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

Deliverable reference number and title:	D.1.1 Report on low-level self-management primitives for structured overlay networks
Due date of deliverable:	July 15, 2007
Actual submission date:	July 15, 2007
Start date of project:	June 1, 2006
Duration:	36 months
Organisation name of lead contractor for this deliverable:	UCL
Revision:	1.0
Dissemination level:	CO

Contents

1 Executive summary **1**

2 Contractors contributing to the Deliverable **2**

3 Results **3**
3.1 Introduction 3
3.2 Ring Topology 4
3.3 Relaxed-Ring Topology 5
3.4 Handling Network Partitions and Merges 8
3.5 Topology for Multi-Dimensional Queries 10
3.6 Future Work 11

4 Papers and publications **12**

A A Structured Overlay for Multi-Dimensional Range Queries **16**

B Handling Network Partitions and Mergers in Structured Overlay Networks **28**

C PEPINO: PEer-to-Peer network INspectOr **37**

D Improving the Peer-to-Peer Ring for Building Fault-Tolerant Grids **40**

1 Executive summary

In order to build self-managing large-scale distributed systems, SELFMAN is aiming for a combination of component models and structured overlay networks. The goal is to achieve self management along four axes: self-configuration, self-healing, self-tuning and self-protection. This deliverable presents results on low-level primitives for structured overlay networks, providing self-configuration and self-healing properties.

Structured overlay networks are used for building self-organising peer-to-peer systems with efficient communication between them. The most popular network topology is based on a ring where every peer knows its successor and predecessor, and it knows a set of other peers that are used for efficient communication with any other part of the network. The resources of the network are uniformly distributed among the peers providing a Distribute Hash Table (DHT). If a uniform distribution of the resources is not feasible, it is recommended to use a topology different to the ring.

Results using the ring topology are achieved with DKS and P2PS, implemented in J2EE and Mozart respectively. Research on P2PS offers a novel relaxed-ring topology where the primitives for peers to join and leave the network are simplified. The resulting network is able to handle high churn and multiple failures of peers. Thanks to the correct failure handling, P2PS, as well as DKS, is able to survive a network partition where two new rings will coexist. Even when nearly every structured overlay network using ring topology is able to survive a network partition, the problem of merging the network has been often ignored by all existing networks. This deliverable presents results on how to efficiently merge rings using gossips based algorithms. The result uses DKS as platform.

Being aware that some problems are not easy to model using a ring topology, this deliverable also presents SONAR, a structured overlay network using a multi-dimensional torus as network topology. This topology is more convenient for resources using multi-dimensional keys, and that are not organised with a uniformly distributed key-space. SONAR deals with this kind of resources by avoiding hashing and working independently of the distribution.

In this deliverable, self-configuration is achieved by providing networks that self-organise their peers independently of how and when they join or leave the network. Different network topologies are provided in order to configure most adequately every problem. Self-healing is achieved by correct failure handling mechanisms, and by providing an efficient solution to network merges after network partitioning.

2 Contractors contributing to the Deliverable

Most of the contractors working on WP1 have contributed to this deliverable. Université catholique de Louvain (UCL) has participated as the lead contractor for this deliverable working closely with the Kungliga Tekniska Högskolan (KTH) and the Zuse Institute Berlin (ZIB). The integration of the work has been discussed and developed during several meetings hold during the first year. Contractor National University of Singapore (NUS) has participated in deliverable D1.3a.

UCL(P1) has focused its work in the analysis of the ring topology for structured overlay networks. As result, a new topology based on a *relaxed-ring* has been developed, providing better guarantees for self-organisation and self-healing properties. This new ring maintenance is designed making realistic assumptions with respect to failure detection. An application for network evaluation called PEPINO has also been developed.

KTH(P2) has also contributed with important results on structured overlay networks based on the ring topology. The work solves the problem of efficiently *merges of rings* for handling network partitioning. The solution is based on a gossip algorithm, allowing system designer to adjust the trade-off between message complexity and time complexity through a *fanout parameter*. The algorithm is highly scalable with respect to the size of the networks, avoiding positive feedback loops.

ZIB(P5) has focused its works in a different topology for structured overlay networks. Since ring topology relies strongly on a uniform data distribution, SONAR, the proposed solution, uses a multi-dimensional torus as overlay topology. The solutions allows storing and retrieving objects addressed by multi-dimensional keys. It avoids hashing and supports logarithmic routing. This solution allows the project to address problems where the ring-based network does not perform well due to skewed distributions or where one-dimensional keys are not sufficient.

3 Results

This section summarises the results obtained during the first year of the project in the context of low-level self-management primitives for structured overlay networks. Section 3.1 gives a very brief introduction to decentralised systems and the challenges that motivate the work that is presented in this deliverable, which is focused on structured overlay networks.

Section 3.2 describes the ring topology, the most representative network topology used by existing structured overlay networks. This topology is quite robust and efficient, but existing ring-based systems still let several open issues to be solved. Sections 3.3 and 3.4 shows results on solving some of those issues, providing guarantees that are required by deliverable D3.1a. The work provides a new network topology in the form of a relaxed-ring, derived from the original ring. The deliverable also includes an efficient algorithm for handling network partition and merges of the ring. These solutions are designed having in mind the ideas presented in deliverable D2.1a. But not all the open issues can be solved with a ring-alike topology, for instance, the ring needs a uniform distribution of the key-space in order to provide efficient routing. This is the main reason for the results presented in section 3.5, where a multi-dimensional network topology is provided as a solution to the problem of not having a uniform distribution of the key-space. More details of these work can be found in the annexes of the this deliverable.

3.1 Introduction

Decentralised applications has rapidly increased their popularity in the last years over the classical client-server architecture. Several phenomena have contributed to remove from distributed applications the dependency on big servers. Among those phenomena we can find the increase of computing-power of home computers, and the increase in Internet bandwidth with a sufficient reliability. These facts have led distributed applications to run on top of peer-to-peer networks. Every peer provides and requests resources and services. Every peer acts as a client and as a server. There is no central point of control or synchronisation, and hence, no single point of failure, and no traffic bottleneck.

Despite the advantages of decentralised networks with respect to the classical client-server architecture, the complexity of the system is highly compromised, ironically, due to the lack of a single point of control. Increasing self-management of the system can help us to reduce this new complexity. One of our first objectives in order to reach self-management properties is to address the issues of *self-configuration* and *self-healing* of the network. These are two of the four axes tackled by SELFMAN. This deliverable presents results on self-organisation at the level of the network topology, contributing to self-configuration. Results providing self-healing are also presented. These results involve failure handling and merging networks after network partitioning.

Within peer-to-peer networks is possible to observe many strategies for organisation. Pioneer Internet applications for file sharing, such as Napster [9], organised their networks using a mixed-architecture where servers were still needed for finding other peers. File sharing applications evolved later to unstructured networks with

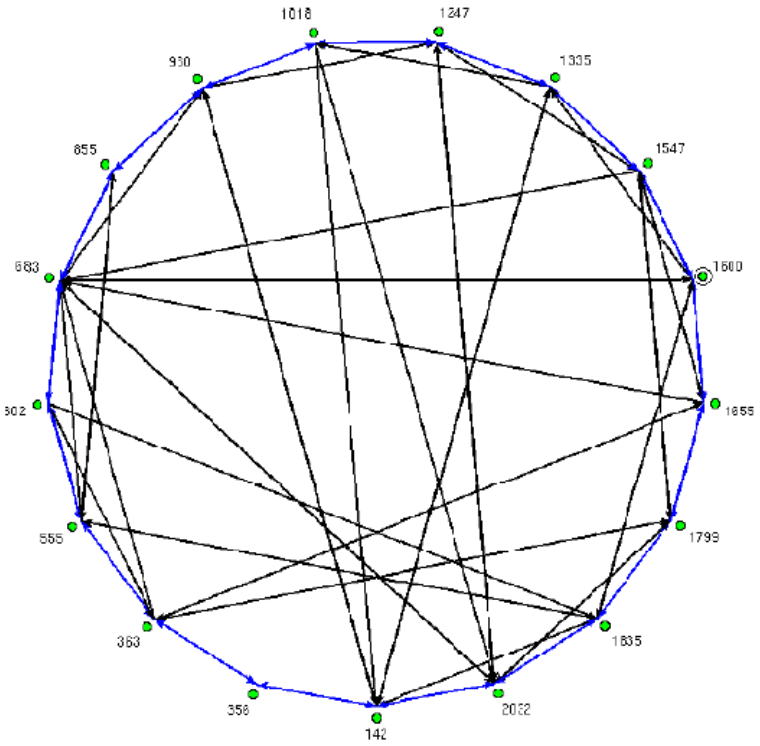


Figure 1: Structured Overlay Network using ring topology

random connections, as in Gnutella [6]. These networks formed an overlay network with the ability to route messages on top of the Internet, but with a high amount of message sending and an inefficient use of the bandwidth. In order to provide scalability and efficient routing, peer-to-peer networks finally evolved to structured overlay networks (SONs) [4] which is the subject of this research.

