. In Cyclon, the essential timing-related invariant is that every peer gossips with one other peer in one cycle. This invariant may be broken if events are delayed for longer than the gossip period, and the observed properties of the Cyclon overlay are inaccurate. In this implementation for each peer we have 4 components: *Cyclon*, *BootstrapClient*, *PeerMonitor* a *JettyWebServer*. We bootstrapped the system with 2000 peers gossiping every 10 seconds and we measured the event queuing time for every event in the system. We continued to join 2000 peers at a time until the 99th percentile of the event queuing time exceeded the 5 seconds bound.

In Figure 7 we plot the 99th percentile of event queuing time for all events in the system. Firstly, as expected, we can see that for increasingly larger system sizes, the event queuing time increases. We can also observe that even for 20 000 peers, for 99% of the events in the system, the observed queuing delay is less than 5 seconds, which constitutes half of the Cyclon cycle period, thus not invalidating its application invariant. For the scalability of our emulator, we can see that event queuing times are consistently lower when a system with the same number of peers is executed using an increased number of processing cores. So although the 99th percentile of the event queuing time for 20 000 peers is 5 seconds when using 1 processing core, it drops to under 1 second when using 8 cores.

4.3 Distributed emulation scalability

We have used the Cyclon implementation described in the previous section to investigate the extent to which the size of the emulated system can be scaled by distributing the simulation over multiple machines. We executed the experiment on a set of 10 IBM blades each having 2 hyper-threaded 3GHz Intel Xeon CPUs using SMP Linux 2.6.24-19-server and the Sun Java server VM version 1.6.0 update 7, with a heap size of 2698 Mb and a parallel garbage collector. We used 2 Kompics workers on each machine.

We bootstrapped the system with 1000 peers on each machine,

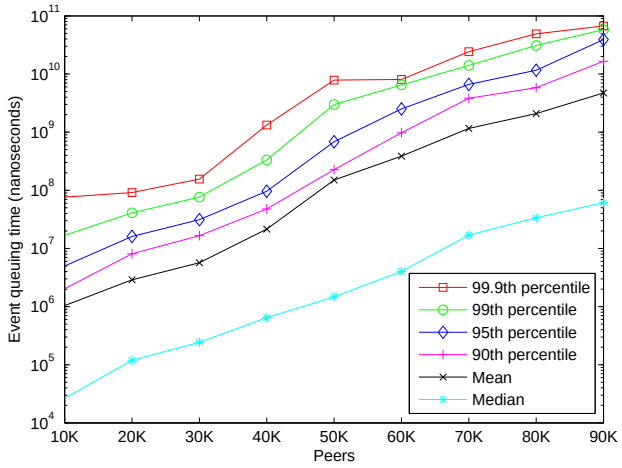


Figure 8: Event queuing time as a function of simulated system size in distributed emulation mode.

gossiping every 10 seconds and we measured the queuing time of all events in the system for a duration of 30 seconds. We continued to join 1000 peers per machine, at a time, and measured the event queuing time for 30 seconds at each step. We stopped at 9000 peers per machine, i.e., a total of 90 000 peers.

We plot the measured event queuing times in Figure 8. The results show that we can simulate around 40 000 Cyclon peers while 99% of all events in the system are not delayed by more than 300 milliseconds. This compares with roughly 16 000 Cyclon peers for a single host with two cores (albeit running on higher performance hardware), from Figure 7. This demonstrates the potential of distributed emulation in Kompics for increasing the scalability of experiments by adding additional hosts with a LAN environment.

4.4 Analysis

The results here show that emulation mode for Kompics P2P applications enables experiments to scale in number of peers, while maintaining bounded event queuing time. Experiments can be scaled simply by adding more CPU cores to a host or by adding additional hosts to an experimental configuration. Experiment runs introduce minor variations in the order of processing events, caused by worker scheduling and hosts running in parallel without agreement on the passage of physical time. In agreement with Lin et al. [18], we argue that these minor variations are useful when testing distributed systems, as they model types of event processing delays that can be expected in production systems, such as those caused by network faults and congestion. As such, our emulation mode provides a useful stage in the development of P2P systems, in that it enables the testing of larger-scale P2P systems in a more challenging environment. This stage of testing for production systems could complement traditional stress-testing stages, by helping to build sufficiently large experiments that are able to identify unexpected behaviours that arise at large system sizes.

5. RELATED WORK

The Kompics P2P component framework is closely related to work in the area of frameworks for building P2P and component-based systems software, discrete event simulators, and message-passing languages.

An overlay network is defined as a distributed algorithm that establishes logical neighbour connections between subsets of peers from a set of global participants, where the peer connections are overlayed atop the IP substrate [20]. Due to their high complexity and dynamism, overlay network systems are generally not amenable to study using analytic methods. Evaluation of overlay systems is typically first done in simulation. In particular, large-scale overlay networks are simulated at the message level, rather than network packet level. There are several popular P2P simulators including P2PSim [17], Peersim [15] and Oversim [6], which extends a domain-independent discrete-event simulator called Omnet++. More relevant to Kompics, however, are the frameworks for building overlay network applications that support using the same code in both simulation and production. The two most significant of these are Mace [16, 20] and Wids [18]. In Mace and Wids, programmers specify system logic using a high-level event-based language that is subsequently compiled to C++ code, which, in turn, uses APIs for framework-specific libraries. For execution in a network environment, Wids and Mace require the programmer to rebuild the system and link it to network libraries. Oversim code cannot be directly executed in production environments.

There have been several component models designed for building reconfigurable systems software, such as OpenCom [10], K-Components [12] and Fractal [8] for building middleware, and Live objects [19] for composing distributed protocols. In contrast to these systems with shared-state component models and request-reply invocation semantics, the concurrency of Kompics components is controlled by the Kompics execution model, thus simplifying their programming.

Kompics' message-passing model was inspired by Erlang [5]. Message-passing as a programming and concurrency model allows the easy expression of distributed systems as state machines. In contrast to existing event-based languages and frameworks, where workers execute events from a single shared event queue, Kompics supports private queues at workers.

In the area of simulation, discrete-event simulators (DES) have been long used to simulate distributed systems. In existing DES simulators, a single controller manages an event queue and executes events. Parallel and distributed DES attempt to take advantage of the availability of multiple processors, by having multiple Logical Processors (LP) drive the simulation concurrently. However, the LPs need to reach agreement on the passage of simulation time. There are two main approaches to this: using a pessimistic or an optimistic policy. In the pessimistic approach, all LPs advance the logical time in lock-step, and the only events that are scheduled to execute at the current simulated time step are those whose scheduled time is below the minimum network delay [18]. In the optimistic approach, known as time-warping [14], events whose scheduled time is greater than the minimum network delay can be executed, but a rollback may be required if an application invariant is violated, that is if some property of the application is violated by delays in events, generally due to network or processing delays. In this rollback case, the LPs restore the system state and recall events for the scheduled execution time of the event. Both of these approaches suffer from scalability problems for multi-core hardware, with either excessive synchronization for the pessimistic policy or excessive rollbacks for the optimistic policy [18]. The Wids simulator [18] offers an alternative approach, called Slow Message Relaxation, that exploits the fact that P2P systems generally tolerate message delays up to the level where timers do not expire because messages were delayed too long. We exploit this same property of P2P systems in our stress-test emulation mode. In Wids, if an event arrives late (which would cause a rollback in the optimistic approach), its timestamp is updated to the current

simulation time, and then the event is executed as if it were delayed in the network. As long as events do not arrive so late as to cause application-specific timers to expire, the protocols should work properly.

Existing DES are not designed for scalability on multi-core architectures, in particular, due to contention on a single event queue. While multiple cores can reduce overall simulation time by running simulations in parallel on different cores, the number of cores available on commercial systems has already exceeded the typical number of simulation replications required to establish reasonable confidence interval lengths, that is, 10-12 replications. This means that to better utilise multi-core hardware, DES should increase their level of concurrency. Existing DES provide some concurrency, as in Wids, by scheduling events that occur in the same logical time step on different LPs. However, synchronization is still required between LPs on the passage of simulation time, for example, as provided by a master server in Wids. As such, existing DES contrast with Kompics's emulation mode. In Kompics's emulation mode, we remove the single event queue bottleneck and remove the need for synchronization between workers on the passage of simulation time. We trade-off complete reproduceability of simulation for increased scalability of simulations.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how overlay systems implemented with the Kompics P2P component framework have the ability to run the same code in simulation, emulation, and production, enabling a more integrated iterative development methodology. We presented Kompics and its component execution model, based on shared-nothing workers with private work queues, and showed how it was used to build a reproducible simulator for P2P systems, a scalable emulator for P2P systems, and production P2P systems.

We evaluated our Kompics P2P framework using an implementation of the Cyclon overlay network. We showed how, for our emulation mode, the size of the simulated system scales with an increasing number of processing cores, while maintaining emulation quality and without requiring source code modifications. We also showed how our distributed simulator enables the simulation of even larger scale systems.

We are currently building different types of P2P systems with Kompics and investigating the use of Kompics in building different types of scalable multi-core distributed systems.

7. REFERENCES

[1] Apache MINA project. In *http://mina.apache.org*. The Apache Software Foundation, 2008.

[2] Kompics: Reactive Component Model for Distributed Computing. In *http://kompics.sics.se*, 2009.

[3] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The essence of P2P: A reference architecture for overlay networks. In *The 5th IEEE International Conference on Peer-to-Peer Computing*, pages 11–20, 2005.

[4] C. Arad and S. Haridi. Practical protocol composition, encapsulation and sharing in Kompics. *Workshop on Decentralized Self Management for Grids, P2P, and User Communities*, 2008.

[5] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[6] I. Baumgart, B. Heep, and S. Krause. OverSim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84, 2007.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[9] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[10] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, February 2008.

[11] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. *In Proc. of IPTPS'03 Workshop*, pages 33–44, 2003.

[12] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[13] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *SIGCOMM Internet Measurement Workshop*, 2002.

[14] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. *SIGOPS Oper. Syst. Rev.*, 21(5):77–93, 1987.

[15] M. Jelasity, A. Montresor, and O. Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems, G. Di Marzo Serugendo*, volume 2977, pages 265–282, 2004.

[16] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. *SIGPLAN Not.*, 42(6):179–188, June 2007.

[17] J. Li, J. Stribling, R. Morris, M. Kaashoek, and T. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. *INFOCOM 2005*, 1:225–236, March 2005.

[18] S. Lin, A. Pan, R. Guo, and Z. Zhang. Simulating large-scale P2P systems with the WiDS toolkit. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 415–424, Washington, DC, USA, 2005. IEEE Computer Society.

[19] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn. Programming with live distributed objects. In *ECOOP*, pages 463–489, 2008.

[20] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. Macedon: methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 267–280, Berkeley, CA, USA, 2004. USENIX Association.

[21] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, February 2003.

[22] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.

[23] S. Voulgaris, D. Gavidia, and M. Steen. Cyclon: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

A.13 Practical Protocol Composition, Encapsulation and Sharing in Kompics

Practical Protocol Composition, Encapsulation and Sharing in Kompics*

Cosmin Arad†
†*Royal Institute of Technology (KTH)*
{icarad,haridi}@kth.se

Seif Haridi†,‡
‡*Swedish Institute of Computer Science (SICS)*
seif@sics.se

Abstract

At the core of any distributed system is a set of concurrent distributed algorithms that coordinate the functionality of the distributed system. We present a software architecture, Kompics that is component-based and compositional which facilitates building distributed protocols. The underlying computation model subsumes that of event-based systems, SEDA (staged event-driven architecture) and thread-based models. We illustrate various salient features of Kompics such as ease of use, compositionality and configurability through a series of well chosen distributed protocols.

1. Introduction

Programming distributed systems, applications and services is difficult due to concurrency and partial failures, which are fundamental characteristics of distributed systems, but also due to the complexity of the software architecture of any non-trivial distributed system. One way to simplify the understanding and programming of complex distributed protocols is to split them into their orthogonal aspects and encapsulate each of these aspects in reusable abstractions [6]. Combining multiple interdependent distributed protocols that are concurrently working together gives rise to complex interactions.

We introduce Kompics, a novel software framework for programming, configuring, and executing distributed protocols as software components that interact asynchronously by passing data-carrying events. Kompics components are reactive, concurrent, and they can be composed into complex architectures of composite components that are safely reconfigurable at runtime and allow for sharing of common subcomponents at various levels in the component hierarchy. Software faults occurring in components are isolated and handled by supervisor components organized in flexible supervisor hierarchies [2].

*This work was funded by the European Union in the SELFMAN project (contract 34084). We would like to thank Roberto Roverso for his contribution to the development of earlier versions of Kompics.

The component programming style encompasses both the event-based and the thread-based styles to facilitate programming most distributed protocols, and the execution model allows a flexible allocation of components to threads which enables various resource allocation policies, and readily takes advantage of multi-core hardware architectures.

The Kompics component model targets the development of reliable and adaptable long-lived, dynamic, and self-managing distributed systems. Such systems are composed of many software modules which implement various distributed protocols (e.g. failure detectors, reliable group communication, agreement protocols, gossip protocols, etc.) and interact in complex ways. Kompics aims to facilitate the construction of complex distributed systems by providing a computation model that accommodates their reactive nature and makes their programming as easy as possible.

The rest of this paper is structured as follows: in the next section we present the Kompics component model and some example component architectures. In Section 3 we describe the semantics of component interaction and execution in Kompics. In Section 4 we give a qualitative evaluation of the model. In Section 5 we survey related work. We discuss future work and we conclude in Section 6.

2. Component model

In Kompics, distributed abstractions are encapsulated into components that can be composed into hierarchical architectures of composite components. Subcomponents can be safely shared by multiple components at any level in the component hierarchy. Kompics components interact by passing asynchronous data-carrying events and they are decoupled by a flexible event publish-subscribe system.

Components A component is a unit of functionality and management. Components are active entities that interact with each other by triggering (sending) and handling (receiving) events. Components react to events by executing event specific procedures to handle the received

events. Components are decoupled (by channels) which makes them independent and reusable.

Every component contains some internal state and a set of event-handling procedures. Composite components also contain subcomponents and thus form a component hierarchy. We sometimes call subcomponents *children* components and the containing composite component *parent* component. We say that the parent component is at a higher level in the component hierarchy than its children components.

Events Events are passive objects that contain a set of immutable attributes. Events are typed and they can form type hierarchies. Components subscribe for events to channels and publish events into channels.

Channels Channels are interaction links between components. They carry events from publisher components to subscriber components. Every channel is parameterized with a set of event types that can be subscribed for or published into the channel. Channels exist in the context of the composite components which create them. However, references to channels can be passed between components through events.

Event handlers An event handler is an event-specific procedure that a component executes as a reaction to a received event. An event handler takes as argument one event of a certain type. While being executed, event handlers may trigger new events. Event handlers can be guarded by boolean guards.

Event subscriptions Components subscribe their event handlers to channels by registering event subscriptions at the respective channels. Event subscriptions can be made either by event type or by both event type and event attributes, whereby a subscription contains a set of (attribute, value) attribute filters. Events published into a channel are delivered to all subscriber components which registered at the channel subscriptions that match the published events. Components can publish or subscribe for subtypes of the event types carried by the channel.

Component types A component interacts with its environment (other components) by triggering (output) and handling (input) events. The component is subscribed for the input events to input channels and publishes output events into output channels. Kompics components are parameterized by their input and output channels. The types of input and output events of a component together with the input and output channel parameters that carry them represent the component's type. Component types are useful for expressing a composite component's dependencies on subcomponents in terms of what they do and not on how they do it.

Component membranes A component membrane is the runtime correspondent of the component's type. The component membrane is a set of references to the actual channels that the component is using as input and output channel parameters. The membrane maps every pair (event type, in/out direction) to an actual channel reference.

Component sharing A component is shared between multiple composite components essentially by sharing the channels in its membrane. To share one of its subcomponents, a composite component registers the subcomponent's membrane under a name, in a registry of shared components. Other composite components can retrieve the membrane (by name) from the registry and use its channels to communicate with the subcomponent. This registry is hierarchical in the following sense: (1) names registered at some level in the component hierarchy (the level of the parent component of the shared component) are not visible at higher levels, and (2) names registered at a lower level in the component hierarchy shadow the same names registered at a higher level.

2.1. Examples

We illustrate some of the concepts in Kompics by examples from Guerraoui et al. [6]. Figure 1 shows an example graphical representation of Kompics components. Here we have two components: A and B. They communicate through channel x which carries events of type E_1. Component A has an output channel formal parameter and component B has an input channel formal parameter. Both A and B are parameterized by channel x. A uses x for its output channel formal parameter and B uses x for its input channel formal parameter. Component B has an event handler that is subscribed to B's input channel and handles events of type E_1. Component A has an event handler that publishes events of type E_1 in A's output channel. Both components A and B and channel x exist in the context of a parent component, Node.

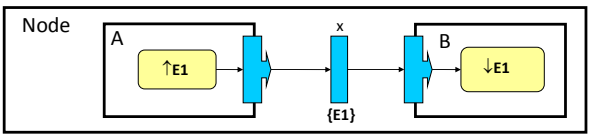


Figure 1. Graphical representation of Kompics components.

Figure 2 shows an example graphical representation of a Kompics composite component, namely a Best-Effort Broadcast (BEB) [6] component that contains a Perfect Point-to-point Links (PP2P) [6] subcomponent. The BEB component is parameterized by an input channel carrying $BebBroadcast$ events and an output channel carrying $BebDeliver$ events. The PP2P component is parameterized by an input channel carrying $Pp2pSend$ events and an output channel carrying $Pp2pDeliver$ events. The BEB component contains two local channels that are used as the actual channels that parameterize the PP2P subcomponent and two event handlers that handle $BebBroadcast$ and $Pp2pDeliver$ events respectively, and trigger $Pp2pSend$ and $BebDeliver$ events respectively. The arrows indicate event subscriptions and publications.

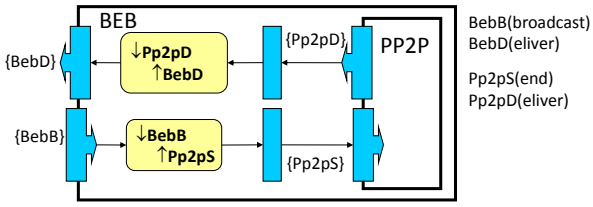


Figure 2. Graphical representation of a Kompics composite component.

Figure 3 shows an example of two composite components sharing a common subcomponent. A Perfect Point-to-point Links (PP2P) [6] component and a Fair-Loss Point-to-point Links (FLP2P) [6] component share a Network subcomponent. The shared component as well as the channels in its membrane are represented with double borders. The Network component accepts $NetSend$ events (in the sender process) and triggers $NetDeliver$ events (in the receiver process). The $NetDeliver$ event that is to be delivered in the receiver process is encapsulated in a $NetSend$ event on the sender process.

$NetSend$ events triggered by PP2P in the sender process are to result in $NetDeliver$ events handled only by PP2P in the receiver process. Correspondingly, $NetSend$ events triggered by FLP2P in the sender process are to result in $NetDeliver$ events handled only by FLP2P in the receiver process. Filtering $NetDeliver$ events between PP2P and FLP2P is done by event subtyping, i.e., PP2P and FLP2P subscribe to the Network component's output channels for different subtypes of the $NetDeliver$ event type.

We refer to this pattern of sharing a subcomponent between parent components of different types, whereby the filtering of events delivered to the parent components is done by event subtyping, as *static sharing*. Another pattern of sharing manifests when a subcomponent is shared

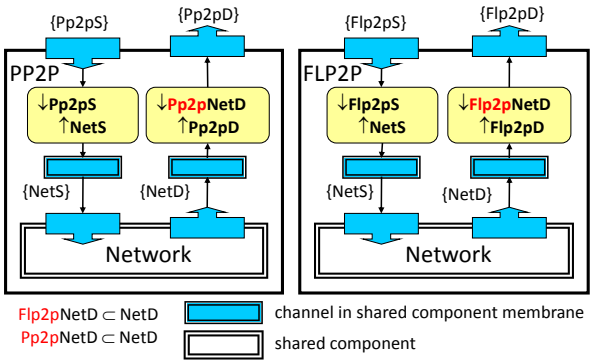


Figure 3. Two different composite components sharing a subcomponent.

among multiple parent components of the same type. In this case, the filtering of events is done by event attributes. The parent components subscribe for the output events of the shared component by event attributes with distinguishing values. We call this sharing pattern, *instance sharing*. An example of instance sharing is illustrated in Figure 4. Here we have multiple Virtual Node (VN) components running on the same physical node and sharing a Network component. Each VN component has subscribed for $NetDeliver$ events with a destination address attribute equal to its own address. Consequently, each VN component receives only the $NetDeliver$ events that it is interested in.

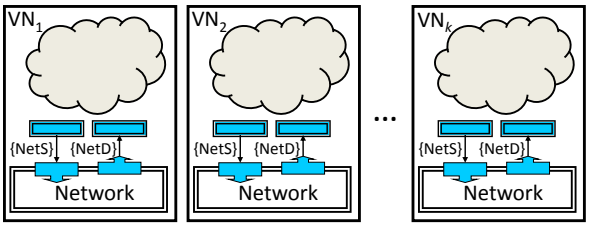


Figure 4. Multiple similar composite components sharing a subcomponent.

3. Component execution and interaction semantics

Event handlers are executed on behalf of components by worker threads from a *worker pool*. Components that have received events are scheduled for execution to one of the worker threads. Components that embed third-party software or use blocking services may use extra threads. We call these threaded components.

Concurrent component execution The event handlers of the same component instance are guaranteed to be executed sequentially, but different component instances can execute event handlers concurrently (or in parallel on multi-core machines). In other words, the event handlers of the same component instance are mutually exclusive, while the event handlers of different component instances are not. However, the execution of an event handler is not atomic (in the all-or-nothing sense). That means that events triggered by one event handler are visible to the corresponding subscriber components (thus, executable) immediately after they are triggered. This entails that the execution of an event handler is not failure atomic, i.e., it can fail before completion with observable partial side effects (some of the events that are supposed to be triggered by the handler are indeed triggered while others are not). The motivation for this design decision is twofold: first, making events available for execution as soon as they are triggered increases the potential parallelism in the system. Second, it provides a uniform observable behavior for threaded and non-threaded components.

Event subscription Components subscribe their event handlers to input channels for a particular event type. When component a subscribes to channel x an event handler h for events of type T with attributes matching $filters(T)$, a subscription of the form (a, T, h, $filters(T)$) is registered at channel x. At the same time, a FIFO *work queue*, q_x, is created at a and is associated with channel x (if q_x is not already existing from a previous subscription of a to x). Thus, a channel y has an associated work queue q_y in every component that has subscribed to y. A component can subscribe multiple event handlers to the same channel and can subscribe the same event handler to multiple channels.

Component scheduling A component a can be in exclusively one of the following three scheduling states: BUSY, READY, or IDLE. We say that a is BUSY if one of the worker threads is being actively executing one of a's event handlers. We say that a is READY if it is not BUSY and at least one of its work queues q_x is not empty, so a is ready to execute some event. We say that a is IDLE if it is not BUSY and all its work queues q_x are empty, so a has no event to execute.

Event publication While executing event handlers, components may publish events into output channels. Assume component a triggers event e of type T in channel x. Let S be the subset of all subscriptions of the form (b, T', h, $filters(T')$), to channel x, where T' is either T or a super-type of T and e matches $filters(T')$. For each subscription (b, T', h, $filters(T')$) in S, a work item of the form (e, h) is enqueued at b in work queue q_x and if b was IDLE then b becomes READY.

Channel FIFO guarantees The execution model guarantees the following FIFO semantics for channels. Each component a subscribed to a channel x, receives events published in x, in the same order in which they are published. Events triggered sequentially by one component instance will be published in the channel in the order in which they were triggered. A channel serializes the concurrent publication of events into the channel, i.e., events that are published concurrently in the same channel by different component instances. This means that all subscribers to channel x for event type T observe the same order of publications of events of type T in their local work queues q_x.

Event handler execution Worker threads execute event handlers on behalf of components. Worker threads atomically pick READY components and make them BUSY. When a worker picks READY component a, it immediately makes a BUSY. An invariant of the execution model is that at this point a has at least one work queue q_x that is not empty. After making a BUSY, the worker dequeues one work item (e, h) from some work queue q_x selected according to some fairness criteria. Thereafter, the worker proceeds to execute a's event handler h by passing it as an argument the event e. Upon completing the execution of h, if all a's work queues q_x are empty, then the worker makes a IDLE. Otherwise it makes a READY.

Worker threads loop Worker threads wait for components to become READY. When a component a becomes READY, a worker w picks it and executes one work item (e, h), the head of some work queue q_x of a. The execution of event handler h may trigger new events e_i of types T_i, published in channels x_i. All components subscribed to channels x_i for event types T_i become READY if they were not BUSY. Upon completing the execution of event handler h, worker w picks another READY component, if one exists, and it repeats the above steps. If no component is READY, worker w starts waiting for a component to become READY and it repeats the above steps.

4. Evaluation

We implemented Kompics in Java. The component framework library, source code and documentation is publicly available at `http://kompics.sics.se`. In this section we present two case studies of event-based hierarchical component architectures that we implemented with Kompics.

The example in Figure 5 shows composition, encapsulation and sharing of protocols in Kompics. For example the Abortable Consensus [6] component makes use of Best-Effort Broadcast and Perfect Links and the Consensus Instance uses the Leader Detector. The example also shows how different components can be used to separate the implementation of functional and non-functional aspects of an abstraction. The Consensus Instance implements a Paxos uniform consensus algorithm. The Consensus Port offers to an application component a sequence of consensus instances while garbage collecting the already decided instances. The Consensus Service component allocates Consensus Ports to different applications.

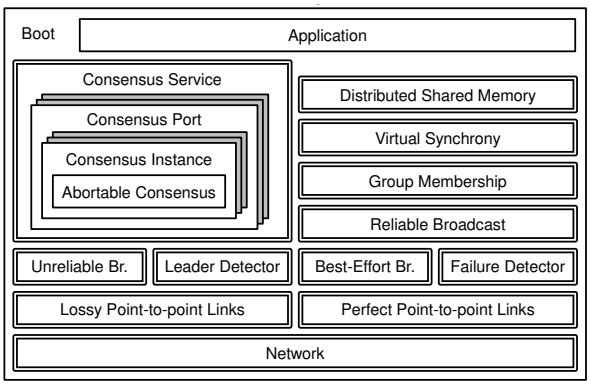


Figure 5. Static membership distributed abstractions system architecture.

In Figure 2 we showed a graphical representation of a Best-Effort Broadcast (BEB)[6] component. In Figure 6 we give the pseudo-code, from Guerraoui et al. [6], and our corresponding Java code for the $BebBroadcast$ event handler of the BEB component. One can observe that Kompics component definitions naturally match the pseudo-code specifications which become straight forward to implement. The Java code corresponds to a *static sharing* implementation, which assumes that the BEB component shares the PP2P component with other components and is itself shared by multiple parent components. Thus, BEB has subscribed for $BebPp2pDeliver$ (a subtype of $Pp2pDeliver$) events and both $Pp2pSend$ and $BebBroadcast$ events encapsulate $Pp2pDeliver$ and $BebDeliver$ events respectively. The pseudo-code assumes no sharing of protocols, hence the slight difference.

The example in Figure 7 shows a peer-to-peer system architecture that supports multiple virtual peers in one node. This is an example of hierarchical sharing where we want to share the Perfect Network abstraction among the protocols of one virtual peer, but have different Perfect Networks in different virtual peers. On the other hand, the Network component is shared and used by all Perfect Network abstractions in an instance sharing pattern. Every Peer component contains subcomponents implementing various protocols like failure detectors, structured and unstructured overlays, gossip protocols, group communication, agreement protocols, transactional storage, etc. The advantage of this architecture is that it allows the execution of the same protocol implementations both in a simulation scenario[1] or in a real deployment. We have executed up to 128 virtual peers implementing the Chord [10] overlay in one address space, where the Perfect Network and Lossy Network components in each peer implemented network emulation by delaying sent messages in the sender peers with latencies extracted from the King [7] matrix, based on the source and destination peer addresses. Exactly the same architecture can be executed in multiple address spaces to scale the network size. The Network component provides communication between the different address spaces.

upon event ⟨ bebBroadcast | m⟩ **do**
  for all $p_i \in \Pi$ **do trigger** ⟨ pp2pSend | p_i, m⟩; **end for**
end event

```
public void handleBebBroadcast(BebBroadcast event) {
   for (Address destination : neighbours.getAll()) {
       BebPp2pDeliver bebPp2pDeliver = new
           BebPp2pDeliver(event.getBebDeliver());
       Pp2pSend pp2pSend = new
           Pp2pSend(bebPp2pDeliver, destination);
       component.trigger(pp2pSend, pp2pSendChannel);
   }
}
```

Figure 6. Pseudo-code and Java code for an event handler of a BEB component.

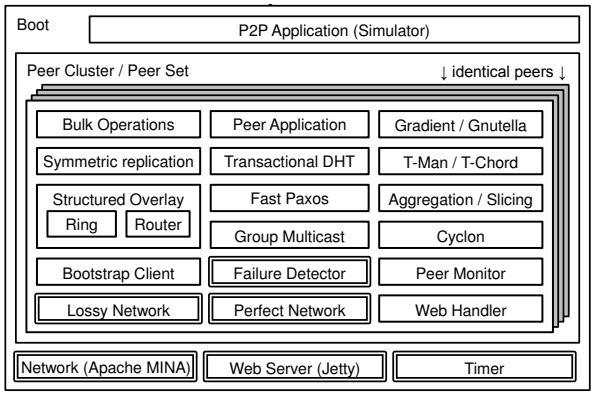


Figure 7. Kompics peer-to-peer system architecture.

[1] by replacing the Application component with a simulator for user actions and peer dynamism and replacing the Network component with a network simulator.

5. Related work

We can position our work with respect to component models, protocol composition frameworks and software architectures.

The Fractal [5] component model allows the specification of components that are reflective, hierarchical, and dynamically reconfigurable. However, the Fractal model is agnostic with respect to the execution model of components. Kompics is a reactive component model that has these desirable properties of Fractal but it enforces a particular execution and component interaction model, that facilitates programming of distributed protocols.

Protocol composition frameworks like Appia [8], Cactus [4] or Coyote [3] allow the building of *protocol stacks* by composing smaller building blocks called *protocol modules* that interact through events. However, these systems focus on the flow of events though the protocol stack rather than on the encapsulation and abstraction of low-level protocols. As a result the degree of protocol composability offered by these systems is limited[2] which prevents the construction of complex hierarchical architectures.

The most notable piece of related work on event-based software architectures is SEDA (staged event-driven architecture) [11]. Here the focus is on performance, namely on self-tuning resource management that adapts dynamically to changes in load to provide graceful degradation in performance, rather than on hierarchical architectures or dynamic reconfiguration. Kompics possesses these properties due to the ability to allocate different worker pools for different groups of components. Additionally, Kompics offers compositionality and encapsulation that are not present in SEDA.

6. Concluding remarks and future work

We presented Kompics, a component model and framework, that facilitates the construction of complex distributed systems by structuring protocols as reactive components and composing them in hierarchical architectures that are reconfigurable at runtime. We have shown the advantages of protocol composition with sharing in two real-world example component architectures. We reported on our experience of using Kompics as a teaching tool which showed that Kompics is very easy to use for prototyping and testing distributed protocols.

Using Kompics we have developed a library of reusable protocol components for building large-scale, decentralized, dynamic distributed systems. We continue to add more protocols to the library with the goal to provide a complete middleware for peer-to-peer systems including: structured and unstructured overlay networks, agreement protocols, scalable transactional storage, gossip protocols for aggregation, slicing, random peer sampling, topology maintenance, information dissemination, etc. Further, we plan to extend the Kompics model with distribution transparency, to work on remote component deployment and distributed architecture reconfiguration, followed by transactional reconfiguration.

We propose Kompics as a tool for teaching and research in distributed algorithms and as a framework for constructing and executing real-world complex distributed systems.

[2] to a set of stacks in Appia and to a 2-level hierarchy in Cactus.

References

[1] The Kompics component framework. `http://kompics.sics.se`, 2008.

[2] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Swedish Institute of Computer Science (SICS), 2003.

[3] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, 1998.

[4] N. T. Bhatti and R. D. Schlichting. A system for constructing configurable high-level protocols. In *SIGCOMM*, pages 138–150, 1995.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[6] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: estimating latency between arbitrary internet end hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, New York, NY, USA, 2002. ACM.

[8] A. Pinto. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 707, Washington, DC, USA, 2001. IEEE Computer Society.

[9] O. Rütti, P. T. Wojciechowski, and A. Schiper. Service interface: a new abstraction for implementing and composing protocols. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 691–696, New York, NY, USA, 2006. ACM.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[11] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

A.14 Kompics Programming Manual

Kompics Programming Manual

For Kompics version 0.4.1

Jim Dowling
Cosmin Arad

Table of Contents

1. Fundamental Concepts 1
    Kompics: Components, Events, Ports and Channels 1
2. Installing Kompics 4
    Install software required for Kompics 4
    Download and Install Subversion and Maven2 4
3. A Minimal Kompics Application 6
    Ping (Example 1) 6
        Ping Event 6
        PingPort Port 6
        Root Component 7
        Host Component 7
    Ping-Pong with 2 Ports (Example 2a) 8
        Pong Event 8
        PongPort 8
        Root Component 9
        Host Component 9
    Ping-Pong with a Single Port (Example 3a) 10
        PingPongPort 10
        Root Component 10
        Host Component 11
    Ping-Pong with a HostPing and a HostPong component (Example 4) 12

List of Figures

1.1. Example port. 3
1.2. Example component. 3
3.1. Ping component inside a Root component. 6
3.2. Ping-Pong component with two Ports inside a Root component. 8
3.3. Ping-Pong component with one Port inside a Root component. 10
3.4. A PingComponent and a PongComponent with two Ports each, inside a parent Root component. 12

List of Tables

1.1. Kompics Programming Abstractions 2
2.1. Install Kompics 5

Chapter 1. Fundamental Concepts

This chapter is a brief informal introduction to Kompics. We begin with a discussion of the main concepts in Kompics, then describe how to download and install Kompics, and finally cover the build environment, based on Maven2.

Kompics: Components, Events, Ports and Channels

A component is a reactive unit of computation that communicates with other components asynchronously by passing data-carrying typed events. Events are passive immutable objects that can be serialized and communicated over network links and between different address spaces.

Components are loosely coupled in the sense that a component does not know the component type, availability or identity of any components with which it communicates. Rather, components are endowed with typed communication ports. A component sends events to and receives events from a local port. Ports of the same type on two different components can be connected by a bidirectional channel. This connection enables the two components to communicate.

As a reaction to received events, components execute event handlers, procedures specific to the types of events being received. During execution, event handlers may trigger new events, by sending them to one of the component's ports. An event handler is associated with events of a certain type, received though a certain port, by means of a subscription.

Table 1.1. Kompics Programming Abstractions

Entity	Description
Component	A *component* is a reactive unit of computation that communicates with other components asynchronously by passing data-carrying typed events over ports. Components contain handlers to execute received events, and components can be composed into *composite components*.
Event	An *event* is a passive immutable object that can be serialized and communicated over network links and between different address spaces.
Port	A port represents a bidirectional event interface of a component and it specifies the types of events that flow into or out of the component. The direction in which an event flows through a port is defined as either *positive* or *negative*. A negative event type flows towards the negative side of the port, while a positive event type flows towards teh positive side of the port. A port is illustrated in Figure 1.1, “Example port.”. By convention, the positive pole (+) of a port is understood to be the provided direction for events, while the negative pole (#) of a port is understood to be the required direction for events. When a component implements (or provides) a port, the port is oriented with its + pole to the outside of the component and its # pole inside. Conversely, a (required) port that is used by a component is oriented with its # pole outside and its + pole inside. The types of events flowing through the port from the # pole to the + pole are tagged with + and the types of events flowing from the + pole to the # pole are tagged with #.
Event Handler	An event handler is a procedure that a component executes as a reaction to receiving a certain event.
Subscription	A subscription binds an event handler to a port pole.
Channel	A channel is a first-class bidirectional connection between two ports of the same type. A channel can connect two ports of the same type and of different polarity.

In Figure 1.1, “Example port.”, we can see two Ports containing 2 events and 3 events, respectively. For `PortType1`, `e1` both goes "out" and comes "in". For `PortType2`, `e2` goes "out", while `e2` and `e3` come "in".

Figure 1.1. Example port.

A component with two ports (one outgoing, one incoming), two handlers and two subscriptions is illustrated in Figure 1.2, “Example component.”. The ports in this component are from Figure 1.1, “Example port.”. `PortType1` is provided by the component and `PortType2` is required by the component. We can see how the `subscriptions` map events from `Ports` to `handlers`, while handlers can trigger (or send) an `event` to a port (if the polarity of that port allows that event to be sent in that direction).

So, we can see for `PortType1`, the event `E1` can be both sent and received over this port. For `PortType2`, we can see that a handler inside the component can send either `E2` or `E3` to the port and handler inside the component can subscribe for `E2` (but not `E3`). For handlers or components outside this component (handlers would have to be in a parent component), they can send and receive events of the opposite type. So, for `PortType2`, they could subscribe for `E2` or `E3` events, and send `E2` events to the component.

Figure 1.2. Example component.

INPUt (Implements Negative, Positive Uses)

An easy way to remember whether a PortType refers to the client-side or server-side is to remember the idiom *INPUt* (Implements Negative, Positive Uses). INPUt reminds you that a negative PortType is one that is provided or implemented by a component (server-side), while a positive PortType is one that is used by a component (client-side).

Summary

We introduced a number of concepts for Kompics, and outlined the software requirements for downloading and installing Kompics.

Chapter 2. Installing Kompics

This chapter describes the software requirements for Kompics version 0.4.1, and the steps required to install Kompics. These instructions cover the operating systems Windows (XP/Vista), Linux (all distros), Mac (OSX).

Install software required for Kompics

The minimal requirements for downloading and installing Kompics are *Java (JDK 5.0 update 6 and above)*, *subversion* and *maven2*. However, we recommend building Kompics from an Eclipse environment (Netbeans should also work fine, but isn't discussed here).

For tutorials on how to use subversion and maven, we refer you to:

- Subversion red book [http://svnbook.red-bean.com/]
- Maven2 book [http://books.sonatype.com/maven-book/reference/public-book.html]

Download and Install Subversion and Maven2

The requirements for installing Kompics are:*maven2* and *subversion*. We present two ways of installing subversion and maven: *using the command-line* and *as eclipse plugins*. The easiest way to build Kompics is to use Eclipse plugins. We outline the steps required for installing subversion and maven2 using either approach in the table below, see Table 2.1, “Install Kompics”. Please refer to the subversion and maven2 books above for additional help.

Table 2.1. Install Kompics

Command-Line	Eclipse

For Ubuntu:
 	
`$ sudo apt-get install subversion`
`$ sudo apt-get install maven2`
`$ svn checkout svn://small.sics.se\`
`/kompics/tags/kompics-manual-0.4.1`
`$ cd kompics-manual-0.4.1`
`$ mvn install`
 	
For Windows:
 	
• Go to http://maven.apache.org/ and download an install Maven2.
 	
• Download and install a subversion client, such as tortoise/svn [http://tortoisesvn.tigris.org/], and then checkout the Kompics code from our subversion server using the address:
 	
`svn://small.sics.se/kompics/tags\`
`/kompics-manual-0.4.1`
 	
. Once you have checked out the Kompics code, you can build Kompics using maven2, by running the following command from the Kompics source code root folder:
 	
`c:\...\>mvn install`
 	
• If you prefer Netbeans, you can generate a netbeans project using the following plugin Netbeans plugin for Maven [http://wiki.netbeans.org/MavenBestPractices].
	You will need to install the following Eclipse tools to build Kompics:
	
	• *Eclipse IDE* (Ganymede version is recommended). You now need to know how to install plugins for Eclipse. For the Ganymede version of Eclipse, you click on `Help->Software Updates->Available Software` . Then click the "Add site" box to add an update site for a plugin.
	
	• *subclipse* - an Eclipse Plugin for Subversion available at Eclipse update site `http://subclipse.tigris.org/update_1.4.x`. It is recommended that you check at least the following boxes for installation: "subclipse", "subclipse client adapter" and the "javahl adapter".
	
	• *m2eclipse* - an Eclipse Plugin for Maven2 available at Eclipse update site `http://m2eclipse.sonatype.org/update/`. It is recommended that you check at least the following boxes for installation: "maven embedder", "maven integration for eclipse", "Maven SCM handler", and the "Maven SCM handler for Subclipse".
	
	In Eclipse, import the maven project from the subversion repository: `File->Import->Other->Checkout Mvn Projects from SCM`.
	
	Then select 'svn' as SCM type, and enter as SCM URL: `svn://small.sics.se/kompics/tags/kompics-manual-0.4.1/`
	
	You should now restart Eclipse, and you will be able to import and build Kompics.

Summary

We specified the software requirements for Kompics and described how to download and build Kompics.

Chapter 3. A Minimal Kompics Application

This chapter shows you how to build a simple Kompics Ping-Pong application. The goal of this chapter is to familiarize you with the basic steps required to build a minimal Kompics application. We do not explain very many details of the source code here, as these details will be introduced later chapters.

See Chapter 1, *Fundamental Concepts* for a basic introduction to Kompics.

Ping (Example 1)

In this first example, see Figure 3.1, “Ping component inside a Root component.”, a `Root` component will send a *ping* message to a `Host` component. `Root` contains the `public void static main` , where a component that sends a `Ping` event to the `Host` component. `Host` has registered a handler, `handlePing`, with its `PingPort` PortType, so when the `Ping` event arrives at `PingPort` it is forwarded to `handlePing`. Finally, `handlePing` prints a message saying received the `Ping` event.

Note

It helps immensely to draw a diagram of your components and their ports along with the ports' polarity. In particular, a diagram will aid you in understanding the polarity of the port based on your context. For example, when you are a client of a Port you have a reference to the Port the opposite polarity.

Also, you should keep in mind the INPUt (Implements Negative, Positive Uses) (Implements Negative, Positive Uses).

Figure 3.1. Ping component inside a Root component.

Ping Event

```
package se.sics.kompics.manual.example1;
import se.sics.kompics.se.sics.kompics.Event;

public class Ping extends Event {
 public Ping()
 {
 }
}
```

PingPort Port

```
package se.sics.kompics.manual.example1;
```

```
import se.sics.kompics.se.sics.kompics.PortType;

public class PingPort extends PortType {

 {
  negative(Ping.class);
 }
}
```

Root Component

An alternative (and more common) way of starting this program is to write a `startHandler` for `Root`. When a `Root` component is constructed, its `startHandler` is automatically called.

```
package se.sics.kompics.manual.example1;
import se.sics.kompics.se.sics.kompics.Component;
import se.sics.kompics.se.sics.kompics.ComponentDefinition;
import se.sics.kompics.se.sics.kompics.Handler;
import se.sics.kompics.se.sics.kompics.Kompics;
import se.sics.kompics.se.sics.kompics.Start;

public class Root extends ComponentDefinition {

 public static void  main(String[] args)
 {
  Kompics.createAndStart(Root.class);
 }
 public Root() {
  subscribe(handleStart,control);
 }

 private Handler<Start> handleStart = new Handler<Start>() {
  public void handle(Start event) {
   Component hostComponent = create(Host.class);
   trigger(new Ping(), hostComponent.getPositive(PingPort.class));
  }
 };
}
```

Host Component

```
package se.sics.kompics.manual.example1;
import se.sics.kompics.se.sics.kompics.ComponentDefinition;
import se.sics.kompics.se.sics.kompics.Handler;
import se.sics.kompics.se.sics.kompics.Negative;

public class Host extends ComponentDefinition {

 Negative<PingPort> pingN = negative(PingPort.class);
```

```
public Host() {
 subscribe(handlePing, pingN);
}

private Handler<Ping> handlePing = new Handler<Ping>() {
 public void handle(Ping event) {
  System.out.println("Received ping..");
 }
};
}
```

Ping-Pong with 2 Ports (Example 2a)

In this example, a Root component will exchange *ping* and *pong* messages with a Host component. The `Root` component sends a `Ping` event to the `Host` component. `Host` has registered a handler, `handlePing`, with its `PingPort` PortType, so when the `Ping` event arrives at `PingPort` it is forwarded to `handlePing`. `handlePing` sends a `Pong` event to its PongPort, which is forwarded to handlePong in Root.

In the examples directory for this manual, you will find a reworking of this example (Example 2b), where we reverse the event directions for the pong port in `PongPortReversed`.

Figure 3.2. Ping-Pong component with two Ports inside a Root component.

Pong Event

```
package se.sics.kompics.manual.example2a;
import se.sics.kompics.se.sics.kompics.Event;

public class Pong extends Event {
 public Pong()
 {
 }
}
```

PongPort

```
package se.sics.kompics.manual.example2a;

import se.sics.kompics.se.sics.kompics.PortType;

public class PongPort extends PortType {

 {
  positive(Pong.class);
 }
```

```
}
```

Root Component

```
package se.sics.kompics.manual.example2a;

import se.sics.kompics.manual.example1.Ping;
import se.sics.kompics.manual.example1.PingPort;
import se.sics.kompics.se.sics.kompics.Component;
import se.sics.kompics.se.sics.kompics.ComponentDefinition;
import se.sics.kompics.se.sics.kompics.Handler;
import se.sics.kompics.se.sics.kompics.Kompics;
import se.sics.kompics.se.sics.kompics.Start;

public class Root extends ComponentDefinition {

 public static void  main(String[] args)
 {
  Kompics.createAndStart(Root.class);
 }
 public Root() {
  subscribe(handleStart,control);
 }

 private Handler<Start> handleStart = new Handler<Start>() {
  public void handle(Start event) {
   Component hostComponent = create(Host.class);
   subscribe(handlePong, hostComponent.getPositive(PongPort.class));
   trigger(new Ping(), hostComponent.getPositive(PingPort.class));
  }
 };

 private Handler<Pong> handlePong = new Handler<Pong>() {
  public void handle(Pong event) {
   System.out.println("Pong received.");
  }
 };
}
```

Host Component

```
  package se.sics.kompics.manual.example2a;

import se.sics.kompics.manual.example1.Ping;
import se.sics.kompics.manual.example1.PingPort;
import se.sics.kompics.se.sics.kompics.ComponentDefinition;
import se.sics.kompics.se.sics.kompics.Handler;
import se.sics.kompics.se.sics.kompics.Negative;
```

```
public class Host extends ComponentDefinition {

 Negative<PingPort> negPing = negative(PingPort.class);
 Negative<PongPort> negPong = negative(PongPort.class);

 public Host() {
  subscribe(handlePing, negPing);
 }

 private Handler<Ping> handlePing = new Handler<Ping>() {
  public void handle(Ping event) {
   System.out.println("Received ping, sending Pong..");
   trigger(new Pong(), negPong);
  }
 };
}
```

Ping-Pong with a Single Port (Example 3a)

We now refactor the section called “Ping-Pong with 2 Ports (Example 2a)” so that `Host` only has a single `PingPong Port`, instead of two ports. This example demonstrates the concept of "two-way event interfaces" (where events flow in and come out of a component).

In the examples directory for this manual, you will find a reworking of this example (Example 3b), where we reverse the event directions in a port called `PingPongPortReversed`.

Figure 3.3. Ping-Pong component with one Port inside a Root component.

PingPongPort

```
package se.sics.kompics.manual.example3a;

import se.sics.kompics.manual.example1.Ping;
import se.sics.kompics.manual.example2a.Pong;
import se.sics.kompics.se.sics.kompics.PortType;

public class PingPongPort extends PortType {

 {
  negative(Ping.class);
  positive(Pong.class);
 }
}
```

Root Component

```
package se.sics.kompics.manual.example3a;
```

```
import se.sics.kompics.manual.example1.Ping;
import se.sics.kompics.manual.example2a.Pong;
import se.sics.kompics.se.sics.kompics.Component;
import se.sics.kompics.se.sics.kompics.ComponentDefinition;
import se.sics.kompics.se.sics.kompics.Handler;
import se.sics.kompics.se.sics.kompics.Kompics;
import se.sics.kompics.se.sics.kompics.Start;

public class Root extends ComponentDefinition {

 private Component hostComponent;

 public static void  main(String[] args)
 {
  Kompics.createAndStart(Root.class);
 }
 public Root() {
  hostComponent = create(Host.class);
  subscribe(handleStart,control);
  subscribe(handlePong, hostComponent.getPositive(PingPongPort.class));
 }

 private Handler<Start> handleStart = new Handler<Start>() {
  public void handle(Start event) {
   trigger(new Ping(), hostComponent.getPositive(PingPongPort.class));  }
 };

 private Handler<Pong> handlePong = new Handler<Pong>() {
  public void handle(Pong event) {
   System.out.println("Pong received.");
  }
 };
}
```

Host Component

```
package se.sics.kompics.manual.example3a;

import se.sics.kompics.manual.example1.Ping;
import se.sics.kompics.manual.example2a.Pong;
import se.sics.kompics.se.sics.kompics.ComponentDefinition;
import se.sics.kompics.se.sics.kompics.Handler;
import se.sics.kompics.se.sics.kompics.Negative;

public class Host extends ComponentDefinition {

 Negative<PingPongPort> negPingPong = negative(PingPongPort.class);

 public Host() {
  subscribe(handlePing, negPingPong);
```

```
 }

 private Handler<Ping> handlePing = new Handler<Ping>() {
  public void handle(Ping event) {
   System.out.println("Received ping, sending Pong..");
   trigger(new Pong(), negPingPong);
  }
 };
}
```

Ping-Pong with a HostPing and a HostPong component (Example 4)

The diagram in Figure 3.4, “A PingComponent and a PongComponent with two Ports each, inside a parent Root component.” shows the same Ping-Pong example factored as two different components, `HostPing` and `HostPong`. The application starts by `Root` sending a start event to `HostPong`, which then sends a `Ping` event to `HostPing`, which then replies to `HostPong` with a `Pong` event.

In the code fragment below, we connect the *positive side* of `PingPort` on `pingHost` to the *negative side* of `PingPort` on `pongHost`, which returns a *Channel* object `x1`.

The code for this example can be found in the examples directory for this manual.

```
Positive<PingPort> pingPosPort = pingHost.getPositive(PingPort.class);
Negative<PingPort> pingNegPort = pongHost.getNegative(PingPort.class);
Channel<PingPort> x1 = connect(pingNegPort, pingPosPort);
```

Figure 3.4. A PingComponent and a PongComponent with two Ports each, inside a parent Root component.

A.15 A Design Methodology for Self-Management in Distributed Environments

A Design Methodology for Self-Management in Distributed Environments

Ahmad Al-Shishtawy*, Vladimir Vlassov*, Per Brand†, and Seif Haridi*†
*Royal Institute of Technology, Stockholm, Sweden
{ahmadas, vladv, haridi}@kth.se
†Swedish Institute of Computer Science, Stockholm, Sweden
{perbrand, seif}@sics.se

***Abstract*—Autonomic computing is a paradigm that aims at reducing administrative overhead by providing autonomic managers to make applications self-managing. In order to better deal with dynamic environments, for improved performance and scalability, we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. We present a methodology for designing the management part of a distributed self-managing application in a distributed manner. We define design steps, that includes partitioning of management functions and orchestration of multiple autonomic managers. We illustrate the proposed design methodology by applying it to design and development of a distributed storage service as a case study. The storage service prototype has been developed using the distributing component management system *Niche*. Distribution of autonomic managers allows distributing the management overhead and increased management performance due to concurrency and better locality.**

***Keywords*-autonomic computing; control loops; self-management; distributed systems;**

I. Introduction

Autonomic computing [1] is an attractive paradigm to tackle management overhead of complex applications by making them self-managing. Self-management, namely self-configuration, self-optimization, self-healing, and self-protection (self-* thereafter), is achieved through autonomic managers [2]. An autonomic manager continuously monitors hardware and/or software resources and acts accordingly. Managing applications in dynamic environments (like community Grids and peer-to-peer applications) is specially challenging due to high resource churn and lack of clear management responsibility.

A distributed application requires multiple autonomic managers rather than a single autonomic manager. Multiple managers are needed for scalability, robustness, and performance and also useful for reflecting separation of concerns. Engineering of self-managing distributed applications executed in a dynamic environment requires a methodology for building robust cooperative autonomic managers. The methodology should include methods for management decomposition, distribution, and orchestration. For example, management can be decomposed into a number of managers each responsible for a specific self-* property or alternatively application subsystems. These managers are not independent but need to cooperate and coordinate their actions in order to achieve overall management objectives.

The major contributions of the paper are as follows. We propose a methodology for designing the management part of a distributed self-managing application in a distributed manner, i.e. with multiple interactive autonomic managers. Decentralization of management and distribution of autonomic managers allows distributing the management overhead, increasing management performance due to concurrency and/or better locality. Decentralization does avoid a single point of failure however it does not necessarily improve robustness. We define design steps, that includes partitioning of management, assignment of management tasks to autonomic managers, and orchestration of multiple autonomic managers. We describe a set of patterns (paradigms) for manager interactions.

We illustrate the proposed design methodology including paradigms of manager interactions by applying it to design and development of a distributed storage service as a case study. The storage service prototype has been developed using the distributing component management system *Niche*[1] [3]–[5].

The remainder of this paper is organized as follows. Section II describes Niche and relate it to the autonomic computing architecture. Section III presents the steps for designing distributed self-managing applications. Section IV focuses on orchestrating multiple autonomic managers. In Section V we apply the proposed methodology to a distributed file storage as a case study. Related work is discussed in Section VI followed by conclusions and future work in Section VII.

II. The Distributed Component Management System

The autonomic computing reference architecture proposed by IBM [2] consists of the following five building blocks.

- **Touchpoint:** consists of a set of sensors and effectors used by autonomic managers to interact with managed resources (get status and perform operations). Touchpoints are components in the system that implement a uniform

This research is supported by the FP6 projects SELFMAN (contract IST-2006-034084) and Grid4All (contract IST-2006-034567) funded by the European Commission.

[1]In our previous work [3], [4] our distributing component management system *Niche* was called DCMS

management interface that hides the heterogeneity of managed resources. A managed resource must be exposed through touchpoints to be manageable.

- **Autonomic Manager:** is the key building block in the architecture. Autonomic managers are used to implement the self-management behaviour of the system. This is achieved through a control loop that consists of four main stages: monitor, analyze, plan, and execute. The control loop interacts with the managed resource through the exposed touchpoints.
- **Knowledge Source:** is used to share knowledge (e.g. architecture information and policies) between autonomic managers.
- **Enterprise Service Bus:** provides connectivity of components in the system.
- **Manager Interface:** provides an interface for administrators to interact with the system. This includes the ability to monitor/change the status of the system and to control autonomic managers through policies.

The use-case presented in this paper has been developed using the distributed component management system *Niche* [3], [4]. Niche implements the autonomic computing architecture described above. Niche includes a distributed component programming model, APIs, and a run-time system including deployment service. The main objective of Niche is to enable and to achieve self-management of component-based applications deployed on dynamic distributed environments such as community Grids. A self-managing application in Niche consists of functional and management parts. Functional components communicate via bindings, whereas management components communicate mostly via a publish/subscribe event notification mechanism.

The Niche run-time environment is a network of distributed containers hosting functional and management components. Niche uses a structured overlay network (Niche [4]) as the enterprise service bus. Niche is self-organising on its own and provides overlay services used by Niche such as name-based communication, distributed hash table (DHT) and a publish/subscribe mechanism for event dissemination. These services are used by Niche to provide higher level communication abstractions such as name-based bindings to support component mobility; dynamic component groups; one-to-any and one-to-all bindings, and event based communication.

For implementing the touchpoints, Niche leverages the introspection and dynamic reconfiguration features of the Fractal component model [6] in order to provide sensors and actuation API abstractions. Sensors are special components that can be attached to the application's functional components. There are also built-in sensors in Niche that sense changes in the environment such as resource failures, joins, and leaves, as well as modifications in application architecture such as creation of a group. The actuation API is used to modify the application's functional and management architecture by adding, removing and reconfiguring components, groups, bindings.

The Autonomic Manager (a control loop) in Niche is organized as a network of *Management Elements* (MEs) interacting through events, monitoring via sensors and acting using the actuation API. This enables the construction of distributed control loops. MEs are subdivided into watchers, aggregators, and managers. Watchers are used for monitoring via sensors and can be programmed to find symptoms to be reported to aggregators or directly to managers. Aggregators are used to aggregate and analyse symptoms and to issue change requests to managers. Managers do planning and execute change requests.

Knowledge in Niche is shared between MEs using two mechanisms: first, using the publish/subscribe mechanism provided by Niche; second, using the Niche DHT to store/retrieve information such as component group members, name-to-location mappings.

III. Steps in Designing Distributed Management

A self-managing application can be decomposed into three parts: the functional part, the touchpoints, and the management part. The design process starts by specifying the functional and management requirements for the functional and management parts, respectively. In the case of Niche, the functional part of the application is designed by defining interfaces, components, component groups, and bindings. The management part is designed based on management requirements, by defining autonomic managers (management elements) and the required touchpoints (sensors and effectors).

An Autonomic Manager is a control loop that senses and affects the functional part of the application. For many applications and environments it is desirable to decompose the autonomic manager into a number of cooperating autonomic managers each performing a specific management function or/and controlling a specific part of the application. Decomposition of management can be motivated by different reasons such as follows. It allows avoiding a single point of failure. It may be required to distribute the management overhead among participating resources. Self-managing a complex system may require more than one autonomic manager to simplify design by separation of concerns. Decomposition can also be used to enhance the management performance by running different management tasks concurrently and by placing the autonomic managers closer to the resources they manage.

We define the following iterative steps to be performed when designing and developing the management part of a self-managing distributed application in a distributed manner.

Decomposition: The first step is to divide the management into a number of management tasks. Decomposition can be either functional (e.g. tasks are defined based which self-* properties they implement) or spacial (e.g. tasks are defined based on the structure of the managed application). The major design issue to be considered at this step is granularity of tasks assuming that a task or a group of related tasks can be performed by a single manager.
Assignment: The tasks are then assigned to autonomic managers each of which becomes responsible for one or more management tasks. Assignment can

be done based on self-* properties that a task belongs to (according to the functional decomposition) or based on which part of the application that task is related to (according to the spatial decomposition).

Orchestration: Although autonomic managers can be designed independently, multiple autonomic managers, in the general case, are not independent since they manage the same system and there exist dependencies between management tasks. Therefore they need to interact and coordinate their actions in order to avoid conflicts and interference and to manage the system properly.

Mapping: The set of autonomic managers are then mapped to the resources, i.e. to nodes of the distributed environment. A major issue to be considered at this step is optimized placement of managers and possibly functional components on nodes in order to improve management performance.

In this paper our major focus is on the orchestration of autonomic managers as the most challenging and less studied problem. The actions and objectives of the other stages are more related to classical issues in distributed systems such as partitioning and separation of concerns, and optimal placement of modules in a distributed environment.

IV. Orchestrating Autonomic Managers

Autonomic managers can interact and coordinate their operation in the following four ways:

A. Stigmergy

Stigmergy is a way of indirect communication and coordination between agents [7]. Agents make changes in their environment, and these changes are sensed by other agents and cause them to do more actions. Stigmergy was first observed in social insects like ants. In our case agents are autonomic managers and the environment is the managed application.

The stigmergy effect is, in general, unavoidable when you have more than one autonomic manager and can cause undesired behaviour at runtime. Hidden stigmergy makes it challenging to design a self-managing system with multiple autonomic managers. However stigmergy can be part of the design and used as a way of orchestrating autonomic managers (Fig. 1).

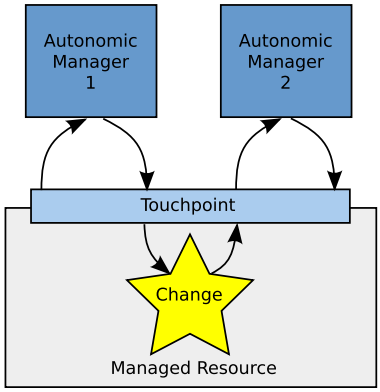


Fig. 1. The stigmergy effect.

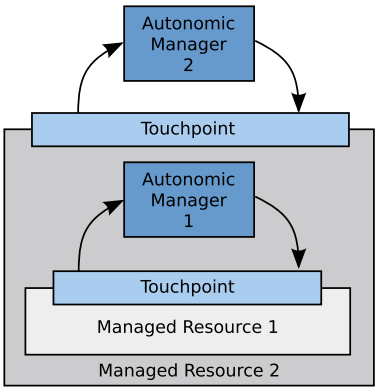


Fig. 2. Hierarchical management.

B. Hierarchical Management

By hierarchical management we mean that some autonomic managers can monitor and control other autonomic managers (Fig. 2). The lower level autonomic managers are considered as a managed resource for the higher level autonomic manager. Communication between levels take place using touchpoints. Higher level managers can sense and affect lower level managers.

Autonomic managers at different levels often operate at different time scales. Lower level autonomic managers are used to manage changes in the system that need immediate actions. Higher level autonomic managers are often slower and used to regulate and orchestrate the system by monitoring global properties and tuning lower level autonomic managers accordingly.

C. Direct Interaction

Autonomic managers may interact directly with one another. Technically this is achieved by binding the appropriate management elements (typically managers) in the autonomic managers together (Fig. 3). Cross autonomic manager bindings can be used to coordinate autonomic managers and avoid undesired behaviors such as race conditions or oscillations.

D. Shared Management Elements

Another way for autonomic managers to communicate and coordinate their actions is by sharing management elements (Fig. 4). This can be used to share state (knowledge) and to synchronise their actions.

V. Case Study: A Distributed Storage Service

In order to illustrate the design methodology, we have developed a storage service called YASS (Yet Another Storage Service) [3], using Niche. The case study illustrates how to design a self-managing distributed system monitored and controlled by multiple distributed autonomic managers.

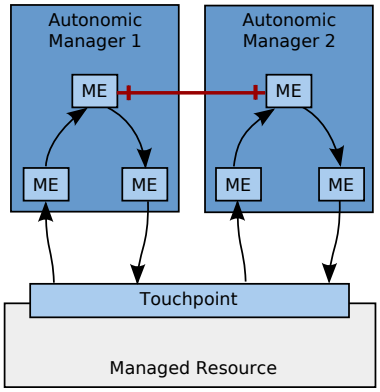


Fig. 3. Direct interaction.

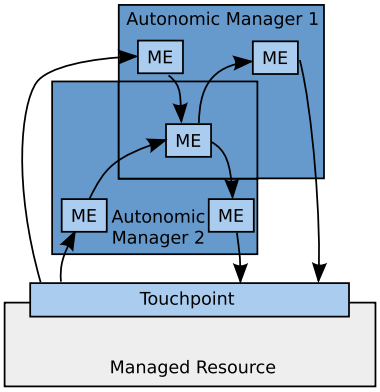


Fig. 4. Shared Management Elements.

A. YASS Specification

YASS is a storage service that allows users to store, read and delete files on a set of distributed resources. The service transparently replicates the stored files for robustness and scalability.

Assuming that YASS is to be deployed and provided in a dynamic distributed environment, the following management functions are required in order to make the storage service self-managing in the presence of dynamicity in resources and load: the service should tolerate the resource churn (joins/leaves/failures), optimize usage of resources, and resolve hot-spots. We define the following tasks based on the functional decomposition of management according to self-* properties (namely self-healing, self-configuration, and self-optimization) to be achieved.

- Maintain the file replication degree by restoring the files which were stored on a failed/leaving resource. This function provides the self-healing property of the service so that the service is available despite of the resource churn;
- Maintain the total storage space and total free space to meet QoS requirements by allocating additional resources when needed. This function provides self-configuration of the service;
- Increasing the availability of popular files. This and the next two functions are related to the self-optimization of the service.
- Release excess allocated storage when it is no longer needed.
- Balance the stored files among the allocated resources.

B. YASS Functional Design

A YASS instance consists of *front-end components* and *storage components* as shown in Fig. 5. The front-end component provides a user interface that is used to interact with the storage service. Storage components represent the storage capacity available at the resource on which they are deployed.

The storage components are grouped together in a storage group. A user issues commands (store, read, and delete) using the front-end. A store request is sent to an arbitrary storage component (using one-to-any binding between the front-end and the storage group) which in turn will find some r different storage components, where r is the file's replication degree, with enough free space to store a file replica. These replicas together will form a *file group* containing the r storage components that will host the file. The front-end will then use a one-to-all binding to the file group to transfer the file in parallel to the r replicas in the group. A read request is sent to any of the r storage components in the group using the one-to-any binding between the front-end and the file group. A delete request is sent to the file group in parallel using a one-to-all binding between the front-end and the file group.

C. Enabling Management of YASS

Given that the functional part of YASS has been developed, to manage it we need to provide touchpoints. Niche provides basic touchpoints for manipulating the system's architecture and resources, such as sensors of resource failures and component group creation; and effectors for deploying and binding components.

Beside the basic touchpoint the following additional, YASS specific, sensors and effectors are required.

- A load sensor to measure the current free space on a storage component;
- An access frequency sensor to detect popular files;
- A replicate file effector to add one extra replica of a specified file;
- A move file effector to move files for load balancing.