3.2 Ring Topology

One of the most popular network topology used by SONs is based on a ring introduced by Chord [13]. This topology is also used by DKS [5] and P2PS [3], both developed by SELFMAN's partners. In the ring, every peer is connected to a *predecessor* and a *successor*. The order is given by their identifiers which represent a *hash-key*. Every peer is responsible for all keys between its predecessor and itself. Like this, the ring provides a Distributed Hash Table (DHT) where values can be stored and recovered using the correspondent key.

Figure 1 depicts a SON using the ring topology. The blue arrows represent connections between successors and predecessors. Every peer is identified by a hash-key, which in this case is an integer. Using the identifiers, successors and predecessors, the network is able to route correctly any message to every responsible of a particular key. To make routing efficient, extra connections to other peers in the network are introduced following a particular algorithm [2, 1]. These connections are called *fingers*, and correspond to the black arrows in Figure 1. Using this fingers for routing, it is guaranteed to reach any peer in the network in maximum $log_k(n)$ hops, where n is the size of the network, and k is the amount of fingers per peer.

Even when the ring topology works correctly and efficiently, its maintainability is not a trivial issue if *fault tolerance* and *consistent lookup* want to be guaranteed in presence of high churn. Churn is a ratio that measures how many peer join and-or leave the network in a given amount of time. By fault tolerance, we mean the ability of the system to reconfigure and continue working despite the failure of some peers. By consistent lookup, we mean that concurrent lookups for the responsible of a particular key will not result in more than one responsible. Lookup consistency is an important property required by deliverable D3.1.

To guarantee lookup consistency, the algorithms for maintenance of the ring must handle correctly the cases of peers joining and leaving the network. It has been shown in [5] that the join algorithm of Chord can generate inconsistencies in the case of multiple joins, and this can happen even in fault-free scenarios. Due to this fact, Chord runs a stabilisation algorithm periodically. Previous work on DKS [5] has concluded that an atomic join/leave operation must be provided in order to provide consistent lookup. To guarantee atomicity, the proposed solution uses an algorithm based on a locking mechanism. But the use of locks in a distributed environment generated several problems of efficiency, and makes maintainability of the ring less tolerant to failures. One of the results of this deliverable is a novel relaxed-ring topology that simplifies self-organisation of the peers, while keeping consistent lookup.

3.3 Relaxed-Ring Topology

One of the main issues we have detected in the existing ring maintenance algorithms, is that they require an agreement of three peers in order to finalise. This synchronisation is not trivially achieve because the communication between peers is asynchronous and expose to failures of nodes and communication channel. Problems in the communication channel are quite difficult to handle because they generate partial false suspicions, meaning that a node can appear as dead to some peers, but as working fine to other set of peers. In such situation is very hard to achieve a consensus between more than 2 peers.

The other issue with ring maintenance is that they try to guarantee reciprocity of *successor* and *predecessor* pointers. We avoid this requirement by stating a separate functionality for each pointer, allowing us to split the join algorithm into two steps where only two peers are involved in each step. This approach is more realistic because it only needs the agreement of two nodes in every step, which is guaranteed with point-to-point communication.

We first state that *every peer is in the same ring as its connected successor*, then, the successor pointer only tells a peer to which ring it is connected to. If the successor fails, a new successor from the same ring must be found. The second statement says that *the responsibility of a peer is determined by the id of the peer and its own id*. Remember that the *id* of peers represents *hash-keys* from DHT. The second statement describes how the responsibility of a peer is determined, but it does not implies a connection between a peer and its predecessor. It only requires the knowledge of its *id*.

When a peer wants to join the network, it triggers a $lookup(q)$ message through an access point searching for the responsible of its key, in this case q. When the

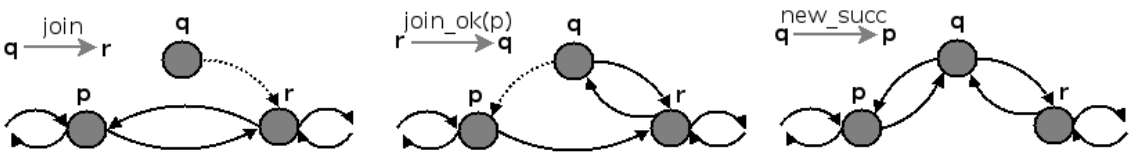


Figure 2: Join algorithm in a relaxed-ring. (1) The joining peer q contacts r, its successor candidate. (2) r accepts q as predecessor and notifies it with a reference to p. At this moment, q is in a branch of the ring. (3) q notifies p about its join, and p accepts q as successor.

responsible answers back, the joining peer triggers a *join* message to its successor candidate. Figure 2 depicts the join algorithm where q joins in between peers p and r. When peer q contacts peer r, this one checks if the joining peer belongs to its responsibility $(p, r]$. When peer r accepts peer q as predecessor, the key-range $(p, r]$ is split into $(p, q]$ and $(q, r]$. At this stage, q has a valid successor r, and then, it is considered inside the ring. It is not in the *core* ring, but in a branch. Peer q also knows its predecessor, and therefore, it knows its range of responsibility in the DHT. This happens even when the connection between p and q is not yet established. Note that there is no overlapping of responsibilities, and this, lookup requests are always resolved consistently. The algorithm finishes when q contacts p presenting itself as new successor. Then, the branch is pruned and the *core* ring becomes a *perfect* ring once again.

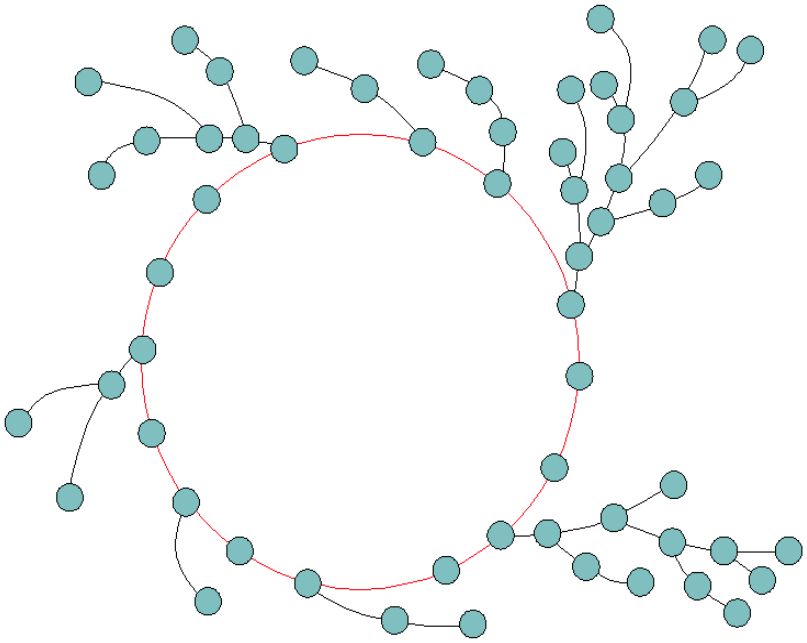


Figure 3: An extreme example of a relaxed-ring. The network is able to route message correctly, and it is constantly converging towards a perfect ring. The red circle represents the core ring.

If the connection between peers p and q can never be establish, the ring continue working correctly because the responsibility of the keys is correctly distributed. If a node joins in between peers q and r, the branch grows and the network continues working. If a new peer joins between p and q, it will fix the core ring if it is able to establish connection with peer p. The branch will just grow otherwise. This approach allows the system to handle multiple joins without stopping the resolution of lookup requests, and it is also fault-tolerant as it will be explain later in this section.

Figure 3 depicts a network with several problems to establish connections between peers. It is an extreme example of a relaxed ring topology. A situation like this would not be possible in a ring requiring reciprocity of successor and predecessor pointers.

Since the system must be fault-tolerant, there is no need for a *leave* algorithm, because leaves are covered by the failure handling. In fact, it is not a good idea that the correctness of the network maintenance relies on gentle leaves of the peers, because the fact that peers will crash and leave the network without respecting any leave algorithm is unavoidable.

In the relaxed ring, as in any structured overlay network, every peer keeps a failure detector to identify crashes of other peers. The failure handling mechanism works as follows. Every peer keeps a list of successors. When the successor is detected to have crashed, the peer pics a successor candidate from the successor list, and triggers a *join* message as it were joining the network. The main difference is that the successor candidate has also identified that its predecessor has crashed, and then, it accepts the *join* message from the predecessor of the crashed node, and the ring is restored. If the successor candidate does not responds, a new candidate is elected from the successor list. Note that the failure handling mechanism is only triggered by the predecessor of the crashed node. The successor only waits the *join* message. This is necessary to keep consistent lookup. More details can be found in appendix D.

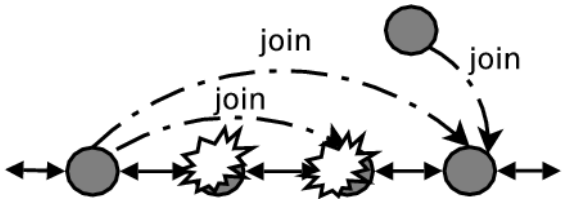


Figure 4: Multiple failures and new peer joining the network just before the successor candidate for recovery.

A situation with multiple failures and a new peer joining the network can be observed in Figure 4. In this case, the new joining peer and the predecessor of the crashed node competes for contacting the successor candidate as they were both newly joining peers. Since the problem is already solved by the joining algorithm, the system is also able to handle the problem of failure handling mixed with join operations.

We can think about the relaxed-ring maintenance in terms of feedback loops as they are described in deliverable D2.1a. Every peer is constantly monitoring the

network by receiving messages from other peers, and by running a failure detector per every node it is connected to. Two event perturb the stability of the system: the joining of a new peer, and the failure of a member of the ring. When these events are detected, corrective actions are triggered in order to include the joining peer in the correct range according to its *id*, or to find a new peer that takes over the crashed peer. The system recovers in the form of a relaxed-ring, converging finally to a perfect ring if all connections are working correctly.

As many other structured overlay network, the relaxed-ring topology can also be used to build a grid networks on top of it. The relaxed ring presents the advantage of being fault tolerance with a self-organising approach, requesting less preparation and less administrative tasks from the grid users. Work done in this area is included in appendix D and presented in [8]. This deliverable also contains a short paper [7] included as appendix C, describing a tool for network analysis and evaluation, called PEPINO. An official release of this tool, together with the release of the third version of P2PS is planed to be done during the second year of the project.

3.4 Handling Network Partitions and Merges

As it has been described in the previous section, structured overlay networks, and in particular using ring based topology, are able to handle simultaneous peers crashes. In this section we analyse what happen when network partitions occur. From the point of view of a single peer, it is impossible to distinguish massive simultaneous node failures from network partitions. If the peer has a reference to another alive node, the ring will survive self-organising and fixing the routing tables. Considering that, almost every ring based network is able to survive to network partitions. But the problem of merging two or more rings when the partition is gone has been ignored. An efficient algorithm for merging rings [12] is presented in this deliverable, and included as appendix B.

The algorithm works as follows. Every peer keeps a *passive list* to store a finite amount of the peers detected to have crashed. If these peers did not crashed and are correctly running in a different network, they will be detected to be alive once the connection between the network is recovered. The merging of the ring is triggered when two nodes from different partitions detect each other. Figure 5 depicts how the algorithm works. In the figure, all peers from two different rings, black and white, are arranged as they were in one ring having the peers distributed along the key-space. When peer p from the black ring detects peer q from the white ring, it triggers a MLOOKUP(q) event in its own ring trying to get closer to id q. Since rediscovering a peer is a reciprocal action, peer q also triggers MLOOKUP(p) in its own ring. When the lookup arrives to the responsible of the searched key, TRYMERGE($cpred, csucc$) is triggered, where *cpred* and *csucc* are predecessor and successor candidates for the merging. The merge proceed in clockwise and anti-clockwise directions.

Even when the algorithm described above works correctly, it is not very efficient. Augmenting MLOOKUP with a gossip-based mechanism for propagation, the algorithm improves considerably its efficiency. The idea is to trigger multiple merges to work simultaneously, making the algorithm to converge faster. Every time a node receives a MLOOKUP(id) request, it picks randomly a peer r from its routing table

and starts a ring merger between id and r. Like this, peers from every ring can find each other faster. To avoid congestion and high traffic, this cannot be trigger eagerly. It has to be done periodically. One advantage of this is that the periodicity can be controlled through a parameter, given the system more control over the trade off between time and bandwidth consumption. More details about how this parameter affects complexity and efficiency of the system can be found in appendix B.

Looking at the system as a model using feedback loops as they are described in deliverable D2.1a, we can identify the *passive list* as an important part of the monitoring component. Once a peer in this list is detected to be alive, a corrective action is triggered, using the merge algorithm as actuator for this correction. The system converge to a stable ring avoiding positive feedback cycles which would congest the network.

The algorithm can also be used to explicitly merge two networks. For instance, if two institutions decide two join their networks without stopping them, it is enough to introduce two nodes, one from each network, and that will trigger the merging algorithm converging to one ring.

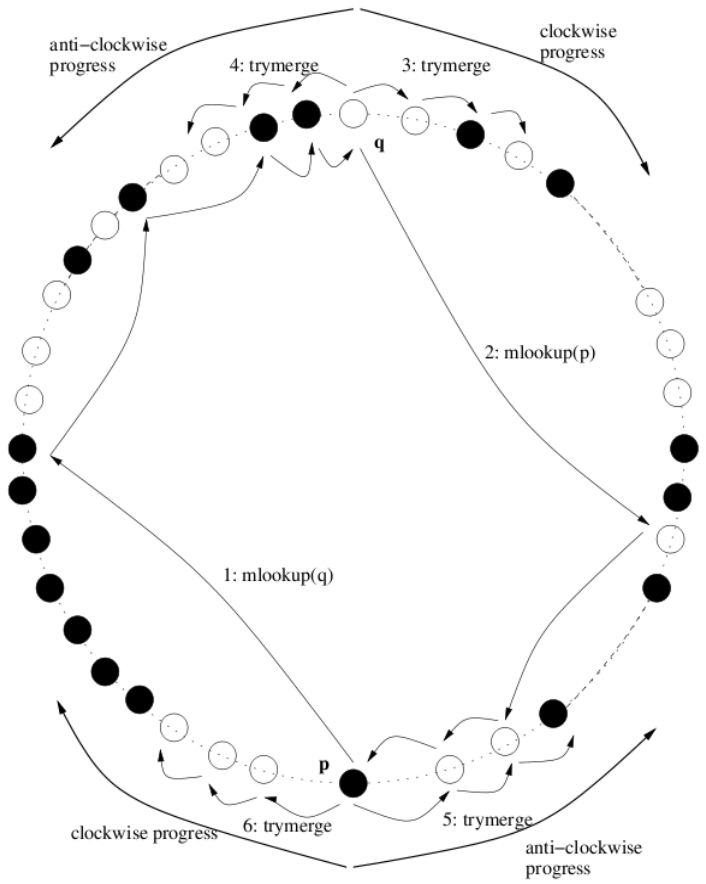


Figure 5: Black peers belong to one structured overlay network, and white peers belong to a different one. Peer p from ring black and peer q from ring white have detected each other. Both of them trigger MLOOKUP and TRYMERGE.

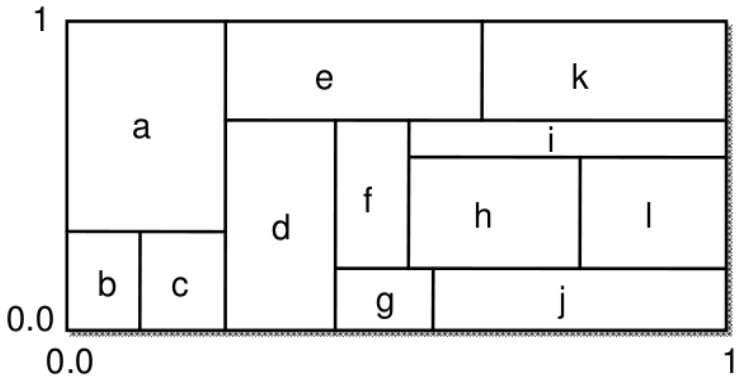


Figure 6: Structured Overlay Network using two dimensions to store data. Every peer in the network is responsible for one rectangle of the *key-space* distribution. Denser areas gets more peers assigned.