D. Self-Managing YASS

The following autonomic managers are needed to manage YASS in a dynamic environment. All four orchestration techniques in Section IV are demonstrated.

1) Replica Autonomic Manager: The replica autonomic manager is responsible for maintaining the desired replication degree for each stored file in spite of resources failing and leaving. This autonomic manager adds the self-healing property to YASS. The replica autonomic manager consists of two management elements, the File-Replica-Aggregator and the File-Replica-Manager as shown in Fig. 6.

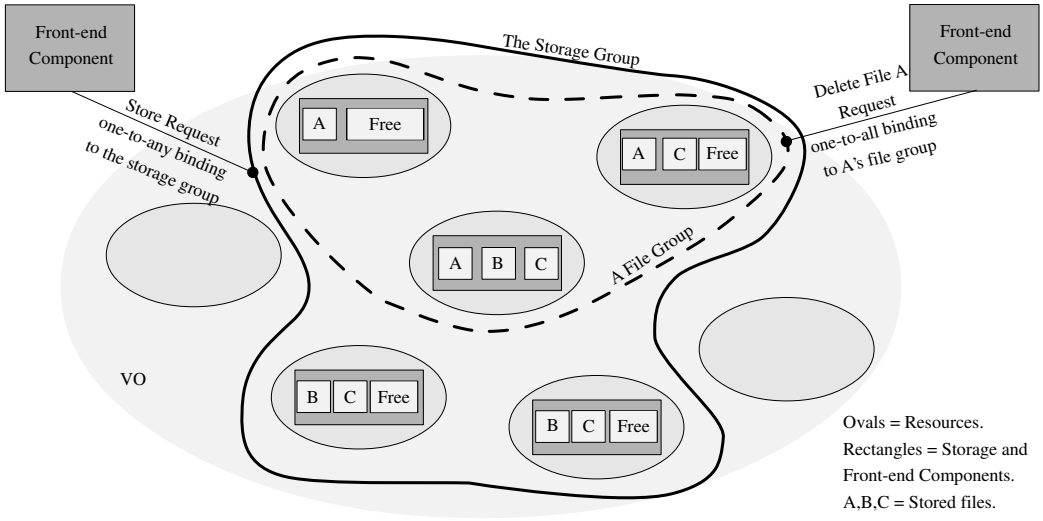


Fig. 5. YASS Functional Part

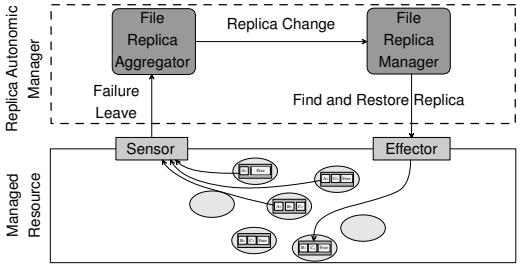


Fig. 6. Self-healing control loop.

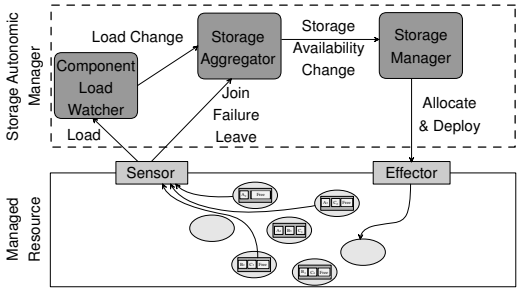


Fig. 7. Self-configuration control loop.

The File-Replica-Aggregator monitors a file group, containing the subset of storage components that host the file replicas, by subscribing to resource fail or leave events caused by any of the group members. These events are received when a resource, on which a component member in the group is deployed, is about to leave or has failed. The File-Replica-Aggregator responds to these events by triggering a replica change event to the File-Replica-Manager that will issue a find and restore replica command.

2) Storage Autonomic Manager: The storage autonomic manager is responsible for maintaining the total storage capacity and the total free space in the storage group, in the presence of dynamism, to meet QoS requirements. The dynamism is due either to resources failing/leaving (affecting both the total and free storage space) or file creation/addition/deletion (affecting the free storage space only). The storage autonomic manager reconfigures YASS to restore the total free space and/or the total storage capacity to meet the requirements. The reconfiguration is done by allocating free resources and deploying additional storage components on them. This autonomic manager adds the self-configuration property to YASS. The storage autonomic manager consists of Component-Load-Watcher, Storage-Aggregator, and Storage-Manager as shown in Fig. 7.

The Component-Load-Watcher monitors the storage group, containing all storage components, for changes in the total free space available by subscribing to the load sensors events. The Component-Load-Watcher will trigger a load change event when the load is changed by a predefined delta. The Storage-Aggregator is subscribed to the Component-Load-Watcher load change event and the resource fail, leave, and join events (note that the File-Replica-Aggregator also subscribes to the resource failure and leave events). The Storage-Aggregator, by analyzing these events, will be able to estimate the total storage capacity and the total free space. The Storage-Aggregator will trigger a storage availability change event when the total and/or free storage space drops below a predefined thresholds. The Storage-Manager responds to this event by trying to allocate more resources and deploying storage components on them.

3) Direct Interactions to Coordinate Autonomic Managers : The two autonomic managers, replica autonomic manager and storage autonomic manager, described above seem to be independent. The first manager restores files and the other manager restores storage. But as we will see in the following example it is possible to have a race condition between the two autonomic managers that will cause the replica autonomic manager to fail. For example, when a resource fails the

storage autonomic manager may detect that more storage is needed and start allocating resources and deploying storage components. Meanwhile the replica autonomic manager will be restoring the files that were on the failed resource. The replica autonomic manager might fail to restore the files due to space shortage if the storage autonomic manager is slower and does not have time to finish. This may also prevent the users, temporarily, from storing files.

If the replica autonomic manager would have waited for the storage autonomic manager to finish, it would not fail to recreate replicas. We used direct interaction to coordinate the two autonomic managers by binding the File-Replica-Manager to the Storage-Manager.

Before restoring files the File-Replica-Manager informs the Storage-Manager about the amount of storage it needs to restore files. The Storage-Manager checks available storage and informs the File-Replica-Manager that it can proceed if enough space is available or ask it to wait.

The direct coordination used here does not mean that one manager controls the other. For example if there is only one replica left of a file, the File-Replica-Manager may ignore the request to wait from the Storage-Manager and proceed with restoring the file anyway.

4) Optimising Allocated Storage : Systems should maintain high resource utilization. The storage autonomic manager allocates additional resources if needed to guarantee the ability to store files. However, users might delete files later causing the utilization of the storage space to drop. It is desirable that YASS be able to self-optimize itself by releasing excess resources to improve utilization.

It is possible to design an autonomic manager that will detect low resource utilization, move file replicas stored on a chosen lowly utilized resource, and finally release it. Since the functionality required by this autonomic manager is partially provided by the storage and replica autonomic managers we will try to augment them instead of adding a new autonomic manager, and use stigmergy to coordinate them.

It is easy to modify the storage autonomic manager to detect low storage utilization. The replica manager knows how to restore files. When the utilization of the storage components drops, the storage autonomic manager will detect it and will deallocate some resource. The deallocation of resources will trigger, through stigmergy, another action at the replica autonomic manager. The replica autonomic manager will receive the corresponding resource leave events and will move the files from the leaving resource to other resources.

We believe that this is better than adding another autonomic manager for following two reasons: first, it allows avoiding duplication of functionality; and second, it allows avoiding oscillation between allocation and releasing resources by keeping the decision about the proper amount of storage at one place.

5) Improving file availability: Popular files should have more replicas in order to increase their availability. A higher level availability autonomic manager can be used to achieve this through regulating the replica autonomic manager. The autonomic manager consists of two management elements. The

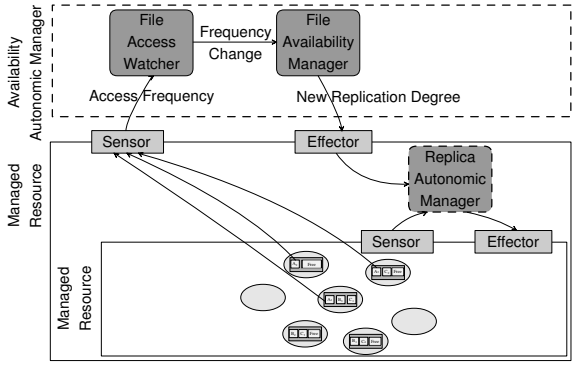


Fig. 8. Hierarchical management.

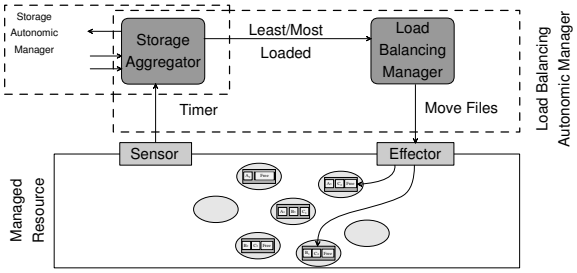


Fig. 9. Sharing of Management Elements.

File-Access-Watcher and File-Availability-Manager shown in Fig. 8 illustrate hierarchical management.

The File-Access-Watcher monitors the file access frequency. If the popularity of a file changes dramatically it issues a frequency change event. The File-Availability-Manager may decide to change the replication degree of that file. This is achieved by changing the value of the replication degree parameter in the File-Replica-Manager.

6) Balancing File Storage: A load balancing autonomic manager can be used for self-optimization by trying to lazily balance the stored files among storage components. Since knowledge of current load is available at the Storage-Aggregator, we design the load balancing autonomic manager by sharing the Storage-Aggregator as shown in Fig. 9.

All autonomic managers we discussed so far are reactive. They receive events and act upon them. Sometimes proactive managers might be also required, such as the one we are discussing. Proactive managers are implemented in Niche using a timer abstraction.

The load balancing autonomic manager is triggered, by a timer, every x time units. The timer event will be received by the shared Storage-Aggregator that will trigger an event containing the most and least loaded storage components. This event will be received by the Load-Balancing-Manager that will move some files from the most to the least loaded storage component.

VI. Related Work

The vision of autonomic management as presented in [1] has given rise to a number of proposed solutions to aspects of

the problem.

An attempt to analyze and understand how multiple interacting loops can manage a single system has been done in [8] by studying and analysing existing systems such as biological and software systems. By this study the authors try to understand the rules of a good control loop design. A study how to compose multiple loops and ensure that they are consistent and complementary is presented in [9]. The authors presented an architecture that supports such compositions.

A reference architecture for autonomic computing is presented in [10]. The authors present patterns for applying their proposed architecture to solve specific problems common to self-managing applications. Behavioural Skeletons is a technique presented in [11] that uses algorithmic skeletons to encapsulate general control loop features that can later be specialized to fit a specific application.

VII. Conclusions and Future Work

We have presented the methodology of developing the management part of a self-managing distributed application in distributed dynamic environment. We advocate for multiple managers rather than a single centralized manager that can induce a single point of failure and a potential performance bottleneck in a distributed environment. The proposed methodology includes four major design steps: decomposition, assignment, orchestration, and mapping (distribution). The management part is constructed as a number of cooperative autonomic managers each responsible either for a specific management function (according to functional decomposition of management) or for a part of the application (according to a spatial decomposition). We have defined and described different paradigms (patterns) of manager interactions, including indirect interaction by stigmergy, direct interaction, sharing of management elements, and manager hierarchy. In order to illustrate the design steps, we have developed and presented in this paper a self-managing distributed storage service with self-healing, self-configuration and self-optimizing properties provided by corresponding autonomic managers, developed using the distributed component management system Niche. We have shown how the autonomic managers can coordinate their actions, by the four described orchestration paradigms, in order to achieve the overall management objectives.

Dealing with failure of autonomic managers (as opposed to functional parts of the application) is out of the scope of this paper. Clearly, by itself, decentralization of management, might make the application more robust (as some aspects of management continue working, while others stop), but also more fragile due to increased risk of partial failure. In both the centralized and decentralized case, techniques for fault tolerance are needed to insure robustness. Many of these techniques, while ensuring fault recovery do so with some significant delay, in which case a decentralized management architecture may prove advantageous as only some aspects of management are disrupted at any one time.

Our future work includes refinement of the design methodology, further case studies with the focus on orchestration of autonomic managers, investigating robustness of managers by transparent replication of management elements.

Acknowledgements

We would like to thank the Niche research team including Konstantin Popov and Joel Höglund from SICS, and Nikos Parlavantzas from INRIA.

References

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] IBM, "An architectural blueprint for autonomic computing, 4th edition," http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf, June 2006.

[3] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing*, T. Priol and M. Vanneschi, Eds. Springer US, July 2008, pp. 163–174.

[4] P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy, "The role of overlay services in a self-managing framework for dynamic virtual organizations," in *Making Grids Work*, M. Danelutto, P. Fragopoulou, and V. Getov, Eds. Springer US, 2007, pp. 153–164.

[5] Niche homepage. [Online]. Available: http://niche.sics.se/

[6] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The fractal component model," France Telecom R&D and INRIA, Tech. Rep., Feb. 5 2004.

[7] E. Bonabeau, "Editor's introduction: Stigmergy," *Artificial Life*, vol. 5, no. 2, pp. 95–96, 1999. [Online]. Available: http://www.mitpressjournals.org/doi/abs/10.1162/106454699568692

[8] P. V. Roy, S. Haridi, A. Reinefeld, J.-B. Stefani, R. Yap, and T. Coupaye, "Self management for large-scale distributed systems: An overview of the selfman project," in *FMCO '07: Software Technologies Concertation on Formal Methods for Components and Objects*, Amsterdam, The Netherlands, Oct 2007.

[9] S.-W. Cheng, A.-C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste, "An architecture for coordinating multiple self-management systems," in *WICSA '04*, Washington, DC, USA, 2004, p. 243.

[10] J. W. Sweitzer and C. Draper, *Autonomic Computing: Concepts, Infrastructure, and Applications*. CRC Press, 2006, ch. 5: Architecture Overview for Autonomic Computing, pp. 71–98.

[11] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto, "Behavioural skeletons in gcm: Autonomic management of grid components," in *PDP'08*, Washington, DC, USA, 2008, pp. 54–63.

A.16 Using Global Information for Load Balancing in DHTs

Using Global Information for Load Balancing in DHTs*

Mikael Högqvist[1], Seif Haridi[2], Nico Kruber[1], Alexander Reinefeld[1], Thorsten Schütt[1]
[1]Zuse Institute Berlin
{hoegqvist, kruber, reinefeld, schuett}@zib.de
[2]Royal Institute of Technology (KTH)
seif@kth.se

Abstract

Distributed Hash Tables (DHT) with order-preserving hash functions require load balancing to ensure an even item-load over all nodes. While previous item-balancing algorithms only improve the load imbalance, we argue that due to the cost of moving items, the competing goal of minimizing the used network traffic must be addressed as well.

We aim to improve on existing algorithms by augmenting them with approximations of global knowledge, which can be distributed in a DHT with low cost using gossip mechanisms. In this paper we present initial simulation-based results from a decentralized balancing scheme extended with knowledge about the average node load. In addition, we discuss future work including a centralized auction-based algorithm that will be used as a benchmark.

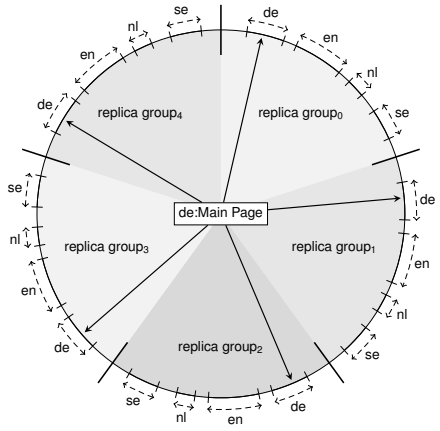


Figure 1. Geographic Load Balancing for Wikipedia.

1 Introduction

This work is motivated by research on self-managing distributed databases for use as a storage layer in large-scale Internet services. We envision that load balancing in such a system will not only consider node capacities, but can also be based on geographic location and application policies.

As an example, Wikipedia provides encyclopedias in different languages. Figure 1 shows how the Wikipedia articles and their respective replicas can be stored on a Distributed Hash Table (DHT) [12]. In such an application it is beneficial to host data nearby the users, i.e. in the geographic area where the language is used. We can use load balancing algorithms to implement location and application policies.

DHTs extend structured overlay networks (SON) with primitives for storing (key, value)-pairs and for retrieving the value associated with a key. Their functionality include support for both direct key lookups [13, 10] and range queries [11]. When a key is stored in a DHT with range queries, its location is decided using an order-preserving hash function. Depending on the distribution of the inserted keys the nodes in the DHT can quickly become unbalanced, which can lead to, for example, network congestion and unresponsive nodes.

Load balancing algorithms in DHTs focus on three different problems. First, when each item is hashed uniformly over the ID space, some nodes can have an $O(\log N)$ imbalance in terms of stored items [9, 6]. Second, when using order-preserving hash function [11], the items are mapped to the ID space such that they keep their original distribution. Therefore, for the system to be balanced, i.e. nodes storing an equal number of items, their node IDs must be distributed according to the key distribution [8, 4]. Third, independent of the item distribution, certain items can have much higher request rates than others. This is typically solved through caching, replication or by exploiting redundant network routes [3].

*This work was partly funded by the EU projects SELFMAN under grant IST-34084 and XtreemOS under grant IST-33576.

A common solution to the first problem is to maintain a set of virtual DHT nodes, or servers, at each physical machine. Virtual servers migrate between physical hosts to balance the system load [6, 13]. However, virtual servers have several issues such as increased churn when a physical host fails and increased state maintenance. In addition, in order-preserving DHTs, a single virtual server can still become overloaded when being responsible for a popular key range. In this paper, we are investigating solutions to the second problem, i.e. algorithms that are balancing the item-load at each node.

Since the network connecting the DHT nodes is the only shared resource, it is vital that DHT maintenance and tuning-algorithms use the network efficiently. This is especially the case for load balancing algorithms since their main operations trigger data movements. We aim to improve current algorithms by introducing approximations of global knowledge at each node, thereby helping them to take informed decisions in order to reduce data transfers. Examples of such information is the average node load or the network topology, which has already proved useful when balancing virtual servers [16]. Recent developments in gossiping for unstructured P2P networks and DHTs has shown that it is possible to obtain estimates of global properties with high confidence and low overhead [15, 14, 5].

In this paper, we argue for the benefits of introducing approximations of global knowledge to DHT load balancing algorithms with the goal of reducing the network utilization while maintaining a balanced system. To support this claim, we extend a well-known decentralized load balancing algorithm [8] to take the information about the average node load into account. The modified algorithm shows direct improvements on the overall items moved during balancing. We further argue for the use of centralized algorithms as a comparative benchmark.

2 Background

In this section we give an overview of current approaches for DHT load balancing with respect to virtual servers and item-balancing algorithms. This paper does not cover techniques for request balancing which is often solved through caching and/or replication.

Virtual Servers is a technique where each physical host maintains a set of virtual nodes. Load balancing is done by moving virtual servers, without changing their item range, from overloaded physical hosts to more lightly loaded hosts. The assignment of virtual nodes to physical hosts is typically performed by one or more directory nodes. A directory node periodically receives load information from random nodes in the system. When it has received load data from a sufficient amount of nodes it executes the load balancing algorithm [9, 6, 2]. An advantage of the virtual server scheme is that it does not require any changes to the DHT routing algorithm and allows for re-use of the join and leave overlay primitives.

An immediate issue with virtual servers is the increase of the routing table state maintained at each host. Godfrey et al. [7] introduce a scheme where a physical host maintains a set of virtual servers which have overlapping links in the routing table. With this placement restriction, a physical host only needs $\Theta(logN)$-links while hosting $\Theta(logN)$ virtual servers.

The above approaches use simple metrics for the cost of the load balancing operations, like the number of transferred items or bytes. A better cost metric could include the overall network utilization. In [16], Zhu et al. investigate how to minimize network usage by introducing proximity-aware load balancing algorithms for virtual servers. In [2], the assignment of virtual servers to physical hosts is modeled as an optimization problem which allows for an arbitrary cost function.

Another issue with virtual servers is that a physical host failure causes the hosted virtual nodes to fail as well. This increases the churn in the system and must be considered when selecting global parameters such as the replication factor.

Item-balancing Most of the research on load balancing in DHTs has focused on virtual servers. However, these approaches assume that items are uniformly distributed over the ID space using a hash function. In a DHT with an order-preserving hash function, a single virtual server can be overloaded if it is responsible for a popular key range. For example, when storing a dictionary, keys with the prefix "e" are more common than "w", resulting in the nodes storing items with prefix "e" being responsible for more items. The goal of item-balancing schemes is to adapt the location of the nodes in the system to correspond to the item distribution. This is performed using two operations, *jump* and *slide*. Jump transfers a node to an arbitrary ID in the system, while a slide operation only exchanges items with a node's direct neighbor.

Karger et al. [8] present a randomized item balancing scheme where each node contacts another random node periodically. If the load of the nodes differs by more than a factor $0 < \epsilon < \frac{1}{4}$, they share each others load by either jumping or sliding. Karger provides a theoretical analysis of the protocol, but does not evaluate the algorithms in an experimental or real-world setting. In addition, Karger's algorithm does not aim to minimize network traffic.

Ganesan et al. [4] use a reactive approach which triggers an algorithm when the node utilization super-cedes a threshold value. A node executing the algorithm first checks whether it should slide by comparing the load with

its neighbor's load. If this is not possible, it finds the least loaded node in the system and requests that it jumps to share the overloaded node's load. The least and most loaded node is located through a lookup to a separate DHT which stores all nodes sorted by their load.

We are basing our algorithms on the proactive approach presented by Karger, but aim to minimize the network utilization. This is achieved by making the algorithm aware of approximations of global parameters. While Ganesan stores the global knowledge in an additional DHT, we distribute this information through gossiping. This is more lightweight since it avoids the maintenance of another DHT and combines the strength of both structured and unstructured networks.

3 System model and problem definition

A DHT consist of N nodes, where each node has an ID in the range $[0, 1)$. This range wraps around at 1,0 and can be seen as a ring. A node, n_i has a *successor*-pointer to the next node in clockwise direction, n_{i+1}, and a *predecessor*-pointer to the first counter-clockwise node, n_{i-1}. The node with the largest ID has the node with the lowest ID as successor. Thus, the nodes and their pointers create a double linked list where the first and last node are linked.

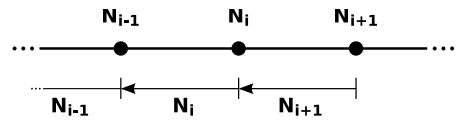


Figure 2. A node N_i with successor and predecessor and their responsibilities.

Each node in the DHT stores a subset of items, $I(n_i)$, where each item has a key in the range $[0, 1)$ and a uniform weight of one. A node n_i is *responsible* for a key iff it falls within the node's key range $(n_{i-1}, n_i]$. Each node has a load $l(n_i)$ indicating the number of stored data items. Figure 2 shows three nodes and their respective responsibilities.

A node is balanced when it is neither *underloaded* nor *overloaded* relative to any other node in the system times a factor ϵ [8]. That is, when $l(n_i) < \epsilon l(n_j)$, $l(n_j)$ is overloaded compared to n_i and n_i is underloaded compared to n_j. The goal of the load-balancing algorithm is to make all nodes balanced. ϵ is a system-defined parameter with values between 0 and $\frac{1}{4}$.

In order to change the load of nodes in the system, two types of operations are used: jump and slide.

Jump allows a node to move to an arbitrary position in the ID space. A jumping node n_i first leaves its current position and re-joins at its new location, ID_k, with n_j as its successor. Data is moved two times. First, the items in the range $(n_{i-1}, n_i]$ are transferred to n_{i+1}. Second, when n_i joins at ID_k, all data in the range $(n_{j-1}, \text{ID}_k]$ is transferred from n_j to n_i.

Slide is a specialized form of jumping where n_i moves to an ID in the range (n_i, n_j), assuming that the overloaded node n_j is n_{i+1}. Since a node does not need to leave and re-join the system, which results in extra data transfer, sliding is always preferred over jumping.

We define a load-balanced configuration as a system state where all nodes are balanced. The maximum load in a configuration, C_i is denoted by $l_{max}(C_i)$, while the minimum load is $l_{min}(C_i)$, respectively. We use the standard deviation of a configuration, $\sigma(C_i)$, as a measure to indicate its imbalance. A jump or slide changes a configuration C_i to a new configuration C_{i+1}. For a jump or a slide operation performed by any node the algorithm must meet the following properties for the load to converge towards a balanced configuration.

$$l_{max}(C_i) \geq l_{max}(C_{i+1}) \quad (1)$$

$$l_{min}(C_i) \leq l_{min}(C_{i+1}) \quad (2)$$

$$\sigma(C_i) > \sigma(C_{i+1}) \quad (3)$$

Following these properties, an algorithm will reach a balanced configuration after a finite number of iterations.

Problem definitions. The load balancing problem can be summarized as follows: given a configuration C_0 with a set of nodes $\mathbf{N}$ and items $\mathbf{I}$, where each item $i \in \mathbf{I}$ is assigned to a responsible node, find a configuration C_b that only contains balanced nodes using the operations jump and slide. A solution to the load balancing problem is a sequence of operations transforming C_0 to C_b.

In addition to the load balancing problem, we search for a solution that minimizes the data movement cost of the transition from C_0 to C_b. That is, given a set of solutions, $\mathbf{S}$, find a solution S_i with minimal cost. The cost-function is $cost(op)$, where op is either a $slide$ or $jump$ operation. The cost-function can be chosen arbitrarily, but is typically based on the number of bytes moved or the network utilization.

4 Decentralized Algorithms

Unlike a centralized algorithm, a decentralized algorithm can only use the information locally available at each node. We modify Karger's randomized item-balancing algorithm to work with different globally approximated parameters. In this paper we use the system's average load.

Global information is, by definition, not available in peer-to-peer systems, unless aggregation algorithms are employed. However, by using gossiping techniques such as Vicinity and Cyclon [15, 14] or DHT gossip [5] it is possible to get a good approximation locally at each node of a parameter's value with low network traffic overhead.

Karger's Algorithm. In order to reach a load-balanced configuration, we rely on the heuristics introduced for Karger's item balancing algorithm [8]. Expressed in our notation, a load-balance operation is only performed between any pair of nodes n_i, n_j, iff $l(n_i) < \epsilon l(n_j)$ or $l(n_j) < \epsilon l(n_i)$, $0 < \epsilon < \frac{1}{4}$. When these restrictions are satisfied, the following cases are possible (assuming $l(n_i) > l(n_j)$).

Case 1, $i = j + 1$ n_i is the successor of n_j. Slide n_j towards n_i, letting n_j take responsibility for $\frac{l(n_i)-l(n_j)}{2}$ of n_i's items.

Case 2, $i \neq j + 1$ If $l(n_{j+1}) > l(n_i)$, set $i = j + 1$ and go to case 1. That is, when the load of n_j's successor is larger then the load of n_i, slide n_j towards the overloaded node n_{j+1}. Otherwise, n_j jumps to a position in the range (n_{i-1}, n_i), taking half of $l(n_i)$.

Modified Karger. Karger's randomized algorithm is based on two decisions. (1) Which nodes should balance? (2) Should they use jump or slide? The new location of a node performing a jump or slide is calculated such that the load is shared evenly by the two participating nodes. We want to show that global information can be used to reduce the number of transferred items, which indirectly impacts the network usage. Therefore, we introduce a heuristics based on the average load. Our modification is a restriction on the position a node takes after an operation. Instead of as in Karger, sharing the load evenly, we ensure that an underloaded node never takes more than the average load, L_{avg}. This effectively limits the amount of unnecessary data item transfers.

The described strategy has the biggest advantage when a node is overloaded relative to another node and it's load is much larger than the average load. Figure 3 shows a scenario where a node N_{i+1} has a load much larger than the average load, i.e. $l(N_{i+1}) > 3L_{avg}$. Let two nodes, A and B, execute the load balancing algorithm in that order. In Karger, assuming that A balances with N_{i+1}, it would first take more than $1.5L_{avg}$ load. If B then chooses to balance with node A, which is possible if A is still overloaded in relation to B. Then the data in A's range is transferred twice, first from N_{i+1} to A and then from A to B. With our modified version, since node A would take at most L_{avg} load from N_{i+1}, the probability that B balances with A is lower as well as the transferred data items if A decides to balance with B.

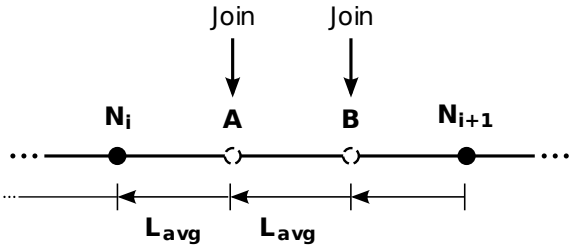


Figure 3. Two consecutive slots being filled by joining nodes, taking at most L_{avg} load.

5 Evaluation

We simulated Karger's algorithm and a version with knowledge of the systems average load. The effect of knowing the average load can be seen in Figures 4 and 5.

Experiments. The experiments are performed in a discrete time simulator where a single operation represents a step in time. The system contains 100 nodes, and the items are distributed such that the first 90 nodes have one item and the remaining ten nodes have 10000, 20000, ..., 90000 items, respectively. We measure the number of moved items as operation cost and the standard deviation is used to indicate the load imbalance of the system.

Figure 4 shows the sum of moved items for the operations necessary to go from the initial configuration to a load balanced configuration. Increasing ϵ values, between $0 < \epsilon < 0.25$ as suggested by Karger, shows a linear increase in the balance cost. Interestingly, the comparison between Karger and the modified Karger shows that in many cases the latter moves half as many items to reach a balanced configuration.

In Figure 5 we set $\epsilon = 0.21$ and study how each operation influences the load imbalance. The x-axis represents the aggregated number of moved items for each round and the y-axis is the standard deviation for the current configuration. The simulation is continued until no further balance operations can be performed. Our main conclusion from this experiment is that the modified Karger decreases the load imbalance of the system faster, even though it moves less items than the basic Karger.

6 Outlook

In this section we discuss the implications of load balancing algorithms for other DHT services. We also outline

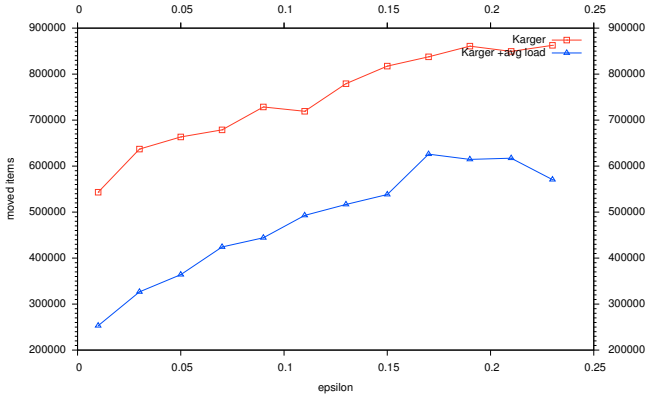


Figure 4. Number of moved items with epsilon between 0 and 0.25.

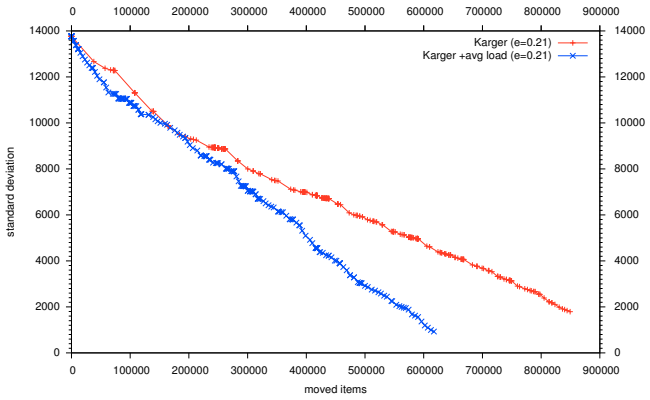


Figure 5. Load imbalance as a function of the number of moved items.

our approach to a centralized algorithm which we plan to used as a comparative benchmark for the decentralized algorithms.

Additional Global Estimates. In the evaluation section, we showed that using the average load can have an impact on the load balancing performance. In addition to the average load, we are interested in evaluating the following parameters.

Standard Deviation In section 3 we used the standard deviation as an invariant for the progress of an algorithm. If each node has knowledge of this value they could try to minimize it for each balancing operation they perform.

Location Proximity-information allows a node which will transfer load to select a target node which minimizes the network utilization [16].