3.5 Topology for Multi-Dimensional Queries

The work presented in the previous sections of this deliverable is based on the structured overlay networks using ring topology. Even when this topology is suitable for many problems, it requires a uniform data distribution. If the network is build with a skewed distribution of the *key-space* it behaves quite inefficiently. Then, an alternative topology is needed in order to model problems such as a database for geographical locations, where the distribution of cities follows a Zipf distribution [14]. The proposed solution presented in this deliverable is to organise the network as a multi-dimensional torus, where more dense areas get assigned more nodes, dealing efficiently with skewed distributions. The structure overlay network built on top of this topology is called SONAR [11], and detailed description of its functionality is included in appendix A.

As any other structured overlay network, SONAR is used to store and retrieve objects identified with a *key*. Instead of using a number for the key as in DKS or P2PS, SONAR uses a vector of d components, called *attributes*. Every attribute is stored in one dimension of the torus. Figure 6 depicts a two-dimensional key-space, where every attribute can take a value in the interval [0,1]. The whole key-space is split into hypercuboids. Each hypercuboid is assigned to a peer in the network in order to share responsibilities and balance the load.

Neighbours are determined by adjacent areas, as in CAN [10], with the difference that not only the neighbours are used for routing but also additional fingers as in e.g. Chord. The routing table is generated with a set of *successors* and *fingers*. Successors are chosen taken the peer to the middle of every adjacent side. Fingers are taken following the successors in any dimension of the torus. This routing table allows a routing of any message in $O(logN)$, which is much better than $O(\sqrt[d]{N})$ of CAN. Another innovation of SONAR is that it avoids hashing. Due to that, any range query can be resolved in a logarithmic number of steps independent of the number of query-dimensions.

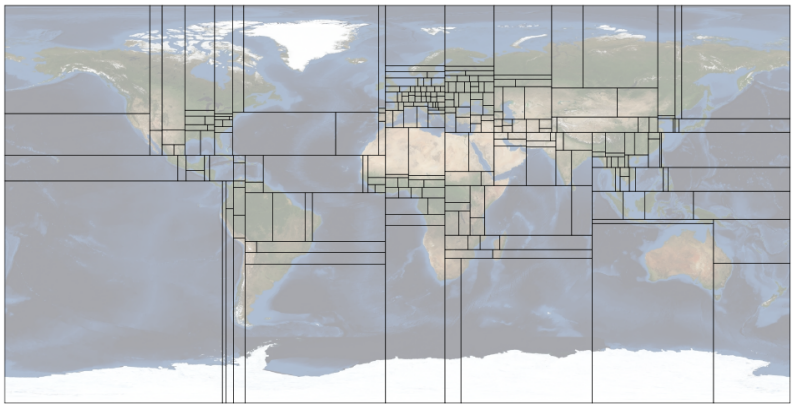


Figure 7: The graphical location data of 1,904,711 cities follows a Zipf distribution, which can be modelled by SONAR overlay network. This examples represents a network of 256 peers.

Figure 7 shows a use-case scenario to evaluate SONAR. The chosen example corresponds to an overlay network storing the locations of 1,904,711 cities. The graphical location of the cities follows a Zipf distribution. As it is observed in the figure, some areas are more dense than others, and therefore, all areas have different sizes. The figure shows a network of 256 nodes, but experiments were run from 128 nodes and up to 2^{14} nodes. All-to-all searches behaved as expected following $O(logN)$ complexity for routing messages.

3.6 Future Work

This deliverable is related with deliverable D1.2, D1.4 and D1.5, to be presented at the end of next SELFMAN's period (M24). All contributors are carrying work towards high-level self-management primitives, to be presented in D1.2. KTH continues with the development of DKS, work to be included in D1.4. UCL will carry on the work on the development of the relaxed-ring topology towards deliverable D1.5. Releases of P2PSv3 and PEPINO are planed for the next period.

Even when the results for this deliverable are quite satisfactory, some work directly related is still in progress. There will be another publication with more results obtained with the multi-dimensional torus of SONAR. A formalisation of the relaxed-ring topology is also expected to be published in the next period.

4 Papers and publications

Most of the work presented in section 3 has been accepted for publication in different conferences and workshops. A list of these publications with their abstracts are presented here. Full papers are provided in the respective appendices.

A Structured Overlay for Multi-Dimensional Range Queries . Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. In *proceedings of Euro-Par 2007* (to appear).

Abstract: We introduce SONAR, a structured overlay to store and retrieve objects addressed by multi-dimensional names (keys). The overlay has the shape of a multi-dimensional torus, where each node is responsible for a contiguous part of the data space. A uniform distribution of keys on the data space is not necessary, because denser areas get assigned more nodes. To nevertheless support logarithmic routing, SONAR maintains, per dimension, fingers to other nodes, that span an exponentially increasing number of nodes. Most other overlays maintain such fingers in the key-space instead and therefore require a uniform data distribution. SONAR, in contrast, avoids hashing and is therefore able to perform range queries of arbitrary shape in a logarithmic number of routing stepsindependent of the number of system- and query-dimensions. SONAR needs just one hop for updating an entry in its routing table: A longer finger is calculated by querying the node referred to by the next shorter finger for its shorter finger. This doubles the number of spanned nodes and leads to exponentially spaced fingers.

PEPINO: PEer-to-Peer network INspectOr . Donatien Grolaux, Boris Mejías, and Peter Van Roy. In *proceedings of The Seventh IEEE International Conference on Peer-to-Peer Computing* (to appear).

Abstract: PEPINO is a simple and effective peer-to-peer network inspector. It visualises not only meaningful pointers and connections between peers, but also the exchange of messages between them, providing a useful tool for debugging purposes. It can monitor running networks, simulate them and log them in order to reproduce interesting case scenarios. Failures can be explicitly introduced to study fault tolerant algorithms. The graphical representation of the network uses a physical model to attract or repel peers, allowing the user to study the system from different points of view. This demo aims to present the use of PEPINO in the development of a novel relaxed-ring topology for fault tolerant networks, where the representation of the ring based on predecessors may differ from the ring based on successors. We show how PEPINO is also useful for visualising other network topologies such as perfect ring or unstructured networks.

Improving the Peer-to-Peer Ring for Building Fault-Tolerant Grids . Boris Mejías and Donatien Grolaux and Peter Van Roy. In *CoreGRID Workshop on Grid-* and P2P-**.

Abstract: Peer-to-peer networks are gaining popularity in order to build Grid systems. Among different approaches, structured overlay networks using ring topology are the most preferred ones. However, one of the main problems of peer-to-peer rings is to guarantee lookup consistency in presence of multiple joins, leaves and failures nodes. Since lookup consistency and fault-tolerance are crucial properties for building Grids or any application, these issues cannot be avoided. We introduce a novel relaxed-ring architecture for fault-tolerant and cost-efficient ring maintenance. Limitations related to failure handling are formally identified, providing strong guarantees to develop applications on top of the relaxed-ring architecture. Besides permanent failures, the paper analyses temporary failures and broken links, which are often ignored.

Handling Network Partitions and Mergers in Structured Overlay Networks . Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. In *proceedings of The Seventh IEEE International Conference on Peer-to-Peer Computing* (to appear).

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 false 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.

References

[1] 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.

[2] Bruno Carton and Valentin Mesaros. Improving the scalability of logarithmic-degree dht-based peer-to-peer networks. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 1060–1067. Springer, 2004.

[3] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. *http://gforge.info.ucl.ac.be/projects/p2ps*, 2007.

[4] Sameh El-Ansary and Seif Haridi. An overview of structured overlay networks. In *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*. 2005.

[5] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.

[6] Gnutella. http://gnutella.com, 2003.

[7] Donatien Grolaux, Boris Mejías, and Peter Van Roy. PEPINO: PEer-to-Peer network INspectOr. In *The Seventh IEEE International Conference on Peer-to-Peer Computing*, 2007. To appear.

[8] Boris Mejías, Donatien Grolaux, and Peter Van Roy. Improving the peer-to-peer ring for building fault-tolerant grids. In *CoreGRID Workshop on Grid-* and P2P-**, july 2007.

[9] Napster. Open source napster server, 2002.

[10] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[11] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. A structured overlay for multi-dimensional range queries. In *Euro-Par 2007*, 2007. To appear.

[12] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Handling network partitions and mergers in structured overlay networks. In *The Seventh IEEE International Conference on Peer-to-Peer Computing*, 2007. To appear.

[13] 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.

[14] G. K. Zipf. Relative frequency as a determinant of phonetic change. In *Harvard Studies in Classical Philology*, volume 15, pages 1–95, 1929.

A A Structured Overlay for Multi-Dimensional Range Queries

A Structured Overlay for Multi-Dimensional Range Queries*

Thorsten Schütt, Florian Schintke, Alexander Reinefeld

Zuse Institute Berlin

Abstract. We introduce SONAR, a structured overlay to store and retrieve objects addressed by multi-dimensional names (*keys*). The overlay has the shape of a multi-dimensional torus, where each node is responsible for a contiguous part of the data space. A uniform distribution of keys on the data space is not necessary, because denser areas get assigned more nodes. To nevertheless support logarithmic routing, SONAR maintains, per dimension, fingers to other nodes, that span an exponentially increasing number of *nodes*. Most other overlays maintain such fingers in the *key-space* instead and therefore require a uniform data distribution. SONAR, in contrast, avoids hashing and is therefore able to perform range queries of arbitrary shape in a logarithmic number of routing steps—independent of the number of system- and query-dimensions. SONAR needs just one hop for updating an entry in its routing table: A longer finger is calculated by querying the node referred to by the next shorter finger for its shorter finger. This doubles the number of spanned nodes and leads to exponentially spaced fingers.

1 Introduction

The efficient handling of multi-dimensional range queries in Internet-scale distributed systems is still an open issue. Several approaches exist, but their lookup schemes are either expensive (space-filling curves) [2] or use probabilistic approaches like consistent hashing [10] to build the overlay.

We propose a system for storing and retrieving objects with d-dimensional keys in a peer-to-peer network. SONAR (Structured Overlay Network with Arbitrary Range-queries) directly maps the multi-dimensional data space to a d-dimensional torus. It supports range queries of arbitrary shape, which are useful, for example, in geo-information systems where objects in a given distance of a position are sought. SONAR can also be employed in Internet games with millions of online-players who concurrently interact in a virtual space and need quick access to the local surroundings of their avatars. In a broader context, SONAR can be employed as a hierarchical publish/subscribe system, where published events are categorized by several independent attributes. The category of published events addresses a data point in the d-dimensional space and consumers subscribing to subareas will receive all events published in their subarea.

* Part of this work was carried out under the SELFMAN and XtreemOS projects funded by the European Commission.

The paper is organized as follows: First, we discuss related work. Then, in Section 3, we introduce SONAR. In Section 4, we present empirical results and in Section 5 we conclude the paper with a brief summary.

2 Related Work

Several systems [1] have been proposed that support complex queries with multi-dimensional keys and ranges. They can be split into two groups.

a) Space filling curves. These systems [2,9,16] use locality preserving space-filling curves to map multi-dimensional to one-dimensional keys. They provide less efficient range queries than the space partitioning schemes described below, because a single range query may cover several parts of the curve, which have to be queried separately (Fig. 5a). Chawathe et al. [7] present performance results of a real-world application using Z-curves on top of OpenDHT. The query performance (≈ 2 sec. for ≤ 30 nodes) is rather low due to the layered approach.

b) Space partitioning. The schemes using space partitioning split the key-space among the nodes. SONAR belongs to this group of systems. The proposed systems mainly differ by their routing strategies.

CAN [14] was one of the very first DHTs. It hashes the key-space onto a multi-dimensional torus. While the topology resembles that of SONAR and MURK (see below), CAN uses just the neighbors for routing and it does not support range queries.

SWAM [4] employs a Voronoi-based space partitioning scheme and uses a small-world graph overlay with routing tables of size $O(1)$. The overlay is not built by some regular partitioning scheme (e.g. kd-tree [5]) but uses a sample technique to place the fingers.

Multi-attribute range queries were also addressed by Mercury [6] which needs a large number of replicas per item to achieve logarithmic routing performance. SWORD [12] uses super-peers and query-caching to allow multi-attribute range queries on top of the Bamboo-DHT [15].

Ganesan *et al.* [9] proposed two systems for multi-dimensional range queries in peer-to-peer systems: SCRAP and MURK. SCRAP uses the traditional approach of mapping multi-dimensional to one-dimensional data with space-filling curves which destroys the data locality. Consequently, each single multi-dimensional range-query is mapped to several one-dimensional queries. MURK is more similar to our approach, as it divides the data space into hypercuboids with each partition assigned to one node. In contrast to SONAR, MURK uses a heuristic approach based on skip graphs [3] to set routing fingers.

3 System Design

We first present the overlay topology of SONAR and then discuss its routing and lookup strategy. Thereafter we present mechanisms that make SONAR robust under churn.

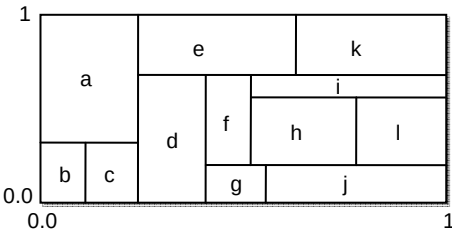


Fig. 1: Example two-dimensional overlay with attribute domains $[0, 1]$.

SONAR is used to store and retrieve *objects*. It works on a d-dimensional torus, the *key-space*. Objects have a name, the *key*, which is a vector of d components, the *attributes* of the key. Each *dimension* of the torus is responsible for one *attribute domain*. Figure 1 illustrates a two-dimensional key-space ($[0, 1]^2$). Arbitrarily located computers, the *nodes*, are each responsible for a dedicated area (hypercuboid) in the key-space of the overlay (rectangles in Fig. 1). The *node-space* has the same extent as the key-space, but is completely filled with nodes. Two nodes in the node-space are *adjacent* (or *neighbors*) when their key-space is adjacent. The direct mapping between key-space and node-space guarantees adjacent keys to be stored on the same or adjacent nodes, which enables efficient range queries across node boundaries by local query propagation.

Nodes are dynamically assigned to the key-space such that each node serves roughly the same number of objects. Load-balancing is done by changing the responsibility of nodes instead of moving around objects in the key-space. That becomes necessary when the number of objects or nodes in the system changes (Sect. 3.4).

3.1 Overlay Topology

As illustrated in Figure 1, the two-dimensional key-space is covered by rectangles, each of them containing about the same number of objects. Because the keys are generally not uniformly distributed, the rectangles have different sizes and thus may have more than one neighbor per direction. The neighbors are stored in *neighbor lists*, one per dimension.

The overlay described so far resembles that of CAN [14] except for the hashing in CAN, which prevents efficient range queries. Consequently, SONAR would also need $O(\sqrt[d]{N})$ network hops if it would just use the neighbors for routing. In the following, we introduce routing tables to achieve logarithmic routing performance.

3.2 Routing

For routing, SONAR uses separate *routing tables*, one per dimension. Each routing table contains *fingers* spanning an exponentially increasing number of nodes

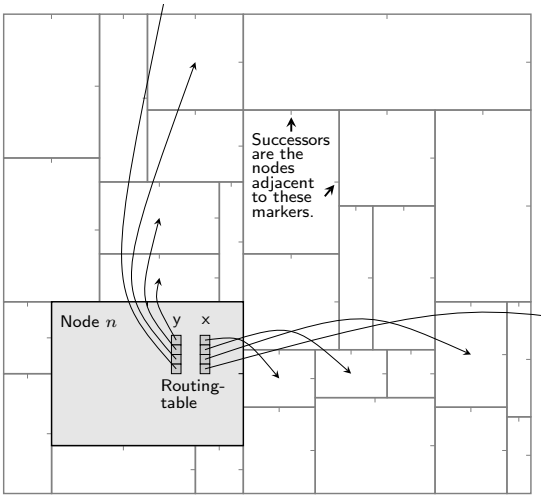


Fig. 2: Routing table for the two-dimensional case

(Fig. 2). With a total of $\log N$ routing fingers, the average number of hops is reduced to $O(\log N)$ [17].

To calculate its i^{th} finger in the routing table, a node looks at its $(i-1)^{th}$ finger and asks the remote node listed there for the $(i-1)^{th}$ finger. At the lowest level, the fingers point to the successor.

$$\mathit{finger}_i = \begin{cases} \mathit{successor} & : i = 0 \\ \mathit{finger}_{i-1}.\mathit{getFinger}(i-1) & : i \neq 0 \end{cases}$$

This update process works in a running system, but also during startup. Initially, all fingers are set to *unknown* except for the finger to the successor. Filling the second entry will always succeed, because the successor knows its successor. Filling further entries may fail (result *unknown*), because the remote node may not have determined the corresponding entry yet. But with subsequent periodic updates, eventually all nodes will get their entries filled. The resulting structure is similar to skip lists [13], but the behavior is more deterministic.