Over- and Underloaded nodes A list of the k most overloaded and most underloaded nodes. These lists can be used for example in the Karger-algorithm to improve the convergence rate.

Implications of Load Balancing on a DHT. As argued throughout this paper, load balancing is an important part in an efficient and self-tuning DHT. However, the load balancing algorithms must work seamlessly together with other components in a DHT-based storage layer such as replication and transactions.

The jump and slide primitives are using the basic join and leave operations from the overlay. Since these operations are triggered by the load balancing algorithms, the balancing itself incurs extra churn in the system. Therefore, it is important that, for example, the systems replication factor is chosen with this in mind. Tuning the load balancing to work at a rate acceptable for the system is an important trade-off that needs to be evaluated for a working system.

A Centralized Auction-based Algorithm. In a centralized algorithm the global state of the system is known. A centralized algorithm can be used as a reference benchmark for decentralized algorithms. We aim to base our centralized algorithm on an auction algorithm [1] where overloaded and underloaded are matched to find an optimal assignment.

An auction algorithm finds an optimal one-to-one assignment of persons to objects in polynomial time. The assignment depends on the cost of the object and the benefit of the person being assigned to the object. For the load balancing problem this is analogous to finding a lowest cost match between underloaded and overloaded nodes.

More formally, each person i has a benefit a_{ij} of selecting an object j with price p_j. The net value for a person i of choosing object j is $a_{ij} - p_j$. The goal of the auction is to find an assignment where every persons find an object which maximizes the total net value. Thus, an auction is finished when the equilibrium $a_{ij} - p_j = \max_{j \in Objects}(a_{ij} - p_j)$ is reached.

Each iteration of the algorithm consists of a bidding phase followed by an assignment phase. During the bidding phase, each person finds an object resulting in maximum net value after which it computes a bidding increment. The value of the bidding increment is used after the assignment phase to increase the price of the object. In the assignment phase the persons with the highest bids are assigned to the respective objects. When all people are assigned to an object the algorithm terminates. This also means that the equilibrium has been satisfied.

An advantage of the auction algorithm is that the benefit function and the object price can be chosen arbitrarily. This allows us to explore more complicated costs than e.g. moved data items. Furthermore, the order of the load balancing operations slide and join are affecting the total price of the load balancing process. Due to the apparent advantages in computational complexity of the auction algorithm approach, we are actively investigating an appropriate cost-function which can include proximity information and the order of operations.

7 Conclusion

We showed that it is possible to reduce the cost of load balancing by introducing simple heuristics and knowledge about basic global parameters. We plan to continue this work by evaluating the effects of more properties such as the network topology. In addition, a centralized algorithm can give the optimal cost for balancing a given configuration. This can be used as a reference to evaluating the performance of the decentralized algorithms.

References

[1] D. P. Bertsekas. *Network Optimization: Continuous and Discrete Models (Optimization, Computation, and Control)*. Athena Scientific, 1998.

[2] C. Chen and K.-C. Tsai. The server reassignment problem for load balancing in structured p2p systems. *IEEE Trans. Parallel Distrib. Syst.*, 19(2):234–246, 2008.

[3] A. Datta, R. Schmidt, and K. Aberer. Query-load balancing in structured overlays. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*, 2007.

[4] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, pages 444–455. Morgan Kaufmann, 2004.

[5] A. Ghodsi, S. Haridi, and H. Weatherspoon. Exploiting the synergy between gossiping and structured overlays. *Operating Systems Review*, 41(5):61–66, 2007.

[6] B. Godfrey, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM*, 2004.

[7] B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. In *INFOCOM*, pages 596–606. IEEE, 2005.

[8] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *IPTPS*, volume 3279 of *Lecture Notes in Computer Science*, pages 131–140. Springer, 2004.

[9] A. Rao, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica. Load balancing in structured p2p systems. In *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2003.

[10] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[11] T. Schütt, F. Schintke, and A. Reinefeld. Structured overlay without consistent hashing: Empirical results. In *CCGRID*. IEEE Computer Society, 2006.

[12] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. In *ACM SIGPLAN Erlang Workshop*, 2008.

[13] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[14] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. Network Syst. Manage.*, 13(2), 2005.

[15] S. Voulgaris, M. van Steen, and K. Iwanicki. Proactive gossip-based management of semantic overlay networks. *Concurrency and Computation: Practice and Experience*, 19(17):2299–2311, 2007.

[16] Y. Zhu and Y. Hu. Efficient, proximity-aware load balancing for dht-based p2p systems. *IEEE Trans. Parallel Distrib. Syst.*, 16(4):349–361, 2005.

A.17 Node Placement in a Distributed Key/Value-store

Workload and Key Distribution Aware Node Placement in a Distributed Key/Value-store

Mikael Högqvist and Nico Kruber

Zuse Institute Berlin
hoegqvist@zib.de kruber@zib.de

Abstract. Distributed key/value stores are a basic building block for large-scale Internet services. Support for range queries introduces new challenges to load balancing since both the key and workload distribution can be non-uniform.
We build on previous work based on the power of choice to present algorithms suitable for active and passive load balancing that adapt to both the key and workload distribution. The algorithms are evaluated in a simulated environment, focusing on the impact of load balancing on scalability under normal conditions and in an overloaded system.

1 Introduction

Distributed key/value stores [1–3] are used in applications which require high throughput, low latency and have a simple data model. Examples of such applications are caching layers and indirection services. Federated key/value-stores, where the nodes are user contributed, require minimal management overhead for the participants. Furthermore, the system must able to deal with large numbers of nodes which are often unreliable and have varying network bandwidth and storage capacities. In addition, centralized key/value-stores such as Berkeley DB and Tokyo Cabinet provide both exact-match and range queries. We argue that to provide increased flexibility for applications, their distributed counterparts should do as well.

Ring-based Structured Overlay Networks (SONs) provide algorithms for node membership (join/leave/fail) and to find the node responsible for a key within $O(\log N)$ steps, where N is the number of nodes. One of the main advantages of SONs for large-scale services is that each node has to maintain state of a small number of other nodes, typically $O(\log N)$. SONs also define the partitioning strategy for stored keys since a node is responsible for a distinct range of keys in the ring.

At first glance SONs may therefore seem to be a good fit for distributed key/value stores. However, the tight coupling between the overlay and data layers in combination with the dynamic nature of user-donated nodes make the design of the data storage layer especially challenging in terms of reliability and load balancing.

The goal of load balancing is to improve the fairness in terms of storage as well as network and CPU-time usage between the nodes. Imbalance mainly

occurs due to: 1) non-uniform key distribution, 2) skewed access frequency of keys and 3) node heterogeneity. First, by supporting range-queries, an order-preserving hash function is used to map keys to the overlay's identifier space. With a non-uniform key distribution a node can become responsible for an unfair amount of items. Second, keys are typically accessed with different popularity which creates uneven workload on the nodes. The third issue, node capacity differences, also impacts the imbalance. For example, a low capacity node gets overloaded faster than a high capacity node.

Our main contribution is a load balancing algorithm which is aware of both the key distribution and the item load, i.e. used storage and access-frequency. The algorithm has two modes: *active*, which triggers a node already part of the overlay to balance with other nodes and *passive*, which places a joining node at a position that reduces the overall system imbalance. In both the passive and active mode, a set of nodes are sampled and the algorithm balance using the node with the highest load.

Section 2 contains the model, assumptions and definitions that are used for the load balancing algorithm presented in Section 3. In Section 4, we evaluate the system using a simulated environment. Results from the simulation show that the algorithm improves the load imbalance within a factor 6-12 in a system with 1000 nodes. In addition, we also show that load balancing reduces the storage capacity overhead necessary in an overloaded system from a factor 10 to 8.

2 System Model

A DHT consists of N nodes and an identifier space in the range $[0, 1)$. This range wraps around at 1.0 and can be seen as a ring. A node, n_i, at position i has an identifier n_i^{ID} in the ID space. Each node n_i has a *successor*-pointer to the next node in clockwise direction, n_{i+1}, and a *predecessor*-pointer to the first counter-clockwise node, n_{i-1}. The last node, n_{N-1}, has the first node, n_0 as successor. Thus, the nodes and their pointers create a double linked list where the first and last node are linked. We define the distance between two identifiers as $d(x, y) = |y - x| \bmod 1.0$.

Nodes can fail and join the system at any time. When a node joins, it takes over the range from its own ID to the predecessor of its successor. Similarly, when a node n_i fails, its predecessor becomes predecessor of n_i's successor.

Storage When a key/value-pair or item is inserted in the system it is assigned an ID using an order-preserving hash-function in the same range as the node IDs, i.e. $[0, 1)$. Each node in the system stores the subset of items that falls within its responsibility range. That is, a node n_i is *responsible* for a key iff it falls within the node's key range $(n_{i-1}^{ID}, n_i^{ID}]$.

Each item is replicated with a replication factor f. The replicas are assigned replica keys according to symmetric replication where the identifier of an item replica is derived from the key and the replica factor using the formula $r(k, i) = k + (i - 1) * \frac{1}{f} \bmod N$, k is the item ID and i is the ith replica [4]. An advantage

of symmetric replication is that the replica keys are based on the item key which makes it possible to perform the look-ups in parallel instead of requiring a look-up to the responsible node first like in successor-list replication [5].

A replica maintenance protocol ensures that a node stores the items and the respective replicas it is responsible for. The protocol consist of two phases; the synchronization phase and the data transfer phase. In the synchronization phase, a node determines which items should be stored at the node and if they are not stored, which replicas need to be retrieved. The retrieval is performed during the data transfer phase by issuing a read for each item using the transaction layer. If an item is busy in a concurrent write operation it is ignored and retrieved during the next round of maintenance. While it is rather expensive to get all items in a range it is only performed when a node joins or when its responsibility changes.

Load and Capacity Each node has a workload and a storage capacity. The workload can be defined arbitrarily, but for a key/value-store this is typically the request rate. Each stored item has a workload and a storage cost. The workload of a single node is the sum of the workload of the stored items. Note that a node cannot store more items than its storage capacity allows. The workload, on the other hand, is limited by for example bandwidth, and a node can decide if a request should be ignored or not. We model the probability of a request failure as $P(fail) = 1 - \frac{1}{\mu}$, where μ is the current node utilization, i.e. the measured workload divided by the workload capacity.

Imbalance We define the system imbalance of a load attribute (storage or workload) as the ratio between the highest loaded node and the system average. For example, for the storage, the imbalance is calculated as $\frac{L_{max}}{L_{avg}}$, if L_{max} is the maximum number of items stored by a node and L_{avg} is the average number of items per node.

3 Load Balancing algorithm

The only way to change the imbalance in our model is to change the responsibility of the nodes. A node's responsibility changes either when another node joins between itself and its predecessor, or when the predecessor fails. Thus, we can balance the system either *actively* by triggering a node to fail and re-join or *passively* by placing a new node at an overloaded node when joining. We first present the load balancing algorithm followed by the placement function taking both key distribution and workload into account.

The passive and active balancing algorithms presented in Figure 1 use only local knowledge and can be divided into three parts. 1) sample a set of k random nodes to balance with, 2) decide the placement of a potential new predecessor and 3) select one of the k-nodes that improves the imbalance the most. We assume that there is a join function which is used to join the overlay given an ID. `passive` is called before a node is joining and `active` is called periodically. `active` is inspired by Karger's [6] balancing algorithm, but we only consider the

```
1  def placement():
2      balanced_ID = ⊥
3      current_dist = ∞
4      for x in (n_{i-1}^{ID}, n_i^{ID}]:
5          dist = f(x)
6          if dist < current_dist:
7              balanced_ID = x + d(x, next(x))/2
8              current_dist = dist
9
10     return balanced_ID
11
12 def sample():
13     samples = [(n.load(), n)
14                for n in random_nodes(k)]
15     return max(samples)
16
17 def passive():
18     (n_load, n) = sample()
19     join(n)
20
21 def active():
22     (n_load, n) = sample()
23     if n_load > local_load * ϵ:
24         leave()
25         join(n.placement())
```

Fig. 1. Passive and Active load balancing

case where the node has a factor ϵ less load than the remote node. The ϵ is used to avoid oscillations by creating a relative load range where nodes do not trigger a re-join. `sample` calls a function `random_nodes` that uses a random walk or generates random IDs to find a set of k nodes. The node with the highest load is returned.

Placement Function

The goal of the placement function is to find the ID in a node's responsibility range that splits the range in two equal halves considering both workload and key distribution. When defining the cost for a single load attribute, it is optimal to always divide the attribute in half [7]. We use this principle for each attribute by calculating the ratio between the range to the left of the identifier x and the remaining range up to the node's ID. The optimal position is where this ratio approaches 1. A ratio therefore increases slowly from 0 towards 1 until the optimal value of x is reached, and after 1 the value approaches the total cost for the attribute.

First, let $l_r(a,b) = \sum_{i=0}^{items \in (a,b]} l(item_i)$ be a function returning the load of the items in the range $(a,b]$. $l(item_i)$ is the load of a single item and is defined arbitrarily depending on the load attribute. Second, let n_i be the node at which we want to find the best ID, then the ratio function is defined as follows

$$r(x) = \frac{l_r(n_{i-1}^{ID}, x)}{l_r(x, n_i^{ID})}$$

The workload ratio, $r_w(x)$, could for example be defined using $l(item_i) = weight(item_i) + (rate_{access}(item_i) \times weight(item_i))$. The weight is the total bytes of the item and the access rate is estimated with an exponentially weighted moving mean. For the key distribution ratio, $r_{ks}(x)$, the load function is $l(item_i) = 1$. This means that $r_{ks}(x) = 1$ for the median element in n_i's responsibility range. An interesting aspect of the ratio definitions is that they can be weighted in order to ignore fast changing or load attributes taking on extreme values.

In order to construct a placement function acknowledging different load attributes, we calculate the product of their respective ratio function. The point x where this product is closest to 1 is where all attributes are being balanced equally. Note that when it equals 1, it means that the load attributes have their optimal point at the same ID.

The placement function we use here considers both the key-space and workload distribution and is more formally described as

$$f(x) = |1 - r_w(x) \times r_{ks}(x)|$$

where x is the ID and n_j is the joining node. The ratio product value is subtracted from 1 and the absolute value of this is used since we are interested in the ratio product value "closest" to 1. Finally, when the smallest value of $f(x)$ is found, a node is placed at the ID between the item, $item_i$ preceding x and the subsequent item, $item_{i+1}$. That is, the resulting ID is $item_i^{ID} + d(item_i^{ID}, item_{i+1}^{ID})/2$.

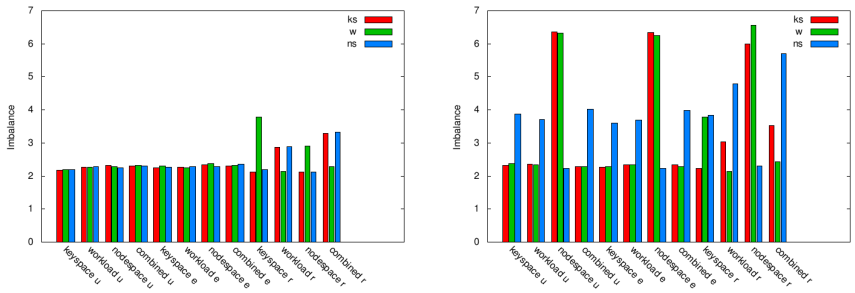
4 Evaluation

This section present simulation result of the passive and active algorithms. The goal of this section is to 1) show the effects of different access-load and key distributions, 2) show the scalability of the balancing strategies when increasing the system size and 3) determine the impact of imbalance in a system close to its capacity limits. Table 1 summarizes the parameters used for the different experiments.

Effect of Workloads In this experiment, we quantify the effect that different access-loads and key distributions have on the system imbalance. The results from this experiment motivate the use of a multi-attribute placement function. Specifically, we measure the imbalance of the nodespace (ns), keyspace (ks) and the access workload (w).

	Nodes	Items	Replicas	k	MTTF	Storage Capacity	Item Size
Effect of Workloads	256	8192	3	7	∞	∞	1
Network costs	256	8192	3	7	1h	∞	1-1MB
Size of k	256	8192	3	0-20	1h	∞	1
System size	64-1024	2^{10}-2^{18}	3	7	1h	∞	1
Churn	256	8192	3	7	1h-1d	∞	1
Overload	256	8192	3	7	1h	$128 * 7$-$1024 * 7$	1

Table 1. Parameters of the different experiments



(a) Uniform key distribution

(b) Dictionary key distribution

Fig. 2. The effect of different access workloads and key distributions.

The simulation is running an active balancing algorithm with $\epsilon = 0.15$, no churn and 7 nodes are sampled for each balance operation. The system has 256 nodes and 2^{14} items. Four different placement functions are used: 1) *nodespace* places a new node in the middle between the node and its predecessor, i.e. $n_i + \frac{d(n_{i-1},n_i)}{2}$. 2) *keyspace* places the node according to the median item, $f(x) = |1 - r_{ks}(x)|$. 3) *workload* halves the load of the node, i.e $f(x) = |1 - r_w(x)|$ and 4) combined uses the placement function defined in section 3.

Workload is generated using three scenarios; uniform (u), exponential (e) and range (r). In the uniform and exponential cases, the items receive a load from either a uniform or exponential distribution at simulation start-up. The range workload is generated by assigning successive ranges of items with random loads taken from an exponential distribution.

From the results shown in Figure 2, we can see that the imbalance when using the different placement strategies are very dependent on the load type. Figure 2(a) clearly shows that a uniform hash-function is efficient to balance all three metrics under both uniform and exponential workload. In the latter case, this is because the items are assigned the load independently. However, for the range workload, the imbalances are showing much higher variation depending on the placement function. We conclude that in a system supporting range

queries, the placement function should consider several balancing attributes for fair resource usage.

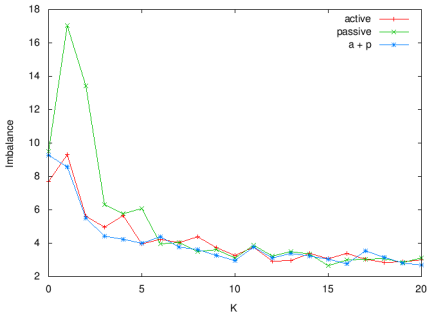


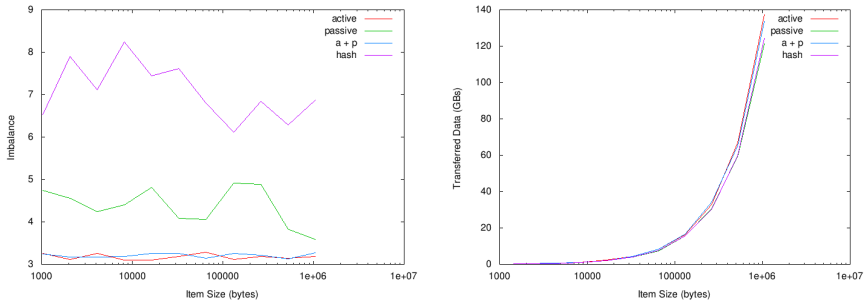
Fig. 3. Imbalance when increasing the number of sampled nodes.

Size of *k* In this experiment, we try to find a reasonable value of the number of nodes to sample, k. A larger k implies more messages, but also reduces the imbalance more. The results in figure 3 imply that the value of k is important for smaller values of between 2-10. However, the balance improvement becomes smaller and smaller for each increase of k, similar to the law of diminishing returns. In the remaining experiments we use $k = 7$.

Network costs We define cost as the total amount of data transferred in the system up to a given iteration. This cost is increased by the item size each time an item is transferred. Since there is no application traffic in the simulation environment, the cost is only coming from replica maintenance. That is, item transfers are used to ensure that replicas are stored according to the current node responsibilities. Active load balancing creates traffic when a node decides to leave and re-join the system.

We measure the keyspace imbalance and the transfer cost at the end of the simulation, which is run for 86400s (1 day). Each simulation has 8192 items with 7 replicas and the size of the items is increased from 2^{10} to 2^{20}. The item size has minor impact on the imbalance (Fig. 4(a)). Interestingly, the overhead when using the hash-based balancing strategy as a reference, of active+passive (a+p in the figure) and active is 5-15% (Fig. 4(b)). The passive strategy does not show a significant difference. Noteworthy is also that in a system storing around 56 GB of total data (including replicas), over 1 TB is transferred. This can be explained with the rather short node lifetime of 3600s.

Churn A node joining and leaving (churn) changes the range of responsibility for a node in the system. Increasing the rate of churn influences the cost of

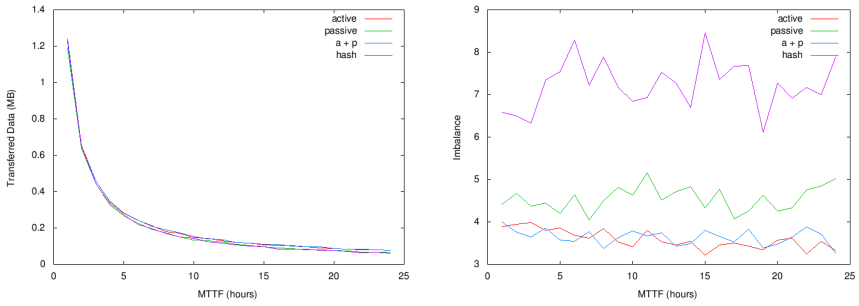


(a) Imbalance vs. Item size (b) Transferred bytes

Fig. 4. Imbalance and cost of balancing for increasing item size.

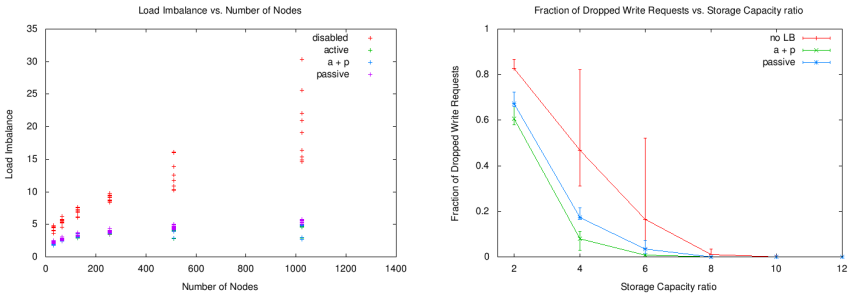
replica maintenance since item repairs are triggered more frequently. In this experiment, we quantify the impact of churn on transferred item cost and the storage imbalance.

In figure 5(a) the MTTF is varied from 1 to 24 hours. As expected the amount of data transferred is decreasing when the MTTF is increasing. Also as noted in the network costs experiment, the different schemes for load balancing have a minor impact on the total amount of transferred data. The overall data transfer costs are still significant compared to the amount of stored data (8192b * 7 replicas). With an MTTF of 1 day, the system still transfers more data than is being stored. This is no surprise since for each node leaving the system, $\frac{\sum items}{N}$ items are transferred on average. Similarly, when a node joins, $\frac{\sum items}{2*N}$ items are transferred on average. Figure 5(b) shows that churn has in principle no impact on the imbalance for the different strategies. This is also the case for the passive approach which only relies on churn to balance the system.



(a) Bytes transferred with increasing MTTF (b) Imbalance with varying MTTF

Fig. 5. Imbalance and network cost for varying levels of churn (MTTF).



(a) Increasing nodes and items (b) Capacity

Fig. 6. Imbalance of the system using different balancing strategies while increasing the system size. The right figure shows the influence of load balancing in an overloaded system.

System size The imbalance in a system with hash-based balancing was shown theoretically to be bounded by $O(\log N)$, where N is the number of nodes in the system [8]. However, this assumes that both the nodes and the keys are assigned IDs from a hash-function. In this experiment, we try to determine the efficiency of the placement function with an increasing number of nodes and items.

We measure the keyspace imbalance for an increasing number of nodes between 2^5 and 2^{10}. In addition, for each system size we run an experiments with an increasing number of items from 2^{10} to 2^{19}. Keys are generated from a dictionary and nodes are balanced using the combined placement function. Each node has a Mean Time to Failure (MTTF) of 1 hour drawn from an exponential distribution and each item is replicated three times. Four different balancing strategies are compared; 1) IDs generated by a uniform hash-function 2) active without any passive placement, 3) passive without any active and 4) active and passive together (a+p). For the last three, 7 nodes are sampled when selecting which node to join at or wether to balance at all.

Figure 6(a) shows that the hash-based approach performs significantly worse with an imbalance up to 6-12 times higher compared to the other balancing strategies. Interestingly, the difference in load imbalance when varying the number of items is also growing with larger system sizes, being around a factor of 2 with 1024 nodes. All three of the balancing strategies show similar performance. The imbalance grows slowly with increasing system size and the difference for different number of items is small. Thus, we draw the conclusion that these strategies are only minimally influenced by system size and number of items. However, note that we need to perform further experiments varying other parameters such as k to validate these results.

Overload In a perfectly balanced system where at most one consecutive node can fail, nodes can use at most up to 50% of their capacity to avoid becoming overloaded when a predecessor fails. This type of overload leads to dropped write

requests when there is insufficient storage capacity and dropped read request with insufficent bandwidth and processing capacity. Since a replica cannot be recreated when a write is dropped, this influences the data reliability. The goal of this experiment is to better understand the storage capacity overhead to avoid dropped writes.

We start the experiment such that the sum of the item weights equals the aggregated storage capacity of all nodes. Then by increasing the node's storage capacity we decrease their fill-ratio and thereby the probability of a dropped write. The system is under churn and lost replicas are re-created using a replica maintenance algorithm executed periodically at each node. The y-axis in Figure 6(b) shows the fraction of dropped write requests and the x-axis shows the storage capacity ratio. We don't add any data to the system which means that a write request is dropped when a replica cannot be created at the responsible node because of insufficient storage capacity. We measured the difference with hash-based balancing vs. the active and active-passive with 7 random nodes and the combined placement function.

Figure 6(b) shows that a system must have at least 10x the storage capacity over the total storage load to avoid dropped write requests when using hash-based balancing. Active and active-passive delays the effect of overload and a system with at least 8x storage capacity exhibits a low fraction of dropped requests.

5 Related Work

Karger et al. [6] and Ganesan et al. [9] both present active algorithms aiming at reducing the imbalance of item load. Karger uses a randomized sampling-based algorithm which balances when the relative load value between two nodes differs by more than a factor ϵ. Ganesan's algorithm triggers a balancing operation when a node's utilization exceeds (falls below) a certain threshold. In that case, balancing is either done with one of its neighbors or the least (most) loaded node found. Aspnes at al. [10] describe an active algorithm that categorizes nodes as closed or open depending on a threshold and groups them in a way so that each closed node has at least one open neighbor. They balance load when an item is to be inserted into a closed node that cannot shed some of its load to an open neighbor without making it closed as well. A rather different approach has been proposed by Charpentier et al. [11] who use mobile agents to gather an estimate of the system's average load and to balance load among the nodes. Those algorithms however do not explicitly define a placement function or use a simple "split loads in half" approach which doesn't take several load attributes into account.

Byers et. al. [12] proposed to store an item at the k least loaded nodes out of d possible. Similarly, Pitoura et al. [13] replicate an item to k of d possible identifiers when a node storing an item becomes overloaded (in terms of requests). This technique, called the "power of two choices" was picked up by Ledlie et. al [14] who apply it to node IDs an use it to address workload skew, churn and het-

erogeneous nodes. With their algorithm, k-Choices, they introduce the concept of passive and active balancing. However, their focus is on virtual server-based systems without range-queries. Giakkoupis and Hadzilacos [15] employ this technique to create a passive load balancing algorithm including a weighted version for heterogeneous nodes. There, joining nodes contact a logarithmic (in system size) number of nodes and choose the best position to join at. Their focus on the other hand is on balancing the address-space partition rather than arbitrary loads. Manku [16] proposes a similar algorithm issuing one random probe and contacting a logarithmic number of its neighbors. An analysis of such algorithms using r random probes each followed by a local probe of size v is given by Kenthapadi and Manku [17]. However, only the nodespace partitioning is examined.

In Mercury [18] each node maintains an approximation of a function describing the load distribution through sampling. This works well for simple distributions, but as was shown in [19] it does not work for more complex cases such as file-names. Instead, [19] introduces OSCAR where the long-range pointers are placed by recursively halving the traversed peer population in each step. Both OSCAR and Mercury balance the in/out-degree of nodes. While this implies that the routing load in the overlay is balanced, it does not account for the placement of nodes according to item characteristics.

6 Conclusions

With the goal of investigating load balancing algorithms for distributed key/value-stores, we presented an active and a passive algorithm. The active algorithm is triggered periodically, while the passive algorithm uses joining nodes to improve system imbalance. We complement these algorithms with a placement function that splits a node's responsibility range according to the current key and workload distribution. Initial simulation results are promising showing that the system works well under churn and scales with increasing system sizes. Ongoing work include quantifying the cost of the algorithms and their implementation within a prototype key/value-store.

Acknowledgments This work is partially funded by the European Commission through the SELFMAN project with contract number 034084.

References

1. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP, ACM (2007) 205–220
2. Rhea, S.C., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: Opendht: a public dht service and its uses. In Guérin, R., Govindan, R., Minshall, G., eds.: SIGCOMM, ACM (2005) 73–84
3. Reinefeld, A., Schintke, F., Schütt, T., Haridi, S.: Transactional data store for future internet services. Towards the Future Internet - A European Research Perspective (2009)

4. Ghodsi, A., Alima, L.O., Haridi, S.: Symmetric replication for structured peer-to-peer systems. In: DBISP2P. (2005) 74–85
5. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM. (2001) 149–160
6. Karger, D.R., Ruhl, M.: Simple efficient load balancing algorithms for peer-to-peer systems. In: IPTPS. Volume 3279 of Lecture Notes in Computer Science., Springer (2004) 131–140
7. Wang, X., Loguinov, D.: Load-balancing performance of consistent hashing: asymptotic analysis of random node join. IEEE/ACM Trans. Netw. **15**(4) (2007) 892–905
8. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: ACM Symposium on Theory of Computing. (May 1997) 654–663
9. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: VLDB, Morgan Kaufmann (2004) 444–455
10. Aspnes, J., Kirsch, J., Krishnamurthy, A.: Load balancing and locality in range-queriable data structures. In: PODC. (2004) 115–124
11. Charpentier, M., Padiou, G., Quéinnec, P.: Cooperative mobile agents to gather global information. In: NCA, IEEE Computer Society (2005) 271–274
12. Byers, J.W., Considine, J., Mitzenmacher, M.: Simple load balancing for distributed hash tables. In: IPTPS. Volume 2735 of Lecture Notes in Computer Science., Springer (2003) 80–87
13. Pitoura, T., Ntarmos, N., Triantafillou, P.: Replication, load balancing and efficient range query processing in dhts. In Ioannidis, Y.E., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C., eds.: EDBT. Volume 3896 of Lecture Notes in Computer Science., Springer (2006) 131–148
14. Ledlie, J., Seltzer, M.I.: Distributed, secure load balancing with skew, heterogeneity and churn. In: INFOCOM, IEEE (2005) 1419–1430
15. Giakkoupis, G., Hadzilacos, V.: A scheme for load balancing in heterogenous distributed hash tables. In Aguilera, M.K., Aspnes, J., eds.: PODC, ACM (2005) 302–311
16. Manku, G.S.: Balanced binary trees for id management and load balance in distributed hash tables. In: PODC. (2004) 197–205
17. Kenthapadi, K., Manku, G.S.: Decentralized algorithms using both local and random probes for p2p load balancing. In Gibbons, P.B., Spirakis, P.G., eds.: SPAA, ACM (2005) 135–144
18. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM, ACM (2004) 353–366
19. Girdzijauskas, S., Datta, A., Aberer, K.: Oscar: Small-world overlay for realistic key distributions. In: DBISP2P. (2006) 247–258

A.18 Security issues in small world network routing

Security Issues in Small World Network Routing

Felix Halim, Yongzheng Wu, Roland H.C. Yap
School of Computing, *National University of Singapore*
Law Link, Singapore
{halim,wuyongzh,ryap}@comp.nus.edu.sg

Abstract

Small World Network (SWN) have been shown to be navigable — a short route can be found using efficiently using decentralized algorithms. This routing relies on nodes having a position to guide the routing such as its coordinates. Even in the absence of positional information such as node coordinates, by using local self-reorganization, it is possible to reconstruct a proxy for the node coordinates which still allows for efficient routing. This paper shows that in the presence of malicious nodes, the self-reorganization mechanism breaks down. We investigate self-protection mechanisms for such SWNs. Preliminary results using a simple restart mechanism for self-tuning shows that much of the effect of malicious nodes can be mitigated.

1. Introduction

Structured Overlay Networks such as Distributed Hash Tables (DHT) provide a self organization network layer which can be used for communication and storage. There has been extensive work on DHTs such as Chord, Pastry, CAN, Kademlia, DKS, etc [1]. However, DHTs have to deal with the maintenance costs of dealing with dynaminism (churn) and be attacked in a variety of ways. Perhaps the worst problem is that in P2P settings, it is difficult to defend against Sybil attacks [3].

Small World Networks (SWN) presents an interesting alternative to self organizing networks. SWN are networks characterized by "small world phenomena". Perhaps, the best example is the idea of "six degrees of separation", that any two persons can be linked through a chain of acquaintances whose path length is at most six. Two important properties of SWN are navigability, being able to route or find information, and low diameter (which accounts for the six in six degrees). The success of social networking websites such as Facebook, LinkedIn, MySpace, etc. are possibly driven also by such small world properties.

One question is given a SWN, how to efficiently navigate the graph, i.e. how to efficient route a message between any two nodes in the graph. Kleinberg [2] proposes a SWN model which is inspired by the Watts-Strogatz model [5]. Starting with a graph which is a lattice, e.g. a 2-D grid, add for each node, a constant number of shortcut edges to other edges with probability proportional to $d(u,v)^{-r}$ where $d()$ is the distance function giving the Manhattan distance between nodes in the lattice. Note that the distribution of shortcut nodes follows a power law distribution. Kleinberg shows that greedy routing, thus using only local operations, is efficient with an expected route length of $O(log^2 n)$ hops where n is the number of nodes in the graph.

In a P2P setting, efficient decentralized routing algorithms using only local information is attractive. However, the assumption in the Kleinberg model that nodes have knowledge of their own location (e.g. lattice coordinates) is rather strong. However nodes or peers may not know their position information which can be thought of as global rather than local information. This would suggest routing difficulties even with a SWN and a low graph diameter.

Sandberg [4] proposed a way around this difficulty. Nodes have a position but this may be incorrect, so a continuous self-tuning approach is used to correct the position. Each node performs a fixed length random walk. The nodes at both ends of the random walk can decide to switch their positions to maximize the maximum likelihood that the node positions follow the edge distribution in the Kleinberg model. This can be achieved by minimizing the product of edge distances. The self tuning employed is somewhat analogous to self stabilization in DHTs but it can be thought of as reordering nodes around the graph.

In this paper, we show that routing and self-tuning strategy used by Sandberg breaks down when there are malicious nodes which can infect nodes with invalid positions. We discuss some attacks by malicious nodes and investigate self-protection strategies. We investigate a simple self-healing strategy where nodes reset themselves with a certain probability. Preliminary results show that without self-protection, a few malicious nodes will be able to infect the whole network. Self-healing protection using decentralized

local reset of node position is able to reduces the percentage of nodes infected. While this is not a complete solution, it means that the effect of malicious nodes can be mitigated to some extent.

2. Malicious Nodes and Self-Protection

The self-tuning in Sandberg assumes all nodes are good. If there are malicious nodes in the network, they can attack self-tuning by lying about their neighbors positions. This results in a malicious node being able to disseminate false position information which in turn affects the self-tuning and routing algorithms. Malicious nodes can also collude which makes it harder to detect such cheating.

After switching position with a good node, the malicious node has the good node's position and can reset it to a particular position X. After a while, there will be many good nodes having position X. However, duplicate positions can be detected once the nodes contact each other. One way to recover from duplicate position is to reset to a random position once duplicate is detected.

Another attack is to reset to a position close to X, so that, after a while, many good nodes are concentrated near X. This has two effects: firstly, nodes which are not near X will have high load; secondly, routing success rate will drop. Fig. 1 shows the success rate drops below 20%.

A decentralized self-protection strategy which doesn't need global information is for nodes to reset/restart their position with a certain probability. For simplicity, we can consider that the node restart decision occurs in rounds, a round is the time unit when all nodes decided whether or not they switch with another node. The restart probability can be calculated from different measures observed locally on each round (such as the density of the position distribution, the successful routes percentage, the switch percentage).

A malicious node succeeds when it switches with a good node. Thus it *poisons* a good node, *infect* good node with a false position. Fig. 1 shows that without self-protection (*Without Restart*), the entire network can be poisoned.

3 Experimental Results

We give some initial experiments to demonstrate the effect of malicious nodes and the effectiveness of the restart strategy. Our initial experiments focus on malicious nodes which attack by actively perform random walks to good nodes to infect them. Fig. 1 shows experimental results with 10^5 nodes in the SWN, the random walk length is 8 (same as [4]). The graphs show the percentage of successful routing, infected nodes and node switching over time. Without self-protection, all nodes will eventually become infected (*Without Restart % Infected*) and number of successful routes drop significantly (*% Success Route*). With

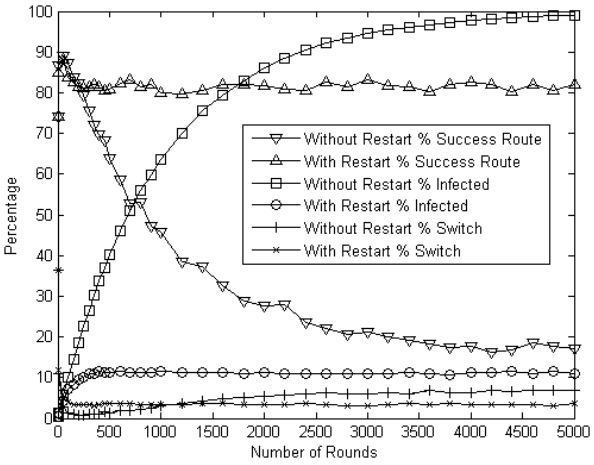


Figure 1. SWN with 0.1% malicious nodes and self-healing protection strategy with reset probability 0.008.

the restart strategy, successful routing can be maintained around 80% (without malicious nodes, convergence is rapid and routing success is $> 90\%$ after ~ 100 rounds), the infection rate is contained to about 10%.

4. Discussion

We show that the self-tuning which allows a small world network to function with decentralized routing fails when there are malicious nodes. Thus, self-protection mechanisms are necessary. Our preliminary experiments using self-protection using a self-healing strategy with partial restart shows that simple decentralized security mechanisms have promise. We observe that: i) without protection, all nodes will be eventually infected by even a single malicious node; ii) a small number of malicious nodes require a small restart probability, while more malicious nodes require larger restart probability. For more information and experimental results, see: `http://www.comp.nus.edu.sg/~halim/drswn`

References

[1] S. El-Ansary and S. Haridi. An overview of structured overlay networks. In *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*. Auerbach, 2005.

[2] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. *ACM Symp. on Theory of Computing*, 2000.

[3] B. Levine, C. Shields, and N. Margolin. A survey of solutions to the sybil attack. Technical Report 2006-052, U. of Massachusetts Amherst, 2006.

[4] O. Sandberg. Distributed routing in small-world networks. *Workshop on Algorithm Engineering and Experiments*, 2006.

[5] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, pages 440–442, 1998.

A.19 Small world networks as (semi)-structured overlay networks

Small World Networks as (Semi)-Structured Overlay Networks

Felix Halim, Yongzheng Wu, Roland H.C. Yap
School of Computing
National University of Singapore
Law Link, Singapore
{halim,wuyongzh,ryap}@comp.nus.edu.sg

Abstract

Recent research has shown that Small World Network (SWN) is navigable. In this position paper, we propose that SWN, for example those which are social networks, have nice properties which make them attractive as overlay networks. Such networks occupy a space between structured and unstructured overlay networks. Our thesis is that SWN may be attractive enough to be a replacement for traditional structured overlay networks which are usually based on Chord-style Distributed Hash Tables. Preliminary experiment results show that without node failure, the performance of greedy routing in SWN works very well and with additional links in SWN the robustness in routing can be improved as well as the resilience against node/link failure.

1. Introduction

In the peer-to-peer (P2P) system, the overlay networks are divided into two realms: unstructured and structured. The unstructured overlay networks have the simplicity in maintenance but very poor in routing (to find a node or an item, flooding is required) and scalability issues. In the next generation of P2P system, the structured overlay networks (SON), the scalability and the routing problems are addressed. Each node only requires small amount of routing table information $O(\log n)$ (depending on the overlay network topology) to route to any node efficiently $O(\log n)$. However, in order to keep the high performance, the consequence is the cost of maintaining the network topology. Another problem is that in a P2P setting, both structured and unstructured overlay networks are vulnerable to Sybil attacks.

There is another approach that has a potential to address the above issues: the Small World Network (SWN). The SWN resembles a SON but it has less structure.

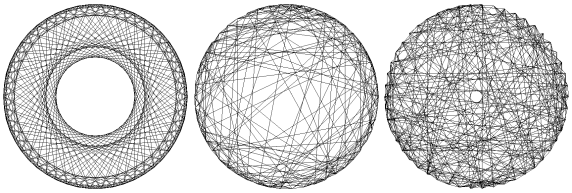


Figure 1. SON, SWN, and Random Network

Figure 1 shows the structure of the graphs for a SON[1], a SWN, and a Random Network. All the graphs have the same number of nodes and links. The leftmost graph is a SON (in this case a Chord ring [18]) where the links are very structured with distance in the form of 2^i. The middle graph is a SWN where the links are created according to power-law distributions. The graph is more relaxed and less structure. However, it has a high clustering coefficient, low diameter [19], and has been shown to be navigable [6] (that is, short-paths can be found between any two nodes in a few steps $O(\log^2 n)$). The rightmost graph is a Random Network where the links are randomly connected between any two nodes. Although each of the three graphs has the same number of nodes and edges, they look rather different because of how they are structured.

There are a number of research papers which look at SWN: [19] described characteristics of SWN as a network with low diameter and high clustering coefficient, [12] described a number of SWN models, [6] showed that SWN are navigable, and many following works of SWN [11, 7, 4, 15, 17, 21, 14, 13].

We are arguing whether a highly structured overlay is needed. SWN is in-between SON and random networks thus making it as a semi-structured overlay network (SSON). In this paper, we take the position that SWN as SSON can be a potential replacement for SON.

[1] Since n is not a power of two, the longest edges do not cross the center.

1.1 Outline

Section 2 explains the issues with SON (the various problems and attempts in the literature) Section 3 briefly introduce the background of SWN and related works. Section 4 shows some preliminary results which explain why we think SWN as Semi-Structured Overlay Network is promising. Finally, Section 5 concludes.

2. SON Issues

SON (also the unstructured one) suffers from the Sybil attack [3]. A Sybil attack is an attack where an attacker can present multiple identities, and uses them gain a disproportionately large influence in the system. Many SONs assume an upper bound on the fraction of malicious nodes. For example, the success rate of a 5-hop route is larger than 77% if the fraction of malicious nodes is less than 5%. However, when under a Sybil attack, a single attacker can create a large number of malicious nodes, and thus breaks the assumption.

Levine et al. [8] surveyed 90 papers on Sybil attack and categorized them into eleven categories. Approximately half of the published papers either suggest certification as a solution to the Sybil attack, following Douceur's [3] approach, or simply state the problem without giving a solution. He concluded that there is no general solution to the Sybil attack, but there are a variety of solutions that can limit or prevent the attack in several individual application domains.

Yu et al. [20] shows that under the assumption that a malicious user has limited social connections, the Sybil attack can be limited. Their protocol limits the number of Sybil nodes to $\log n$ per attack edge. An attack edge is a social connection from an attacker to an honest user. This means Sybil attack is much less effective in social networks.

Study [10, 9, 16] shows that SON incurs communication cost during churn (nodes join/leave) to maintain its structure. However, social network is more static comparing to SON, because the frequency of people making friends is smaller than the frequency of nodes joining/leaving. This means social networks have less communication cost to maintain its structure.

3. Background

In this section we briefly give an overview of SWN: the motivation, some SWN models, the strengths and guarantees, an example of SWN, and related work.

3.1 Small World Network

A Small World phenomenon first experimented by Stanley Milgram showing that the chain of social acquaintances required to connect one arbitrary person to another arbitrary person anywhere in the world is generally short (this is the origin of the phrase "*six degrees of separation*" and concepts such as Erdos number).

The definition of a "small-world" networks (SWN) was first introduced by Watts and Strogatz [19]. The SWN is created from a regular graph where each node is connected with k-nearest neighbors. Then each links are rewired to other nodes at random with probability p. As shown in [19], with $p \approx 0.01$, the resulting graph has small diameter and high clustering coefficient. If $p = 1$ then all links are rewired at random thus creating a random graph.

A random graph from routing stand point is like a completely unstructured network which has problems in navigability (finding short paths between any two nodes). Currently, the best way to do routing in unstructured network is by flooding which consumes a lot of network resources and bandwidth as in Gnutella-based systems. Although improvements can be made out of the system, the solution can be rather complex by using topology adaptation, active flow control, one-hop replication, and biased random walks [1].

However, for a SWN there exists a simple decentralized algorithm, shown by Jon Kleinberg [6], that can route between any two nodes in SWN with expected routing length of $O(\log^2 n)$, thus made SWN navigable.

In this position paper, we are interested in one example of SWN, a Social Network. A Social Network has the ability to self-organize such that the resulting network forms a small-world which has high clustering coefficient, low diameter, and navigable [2]. Moreover, in a Social Network, the identities of the nodes are verified by their neighbors directly so the significance of Sybil attack is minimized. For example, in a social networking site (such as Facebook, Friendster, LinkedIn, etc...), we usually verify the identity of the people who are requesting to be our friends, so it is very hard to create many virtual identities in a Social Network. In the remaining sections of the paper, the Social Network is referred to as SWN or a Semi-Structured Overlay Network (SSON).

3.2 Related Work

Hui et al. [5] constructed a Small World Overlay Protocol to investigated the behavior of structured P2P network under flash crowds, improved the object lookup performance, and handled heavy object lookup by exploiting the high clustering coefficient. This work is related with this paper in terms of showing that SWN can also work under flash crowds, while our work investigates SWN in terms of

node/link failure (churn), routing length performance, and routing robustness.

4 Towards SWN as a SSON

SON is used in a dynamic environment where conditions like with churn, node failures can happen. So, we would like to test using the same environment in SSON. In this section, we will test SSON behavior in dynamic environments. Empirical tests show that good performance is feasible and that provides evidence that SWN may be viable as SSON.

4.1 SWN Testbed and Simulator

As part of the SELFMAN project to investigate self properties, we have built a simulator and testbed for experimenting with SWNs.

The simulator interface is shown in Figure 2. The testbed contains a number of generators for SWNs and also SONs. It has a simulator which monitors routing performance (average routing hops), routing hops percentiles, success percentage (showing how many percent of the routing are successful), node positions (visualized as a ring), and other statistical distributions such as hop-counts, edge-distance, edge-count, etc. The simulator allows us to easily build new SWN algorithms and experiment with them. The SWN testbed GUI has extensive use of visualization and animation which are useful for understanding performance of the SWN. To handle large networks, we have a parallel version of the simulator which runs on our cluster. We have run networks up to 100000 nodes and depending on the type of experiment, typically these take between 1 minute to under an hour. Thus, our SWN testbed platform can handle realistic network sizes.

4.2 Simulator Settings

We experimented three SWN models: Kleinberg, Normal, and Sandberg. In Kleinberg model [6], the SWN are constructed from regular graph (k-nearest neighbor) with two long range contacts (shortcut links) where two nodes u and v are connected with probability proportional to $d(u,v)^{-r}$ where $d()$ is the distance function between the two nodes. Routing in this graph is possible using only local decision which usually called a "greedy routing". The routing only needs the coordinates of their neighbors and the target node in order to route. No other global information or states are needed to do the routing. In Kleinberg model, the routing between any two nodes are expected to complete in $O(\log^2 n)$. The graph of Normal model is constructed by taking the Kleinberg model and add more shortcut links up to $\log n$ links to mimic DHT (such as Chord). The graph of Sandberg model further add the number of links to $6 \log n$. All graphs are of one dimensional ring-like structure.

The three SWNs models are generated each with 100K nodes with a number of links according to each model. The graphs then are analyzed by running 10K routing tests. The routing is the greedy routing which only uses local information, that is by passing the message to the closest node to the target node. In case if a cycle is detected, or a node failed, or a link failed, the cost of 2 hops will be added to total hops and the message will be bounced back to the previous node and then it will be routed to the next closest neighbor and so on. If the total hops reaches more than $\log^2 n$ or no previous nodes can find any next neighbor then the routing is considered as failed. The routing tests only consider nodes that are not failed or leaved.

4.3 Preliminary Results

Preliminary results show that routing works rather well, it appears better than expected given the fact that the theoretical results are only probabilistic.

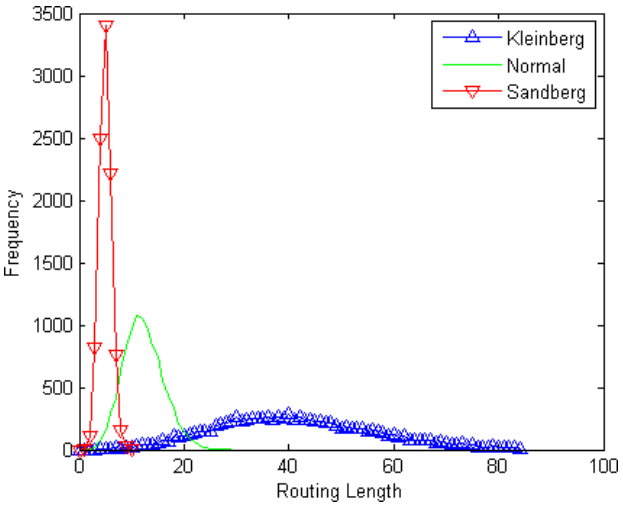


Figure 3. Routing Length Distributions

When there are no failures involved, we observed 100% success rate in the routing tests. Figure 3 shows the routing length distribution for Kleinberg, Normal, and Sandberg models under no node failures. In the Kleinberg model, the deviation in routing length is quite high (the routing tests completed in between 1 to 84 hops). However, all of them manages to complete below $\log^2 100000 = 275$ hops. In the Normal model, the deviation is smaller (the routing tests completed in between 1 to 31 hops). This shows that by having more edges, the routing lengths get shorter. In the Sandberg model, the deviation is very small (the routing tests are completed in from 1 to 10 hops) which means it's

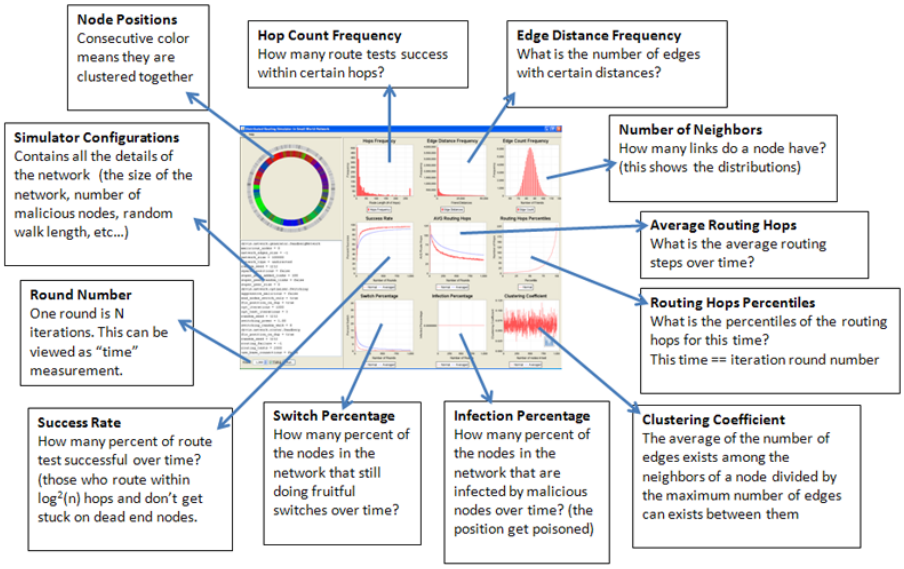


Figure 2. SWN Simulator Interface

very consistent in routing. Most of the routing tests only require 5 hops to complete.

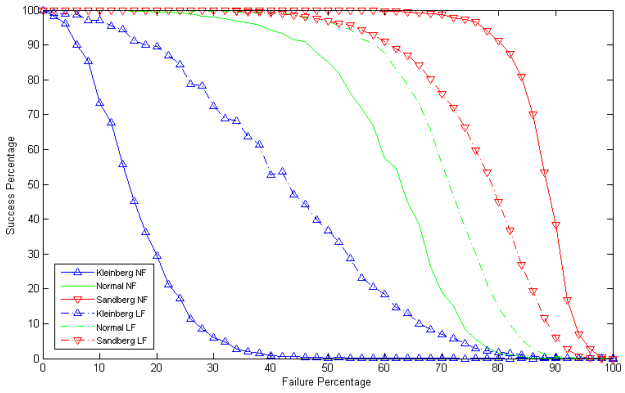


Figure 4. Comparisons between 3 models

Figure 4 shows the comparison between the 3 SWN models in terms of their resilience against node/link failure (churn). Node failure is equivalent to as having all links to the node failed where link failure is only a fraction of links of a node failed. In Kleinberg model, it can be seen that link failure is less damage than node failure. Since Kleinberg only has 2 additional links, it's very vulnerable to node/link failure. The success percentage drops below 50% with only 15% of node failure or with 40% of link failure. In Normal model, with $\log n$ shortcuts instead of 2 constant additional shortcuts, the robustness increases. The success percentage drops below 50% with 60% of node failure or with 70% of link failure. In Sandberg model, with $6 \log n$ shortcuts, the success percentage drops below 50% with 88% of node failure or with 78% of link failure. This is shows that by having more edges, the resilience against node/link failure increases but there is a strange behavior: the link failure does more damage than node failure. This merits further investigation.

In Chord, the routing is using 100% greediness that is by routing to the neighbor which is the closest to the target node. Hence the performance of Chord is expected be similar to the SWN Normal model above with $\log n$ neighbors. To measure the robustness of a routing, the greediness of the routing algorithm is tuned. Less greedy means we can have more routing alternative (by not picking the best one). By having more alternative, the routing can withstand from node failure. We defined a greedy routing with a greediness probability G which means with probability G the closest node (in the neighbors) to the destination will be picked to route the message. If failed then the next closest node will be picked with probability G, and so on. If the greediness probability G is very low, then the neighbor will be selected at random.

Figure 5 shows the effect of greediness in routing. In Kleinberg model where the number of edges is very few, greediness affects a lot on routing success percentage. As the number of edges increases to $\log n$ in Normal model the success percentage increases. With even more links ($6 \log n$) as in Sandberg model, the routing can achieve a good success percentage even with very low greediness.

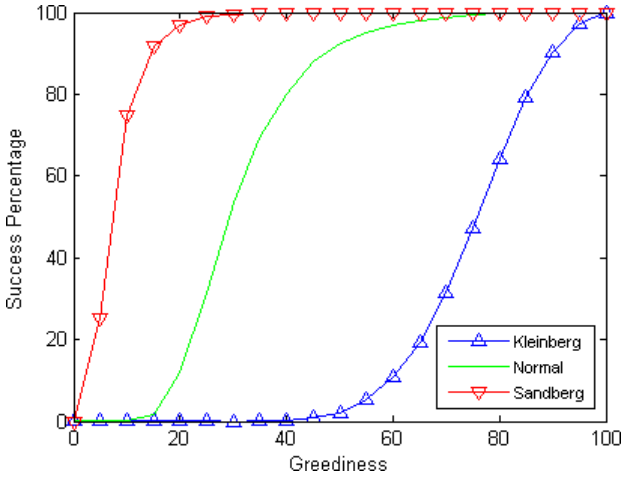


Figure 5. Greediness

5. Conclusion

In the experiments, the number of edges plays an important role in the robustness of routing performance as well as the resilience against node/link failure. In highly structured SON such as Chord the number links are the same for all nodes thus are not flexible and needs a high maintenance costs especially with larger links. In SWN the edges are flexible (nodes are free to choose how many edges they want) and the maintenance are cheap (they tend to self-organize like Social Network, the nodes choose their friends manually and not controlled by the system). In terms of node identity, SWN can be thought as ”full” of identities since the identities of the nodes that join into the network are verified by their friends thus minimizing the impact of Sybil attack. We argued that SON has many drawbacks that can be covered by SWN strengths. Preliminary results suggest that a SWN are quite efficient and thus able to function as a SON replacement. By increasing the number of edges, a SWN can also be made more robust but not have the drawbacks of SON in maintaining a large routing table.

Acknowledgements

We acknowledge the support of project SELFMAN (contract number: 034084)

References

[1] Y. Chawathe. Making gnutella-like p2p systems scalable. 2003.

[2] A. Clauset and C. Moore. How do networks become navigable? *Technical Report*, 2003.

[3] J. R. Douceur. The sybil attack. *1st International Workshop on Peer-to-Peer Systems*, 2002.

[4] P. Fraigniaud. Small worlds as navigable augmented networks: Model, analysis, and validation. *European Symposium on Algorithms*, 2007.

[5] K. Y. K. Hui, J. C. S. Lui, and D. K. Y. Yau. Small world overlay p2p networks. *The Twelfth IEEE International Workshop on Quality of Service*, 2004.

[6] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. *32nd ACM Symposium on Theory of Computing*, 2000.

[7] R. Kumar, D. Liben-Nowell, and A. Tomkins. Navigating low-dimensional and hierarchical population networks. *European Symposium on Algorithms*, 2006.

[8] B. Levine, C. Shields, and N. Margolin. A survey of solutions to the sybil attack. *Tech report 2006-052, University of Massachusetts Amherst*, 2006.

[9] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. 2004.

[10] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil. A performance vs. cost framework for evaluating dht design tradeoffs under churn. *24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005)*, 2005.

[11] C. Martel and V. Nguyen. Analyzing kleinberg's (and other) smallworld. *Principles of Distributed Computing*, 2004.

[12] M. E. J. Newman. Models of the small world (a review). *J. Stat. Phys., 101*, 2000.

[13] V. Nguyen and C. Martel. Analyzing and characterizing small-world graphs. *Symposium on Discrete Algorithms*, 2005.

[14] V. K. Nguyen. Small-world graphs: Models, analysis and applications in network designs. *Dissertation of graduate studies*, 2006.

[15] A. Ohtsubo, S. Tagashira, and S. FUJITA. A content addressable small-world network. *International Multi-Conference: parallel and distributed computing and networks*, 2007.

[16] S. Rhea, D. Geels, T. Roscoe, , and J. Kubiatowicz. Handling churn in a dht. *Proceedings of the USENIX Annual Technical Conference, June 2004.*, 2004.

[17] O. Sandberg. Distributed routing in small-world networks. *The Eighth Workshop on Algorithm Engineering and Experiments*, 2006.

[18] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Special Interest Group on Data Communications*, 2001.

[19] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature: Macmillan Publishers Ltd*, 1998.

[20] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[21] J. Zeng and W.-J. Hsu. Optimal routing in a small-world network. *Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2005.

A.20 Wiki credibility enhancement

This is a preliminary draft of the paper accepted at the WikiSym 2009 conference.

Wiki Credibility Enhancement

Felix Halim, Wu Yongzheng, Roland Yap
School of Computing
National University of Singapore
13 Computing Drive
Singapore
{halim,wuyongzh,ryap}@comp.nus.edu.sg

ABSTRACT

Wikipedia has been very successful as an open encyclopedia which can be edited by anybody. However, the anonymous nature of Wikipedia means that readers may have less trust since there is no way of verifying the credibility of the authors or contributors. We propose to transfer external information from outside Wikipedia to Wikipedia pages. These additional information is meant to enhance the credibility of the content. For example, it could be the education level, professional expertise or affiliation of the author. We do this while maintaining anonymity. In this paper, we present the design and architecture of such system together with a prototype.

1. INTRODUCTION

Wikipedia is perhaps one of the most successful efforts to create collaborative content. It is an encyclopedia covering a wide range of knowledge to exploit the "wisdom of the crowds" and to which anybody can contribute. Arguably, the success of Wikipedia is due to its open and self-policing nature. Anonymity is also a key feature – anybody can create an online persona with an account, or alternatively, the IP address is used.

One of the criticisms of Wikipedia is that the material is written by "anonymous strangers of unknown qualifications" [6]. Consider an entry on a technical subject, say a medical article, one might prefer an article written by a qualified physician. In this paper, we propose to enhance the credibility of the information contained in Wikipedia.

Consider the following scenario. The ACM maintains a comprehensive library of computer science publications with author information. If a contributor to a computer science Wikipedia article has credentials such as published in x ACM conferences and journals or affliation being MIT, this adds additional information about the credibility of an author. We call such information, *credibility information*. In Wikipedia, authors are identified by login id or IP address, but as anybody can make one or more login ids, the login information of an author does not by itself lend credibility. Rather, we want to be able to make use of other information from credible and trusted sources outside Wikipedia to transfer credibility information into Wikipedia. In this example, one could choose to retain anonymity while asserting a statement like published papers in ACM Symposium on Operating Systems Principles.

Unlike Wikipedia, Google Knol [2] attempts to add additional credibility information. In Knol, the credibility of the articles is based on the name of the author which can be certified by credential providers such as credit card companies or manually by phone. The verification mechanism is proprietary to Knol. Furthermore, it means that the author cannot be anonymous. Essentially, name verification tells one that a certain individual with a particular name as certified by Google contributed the article. However, the name by itself may not be very credible with the exception of well known authors. However, ambiguity still exists since several individuals could have the same name. For example, a Wikipedia author with pseudonym Essjay [1] claimed to be a (bogus) tenured professor who taught theology. Such an incident could also take place in Knol since a valid real name does not provide information about expertise or profession (i.e. professor of theology).

In this paper, we propose a simple extension to Wikipedia (and MediaWiki) which enhances the information in Wikipedia to make it more credible using credibility information from trusted third parties. Our extension maintains the open and anonymous nature of Wikipedia. We transfer information from trusted third parties and associate that securely with the text written by the author. We have implemented a prototype which utilizes the MediaWiki tag extension together with OpenID [3] as either an authentication or credibility provider although other credibility providers could also be used. Some scenarios where we can enhance Wiki:

- Verifying the author's name: A credit card provider such as Visa can certify that the author is a human and optionally their actual name. This gives a knol-like flavor to Wikipedia. It can also help to make it more difficult for robots to edit Wikipedia.
- Verifying the anonymous membership an organization: A provider like ACM can sign university or expertise credentials for an author.

- Restricting anonymous voting system: A credibility provider can be used to restrict the voting system in Wikipedia to from certain voters without disclosing the name of the voters.
- Other services: can enhance Wiki articles by giving information about the author while preserving the anonymity of the author.

2. DESIGN GOALS

Before discussing the design of the credibility enhancement for Wikipedia, we first give our design objectives:

- **Credibility:** The purpose of the credibility enhancement is to enable Wikipedia to show some external trusted information about the authors. Such information could be the authors' real names, professional affiliation, proof of identity, etc., essentially anything which can give additional credibility to the text in an article. This information has to be reliable so that authors cannot easily lie on the information they provide. We also want to avoid an author stealing other author's identity to publish/edit pages.
- **Anonymity:** We want to preserve the capability of authors to be anonymous if they want to, i.e. we do not want Knol [2] which requires that the real names of users be verified. Furthermore, we want to ensure that users' private data is not stored in Wikipedia's, so that even if Wikipedia is compromised, users' private data will not be exposed.

  There is trade off between credibility and anonymity. Authors sometimes want to be anonymous, but that means their statements/edits may be less credible. Less credible edits may be more likely to be deleted by Wikipedia administrators. We allow the author the freedom of balancing the trade off and provide different levels of credibility information.
- **Ease of Use:** The enhancement should not make Wikipedia much harder to use, e.g. forcing authors to download and run some software on their local machine is inconvenient and should be avoided.

We remark that the credibility information in our proposal is independent of reputation. We preserve reputation [4] on any edits, and, reputation can be linked to the author's credibility as well.

3. PROTOCOL DESIGN

The credibility extension involves four components including the author which work together as shown in Fig. 1 *C1-4*. *C1* is the Wikipedia web server with our credibility extension installed. *C2* is the credibility proxy (we suggest it be run in a different host). The Wikipedia web server stores a certificate of the proxy so that Wikipedia can verify the proxy's signature using its public key. Note that it is possible to have more than one proxy, but we use one for the illustration purpose.

C3 is one or more credibility providers. The credibility providers give credible information specified by the author

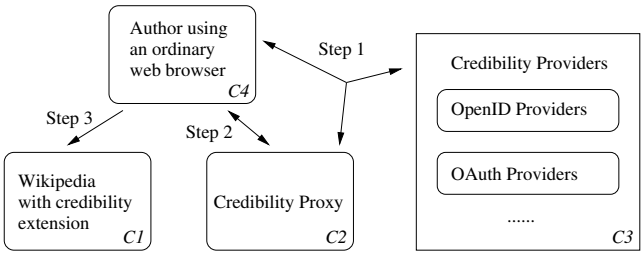


Figure 1: Components and work flow of the credibility extension.

to the credibility proxy. They communicate with the credibility proxy using the respective supported protocol. For example, the OpenID protocol needs three-way communication among the author, OpenID server, and credibility proxy. *C4* is the Wikipedia author using an ordinary web browser.