Successor used for routing. A node may have more than one neighbor per direction. We define the node adjacent to the middle of the respective side to be the *successor*. Successors are marked by small ticks in Figure 2.

Due to the different box sizes and the calculation of longer fingers from shorter ones, fingers are not necessarily straight in one direction. Slight deviations in y-direction might occur when following the fingers of the x-direction (and vice versa), as shown in Figure 2. Our empirical results indicate, however, that this does not affect the logarithmic routing performance (Sect. 4).

```
// calculates the entries of a routing table
void updateRoutingTable(int dim) {
  int  i = 1;
  bool done = false;

  rt[dim][0] = this.Successor[dim];

  while (!done) {
    Node candidate = rt[dim][i - 1].getFinger(dim, i - 1);
    if (IsBetween(dim, rt[dim][i - 1].Key, candidate.Key, this.Key)){
      rt[dim][i] = candidate;
      i++;
    } else
      done = true;
  }
}

// checks whether the resp coordinate of pos lies between start and end
bool IsBetween(Dim dim, Key start, Key pos, Key end);
```

Fig. 3: Finger calculation for dimension **`dim`**.

Routing table size. Each node holds approximately $\log N$ fingers in its routing tables. However, not knowing the total number of nodes N, how many fingers should a node put into each of its d routing table so that the total is $\log N$? Mercury [6] predicts the system size N by estimating the key density. SONAR uses a simpler, deterministic solution with less overhead.

For each dimension *dim*, SONAR's finger update algorithm (Fig. 3) inserts an additional finger $finger_i$ as long as its position is between that of the last routing table entry $finger_{i-1}$ and that of the node itself. Otherwise the new finger circles around the ring and is not inserted.

Our results in Section 4 confirm that each node holds indeed $\log N$ fingers. The construction process guarantees—in contrast to Chord [18]—that no two fingers point to the same node. Since the fingers in the routing tables span an exponentially increasing number of nodes, the routing table of each dimension has a total of $\lceil \log D \rceil$ entries on the average, where D is the number of nodes in this direction on the torus.

Cost of a finger update. Our periodically running finger update algorithm needs just one network hop to determine an entry in the routing table. Chord in contrast needs $O(\log n)$ for the same operation, because it performs a DHT lookup to calculate a finger.

3.3 Lookup and Range Queries

As in other DHTs, SONAR uses greedy routing. In each node the finger that maximally reduces the Euclidean distance to the target in the key-space is followed, independently of the dimension (see Fig. 4).

SONAR supports range queries with multiple attributes. In its most basic form a range query is defined by d intervals for the d attribute domains. The

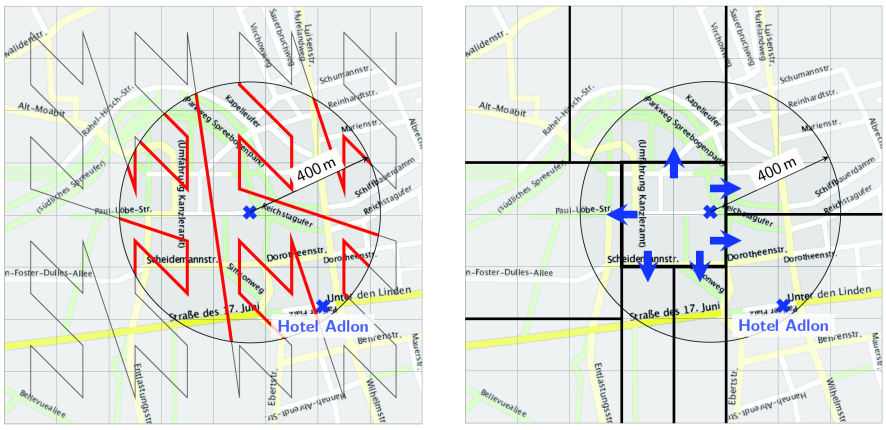
```
// find the responsible node for a given key
Node find(Point target) {
  Node nextHop = findNextHop(target);
  if (nextHop == this)
    return this;
  else
    return nextHop.Find(target);
}

double getDistance(Node a, Point b);

Node findNextHop(Point target) {
  Node candidate = this;
  double distance = getDistance(this, target);

  if (distance == 0.0)
    // found target
    return this;

  for (int d = 0; d < dimensions; d++) {
    for (int i = 0; i < rt[d].Size; i++) {
      double dist = getDistance(rt[d][i], target);
      if (dist < distance) {
        // new candidate
        candidate = rt[d][i];
        distance = dist;
      }
    }
  }
  // will never happen:
  Assert(candidate != this);
  return candidate;
}
```

Fig. 4: Lookup for a target.

range query finds all keys whose attributes match the respective intervals and returns the corresponding objects. Because of their shape, such range queries are called d-dimensional *rectangular* range queries.

In practice, users sometimes need to define circles, polygons, or polyhedra in their queries. Figure 5 illustrates a two-dimensional circular range query defined by a center and a radius. Here, we assume a person located in the governmental district of Berlin searching for a hotel in 'walking distance' (circle around the person). The query is first routed to the node responsible for the center of the circle and then forwarded to all neighbors that partially cover the circle (Fig. 5b). The query is checked against the local data and the results are returned to the requesting node. Figure 6 shows the pseudocode of this algorithm. Note that redundant messages are eliminated. `op` is an additional check for objects in the queried area—in this case for type hotel.

SONAR performs a range query with a single lookup. When the target node does not hold the complete key range, the query is locally forwarded. Systems with space-filling curves, in contrast, usually require more than one lookup for a single range query because they map connected areas to multiple independent line segments, see Figure 5a.



(a) Z-curve (8 line segments): $8 \cdot log_2(N)$ (b) Neighborhood broadcast: $log_2(N)+6$

Fig. 5: Circular range query.

```
// perform a range query
void queryRange(Range r, Operation op) {
  Node center = Find(r.Center);
  center.doRangeQuery(r, op, newId());
}

void doRangeQuery(Range r, Operation op, Id id) {
  // avoid redundant executions
  if (pastQueries.Contains(id))
    return;
  pastQueries.add(id);

  foreach (Node neighbor in this.Neighbors)
    if (r ∩ neighbor.Range != ∅)
      neighbor.doRangeQuery(r \ this.Range, op, id);

  // execute operation locally
  op(this, r);
}
```

Fig. 6: Range query algorithm.

3.4 Topology Maintenance to Handle Churn

Node Join. When a node joins the system, the key-space of a participating node has to be split and the key responsibilities subdivided. To achieve this, two things must be done (we first describe random splitting and then include load balancing):

1. Select a random target node: A random position in the key-space is routed to and a random walk is started from there. The final target node of this is the candidate to be split. The random walk ensures that nodes responsible for larger areas of the key-space are not preferred over smaller ones.

2. Split the key-space and transfer one part to the new node: Splits are parallel to one of the coordinate system axes. The selection of the axis to be split should not strictly favor one dimension over the others because the number of nodes to be contacted for a range query could become disproportionately high, when a large interval for the favored dimension is specified in the query. Also, node leaves could become more expensive.

Node Leave. Handling a leaving node is more difficult, because it is not always obvious which node can fill the area of the leaving node. For example, in Figure 1, the area of node f cannot be merged with any of its neighbors, because this would result in a non-rectangular node-space.

Therefore the node-space is constructed in such a way that the splitting plane forms a *kd-tree* [5]. KD-trees are used only for topology maintenance, similar as in MURK [9], but not as index structures like in database systems. The space of a leaving node can be taken over by a neighboring node which is also a sibling in the kd-tree. By keeping the tree balanced the probability of having a sibling as a neighbor increases. Each node must remember its position in the kd-tree, a bit-string describing the path from the root of the tree to the node itself.

If no neighboring nodes are siblings in the kd-tree, another node must be found to fill the gap. Either a neighboring node additionally takes over the responsibility of the separate area until a free node can be found, or two completely independent nodes that are siblings in the kd-tree have to be found to merge them and thus free a node that takes over the free area. The former concept, called *virtual nodes*, is also used for load-balancing in other systems.

Load Balancing. Load-balancing can be implemented by either adjusting the boundaries of the responsibilities locally or freeing nodes in underloaded areas and moving them to overloaded areas. The former has similar issues as a node leave—the boundaries are interlocked with limited room for adjustments. The latter was shown to be converging [11] with predictable performance.

The balancing can be based on different metrics for load, like object or query load or a combination of both. To avoid thrashing effects a threshold for performing a load-balancing round must be introduced.