There are three main steps get a credible edit in a Wikipedia page:

- **Step 1: acquiring author information**
  In the case of OpenID or OAuth protocol, this step involves three-way authentication. After this step, the credibility proxy should have the author's information. This step can be performed multiple times to get information from multiple providers.
- **Step 2: sign**
  The author selects the appropriate author information to be passed to Wikipedia and enters the text to be published in Wikipedia. The credibility proxy signs the author's information together with the text and generates the signed text, see the screen shot in Fig. 2.
- **Step 3: edit page**
  The author pastes the signed text to Wikipedia (shown in Fig. 3). Note that the author does not have to login to Wikipedia in order to use the credibility extension. The signed text can be published elsewhere and someone else can enter the signed text. It can also be copied between pages.

When the edited page is viewed, the credibility extension verifies that the edit has been signed correctly using the credibility proxy's certificate. If the edit is verified, the author's information will be displayed — this can be done in various ways, e.g. as in Fig. 4. Our credibility extension is compatible with caching which is important for Wikipedia performance, the signed text does not have to be verified every time it is viewed.

The trust relationships among the four components are:

- Wikipedia trusts the credibility proxy to sign the correct information. Wikipedia also trusts that the proxy's key is not compromised.
- The authors trust the credibility proxy to only release information which they authorise. Note that the information can be filtered by the credibility providers

before it is given to the proxy, so the ideal case is that the proxy *only* knows the information to be signed and released. However, some information such as the user ID in the OpenID server and user's IP address are always known to the proxy.

- The credibility proxy does *not* have to trust the credibility providers because the providers' name will be shown together with the signed text. We leave the Wikipedia readers to decide whether to trust the providers or not but Wikipedia could choose to trust predefined providers so as to be able to conveniently display them in the Wikipedia article.
- The authors implicitly trust the credibility providers which are chosen by them.

4. CREDIBLE WIKI PROTOTYPE

We describe a credible Wiki prototype to illustrate our ideas. It consists of a credibility proxy and a MediaWiki extension. Our prototype employs the OpenID 2.0 framework [5] to communicate between the credibility proxy and third party credibility providers to share information about the particular user. However, other open protocols could also be used. The proxy anonymizes the user information selected by the user and signs it along with the text. Wikipedia only needs a lightweight extension to check the signature of the text sent by the proxy. If the signature matches, it will be publish along with the assigned credibility information. Otherwise no special credibility will be given to the text.

4.1 The Credibility Providers

Credibility providers are the source of the additional information for the authors to enhance their credibility of their edits. Recently, `http://www.myid.is` provides a service to certify a digital identity online which is similar to what Knol uses for author name verification. One can imagine a variety of credibility providers to provide a variety of information which could include public and private organizations. The information would be some property associated with the author such as professional association, real name verification, geographic location or country, etc.

The credibility provider must have a protocol to share information to the credibility proxy or any other consumer. We observed that OpenID [3, 5] and OAuth are the two most promising open protocol to be used widely for managing the online identity and sharing information.

OpenID provides a decentralized open standard for user authentication and access control. The user only needs to setup one digital identity on an OpenID provider to gain access to other systems. We take advantage of an OpenID provider not for login but as a way of transferring information about a digital identity, so we use an OpenID provider as a credibility provider.

Our examples with our prototype use a free OpenID provider (`myopenid.com`) as the credibility provider. Since there is no particular trust associated with `myopenid.com`, the information in the examples is only illustrative.

4.2 The Credibility Proxy

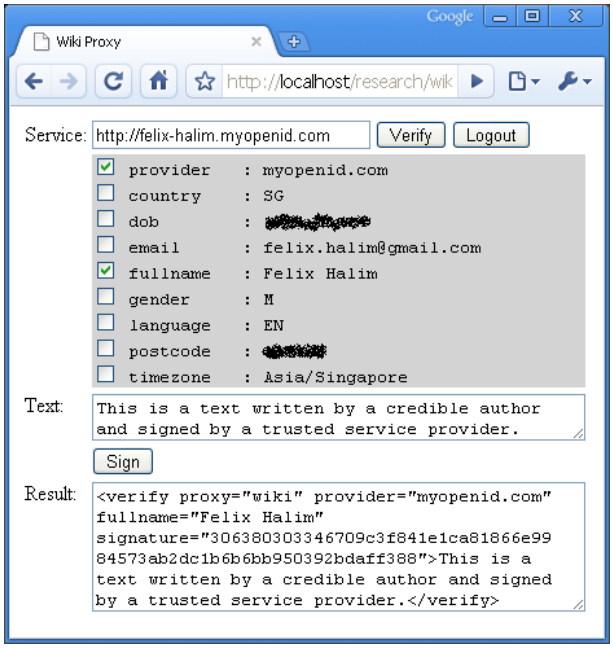


Figure 2: A Proxy for Wikipedia.

Fig. 2 shows our prototype. The service field is filled with the URL of the credibility provider. The gray area is the user information retrieved from the OpenID credibility provider. The *text* area is the text that will be signed by the proxy together with selected user information. The example chooses to include the provider and the full name to be signed with the text. The *result* area is the ready to use Wiki text that can be inserted anywhere in Wiki page.

We allow the author to select which information from credibility providers to be attached. This information should be thought of as credibility attributes to be attached to the edit. Wikipedia could have a policy to require certain attributes from trusted credibility providers in order to achieve a certain category of credibility. For example, to get a credibility of a "scientist", the author have to include information such as: institution, position, and perhaps information about publications (as in the ACM example in Sec. 1).

4.3 Wikipedia Extensions

MediaWiki is the software behind Wikipedia. MediaWiki can be extended using extensions such as tag extensions, parser functions, special pages, or template extensions. We implement our credible Wiki using a tag extension which we call the *verifier extension*.

4.3.1 Wiki Verifier Extension

The text signed by the credibility proxy can be put inside any page in Wikipedia (as well as outside Wikipedia since verifying the signature can be easily done with the certificate of the credibility provider). We created a verifier extension tag to check the that the text and additional attributes inside the tag have been signed by the proxy.

There are three mandatory items and several optional attributes within the verifier tag extension:

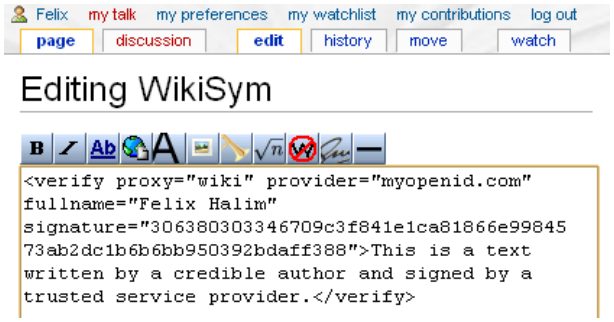


Figure 3: Verifier tag extension for Wiki.

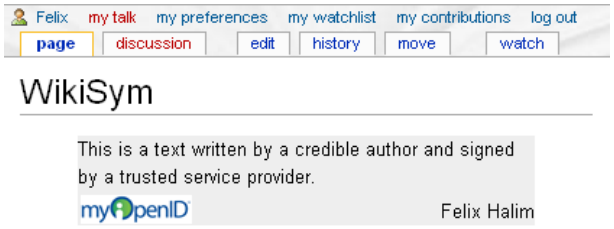


Figure 4: The end result in Wikipedia page.

- proxy: the name or the public key of the proxy. Wikipedia will be able to verify the signed text by having a list of trusted proxies and their certificates.
- signature: the signature of the text inside the tag. The signature should match with the digested text decrypted using the public key of the proxy.
- text: the text to add or edit.
- optional attributes: such as provider, full name, country, email, etc. can be included as the attribute of the verifier tag. Wikipedia then can use the additional information to display the text.

Fig. 3 shows an example of a verifier extension tag. The content of the signature attribute contains the signed digest of the information in the verify tag. If any of the text or attribute values are changed, the verify tag will treat the text content as regular text rather than as credible text.

Credible text in a Wikipedia page should be presented in a way which can show its credibility properties. While there are many ways of doing the presentation, Fig. 4 shows displaying credible text by graying the background. The displayed paragraph with grayed background provides the "context" for the author when editing a paragraph in Wiki. The idea of a context is to make it harder to abuse the credible text (i.e. placing the text in different paragraph or articles that have different context to get different meanings from the same text).

The display of the text can be improved further with more credibility information (other than 8 fields in Fig. 2). For example, a badge-like display can be used to annotate the text with particular properties to be associated with user information matching a Wikipedia credibility category, e.g. "computer scientist".

4.3.2 Wiki Poll Extension

The MediaWiki poll extension can benefit from the credibility extension. Currently, the poll extension stores the IP address and Wikipedia user name pair as the poll account to vote for the poll. If the user does not have a Wikipedia account then only the IP address will be used to vote. The poll doesn't allow duplicate votes for each poll account.

Credibility allows the poll account to have additional information about the account. Alternatively, we may want to restrict the participant of the poll by only accepting users from a particular country. This can be done by requiring a "country" field from a trusted credibility provider (other information could be hidden).

5. DISCUSSION

Wikipedia accumulates information through the efforts of anonymous contributors and volunteers. While this is democratic, it has a weakness that the information may be perceived as being less credible (regardless of whether or not it is actually so). Normally, Wikipedia uses external citations to add credibility to the information entered. However the citation may either not be available or not easily accessible (confidential). The text might also simply be just words of wisdom from an expert author but it is hard to convince the readers that the text they are reading has a certain quality as it may lack sufficient citation.

Well known authors usually have credibility information outside Wikipedia. Our enhancement allows to transfer the rich information about the author available from the third party credential provider to Wikipedia. Our enhancement can be seen as a complement to the citation mechanism. It is important to note that, in the process of transferring the author information, we can maintain the anonymity of the authors which is consistent with the philosophy of Wikipedia and serves to protect the authors.

Our credibility mechanism can be used to enhance any reputation mechanism. It may be also used by administrators to manage edits.

Acknowledgements

We acknowledge the support of project SELFMAN (contract number: 034084)

6. REFERENCES

[1] "Essjay Controversy", `http://en.wikipedia.org/wiki/Essjay_controversy`.
[2] "Knol", `http://knol.google.com/k`.
[3] `http://openid.net/`.
[4] B.T. Adler, K. Chatterjee, L. de Alfaro, M. Faella, I. Pye and V. Raman, "Assigning Trust to Wikipedia Content", WikiSym, 2008.
[5] D. Recordon and D. Reed, "OpenID 2.0: A Platform for User-Centric Identity Management", Digital Identity Management, 2006.
[6] P. Denning, J, Horning, D. Parnas and L. Weinstein, "Wikipedia Risks", Comm. of the ACM, 48(12), 2005.

A.21 A Toolkit for Peer-to-Peer Distributed User Interfaces: Concepts, Implementation, and Applications

A Toolkit for Peer-to-Peer Distributed User Interfaces: Concepts, Implementation, and Applications

Jérémie Melchior[1], Donatien Grolaux[1,2], Jean Vanderdonckt[1], Peter Van Roy[2]
Université catholique de Louvain, – B-1348 Louvain-la-Neuve (Belgium)
[1]Louvain School of Management, Place des Doyens, 1
[2]Dept. of Computing Science and Engineering, Place Sainte Barbe, 2
nediar@gmail.com, {jeremie.melchior, jean.vanderdonckt, peter.vanroy}@uclouvain.be

ABSTRACT
In this paper we present a software toolkit for deploying peer-to-peer distributed graphical user interfaces across four dimensions: multiple displays, multiple platforms, multiple operating systems, and multiple users, either independently or concurrently. This toolkit is based on the concept of multi-purpose proxy connected to one or many rendering engines in order to render a graphical user interface in part or whole for any user, any operating system (Linux, Mac OS X and Windows XP or higher), any computing platform (ranging from a pocket PC to a wall screen), and/or any display (ranging from private to public displays). This toolkit is a genuine peer-to-peer solution in that no computing platform is used for a server or for a client: any user interface can be distributed across users, systems, and platforms independently of their location, system constraints, and platform constraints. After defining the toolkit concepts, its implementation is described, motivated, and exemplified on two non-form based user interfaces: a distributed office automation and a distributed interactive game.

Categories and Subject Descriptors
C2.4 [**Distributed systems**]: Distributed applications. D1.3 [**Concurrent Programming**]: Distributed programming. D2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces; user interfaces*. D2.m [**Software Engineering**]: Miscellaneous – *Rapid Prototyping; reusable software*. D4.7 [**Organization and Design**]: Distributed systems. H.1.2 [**Information Systems**]: Models and Principles – *User/Machine Systems*. H5.2 [**Information interfaces and presentation**]: User Interfaces – *Prototyping; user-centered design; user interface management systems (UIMS)*. I.6.5 [**Model Development**]: modeling methodologies.

General Terms
Design, Experimentation, Human Factors, Verification.

Keywords: Distributed User Interfaces, Multi-Device Environments, Multi-platform user interfaces, Multi-user user interfaces, Peer-to-peer, User Interface Toolkit, Ubiquitous computing.

1. INTRODUCTION
The division of labor in corporate environments requires more and more to allocate tasks to users in a flexible, dynamic, and opportunistic way. For instance, a task that can no longer be ensured by a worker is delegated to another one or offered to single / multiple resources in order to be achieved. Interactive tasks are frequently reallocated or (re)distributed across workers in an organization and the User Interfaces (UIs) should support these interactive tasks. For instance, a particular worker may want to get some advice from a colleague for solving a problem. For this purpose, she can share some part of the information by redistributing parts or whole of her UI, here by duplicating the UI to another user using another computing platform and located in another environment or context of use.

Workers may distribute UIs for several reasons beyond this distribution of task allocations: in order to follow task allocation, to exchange information with co-workers, to balance private and public information [15], to partition the working space into different parts [12] and several displays [22]. They all want to distribute their UI in order to match requirements of dynamically distributed tasks and to keep the same usability quality in the distributed UIs as they had before the distribution. The UIs of these applications are generally unable to accommodate such changes, thus forcing end users to switch from one application to another or to rely on workflow management systems (if any) to reach their goals. In general, the following situations may arise:

- *Multi-monitor usage*: a single user using a single computing platform may want to distribute her UI across various monitors connected to the same platform [12,14]. For instance, a dual display if the graphic card allows it or an external monitor via an external port.
- *Multi-device usage*: a single user may use several different devices together, whether they are running the same operating system or not [15]. For instance, a user may control a music player running on a media center using a remote control running on a handheld device.
- *Multi-platform usage*: a single user may user heterogeneous computing platforms, perhaps running different operating systems [21]. Note that a multi-device usage implies a multi-platform usage (since there are different machines) but the reciprocal does not hold: a user could use several computers (hence, multi-platform) that are similar (hence, no multi-device).
- *Multi-display usage*: we hereby define multi-display as a combination of multi-monitor and multi-device usages [22]. A single user may distribute a UI across multiple monitors and devices simultaneously.
- *Multi-user*: it represents an extension of the previous usages to multiple users concurrently [5]. In this case, one or many users may want to distribute parts or whole of their UI across several monitors, devices, platforms, or displays. For instance, in

EICS'09, July 15–17, 2009, Pittsburgh, Pennsylvania, USA.

a control room setup, users may want to direct portions of a UI to other displays of others users depending on the context of use. When a multi-user interface is of concern, it is also typically used for supporting tasks that are allocated or deallocated from one user to another one, such as in task delegation, task suspension and resuming.

A *Distributed User Interface* (DUI) consists of a UI having the ability to distribute parts or whole of its components across multiple monitors, devices, platforms, displays, and/ or users. Hence, a DUI should support any of the aforementioned usages, the multi-monitor being the minimal.

This paper presents a toolkit with the following properties. Given n application processes and m display processes, the toolkit allows the GUI of each application to be partitioned arbitrarily and dynamically over the m display processes. A single display process can therefore combine parts of the GUI of each application, in one or several windows with no restriction. No other dependency exists beyond those implied by the n+m processes, i.e., if an application process crashes, its GUI disappears and if a display process crashes, the GUI parts that it hosts 'return' to the application process and can be migrated to another display process. Each application has its own point of failure which means that if an application process crashes, other applications still work.

During the past two decades, a lot of work has been dedicated to addressing some of the aforementioned usages at a time, such as multi-device or multi-platform only. With the advent of all these usages, the time has come to enable designers and developers to engineer their interactive applications in a way that they do not need to take care about the functions required to support these usages.

In other words, the user interface of such applications should become completely agnostic, independent from any underlying technology that would allow these usages. Until now, many works addressed these usages explicitly, but in a way that forces designers and developers to think and develop user interfaces in a way that is constrained by distribution. For instance, the underlying software architecture may influence the way the user interface is programmed if it should be migratory [3,26] or multi-user [5].

In this paper, we relax this important constraint by offering to designers and developers a toolkit that would enable them to design and develop user interfaces that support all the aforementioned usages. For this purpose, the remainder of this paper is structured as follows: Section 2 reports on some significant pieces of related work in order to characterize a brief, yet accurate, state of the art in the domain of distributed user interfaces. Section 3 summarizes the benefits brought by the toolkit. Section 4 immediately shows two case studies of interactive applications exhibiting user interfaces covering the above usages that are hard to develop otherwise. It then introduces, describes, and motivates the software architecture of the underlying toolkit that was used for those two case studies and finished with some final examples. It also discusses some selected aspects and properties of this toolkit. Section 5 concludes the paper by presenting some future avenues of this work.

2. RELATED WORK

In this section, we compare the advantages and disadvantages of the major related work in order to identify the specific aspects of our toolkit.

Probably the first DUI ever was developed as a system that distributed a UI over many workstations connected to the same network and running the same operating system [3] thanks a to a connector mechanism. In [4], a program is dynamically changing between centralized, replicated and a hybrid collaboration architecture. There is a notion of masters using a replica of the program and slaves using one of the replicas provided by a master in a centralized architecture. It allows mobile devices with lack of computing power to avoid running the program when it can communicate with a more powerful device.

In [13], a web page is split in partial pages which will be replicated to all the users. The framework supports multi-device and multi-user Web browsing where clients connect to a server which delivers the page. A proxy split the pages in respect to the device and user constraints. Each page is in a XML file with specific tags to configure how the Web page will be split among the different users and devices.

In Luyten & Coninx [17], it is shown how an interactive system can be distributed among several peer devices. Their approach relies on the fact that nowadays most computing resources are network-enabled and publish their device profile like in UAProf or CC/PP. It raises the opportunity for supporting collaborative tasks with the same user interface with little or no extra effort from the user interface designer.

In [2,24], a part or whole of a DUI can be migrated from one platform to another at run-time. The underlying architecture is a client-server architecture that maintains in a central position the internal state of the DUI.

In this paper, the toolkit that will be presented is significantly different from this previous work in that it provides a unique combination of the following features:

- Any DUI developed in the toolkit may benefit transparently from facilities provided to support any of the aforementioned usages.
- The DUI is not restricted to form-based applications as it is possible to distribute any graphical UI, such as from a game, a spreadsheet, a graphic application. Such programs contain native widgets.
- The DUI is not restricted to web applications. The only condition to distribute a UI is to have the platforms connected via a LAN or a wireless connection, but the applications are not necessarily web applications in markup languages. There is no such language restriction like having a UI in HTML or another markup language.
- The toolkit relies on a genuine peer-to-peer architecture in the sense that there is no client and no server: every platform can send and receive any part of a DUI depending on its distribution rights. No single platform maintains the DUI internal state.
- The granularity of distribution can range from the application level to the widget level: an entire application can be distributed across platforms for instance, but also the different components of any widget. For instance, even a radio button, consisting of a circle to be checked and a label can be split across platforms. Even the label could be distributed, although it does not make sense in this case.

3. SOLUTION BENEFITS

The toolkit does not rely on a client-server architecture which in the WebSplitter[13] would allow each device to create a Web page

and to share it with others. There is no need for a server to store the Web page and provide it to the proxy. Each peer may create a user interface and share it through the network. The exchange of data is values passed through a message passing mechanism; there is no static representation of the application in files. The toolkit combines the ability to distribute a part or whole of a user interface among devices without needing to completely share the application. It is possible to design a lot of various applications at a finer granularity than in [3,4,13]. Here is how the dimensions are possible in the toolkit:

- *Multiple displays*: the system may be a single device connected to multiple displays or multiple devices. For a single device with many displays, many rendering engines on the same device might be used. For multiple devices, each device will create at least one rendering engine.
- *Multiple users*: the user interface can be distributed in part of whole for each user independently of the situation.
- *Multiple operating systems*: the toolkit is developed in the Mozart environment which supports a lot of different operating systems.
- *Multiple platforms*: it is supported by combining the multiple operating systems dimension and the distribution of parts of the user interface. The smallest form factors which are not able to render the whole user interface might just get what they need and are able to display.

In sort, any computing platform, regardless its operating system (i.e., Linux, MacOs or Windows), can trigger a distribution of a user interface to other platforms provided that they are connected together through any local or wireless network. Each computing platform can be used by a single user or by multiple users. The distribution of the UI is not governed by a single machine since any portion of a UI can be, for instance, forwarded to another one, and vice versa.

4. DISTRIBUTED USER INTERFACES

4.1 Case Studies

Two case studies have been chosen to evaluate the power of the toolkit. The evaluation of a toolkit is never easy because there are always a lot of different and complex applications and it is not possible to test every single feature and his behaviors. The choice of case studies is thus quite important and difficult.

The *first case study* is a distributed office suite. The goal is to show that even a complex application with many toolbars and buttons, such as those found in office suites, can be distributed over multiple devices and many users. The interface can be decomposed in a lot of components. Going further in the decomposition of the interface, the work space can also be divided into a lot of different migratable regions. Distributing components of any commercially-available application belonging to an office suite cannot be achieved today.

The *second case study* is a distributed Pictionary. This is a multiplayer game where each player has his own role. To prevent players from cheating, the toolkit provides some distinctions between the users. Players trying to guess the word cannot see it and the player who has to describe the word by drawings should see the word and draw in an area. The drawing area then is seen by the other players but none of them can edit it.

Pictionary is naturally distributed because it needs at least a drawer, a player trying to guess the word the other is trying to draw. This game is task-driven distributed, the choice of by whom and where the task is realized depends on the task itself. A player has to pick a word and then another have to guess that word. These two tasks have to be realized by two different users, otherwise the player will know the word she has to guess. These two tasks should also happen at different places to prevent the players seeing the chosen word. The game is distributed across players.

Contrarily to Pictionary, the distributed office suite is not naturally distributed. Someone may write a text or draw a picture but nobody is able to write in the text at the same time or to draw on the same picture. One of the objectives of the first case study is to distribute an application that is not naturally distributed. Many users will be able to draw on the same place, to write on the same text at the same time and to work on the same spreadsheet. Another objective is to be able to distribute atomic elements of the user interface. Buttons on the drawing toolbar shall be detachable and distributable. A paragraph or a line of the text shall also be distributable without the need to distribute the whole text. A task like writing a text may be realized by more than one user, which is different than usual office suites.

4.1.1 The Distributed Office Case Study

The office suite is a bundle of applications running separately in order to write text, to draw something, to make a presentation, and to do other typical office tasks.

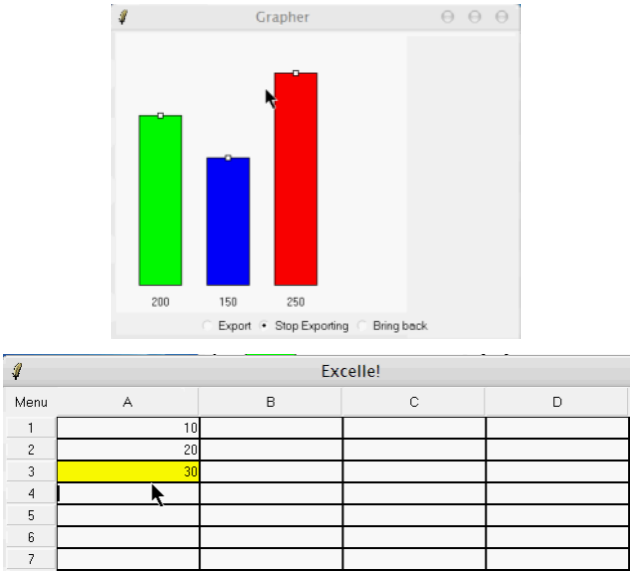


Figure 1. A graphing and a spreadsheet application.

Figure 1 reproduces a screenshot of two typical office applications: a graphing that shows a histogram with 3 columns. The value of each column can be edited by height and a spreadsheet. Any component of either application can be marked for export and exported to another display, device, platform or user. In particular, a whole UI like the one of graphing application itself is entirely migratable from one platform to another. In the second case, every cell is migratable from one platform to another. The value of a cell, even if it is the result of a formula, will remain consistent with the formula, no matter where the cell is displayed.

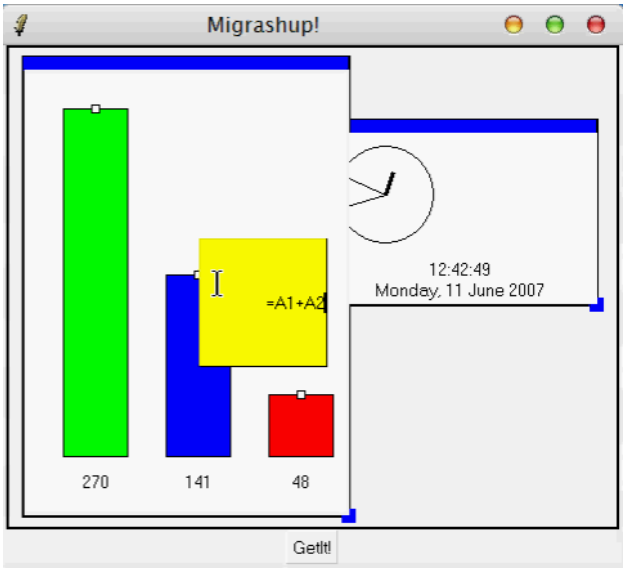


Figure 2. Example of migration realized from the two applications.

The result of a possible migration of components exported from both applications is illustrated in Figure 2. For each application, parts or whole can be marked for export, exported to another platform where it continues to run its own life, and re-imported back from where it was initially exported. This migration allows a user to enter some data in the spreadsheet while another user is looking at the result. The other user may adapt the values of the graphing thanks to the values of the cells he gets from the spreadsheet. The work is concurrently achieved and many users are working together no matter where they are.

Applications with migratable abilities allow many users working together on the same application and on the same document. The sequential work is thus converted to a concurrent multi-task system. A full video of a typical session demonstrating this case study is provided attached on the PCS system for this submission.

4.1.2 The Pictionary Case Study

The Pictionary is a multi-player game with a board. Players are separated into teams that will have to guess words. Teams are represented by their position on the board. They win the game when they reach the last square of the board. In each team, a player has to make his team guess a word by drawing some clues or the word itself but without talking and without writing it. Here, the game is a multi-user application with two teams. Both teams have to play alternatively. The needs of this case study are some devices able to do the tasks. The first task is the selection of a word on a computer by a team, let's call it Team 2. This computer may be a PDA, a laptop or a desktop. The other team, let's call it Team 1, has to guess the word selected by Team 2. A member of Team 1, denoted Player, has to draw something to help his teammates find the word. This task has to be realized on a PDA to allow drawing like on a paper sheet. In order to see what Player is drawing, another device is set up to display the drawing. A computer with a big screen or a video projector can be used to realize this task. An example of architecture able to run the application is illustrated in Figure 3.

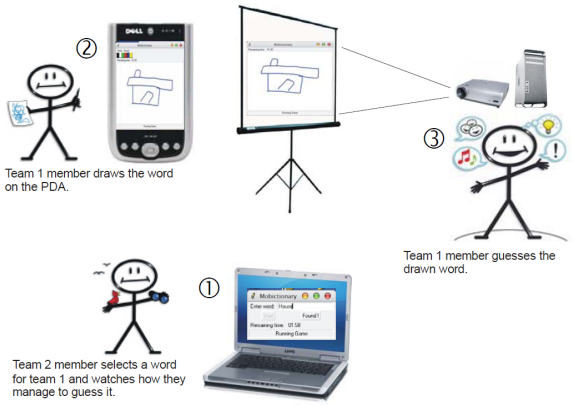


Figure 3. Example of architecture for the Pictionary.

To start the game, the application must be run on the three devices. Once the connection is established, Team 2 goes to a computer to enter a word. Once entered, a countdown starts to prevent Team 1 using too much time to find the word. The PDA is given to Player and the word appears on the screen. He may now draw whatever he wants. The video projector displays the drawing for everyone. These different steps are reproduced in Figure 4. In this case, the different screen shots are taken from the different devices running the same operating system, but nothing would prevent this application to run between devices running different operating systems.

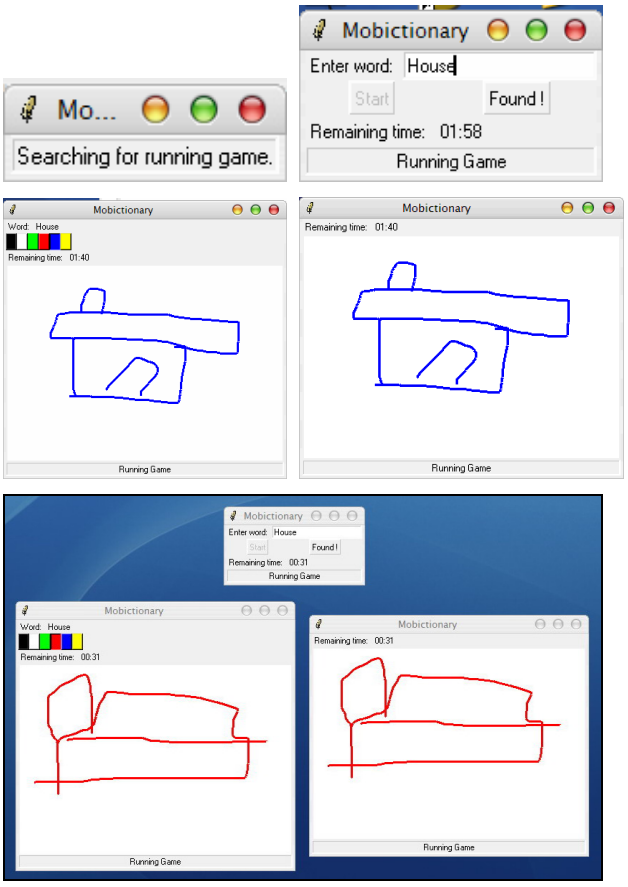


Figure 4. Screenshots of the Pictionary.

Indeed, the toolkit used for this purpose allows deploying DUI on top of three operating system: Linux Ubuntu distribution, Mac OS X and Windows XP/Vista any version.

The toolkit allows designing and developing the DUI of this application and supports the multi-user aspects of the game. To prevent cheating, some widgets are created and migrated to the PDA as well as the toolbar which allows drawing. The application can also be run on a single computer in multiple windows but we recommend using at least three devices: one to enter the word, one for the drawings and one to find the word.

Let us design this application with our toolkit, using the simple multi-user aspects of the toolkit. Even if three different devices are used in a distributed fashion, we must think of this application as a single process application, with transparently distributed user interfaces. The widgets used are:

- A text field for entering the text (Laptop)
- A label for displaying this text (PDA)
- A start button for accepting the text input and starting the countdown (Laptop)
- A clock (PDA, Laptop, and PC)
- A free drawing area (PC and PDA)
- A toolbar for selecting the drawing tools (PDA)
- A "Win" button to click if the word is guessed on time (Laptop)

We end up with seven different components, some of them present only on one device, others at different devices simultaneously. We use the proxy-renderer relationship: the underlying principle is that the proxy serves as the reference for the state of the widget while the renderer follows the instructions of the proxy for updating its incarnation. This principle allows the existence of several renderers; each of them following the same instructions. Basically they all act as mirror views of the same proxy. Note that this is against the principle of not disrupting the stationary behavior as we introduce multi-user capabilities. New complexity is introduced because of concurrency and coherency problems. The toolkit itself provides a very basic way of dealing with this complexity: each widget can be configured so as to have at most one renderer at a time (the default), or to let an arbitrary number of renderers be connected at the same time. For the Pictionary application, we have the following functionalities:

- The application can be run from any device, including the PC, the laptop, the PC or even another computer. For our demonstration, we use the laptop.
- The text field, Start and Win buttons are created and migrated to the laptop
- The clock is created, configured to support multiple renderers and displayed on all the devices
- The label and the toolbar are created and migrated to the PDA.
- The canvas is created, configured to support multiple-renderers and displayed on the PDA and the PC.