4 Empirical Results

For testing the performance of SONAR we used a traveling salesman data set with 1,904,711 cities[1]. The cities' geographical locations follow a Zipf distribution [19] which is also common in other scenarios.

We assigned the responsibility of nodes by recursively splitting the key-space at the longer side, so that each part gets half of the cities until enough rectangles are created. Figure 7 shows a sample splitting for 256 nodes.

The coordinates were mapped onto a doughnut-shaped torus rather than a globe, because in a globe all vertical rings meet at the poles. This would not only

[1] http://www.tsp.gatech.edu/world

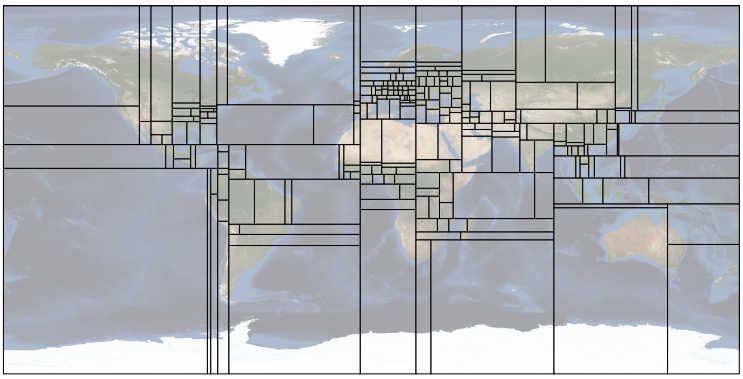


Fig. 7: 1,904,711 cities split evenly into 256 rectangular nodes.

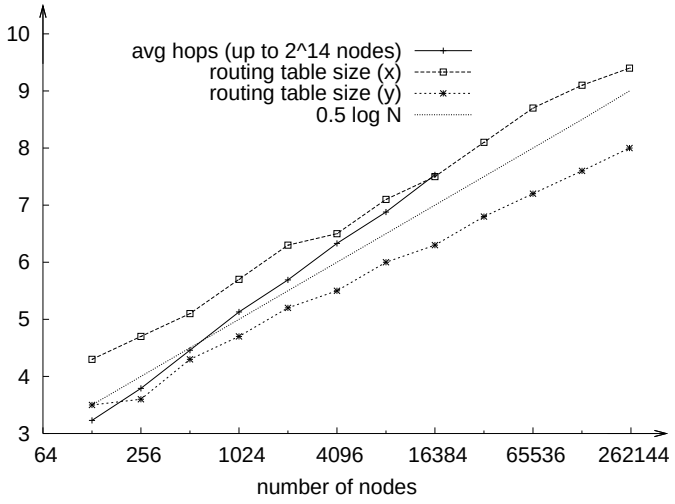


Fig. 8: SONAR results for increasing system sizes (2-dimensional).

cause a routing bottleneck at the poles but would also result in different ring directions for the western and eastern hemisphere (southwards vs. northwards).

Figure 8 shows the results for various all-to-all searches in networks of different sizes. The routing performance, depicted by the '+' ticks, almost perfectly matches the expected $0.5 \log_2 N$ hops. Only in the larger networks the expected value slightly deviates.

We also checked whether the number of fingers in the routing tables, which are calculated without global information (Fig. 4), meets our expectations. The '$\square$' ticks give the routing table sizes in horizontal direction, and the '$*$' ticks represent the sizes in vertical direction. As expected, both graphs have the same

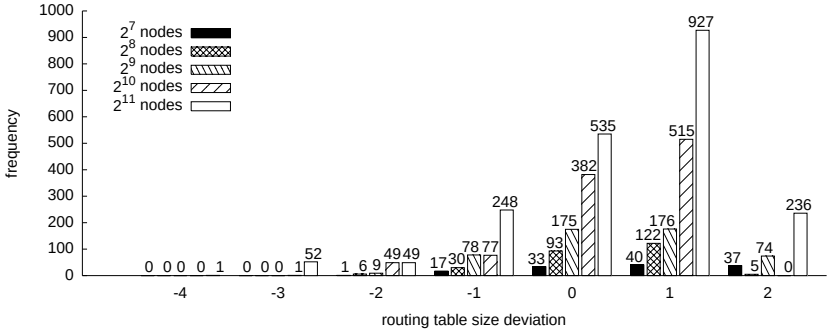


Fig. 9: Routing table size deviation from the expected value.

slope of $0.5 \log_2 N$: One lies consistently above, the other below. This is attributed to the different domain sizes of the coordinate system (360 versus 180 degrees) and to the uneven number of splitting planes.

Figure 9 gives further insight into the characteristics of SONAR's routing tables. It shows—again for various network sizes—the deviation of the table sizes from their expected size $\log_2 N$ (denoted by '0'). As can be seen, the same pattern applies for all network sizes: About 50% of the tables contain one extra entry, about 25% meet the expected size of $\log_2 N$, while there is a decreasing number of tables with fewer entries. These deviations are caused by the uneven key distribution and by SONAR's finger update algorithm which has a tendency to insert in some cases an extra finger that is more than halfway around the ring (but still 'left' of the own node).

5 Conclusion

SONAR efficiently supports range-queries on multi-dimensional data in structured overlay networks. It needs $O(\log N)$ routing steps for processing range-queries of arbitrary shapes and an arbitrary number of attribute domains. The finger calculation needs just one hop for updating an entry in the routing table.

We presented empirical results from a Zipf distributed data set with approximately two million keys. The results confirm that SONAR does its routing with a logarithmic number of hops—even in skewed data distributions. Additional tests with other practical and uniform distributions (not shown here) gave the same logarithmic routing performance. Furthermore, we observed that the sizes of the distributed routing tables are always $O(\log N)$ although they are autonomously maintained by the nodes with local information only.

Acknowledgements

Thanks to the anonymous reviewers for their valuable comments. The topographic images were taken from the 'Blue Marble next generation' project of NASA's Earth Observatory. Thanks to Slaven Rezić for the street map of Berlin.

References

1. K. Aberer, L. Onana Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The essence of P2P: A reference architecture for overlay networks. P2P 2005, 2005.
2. A. Andrzejak and Z. Xu. Scalable, efficient range queries for Grid information services. P2P 2002, 2002.
3. J. Aspnes and G. Shah. Skip graphs. *SODA*, Jan. 2003.
4. F. Banaei-Kashani and C. Shahabi. SWAM: A family of access methods for similarity-search in peer-to-peer data networks. *CIKM*, Nov. 2004.
5. J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, Vol. 18, No. 9, 1975.
6. A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. *ACM SIGCOMM 2004*, Aug. 2004.
7. Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A Case Study in building layered DHT applications. SIGCOMM'05, Aug. 2005.
8. V. Gaede and O. Günther. Multidimensional access methods. ACM Computing Surveys, 30 (2), 1998.
9. P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in P2P systems. *WebDB 2004.*
10. D. Karger, E. Lehman, T. Leighton, R. Panigrah, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. *ACM Sympos. Theory of Comp.*, May 1997.
11. D. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. *IPTPS 2004*, Feb. 2004.
12. D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. *14th IEEE Symposium on High Performance Distributed Computing (HPDC-14)*, Jul. 2005.
13. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, June 1990.
14. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *ACM SIGCOMM 2001*, Aug. 2001.
15. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. *Proceedings of the USENIX Annual Technical Conference*, Jun. 2004.
16. C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 19-26, May/June 2004.
17. T. Schütt, F. Schintke, and A. Reinefeld. Structured overlay without consistent hashing: Empirical results. *GP2PC'06*, May. 2006.
18. 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.
19. G. Zipf. Relative frequency as a determinant of phonetic change. *Harvard Studies in Classical Philiology*, 1929.

B Handling Network Partitions and Mergers in Structured Overlay Networks

Handling Network Partitions and Mergers in Structured Overlay Networks *

Tallat M. Shafaat† Ali Ghodsi‡ Seif Haridi†

†Royal Institute of Technology (KTH), ‡Swedish Institute of Computer Science (SICS)
{tallat,haridi}(at)kth.se ali(at)sics.se

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 to be difficult or impossible. We evaluate the algorithm for various scenarios and show that even when false 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.

1 Introduction

Structured Overlay Networks (SONs)—such as Chord [28], Pastry [26], and SkipNet [13]—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 those of distributed databases [5] and distributed file systems [29].