When the start button is clicked, the content of the text entry is placed into the label, and a countdown thread is created: each second the clock is updated to reflect the remaining time. The toolbar chooses the active drawing tool, while the clicks on the drawing area issue commands to apply it at that place. And that's it; we have a functional simple multi-player game! Note that the three processes find each other using the Discovery module of Mozart [6], which uses a broadcasting message on a LAN to find providers. The Pictionary is inspired by the HyperPalette [1], but in a distributed way that can be tailored at run-time.

4.2 SOFTWARE TOOLKIT FOR PEER-TO-PEER DUIS

4.2.1 Software architecture

Each application using the toolkit relies on a three-layer model as in Figure 5. The Application layer describes the different services offered, the toolkit for migration and adaptation and the distribution layer for peer-to-peer network. At the top, the Application layer is the part developed for the application itself. It differs between applications depending on their needs. They support migratable and adaptable user interfaces. This part is independent from the toolkit and the distribution layer. The application has a standard graphical user interface. The middle layer is the toolkit supporting the different features for migration and adaptation. It extends Tcl/Tk which is a toolkit for graphical user interfaces supporting several platforms. An extension Ext is added to this toolkit to provide the needed features. The lowest layer is the base for the toolkit. Mozart [6] implements the Oz programming language [25]. It supports several paradigms and distributed applications [20]. It relies on a distribution layer which relies on TCP/IP protocol.

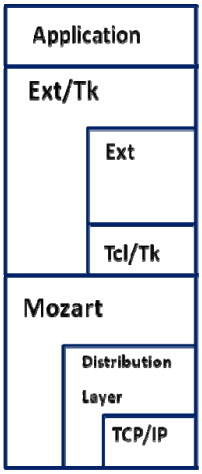


Figure 5. Software architecture of applications using the toolkit.

4.2.2 Granularity of migration & adaptation

A running application could have its UI migrated and/or adapted at different levels of granularity:

1. Whole screen containing the UI of the application (which may also contain the UI of other running applications).
2. Whole UI of the application, typically contained in a single window.
3. Subset of widgets of the application:
   a. Limited to a single widget.
   b. Limited to widgets that respects some placement constraints (for example respecting a rectangular shape).
   c. Any arbitrary selection of widgets.
4. Arbitrary pixel area.

Not all these levels are interesting for our purpose. In level 1, we lack information regarding the remaining of the screen which makes virtually impossible to provide interesting adaptation. In level 4, the arbitrary nature of the area also makes it virtually impossible to provide interesting adaptation. Level 2 is a particular case of level 3, where the whole UI of the application is used instead of a particular subset of it.

Consequently **Ext/Tk provides migration and adaptation support at the widget level** [11]. We want a maximum of flexibility: any widget can be migrated to any platform at any time. Two widgets from the same running application can be migrated to the same platform, or to two different ones. As the migration is independent for each widget, covering 3a is enough to also cover 3b and 3c, by executing several migrations at the same time.

4.2.3 Orthogonal migration & adaptation

To be useful, a graphical toolkit with migration and adaptation support must still offer a functionality equivalent to a graphical toolkit with no such support. In other words the migration and adaptation are new functionality on top of the pure graphical toolkit functionality. We argue that this new functionality is important enough to be isolated from the pure toolkit functionality. Ext/Tk is consequently designed to provide the migration and adaptation functionality orthogonally to the pure graphical one

1. The migration functionality is provided as a capability of the widget. This capability is a value which can be passed along freely to another process, on another computer: the migration is a distributed operation between different computers connected over the Internet (local migrations on the same computer/process are of course supported). Once a migration capability has been passed to another process, it can be used to trigger the migration of the widget, like a PULL mechanism. To achieve this, the capability serves two purposes: 1) it contains the authority to migrate the widget, and 2) it is a reference to the home site of the widget over the Internet, like a URL for a web page. Because of 2), we often call the migration capability of the widget the reference of the widget. It is the responsibility of the application to pass the capabilities to interested parties: it has the complete control on who receives them. However it does not have the control on when these capabilities are used by the remote peers, i.e. when the migration really occurs. Consequently the application should be as impermeable to the migration process as possible. The only observable effect is a temporary blocking of the threads interacting with the migrated UI. For this reason we say that the migration is transparent to the application. Note that the application can register its interest for migration events if it wants to be notified of the process. Note also that the application has a direct access to the capabilities of the widgets it has created, and consequently can use them to migrate the widget back into its original place.

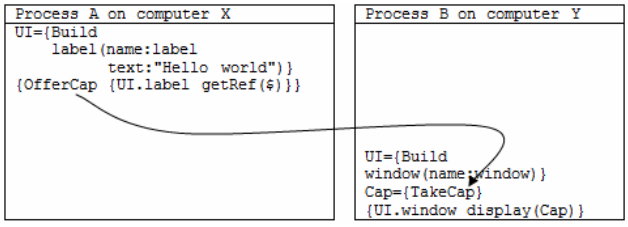


Figure 6. Migration from one platform to another.

This example illustrates the computer X offering the migration capability of a label widget, and the computer Y creating a window, getting the capability, and using it to migrate the label into its local window. Note that the `OfferCap` and `TakeCap` functions are not specified here and can be implemented in numerous different ways. This example assumes that they are able to get in touch with each other, and then exchange the piece of information given to `OfferCap`. A possible implementation is to have a shared file between the processes. Another possible implementation is to rely on emails: `OfferCap` sends an email with the capability attached to it to a mail box that is then read by `TakeCap`. Still another possible implementation is to use a DHT (distributed hash table) based P2P (peer to peer) network, and use a shared name between the two processes to place the value into the network, and obtain it back. And many more implementations are still possible.

2. The adaptation of the widgets consists in changing its representation (presentation and/or interaction) while keeping a useful level of usability. In that sense, the simple reconfiguration of a visual parameter of a widget like its background color is already an adaptation of that widget. Ext/Tk pushes this view forward by introducing a special adaptation parameter to every widget. When this special configuration parameter is changed, it is the whole way the widget is displayed that is changed. Once again, this process is impermeable to the application; the only observable effect is a temporary blocking of the threads interacting with the adapted UI. For this reason we say that the adaptation is *transparent* to the application. Because of this transparency, the application is independent of the representation currently used for a particular widget. For example the target device of a migration could provide its own representation of the widget, adapting it on the fly to its own specifics. Figure 7 illustrates a selector widget that supports different representations. Switching between these representations is achieved by calling the `setContext` method. It is up to the application to define why and when the widget should change the representation. The representation is just changed at different points during the execution unknown to the application.

```
UI={Build window(selector(name:selector
                          text:"Car Model"
                          items:["Ford" "Peugeot" "Renault"]))}

{UI.selector setContext(list)}

...

{UI.selector setContext(default)}
```

Figure 7. Migration from one platform to another.

4.2.4 Distributed structure of a widget

Desktop applications are often centralized applications running the functional core and the UI inside a single process of a computer. Some of them have a distributed functional core (voice over IP applications for example), but that is not what interest us in this work. Once we let parts of the UI migrate from site to site, several devices become involved in the running of the application, and we also shift from a centralized environment into a distributed one. The way Ext/Tk introduces distribution is motivated by two choices:

- Any widget of a running application can be migrated at any time (transparent distribution). Consequently Ext/Tk widgets are distributed entities. At any time they may be situated at the application's process, a remote process, or even nowhere if they are not currently displayed. Later we will see that it is also possible to have several renderers connected to a single proxy, replicating the UI of the widget at several places simultaneously.
- As for the functional core of the application, Ext/Tk does not dictate if it should be distributed or stationary, nor does it offer any support for distribution.

Ext/Tk provides specific distribution support for all widgets, allowing them to dynamically migrate from one site to another. But the widgets are also used by the functional core of the application that interacts with the UI, so part of them should behave in a stationary way. The distribution scheme of widgets is composed of:

- A part that is stationary to the process that created the widget. That part is returned to the functional core of the application so that it can interact with the widget. This part is called the proxy of the widget.
- A part that is distributed, and runs on the site actually displaying the widget. That part is the one the user can interact with. This part is called the renderer of the widget.

In fig. 8, three sites are running on three different computers. Site A creates two widgets that are migrated into Site B. Site C creates one widget that is also migrated into Site B. Each gray area covers a widget in its distributed execution. The proxies stay at the site that created them forever. However the renderers are running at the site the widgets are migrated to. The proxies and the renderers are connected together over the Internet, so as to be synchronized.

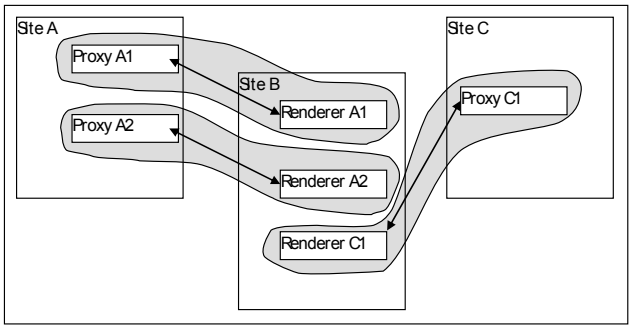


Figure 8. Distributed architecture example.

4.2.5 Runtime architecture

At runtime, each widget is split in two parts: the stationary part that stays at the creator site (the proxy), and the migratable part that is run at a remote site to actually display the widget (the renderer). A notable exception is the top level window widget: the migratable part stays at the creator site; it is created immediately along with the proxy and cannot migrate away. The renderer part of a widget needs a window to be displayed inside, so it can only run at a site were a window proxy is running. Note that the content of a window is a separate widget that can be migrated away. In other words, top level windows provide the physical hook where widgets can be displayed. Also Note that there is no dependency on an external server for this architecture to work. Widget proxies act as servers for their renderers. This is based on the distribution layer of Mozart.

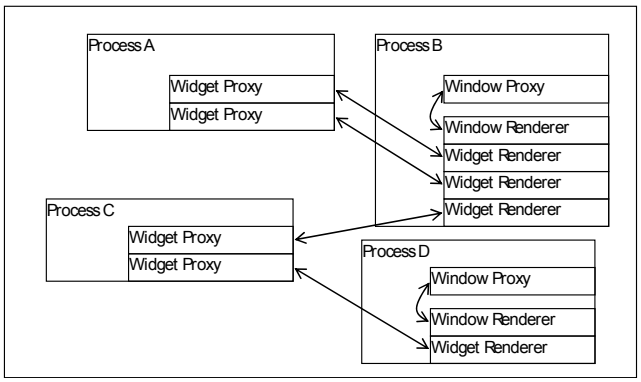


Figure 9. Overview of the runtime architecture.

4.2.6 Trajectory of a universal reference

The universal reference is a capability the creator of the widget can give to a remote site. Typically, an intermediate discovery service allows the sites to exchange these values. Fig. 10 is a typical scenario (dashed arrows are actual connections, plain arrows are the trajectory of the universal reference):

1. Process A running on computer X creates a widget and asks for its migration capability.
2. Process A stores this capability at the discovery service.
3. Process B running on computer Y asks the discovery service for the capability of the widget it wants to display.
4. Process B receives the answer
5. Process B passes it to the proxy of a container widget, here a window.
6. The proxy forwards the capability to its renderer.
7. And lastly the renderer opens a connection with the proxy corresponding to the capability. In section Migration protocol we show how this connection is used for creating a new renderer for this proxy.

Figure 11 displays a more complex scenario where the process B migrates the widget inside a container that is currently displayed at the process C. The migration capability follows the same route as in Figure 10, except that the container proxy forwards the capability to its renderer at process C and not locally anymore.

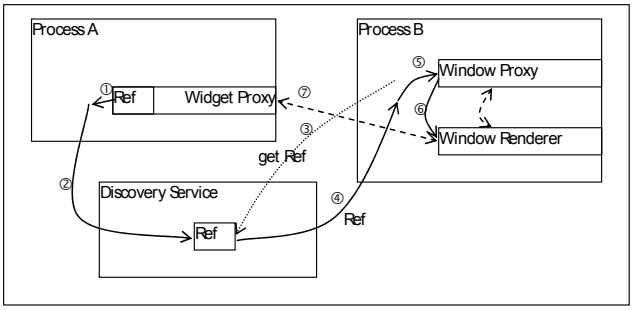


Figure 10. Universal reference trajectory.

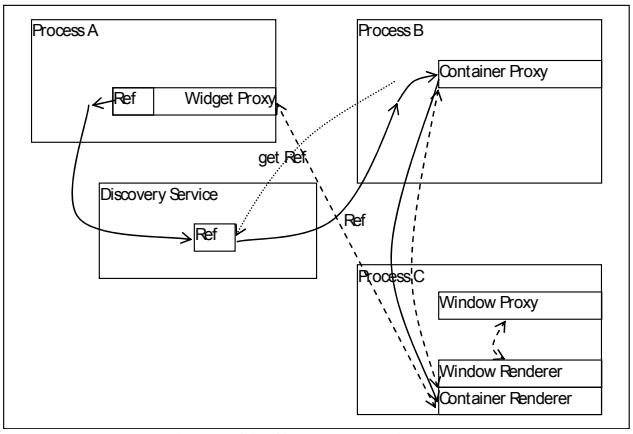


Figure 11. Complex trajectory.

4.2.7 Migration protocol

The migration protocol is a negotiation between the proxy of the receiving container (PC), the proxy of the migrated widget (PM), the renderer of the container (RC) and the renderer of the migrated widget (RM). First, the migration capability of PM has to be given

to PC somehow ①. The migration starts at PC, by using the `importHere` method of its manager using the reference given by PM (the second PI parameter is further placement instructions for example the row/column coordinates of a table container). This method stores this new child; stored children are automatically given to RC ② (either at the child's creation or at the RC creation). RC connects to PM using the reference contained in the capability ③, and in returns PM sends the class definition of the widget renderer ④. RC creates an environment and then asks Ext/Tk to create RM using the class definition just received. If RC fails to create RM (due to PM not responding, or an error while creating RM), RC tells PC to drop this particular child.

In order to create RM, Ext/Tk first creates its manager, connects back to PM ⑤, gets the actual state of all stores ⑥, and then creates the RM object with the manager as parameter ⑦. The initialization of RM should create the actual widget, and update its state according to the current content of the store ⑧ (parameters & event bindings). Once initialized, Ext/Tk automatically calls the methods of RM according to the updates of the store ⑨. If the migrated widget is itself a container, the information necessary to restore its content is in the store it receives from PM, and RM reacts to it like RC after step ②. As a result its content is also migrated along.

Negotiation phase. The step ③ of the protocol above asks a class definition for the renderer and is returned the one currently selected by PM. Indeed there can be different renderers possible for this widget, and the process running PM has selected one of them in particular (using the `setContext` method). However we can extend this protocol further by adding a negotiation phase where RC uses the knowledge of its own available resources (keyboard/mouse presence, screen size...) to hint PM so that it is able to override the current selection for the renderer with another one that is more fit to the device.

The scheme would require:

- A model for describing the platform running the UI.
- Introspection capabilities for renderers determining their level of compatibility with specific platforms.

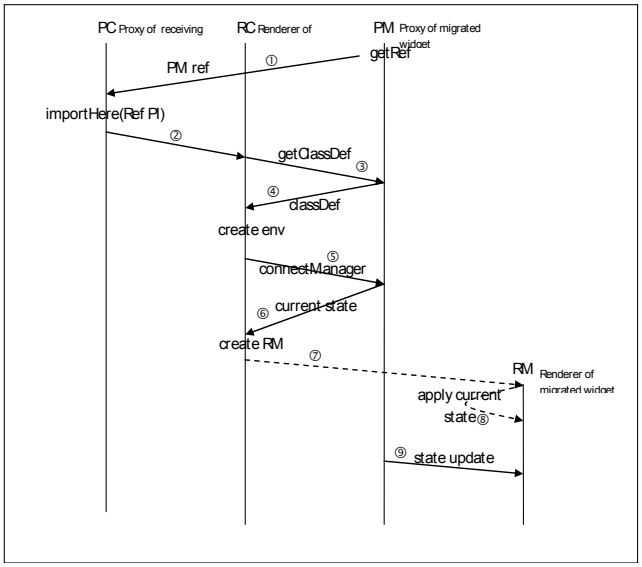


Figure 12. Migration protocol.

Another option is for RC to use its own renderer definition, ignoring the one sent by PM. This may result in an incorrect renderer that is unable to behave correctly with its proxy, however this would open up the possibility of having a target device that adapts the UIs it receives even if the process running those UIs do not know how to adapt them.

Fault tolerance. Network failures can happen at any time, between any of the sites:

- Between PC and PM. There is no direct connection between these two sites: the capability of PM can be brought to PC by a third site.
- Between PC and RC. If message ② cannot be sent be cause of a network failure or because there is currently no RC, then the migration cannot be executed. Nevertheless, the migration instruction is now part of the store of the widget. When a new RC comes in, it will then proceed with the migration of PM. As a result, there may be an arbitrary time between the application command to migrate a widget, and when this command is really executed. If message ② was sent, and there is a network failure between PC and RC then RC eventually disappears. This can happen while the migration protocol is still running, or afterwards. In all cases, the disappearance of RC will result in a disconnection with either PM or with RM. In both situations the migration of RM is cancelled, and it is destroyed if it exists.
- Between RC and PM. The only time this network connection matters is between messages ③ and ⑥. If there is a network failure there, then the migration of PM is aborted. Also RC removes PM from the migration store shared with PC, so that PM is no more considered as a contained widget of PC.
- Between PM and RM. This is the same as between PC and RC.

4.2.8 Final examples

With this toolkit, all the widgets automatically have a migration capability. This capability is controlled by the universal reference of the widget. This universal reference is a simple text string encoding the information needed to find the widget on the Internet. As long as the widget exists, this reference implements the migration capability of the widget. Typically, passing a reference from an application A to an application B is achieved by a discovery service. This service can be implemented in many different ways:

- By human beings, dictating the reference over the phone.
- By email, sending the reference inside an email.
- By using an Internet server, where A registers the reference and B gets it back. This server can be a Web server, an FTP server, or the simple socket server provided by Ext/Tk itself.
- By broadcasting messages over a LAN, allowing B to find A and get the reference. This can be implemented by the Discovery module of Mozart.
- By registering to a peer to peer network and using its functionality to get the reference. This can be implemented by the P2PS module for Mozart.

Figure 13 graphically depicts an example where every DUI component can be distributed: the clock, the buttons, the labels, the calendar, the agenda or even any entry of the calendar and the agenda. Figure 14 represents a case where the clock has been distributed.

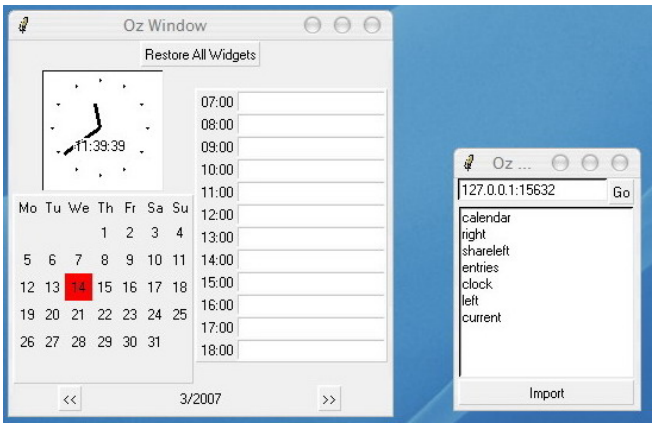


Figure 13. DUI before distribution.

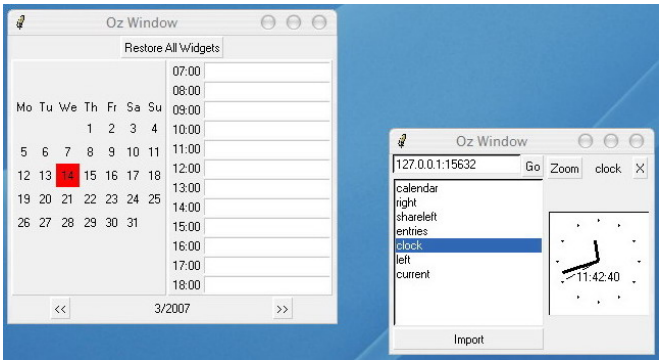


Figure 14. DUI after distribution.

Figures 15 to 17 present another example of distributing portions of a graphical user interface, in this case, a vectorial drawing application. This application is again non-form based and is considered hard to distribute due its tight synchronization between the drawing operations and its effects. Figure 15 presents a screen shot of this application before distribution.

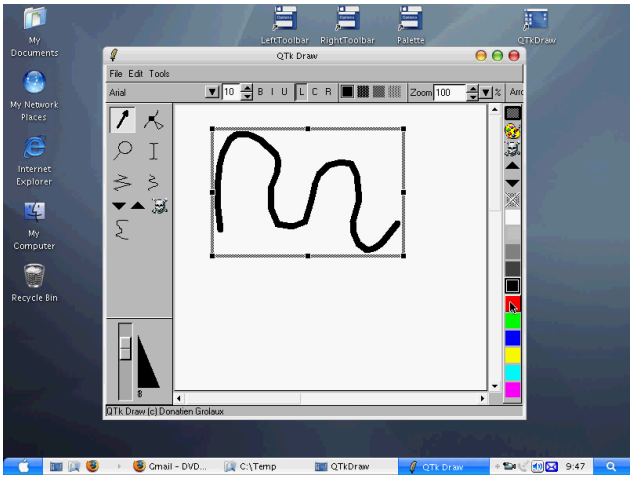


Figure 15. Drawing Application DUI before distribution.

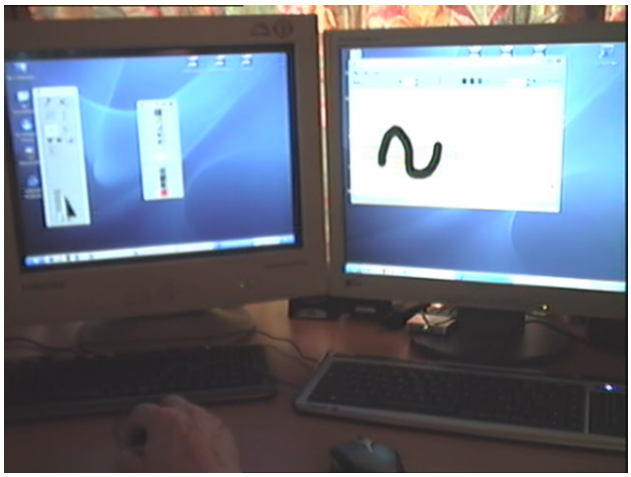


Figure 16. Drawing Application DUI after distribution with another desktop platform.

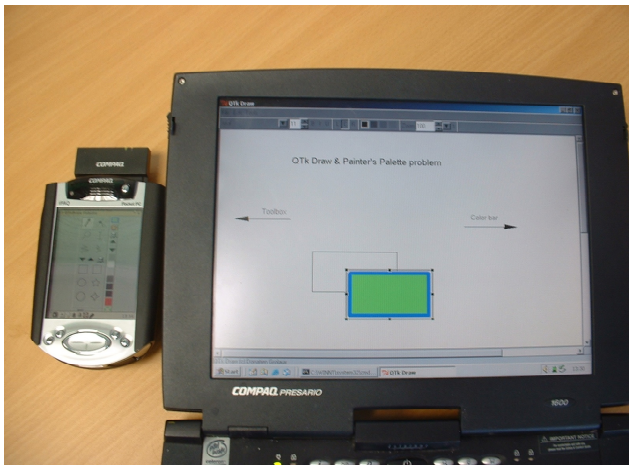


Figure 17. Drawing Application DUI after distribution with another mobile platform.

Figures 16, respectively 17, presents the results of DUI distribution after the toolbars and palettes have been migrated to another desktop, respectively another mobile platform. In this case, the various platforms were used by the same user, but we can also assume that these platforms are used by different users provided that they are connected together via the same network, local or wireless.

5. CONCLUSION

This paper presents a toolkit for peer-to-peer distribution of any graphical UI that supports the following usages: multi-monitor, devices, platform, display, and users. It also presents a detailed description of the software architecture developed for this toolkit. Contrarily to model-based design of multiple UIs [7,8,9, 10], this approach does not rely on any model per se, although each user interface is stored in a distributed way through its properties.

ACKNOWLEDGEMENTS

This work is supported by the SELFMAN (Self Management for Large-Scale Distributed Systems based on Structured Overlay

Networks and Components) European project of the 6th Framework Programme (FP6-IST-2005-034084). We also acknowledge the support of the UsiXML (User Interface extensible Markup Language – http://www.usixml.org) project.

REFERENCES

[1] Ayatsuka, Y., Matsushita, N., and Rekimoto, J. 2000. HyperPalette: a Hybrid Computing Environment for Small Computing devices. In *Proc. of CHI'2000*. ACM Press, New York, pp. 133-134.

[2] Bandelloni, R. and Paternò, F. Migratory user interfaces able to adapt to various interaction platforms. *Int. J. Human-Computer Studies 60*, 5-6 (2004), pp. 621-639.

[3] Bharat, K.A. and Cardelli, L. 1995. Migratory Applications Distributed User Interfaces. In *Proc. of UIST'95* (Pittsburgh, Nov. 1995), ACM Press, New York, pp. 132-142.

[4] Chung, G. and Dewan, P. 2004. Towards Dynamic Collaboration Architectures. In *Proc. of the ACM Conf. on Computer Supported Cooperative Work CSCW'2004*, pp. 1-10.

[5] Dewan, P. and Choudhary, R. Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Computer-Human Interaction 5*, 1 (1998), pp. 34-62.

[6] Distributed Programming in Mozart - A Tutorial Introduction, chapter 3: Basic Operations and Examples, accessible at http://www.mozart-oz.org/documentation/dstutorial/node3.html#chapter.examples

[7] Eisenstein, J., Vanderdonckt, J., and Puerta, A. 2001. Model-Based User-Interface Development Techniques for Mobile Computing. In *Proc. of IUI'01* (Santa Fe, January 14-17, 2001), ACM Press, New York, pp. 69-76.

[8] Griffiths, T., Barclay, P.J., Paton, N.W., McKirdy, J., Kennedy, J., Gray, P.D., Cooper, R., Goble, C.A., and Pinheiro, P. Teallach: a Model-based User Interface Development Environment for Object Databases. *Interacting with Computers 14*, 1 (December 2001), pp. 31-68.

[9] Grolaux, D., Van Roy, P., and Vanderdonckt, J. 2004. Migratable User Interfaces: Beyond Migratory User Interfaces. In *Proc. of 1st IEEE-ACM Annual Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services MOBIQUITOUS'04*, pp. 422-430.

[10] Grolaux, D., Vanderdonckt, J., and Van Roy, P. 2005. Attach me, Detach me, Assemble me like You Work. In *Proc. of INTERACT'05*, pp. 198-212.

[11] Grolaux, D. 2007. *Transparent Migration and Adaptation in a Graphical User Interface Toolkit*, Ph.D. dissertation, Department of Computing Science and Engineering, Université catholique de Louvain, 2007.

[12] Grudin, J. 2001. Partitioning digital worlds: focal and peripheral awareness in multiple monitor use. In *Proc. of CHI'01*, ACM Press, New York, pp. 458-465.

[13] Han, R., Perret, V., and Naghsineh, M. 2000. WebSplitter: A Unified XML Framework for Multi-Device Collaborative Web Browsing. In *Proc. of the ACM Conf. on Computer Supported Cooperative Work*, pp. 221-230.

[14] Hutchins, R., Meyers, B., Smith, G., Czerwinski, M., and Robertson, G. 2004. Display Space Usage and Window Management Operation Comparisons between Single Monitor and Multiple Monitor Users. In *Proc. of AVI'04*, ACM Press, New York, pp. 32-39.

[15] Hutchings, H.M. and Pierce, J.S. 2006. Understanding the Whethers, Hows, and Whys of Divisible Interfaces. In *Proc. of AVI'06*, ACM Press, New York, pp. 274-277.

[16] Loeser, C., Mueller, W., Berger, F., and Eikerling, H.-J. 2003. Peer to peer networks for virtual home environments, in *Proc of HICSS-36*, IEEE Computer Society Press.

[17] Luyten, K. and Coninx, K. 2005. Distributed User Interface Elements to support Smart Interaction Spaces. In *Proc. of the 7th IEEE Int. Symposium on Multimedia*, IEEE Comp. Society, Washington, DC, pp. 277-286.

[18] Luyten, K., Vandervelpen, Ch., and Coninx, K. 2002. Migratable User Interface Descriptions in Component-Based Development, in *Proc. of DSV-IS'2002*, pp. 44-58.

[19] Luyten, K., Van den Bergh, J., Vandervelpen, Ch., and Coninx, K. 2006. Designing distributed user interfaces for ambient intelligent environments using models and simulations. *Computers & Graphics 30*, 5 (2006) 702-713.

[20] Mesaros, V., Carton, B., and Van Roy, P. 2004. P2PS: Peer-to-Peer Development Platform for Mozart. In *Proc. of Second International Mozart/Oz Conference MOZ'04*. LNCS, Vol. 3389, Springer, Berlin, pp. 125-136.

[21] Myers, B.A. Using Handhelds and PCs Together. *Communications of the ACM 44*, 11 (2001), pp. 34-41.

[22] Tan, D.S. and Czerwinski, M. 2003. Effects of Visual Separation and Physical Discontinuities when Distributing Information across Multiple Displays. In *Proc. of INTERACT'03,* IOS Press, pp. 252-260.

[23] Vanderdonckt, J., Furtado, E., Furtado, V., Limbourg, Q., Silva, W., Rodrigues, D., and Taddeo, L. 2001. Multi-model and Multi-level Development of User Interfaces, in "Multiple User Interfaces - Cross-Platform Applications and Context-Aware Interfaces", John Wiley & Sons, pp. 193-216.

[24] Vandervelpen, Ch., Vanderhulst, G., Luyten, K., and Coninx, K. 2005. Light-Weight Distributed Web Interfaces: Preparing the Web for Heterogeneous Environments. In *Proc. of ICWE 2005*, pp. 197-202.

[25] Van Roy, P. and Haridi, S. 2004. Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge.

[26] Yanagida, T. and Nonaka, H. Architecture for Migratory Adaptive User Interfaces. In *Proc. of CIT'2008*, pp. 450-455.

A.22 Decentralized Transactional Collaborative Drawing

Decentralized transactional collaborative drawing

Jérémie Melchior[1], Boris Mejías[2], Yves Jaradin[2], Peter Van Roy[2], Jean Vanderdonckt[1]
[1]Information Systems Unit
[2]Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
{$firstname.lastname$}@uclouvain.be

Abstract

When multiple users collaboratively edit a vector image, avoiding conflicts requires synchronizing exclusive access to the objects of the image. This synchronization needs a true concurrency control algorithm. One of the most common strategy to achieve this synchronization is to use a centralized architecture where a single server becomes the transactional manager. Unfortunately, a central point of control is also a single point of failure. This paper proposes a decentralized architecture based on a peer-to-peer network providing decentralized transactional support with replicated storage. As a consequence, there is a gain in fault-tolerance and the transactional protocol eliminates the problem of network delay improving usability and network transparency. The same result can be applied to text edition and other collaborative editing tasks.

1. Introduction

There are many software applications supporting collaborative work, such as drawing, text editing or software development. Collaborative work can be done synchronously or asynchronously. In the later case, the participants make their modifications on their local copy without direct interaction with the other participants. Once the changes are made, they are committed to the global state. In the former case, which is the focus of this paper, all participants are concurrently working on a shared working space. Such scenario requires continuous synchronization of the participants in order to avoid conflicts. One way of achieving such synchronization is by letting the participants lock the part of the shared space they want to modify, granting exclusive access to that part. Since all participants can take any lock, having a single point of control make sense, resulting in the classical client-server architecture. Unfortunately, it is well known that having a single point of control also means having a single point of failure, because the whole application relies on the stability of the server.

Transdraw [5] is a distributed collaborative vector-based graphical editor with a shared drawing area. Each user runs the application and joins a server to get access to the shared area. When someone is drawing in this area, feedback is sent to other users reflecting the action. In addition, TransDraw uses a transactional protocol to allow users to make optimistic changes on the drawing with immediate conflict resolution. This feature eliminates the problem of performance degradation caused by network latency and it is a crucial property of TransDraw. The synchronization and storage of the global state is done on a server which centralizes the control of the work flow. When users modify an object on the drawing, they request exclusive access for it, which may succeed or fail depending on the behaviour of the other users. All this is reflected graphically in the shared drawing space.

As we have mentioned, a problem of TransDraw, due to its centralized architecture, is its dependency on the server. If the server crashes the work is lost, and the application will not run until the server is rebooted.

Peer-to-peer networks have the nice property of being self-organized, fault-tolerant and fully decentralized. We propose in this paper to redesign the transactional protocol of TransDraw to overcome the problem of the single point of failure. In order to do that, we use Beernet [2], a structured peer-to-peer overlay network providing a fault-tolerant distributed transaction layer with replicated storage. Every time a user attempts to modify a graphical object, this modification will be done inside a transaction with a different transaction manager, which is replicated to allow the transaction to finish in case of failure of the manager. Unfortunately, this fault-tolerance mechanism is not free. Replication requires a higher usage of network resources increasing latency of transactions, but the optimistic approach for starting the modification of an object counteract the latency. We consider this a small drawback because functionality of TransDraw is fully respected and there is an important gain in fault-tolerance.

What follows is a more detailed description of TransDraw and related works in sections 2 and 3. Beernet is described in section 4. The core of the proposal is explained in section 5, being followed by conclusions.

2. TransDraw

2.1 Description

Transdraw is a collaborative vector drawing tool created by Donatien Grolaux using transactions[5]. The toolbar provides, not only the traditional tools of vector editing (eg. lines, ellipse, rectangles), but also a pair of tools supporting collaboration. As soon as a user selects an object, a request is sent to the server for the corresponding lock. However, the user is permitted to edit the object optimistically before the server can answer the request. The optimistic nature of the operation is visually presented to the user by red selection handles. When the server grants the lock, the transaction on the object is committed and the user can continue to edit the object in exclusive mode, indicated by black selection handles until he deselect it at which time the lock will be returned. If the lock was already held by another user, the server has to refuse it to the user and the transaction is aborted. The user see the modification he did optimistically undo themselves and the object is deselected.

A user can also manage explicitly his locks by using the *"take lock"* tool, for example to make a complex reorganisation of the drawing, involving several individual objects. He then has to release the locks manually using the flashing *"release-locks"* button.

In order to prevent starvation which could happen as simply as by a user inadvertently selecting every object before taking a rest, a lock stealing mechanism is provided. The *"steal lock"* tool make a request to steal a lock to the server which forwards it to the current owner of the lock. This user then as a few seconds to accept or reject the stealing of her locks. On timeout, the stealing is considered accepted. Once accepted, the previous owner notifies the server to forward the lock to the stealer.

2.2 Example scenario

Figure 1, presents the view of two users working on the same drawing, each in his own window. Bob, on the right, had the top ellipse selected long enough for the server to grant him the lock has can be seen by the black selection handles around it. Alice, on the left has just tried to select this ellipse. After a, normally brief, period during which she was able to do optimistic changes to this ellipse, her transaction is aborted, and she is notified of it by the disappearance of her selection and the red dot on the ellipse which will

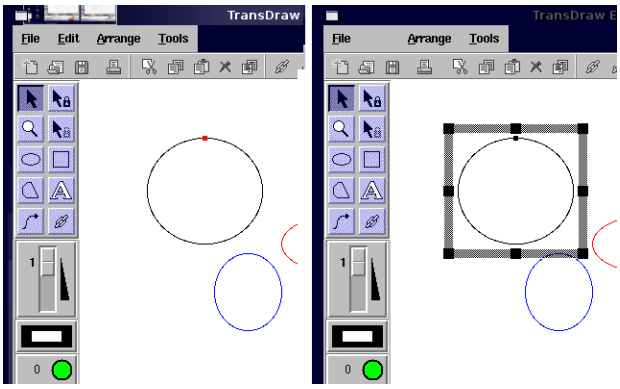


Figure 1. Alice, on the left, see a locked and non-editable ellipse while Bob has it is selected and editable.

blink a few times to explain that Bob is a currently editing this object.

The diagram in Figure 2 describes a possible continuation of the scenario in which Alice steals the lock from Bob to perform the update she wants. Alice ask to steal the lock to the server. Since Bob currently has the lock, the server ask Bob whether he allows his lock to be stolen or not. This is shown to Bob as two blinking buttons at the bottom of his edition window as we can see in Figure 3. If Bob allows his lock to be stolen, either explicitly or by ignoring the request long enough, he loose selection of the object and possession of the lock and the server transfer them to Alice.

Of course, all of this assumes that the server does not crash...

3. Related works

There are some applications that already support collaboration in different ways. We describe and comment some of them briefly.

3.1 BOUML

Some researchers have released an application to provide an easy to use and free UML tool, named BOUML[8]. It allows drawing diagrams and generating code in multiple languages. The tool has been developed as a multiuser application in a sequential way. Each user of the application must choose an identifier which allows working on some diagrams. The work may be done in parallel but there is not any feedback on other users work as there is no support for concurrent work. There are many problems with the tool. The lack of feedback prevents user to know what others are doing and to see their changes. It is also impossible to know

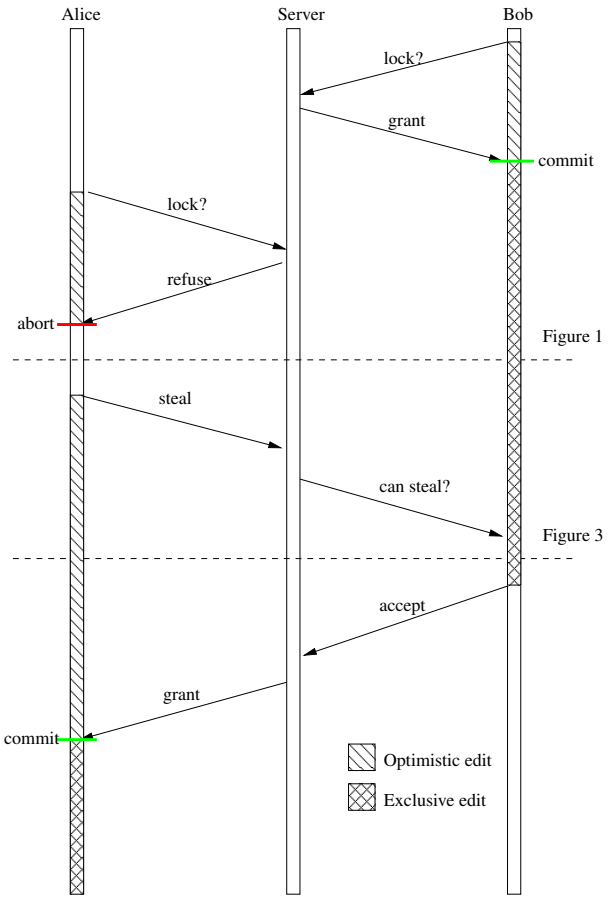


Figure 2. Scenario of complex interaction

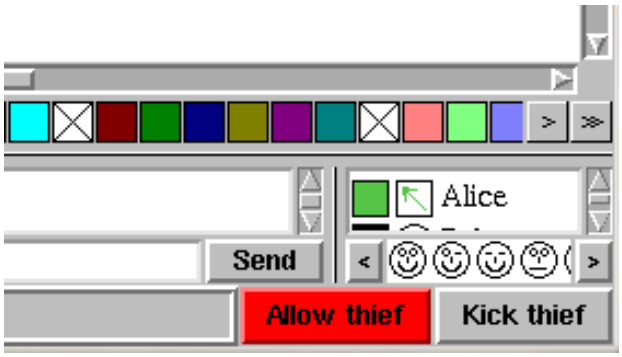


Figure 3. Bob is asked whether he allows his lock to be stolen.

which files are currently being modified or that have been modified and saved. There can be conflicts when saving the project. When users are working collaboratively, the work of a user will be saved but not all the modification of other users. This leads to irreversible lost work without any warning. Another problem is the impossibility to lock part of the work to prevent modification from another user.

3.2 Gobby

Gobby is a free text-editor that allows collaborative work [1]. It supports multiuser parallel edition on multiple documents and a multiuser chat. A user has to start a session and create the documents, he will host the server needed to centralize the information. Other users must choose a name and a color and connect to the server host. The collaboration between all the users is simple thanks to the feedback brought to users with colors. A list of users allows knowing the color of each editor. The application has the ability to recognize patterns of many different text formats and enable syntax coloration. As the BOUML application, Gobby does not support any lock of some part of the text and all the users can edit what they want. This is not a major issue since other users can observe the changes in real time and the team work may rely on trusted users. Nevertheless there is a problem when the server crashes. All the unsaved modifications can be saved by another user but the whole process of creating a server and joining the server must be restarted.

3.3 Google Docs

Google Docs [3] is an online office suite that allows multiple users to modify the same file at the same time. On particular feature, similar to TransDraw, can be seen on spreadsheets. One a user is modifying a cell, this one is coloured differently as in any single user spreadsheet application. When other users connect to Google servers to edit the same file, then, the cells they select will appear with a different colour on the view of the other users, and with a tag identifying the user. Instead of locking the cell, changes are save incrementally using versioning. Google Docs uses also a centralized architecture because everything is control at Google side. But, there is a very important difference. There is not only one server to rely on, but a set of servers with replicated information, so if a server crashes, another one takes over. Of course, these are only conjectures about Google's back-end.

4. Decentralized transactional DHT

Beernet [2] is a structured overlay network providing a distributed hash table (DHT) with symmetric replication.

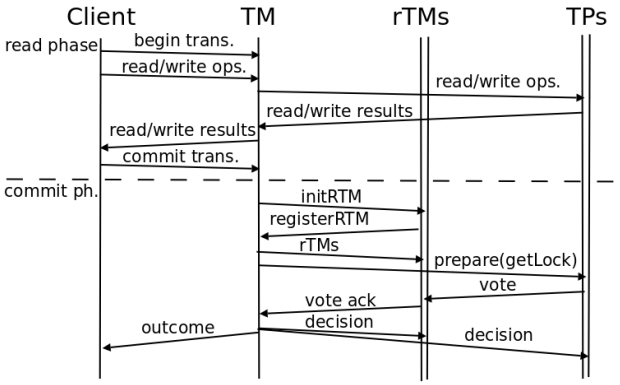


Figure 4. Paxos consensus protocol for distributed transactions.

Peers are self-organized using the relaxed-ring topology [6], which is derived from Chord [10], with cost-efficent ring maintenance and self-healing properties. Data replication is guaranteed with a decentralized transactional protocol allowing the modification of different items within a single transaction. The transactional protocol implements a Paxos-consensus algorithm [7, 4], with requires the agreement of the majority of peers holding the replicas of the items. We will focus on the transactional layer of Beernet because it will be our mean to decentralize TransDraw.

Figure 4 describes how the Paxos-consensus protocol works. The client, which is connected to a peer that is part of the network, triggers a transaction in order to read-/write some items from the global store. When the transaction begins, the peer becomes the transaction manager (TM) for that particular transaction. The whole transaction is divided in two phases: *read phase* and *commit phase*. During the *read phase*, the TM contact all transaction participants (TPs) for all the items involved in the transaction. TPs are chosen from the peers holding a replica of the items. The modification to the data is done optimistically without requesting any lock yet. Once all the read/write operations are done, and the client decides to commit the transaction, the *commit phase* is started.

In order to commit the changes on the replicas, it is necessary to get the lock of the majority of TPs for all items. But, before requesting the locks, it is necessary to register a set of replicated transaction managers (rTMs) that are able to carry on the transaction in case that the TM crashes. The idea is to avoid locking TPs forever. Once the rTMs are registered, the TM sends a $prepare$ message to all participants. This is equivalent to request the lock of the item. The TPs answer back with a $vote$ to all TMs (arrow to TM removed for legibility). The vote is acknowledged by all rTMs to the leader TM. Like that, the TM will be able to take a decision if the majority of rTMs have enough information to take exactly the same decision. If the TM crashes at this point, another rTM can take over the transaction. The decision will be *commit* if the majority of TPs voted for commit. It will be *abort* otherwise. Once the decision is received by the TPs, locks are released.

The protocol provides atomic commit on all replicas with fault tolerance on the transaction manager and the participants. As long as the majority of TMs and TPs survives the process, the transaction will correctly finish. These are very strong properties that will allows us to run TransDraw on a decentralized system without depending on a server.

5. Decentralized TransDraw

Our conjecture about the way Google Docs is designed in order to provide fault-tolerance is strongly based on replication and the possibility of replacing a crashed server with another machine. Not having Google's infrastructure, we can achieve replication and fault-tolerance by building TransDraw on top of a peer-to-peer network, and by decentralizing the synchronization of locks and data storage. Our proposal is to build TransDraw on top of Beernet.

Peers are self-organized using the relaxed-ring topology implemented by Beernet. Data is stored using the DHT with symmetric replication. The transactional layer provides synchronized access to the shared state solving conflicts due to race conditions. But the Paxos-consensus protocol as described in section 4 is not sufficient to provide exactly the same functionality of TransDraw as it was described in section 2. The main difference lies on the moment where the locks are granted. As it is currently, locks are granted too late for TransDraw, because it is not possible to inform users about the intention of the others.

The first modification we have to do to the transactional protocol is to allow eager locking request. One idea is to request the locks when read/write operations are sent to the transaction participants during the *read-phase*. If locks are not granted, the transaction is immediately aborted. The problem introduced by this modification is that if leader TM crashes after requesting the locks, there is no rTM yet to take over the transaction, and items would be locked forever. Considering this, the registration of rTMs must also be moved up to the read-phase. After this two modifications we realized that in fact it is better to avoid the read-phase and start immediately with a extended commit phase that first needs to gather the participants.

The second modification is an eager notification mechanism. Currently, out transactional layer is meant for asynchronous access to the share state. When a peer write a new value for item, other users are not notified unless they read the item. In the case of TransDraw, other users needs to be notified not only of every modification on the value of items, but also on the intention of other users when they

lock items. To achieve this, the leader must broadcast its decision to the network once it get enough locks, and once the final decision is taken.

Note that eager locking and the notification mechanism are only needed on synchronous collaborative work. If the collaborative application relies on asynchronous collaboration it is enough with the Paxos protocol presented in the previous section. Scalaris [9] is an implementation of Wikipedia running on top a structured overlay network with Paxos transactional protocol. This shows that we could already add fault-tolerance and decentralization to TransDraw if the goal is work on asynchronous fashion. The suggested modifications are meant for achieving real-time collaborative work.

6. Conclusion and Future Work

We have seen that several synchronous collaborative applications are currently based on centralized synchronization. This strategy is efficient but not fault-tolerant because it strongly relies on the stability of the server. Some applications achieve fault-tolerant by replicating the state of the server, but this requires a more sophisticated infrastructure and it is still inherently centralized. Single point of control is a single point of failure.

We propose to implement these kind of applications on top of structured overlay networks with symmetric replication, and a transactional layer based on consensus. This strategy provides synchronization and fault-tolerance by decentralizing the control of the work flow. We present our approach by taking the TransDraw application and the Beernet peer-to-peer network.

Beernet as is, can help to decentralize asynchronous collaborative applications. In order to achieve the functionality of TransDraw, which is synchronous, eager locking and a notification mechanism needs to be added to the current transactional protocol.

We still need to study in detail the new transactional protocol, implement it and compare the performance with the centralized approach. We expect to have a small degradation in performance at the level of the transactional protocol due to replication cost, but with a huge gain in fault-tolerance. There is no degradation in performance for the user in case of no conflicts, because its changes are done optimistically, eliminating the problem of network latency.

7. Acknowledgements

This work has been mainly funded by the European Commission FP6 IST Project SELFMAN (Contract 034084), with support of the UsiXML project.

References

[1] 0x539 dev group. The gobby collaborative editor. http://gobby.0x539.de, 2009.

[2] Distoz group. Beernet - the relaxed peer-to-peer network. http://beernet.info.ucl.ac.be, 2009.

[3] Google. Google docs. http://docs.google.com, 2009.

[4] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[5] D. Grolaux. Editeur graphique réparti basé sur un modéle transactionnel, 1998. Mémoire de Licence.

[6] B. Mejías and P. Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *XXVI International Conference of the Chilean Computer Science Society*. IEEE Computer Society, November 2007.

[7] M. Moser and S. Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.

[8] B. Pagès. The bouml tool box. http://bouml.sourceforge.net, 2009.

[9] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for distributed wikis on structured overlays. In A. Clemm, L. Z. Granville, and R. Stadler, editors, *DSOM*, volume 4785 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 2007.

[10] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

A.23 Decentralized Transactional Collaborative Drawing *(demo)*

Decentralized Transactional Collaborative Drawing*

Boris Mejías, Jérémie Melchior, Yves Jaradin
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

1 Demonstrator

DeTransdraw is a decentralized collaborative vector-based graphical editor with a shared drawing area. It provides synchronous collaboration between users with graphical support for notifications about other users' activities. Conflict resolution is achieved with a decentralized transactional service with storage replication, and self-management replication for fault-tolerance. The transactional service also allows the application to prevent performance degradation due to network latency, which is an important feature for synchronous collaboration.

DeTransDraw is a redesign of TransDraw [1], a client-server application providing similar features. Due to its centralized architecture, TransDraw has a single point of failure and does not scale beyond the capacity of the server. DeTransDraw is built on top of a peer-to-peer network, Beernet [4], allowing users to join and leave the application at any time, without relaying on any central point of control. The decentralized architecture of DeTransDraw makes it more scalable and fault-tolerant. Other collaborative applications, either synchronous or asynchronous, can benefit from these properties by reusing the transactional layer over an equivalent peer-to-peer network.

The transactional service we use is based on an eager protocol that is an adaptation of Paxos consensus algorithm [3]. The peer-to-peer network we built uses the relaxed-ring topology [2].

During the demonstration we will built an ad-hoc peer-to-peer network that will be interfaced by three clients running on three different computers. The three clients will run the graphical interface of DeTransDraw, accessing the shared drawing area. Apart from simple drawing actions, conflict resolution will be tested by trying to modify the same graphical objects by more that one client. Fault-tolerance will be tested by killing some of the peers during the drawing actions. For the demonstrations we will need space and power to set up three laptops and a router.

*This research is funded by SELFMAN (contract number: 034084).

2 Innovations

- Replicated storage achieved by decentralized transaction over peer-to-peer networks providing distributed hash table (DHT), providing eager notifications to the participants of a collaborative application
- Prevention of performance degradation due to network latency. Users work on the application almost as if it was a local application.
- Self-management of storage achieved with symmetric replication over a structured overlay network.
- Self-healing of transactions participants. A transaction always terminate if the majority of the peers is alive during the execution. Fault-tolerance is guaranteed depending on the majority.

References

[1] Donatien Grolaux. Editeur graphique réparti basé sur un modéle transactionnel, 1998. Mémoire de Licence.

[2] Boris Mejías and Peter Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *XXVI International Conference of the Chilean Computer Science Society*. IEEE Computer Society, November 2007.

[3] Monika Moser and Seif Haridi. Atomic commitment in transactional dhts. In *Proceedings of the CoreGRID Symposium*, CoreGRID series. Springer, 2007.

[4] Programming Languages and Distributed Computing Research Group, UCLouvain. Beernet: pbeer-to-pbeer network - *http://beernet.info.ucl.ac.be*.

Bibliography

[1] RPM Package Manager (RPM) v4 Homepage.

[2] Kompics: Reactive Component Model for Distributed Computing. http://kompics.sics.se, 2009.

[3] The p2psim simulator. http://pdos.csail.mit.edu/p2psim/, 2009.

[4] The peersim simulator. http://peersim.sourceforge.net/, 2009.

[5] Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov, and Per Brand. Distributed control loop patterns for managing distributed applications. In *SASO SELFMAN Workshop*, October 2008.

[6] Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, and Seif Haridi. A design methodology for self-management in distributed environments. GRID4ALL project.

[7] Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, simulating, and deploying peer-to-peer systems using the Kompics component model. In *COMmunication System softWAre and middlewaRE (COMSWARE)*, Dublin, Ireland, 2009.

[8] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Swedish Institute of Computer Science (SICS), November 2003.

[9] Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, 1996. Erlang system available at `www.erlang.org`.

[10] W. B. Arthur. Complexity in economic theory: Inductive reasoning and bounded rationality. *The American Economic Review*, 84(2):406–411, May 1994.

[11] D. P. Bertsekas. The auction algorithm: a distributed relaxation method for the assignment problem. *Ann. Oper. Res.*, 14(1-4):105–123, 1988.

[12] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the ACM SIGCOMM '04 Symposium on Communication, Architecture, and Protocols*, OR, USA, Mar. 2004. ACM Press.

[13] Aaron B. Brown, Joseph L. Hellerstein, Matt Hogstrom, Tony Lau, Sam Lightstone, Peter Shum, and Mary Peterson Yost. Benchmarking autonomic capabilities: Promises and pitfalls. In *Proceedings of 1st International Conference on Autonomic Computing*, pages 266–267, New York, NY, USA, May 17-19 2004. ICAC, IEEE Computer Society.

[14] Aaron B. Brown and Charlie Redlin. Measuring the effectiveness of self-healing autonomic systems. In *Proceedings of 2nd International Conference on Autonomic Computing*, pages 328–329, Seattle, WA, USA, June 13-16 2005. ICAC, IEEE Computer Society.

[15] Alexandre Bultot. A survey of systems with multiple interacting feedback loops and their application to programming. Technical report, Université catholique de Louvain, 2009. In preparation.

[16] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems, Second Edition*. Springer, 2008.

[17] Huoping Chen and Salim Hariri. An evaluation scheme of adaptive configuration techniques. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 493–496, Atlanta, Georgia, USA, November 5-9 2007. ASE, ACM Press, New-York, NY.

[18] Raphaël Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Dept. of Computing Science and Engineering, Université catholique de Louvain, December 2007.

[19] Tom De Wolf and Tom Holvoet. Evaluation and comparison of decentralised autonomic computing systems. CW Reports CW437, Department of Computer Science, K.U.Leuven, Leuven, Belgium, March 2006.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's

highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 205–220. ACM, 2007.

[21] Jim Dowling and Cosmin Arad. *Kompics Programming Manual*. Swedish Institute of Computer Science and Kungliga Tekniska Högskola, 2009. Available at `kompics.sics.se`.

[22] A. Flissi, J. Dubus, N. Dolet, and P. Merle. Deploying on the Grid with DeployWare. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*. IEEE Computer Society, 2008.

[23] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[24] Thomas Forse. *Qualimtrie des systmes complexes - Mesure de la qualit du logiciel*. Paris, editions d'organisation edition, November 1989.

[25] The Apache Software Foundation. Apache http server, 2009. Available at `www.apache.org`.

[26] E. Gat. Three-layer architectures. In *Artificial Intelligence and Mobile Robots*. MIT/AAAI Press, 1997.

[27] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.

[28] P. B. Godfrey and I. Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *Proc. of the 24th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM'05)*, FL, USA, March 2005. IEEE Comp. Society.

[29] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, Andrew A. Farrell, A. Lain, P. Murray, and P. Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1), 2009.

[30] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[31] The PHP Group. PHP: Hypertext Preprocessor, 2009. Available at `www.php.net`.

[32] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2006.

[33] Felix Halim, Rajiv Ramnath, Sufatrio, Yongzheng Wu, and Roland H. C. Yap. A lightweight binary authentication system for windows. *Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM 2008)*, 2008.

[34] Felix Halim, Yongzheng Wu, and Roland H. C. Yap. Security issues in small world network routing. *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2008.

[35] Felix Halim, Yongzheng Wu, and Roland H.C. Yap. Small world networks as (semi)-structured overlay networks. *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, 2008.

[36] Felix Halim, Wu Yongzheng, and Roland H.C. Yap. Wiki credibility enhancement. In *Fifth International Symposium on Wikis and Open Collaboration (WikiSym)*, to appear, 2009.

[37] Garrett Hardin. The tragedy of the commons. *Science*, 162(3859):133–160, December 1968.

[38] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, August 2004.

[39] Mikael Högkvist, Seif Haridi, Nico Kruber, Alexander Reinefeld, and Thorsten Schütt. Using global information for load balancing in DHTs. In *SASO SELFMAN Workshop*, October 2008.

[40] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology, 2001.

[41] M. Jelasity, A. Montresor, and Ö. Babaoglu. Gossip-based Aggregation in Large Dynamic Networks. *ACM Trans. on Computer Systems (TOCS)*, 23(3), August 2005.

[42] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In Geoffrey M. Voelker and Scott Shenker, editors, *IPTPS*, volume 3279 of *Lecture Notes in Computer Science*, pages 131–140. Springer, 2004.

[43] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[44] Jeong-Rae Kim, Yeoin Yoon, and Kwang-Hyun Cho. Coupled feedback loops form dynamic motifs of cellular networks. *Biophysical Journal*, 94:359–365, January 2008.

[45] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. *32nd ACM Symposium on Theory of Computing*, 2000.

[46] Tetsuya Kobayashi, Luonan Chen, and Kazuyuki Aihara. Modeling genetic switches with positive feedback loops. *J. Theor. Biol.*, 221:379–399, 2003.

[47] Jeff Kramer and Jeff Magee. Self-managed systems: An architectural challenge. In *Workshop on the Future of Software Engineering (FOSE 2007)*, pages 259–268, May 2007.

[48] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, pages 233–242, New York, NY, USA, 2002. ACM Press.

[49] Paul Lin, Alexander MacArthur, and John Leaney. Defining autonomic computing: A software engineering perspective. In *Proceedings of 16th Australian Software Engineering Conference*, pages 88–97, Brisbane, Australia, March 31 - April 1 2005. ASWEC, IEEE Computer Society.

[50] Y. Liu and S. Smith. A Formal Framework for Component Deployment. In *20th ACM Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[51] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, New York, NY, USA, 2002. ACM Press.

[52] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR metric. In *Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, Lecture Notes in Computer Science (LNCS), pages 53–65, London, UK, 2002. Springer-Verlag.

[53] Julie A. McCann and Markus C. Huebscher. Evaluation issues in autonomic computing. In Hai Jin, Yi Pan, and Nong Xiao, editors, *Proceedings of Grid and Cooperative Computing - GCC 2004 Workshops: GCC 2004 International Workshops, IGKG, SGT, GISS, AAC-GEVO, and VVS*, volume 3252 of *Lecture Notes in Computer Science*, pages 597–608, Wuhan, China, October 21-24 2004. GCC, Springer.

[54] Boris Mejías. Beernet: The relaxed beer-to-beer network. Université catholique de Louvain, 2009. Available at `beernet.info.ucl.ac.be`.

[55] Boris Mejías, Alfredo Cádiz, and Peter Van Roy. Beernet: RMI-free peer-to-peer networks. In *First International Workshop on Distributed Objects for the 21st Century*, July 2009.

[56] Boris Mejías and Donatien Grolaux. DeTransDraw: Decentralized transactional collaborative drawing. Demonstrator, Internet of Services 2009 Collaboration Meeting, June 2009.

[57] Boris Mejías and Peter Van Roy. A relaxed-ring for self-organising and fault-tolerant peer-to-peer networks. In *SCCC '07: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.

[58] Boris Mejías and Peter Van Roy. The relaxed-ring: A fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, September 2008.

[59] E. Le Merrer, A.-M Kermarrec, and L. Massoulie. Peer to peer size estimation in large and dynamic networks: A comparative study. In *Proc. of the 15th IEEE Symposium on High Performance Distributed Computing*, pages 7–17. IEEE, 2006.

[60] Drazen Milicic. *Software quality models and philosophies*, chapter 1, page 100. Blekinge Institute of Technology, June 2005.

[61] Monika Moser and Seif Haridi. Atomic commitment in transactional DHTs. In *CoreGRID Symposium*, August 2007.

[62] Mozart Consortium. Mozart programming system version 1.4.0, July 2008. Available at `www.mozart-oz.org`.

[63] The OSGi Alliance. *OSGi Service Platform Release 4, Version 4.1 - Core Specification*, April 2007.

[64] Walamitien H. Oyenan and Scott A. DeLoach. Design and evaluation of a multiagent autonomic information system. In *IAT*, pages 182–188, Silicon Valley, CA, USA, November 2-5 2007. IAT'07, IEEE Computer Society.

[65] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. Transactions for distributed wikis on structured overlays. In Alexander Clemm, Lisandro Zambenedetti Granville, and Rolf Stadler, editors, *DSOM*, volume 4785 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 2007.

[66] Programming Languages and Distributed Computing Research Group, UCLouvain. P2PS: A peer-to-peer networking library for Mozart-Oz - *http://p2ps.info.ucl.ac.be*, 2008.

[67] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[68] J. Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols, 2007.

[69] J. Rosenberg, R. Mahy, and C. Huitema. Traversal using relay NAT (TURN). Internet-Draft http://www.jdrosen.net/midcom_turn.html, September 2005.

[70] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[71] Mazeiar Salehie and Ladan Tahvildari. Autonomic computing: emerging trends and open problems. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[72] Oskar Sandberg. Distributed routing in small-world networks. *The Eighth Workshop on Algorithm Engineering and Experiments (ALENEX06)*, 2006.

[73] Thorsten Schütt. Scalaris: A scalable transactional data store for Web 2.0 services. Technical report, Zuse Institute Berlin, 2008. Available at `code.google.com/p/scalaris`.

[74] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Structured overlay without consistent hashing: Empirical results. In *CCGRID*, page 8. IEEE Computer Society, 2006.

[75] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Range queries on structured overlay networks. *Computer Communications*, 31(2):280–291, 2008.

[76] Peter M. Senge, Art Kleiner, Charlotte Roberts, Richard B. Ross, and Bryan J. Smith. *The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization*. Nicholas Brealey Publishing, London, 1994.

[77] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with network partitions in structured overlay networks. *Journal of Peer-to-Peer Networking and Applications*, 2008.

[78] A. Shaker and D. S. Reeves. Self-Stabilizing Structured Ring Topology P2P Systems. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*, pages 39–46. IEEE Computer Society, August 2005.

[79] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[80] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM New York, NY, USA, 2004.

[81] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[82] Peter Van Roy. Self management and the future of software design. In *Third International Workshop on Formal Aspects of Component Software (FACS '06)*, volume 182. Springer ENTCS, September 2006.

[83] Peter Van Roy. Overcoming software fragility with interacting feedback loops and reversible phase transitions. In *First International Conference on Visions of Computer Science (BCS 08)*, September 2008.

[84] Peter Van Roy, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. Self management for large-scale

distributed systems: An overview of the SELFMAN project. In *FMCO 2007*. Springer LNCS, October 2007.

[85] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. Network Syst. Manage.*, 13(2), 2005.

[86] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An Automated Formal Approach to Managing Dynamic Reconfiguration. In *21st IEEE International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006.

[87] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature: Macmillan Publishers Ltd*, 1998.

[88] Norbert Wiener. *Cybernetics, of Control and Communication in the Animal and the Machine*. MIT Press, 1948.

[89] Wikipedia, the free encyclopedia. Belief propagation. March 2008. See `en.wikipedia.org/wiki/Belief_propagation`.

[90] Jonathan Wildstrom, Peter Stone, and Emmett Witchel. Autonomous return on investment analysis of additional processing resources. In *Proceedings of 4th International Conference on Autonomic Computing*, page 15, Jacksonville, Florida, USA, June 11-15 2007. ICAC, IEEE Computer Society.

[91] David H. Wolpert, Kevin R. Wheeler, and Kagan Turner. Collective intelligence for control of distributed dynamical systems. *Europhys. Lett.*, 2000.

[92] Daniel Worden. Understand autonomic maturity levels, February 2004.

[93] Yongzheng Wu, Sufatrio, Roland H.C. Yap, Rajiv Ramnath, and Felix Halim. Establishing software integrity trust: A survey and lightweight authentication system for windows. In Zheng Yan, editor, *Trust Modeling and Management in Digital Environments: from Social Concept to System Development*, chapter 3. IGI Global, 2009.

[94] Roland Yap, Felix Halim, and Wu Yongzheng. First report on security in structured overlay networks. SELFMAN Deliverable D1.3a, November 2007. Available at `www.ist-selfman.org`.

[95] Y.Takeda. Symmetric nat traversal using stun, 2007.

[96] H. Yu, M. Kaminsky, P.B. Gibbons, and A. Flaxman. Sybilguard: Defending against sybil attacks via social networks. In *Proceedings of the ACM SIGCOMM 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 267–278. ACM New York, NY, USA, 2006.

[97] Benjamin Zeiss, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, and Jens Grabowski. Applying the iso 9126 quality model to test specifications - exemplified for ttcn-3 test specifications. In Wolf-Gideon Bleek, Jörg Raasch, and Heinz Züllighoven, editors, *Procedings of Software Engineering 2007, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 105 of *LNI*, pages 231–244, Hamburg, March 27-30 2007. SE, GI.

[98] Haitao Zhang, Huiqiang Whang, and Ruijuan Zheng. An autonomic evaluation model of complex software. *International Conference on Internet Computing in Science and Engineering*, 0:343–348, 2008.