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, most SONs cope with network partitions, but not with network mergers. We believe that this is because a network partition, as seen from the perspective a single node, is identical to massive node failures. Since SONs have been designed to cope with churn, they can self-manage in the presence of such partitions. However, 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 poorly fit for dealing with network mergers. Datta *et al.* [4] 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". Birman [2] argues that ring-based SONs are inherently ill-suited for dealing with 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 [11, 3, 9].

*This research has been funded by the European Project SELFMAN, VINNOVA 2005-02512 TRUST-DIS, and SICS Center for Networked Systems (CNS).

[1]By availability we mean that a get/put operation should eventually complete.

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 [29, 6].

In this paper, we present an algorithm for merging any number of similar structured overlays. We will limit ourselves to ring-based overlays, since they constitute the majority of the SONs. It is desirable that a solution to the problem of merging rings takes minimum amount of time to complete (time complexity). At the same time, it is desirable that the solution has a minimal bandwidth consumption (message and bit complexity). These two goals are conflicting, as shown by the following two extreme cases. On the one hand, it is possible to construct an algorithm that completes in minimal time by having all the nodes repeatedly spreading all their routing information to every other node through an overlay broadcast [7, 10, 9]. On the other hand, it is possible to construct an algorithm which tries to minimize the bandwidth consumption by passing a "merging" token around each of the rings. Hence, it is desirable to find an algorithm which strikes a balance between time, bit, and message complexity.

The contribution of this paper is a ring merging algorithm, which allows the system designer to adjust, through a *fanout* parameter, the tradeoff between message complexity and 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 widely believed to be to be difficult [2] or impossible [4]. 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.

Outline Section 2 serves as a background by motivating and defining our choice of ring-based SONs. Section 3 introduces the *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.

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 Ring Geometry The reason for confining ourselves to ring-based SONs is twofold. First, ring-based SONs constitute a majority of the SONs, including Chord [28], Pastry [26], SkipNet [13], DKS [9], Koorde [16], Viceroy [23], Mercury [1], Symphony [24], Epi-Chord [17], and Accordion [18]. Second, Gummadi *et al.* [12] diligently compared the geometries of different SONs, 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.

Our results apply to all ring-based SONs. Nevertheless, we assume a SON similar to Chord [28] to simplify the understanding of our algorithms.

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 $\{0, 1, \cdots, N-1\}$, where N is some apriori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $N-1$.

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 $2\log_2(n)$ in an n node system.

Ring-based SONs also maintain additional routing pointers on top of the ring to enhance routing. To keep things concrete, assume that these are placed as in Chord. Hence, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod N$ for $0 < i < \log_2(N)$. 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 $succ$ node 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 [20], it also recovers from network partitions, making each component of the partition eventually form its own ring. Our simulation results confirm this, though they are omitted due to space constraints. The problem that remains unsolved, which is the focus of the rest of the paper, is how several independent rings efficiently can 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, in its passive list. Every node periodically pings nodes in its passive list to detect if a failed node is again alive. 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 lists 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. It might also be the case that a node with the same address and identifier as a failed node joins the ring. The ring merging algorithm should be able to cope with the first case, by trying to ensure that such false-positives will terminate the algorithm quickly. The latter case can be dealt with by associating with every node a globally unique random nonce, which is generated each time a node joins the network. Hence, a new node can always be differentiated from an old node with the same 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. 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. 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 n
2:      id := detqueue.dequeue()
3:      sendto n : MLOOKUP(id)
4:      sendto id : MLOOKUP(n)
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 n, and if it is non-empty the first node p in the list is picked to start a ring merger. Ideally, n and p will be on two different rings. But even so, the distance between n and p 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($cpred, csucc$), which tries to do the actual merging by updating $succ$ and $pred$ pointers.

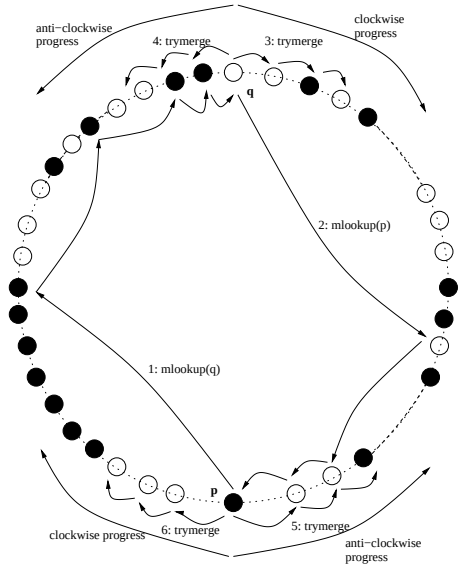


Figure 1: Filled circles belong to SON1 and empty circles belong to SON2. The algorithm starts when p detects q, p makes an MLOOKUP to q and asks q to make an MLOOKUP to p. Steps $3-6$ execute in parallel.

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 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.

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 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.

Algorithm 2 Gossip-based Ring Unification Algorithm

```
1:  every γ time units and detqueue ≠ ∅ at n
2:      ⟨id, f⟩ := detqueue.dequeue()
3:      sendto n : MLOOKUP(id, f)
4:      sendto id : MLOOKUP(n, 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:              r := randomnodeinRT()
10:             at r : detqueue.enqueue(⟨id, f⟩)
11:             f := f − 1
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, partly gossip-based. The algorithm is essentially the same as the simple ring unification algorithm, but it starts multiple such mergers at random places on the rings. The basic idea is to augment MLOOKUP(id), such 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 used to avoid that the algorithm consumes too many messages, and therefore gives rise to positive feedback cycles which 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 only is 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 or equal to 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.

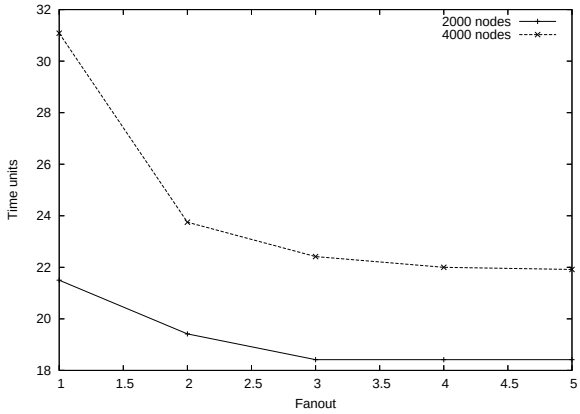


Figure 2: Evaluation of a typical scenario with multiple nodes detecting the merger for various *network sizes* and *fanouts*.

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. If not said otherwise, message complexity is until termination, while time complexity is until completion.

We first evaluate the message and time complexity of the algorithms in the typical scenario where many nodes simultaneously detect alive nodes in their passive lists. For a worst case scenario, we evaluate the algorithms when only a single node detects the existence of another ring. Thereafter, we evaluate the performance of the algorithms while joins and failures are taking place during the ring merging process. Finally, we evaluate message complexity of the algorithms when a node falsely believes that it has detected another ring.

The evaluations are done in a stochastic discrete event simulator, 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.

Each 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, on which the simulator divides the set of nodes into as many components as requested by the partition event. A partition event is implemented using lottery scheduling [31] 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 Section 2 and 3). Thereafter, a network merger event simply again 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 our simulation graphs, a fanout of 1 represents the simple ring unification algorithm. As the simulations show, fanout values less than 5 create very few messages even with thousands of nodes in the system.

Figure 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 unification algorithm. As can be seen in Figures 2 and 3, the simple ring unification algorithm ($F = 1$) consumes minimum messages but takes maximum time. 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 messages. Figure 4 shows a tradeoff between time complexity and message complexity. Choosing to have less time for completion of mergers will create more messages, and vice versa.

For the rest of the evaluations, we use a worst case scenario where only a single node detects the merger.

Next, we evaluate rings unification under churn, *i.e.* nodes join and fail during the merger. The algorithm may fail to complete and the merged overlay may not converge under churn, especially for simple ring unification and low

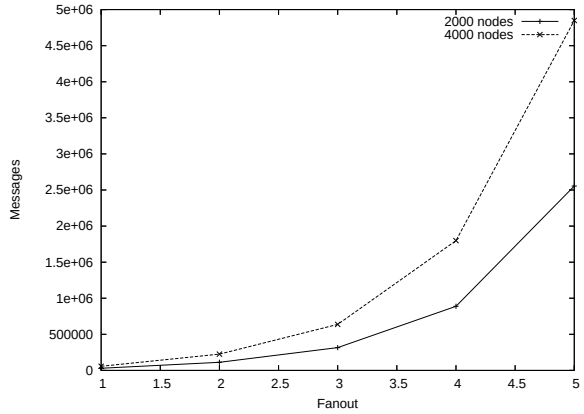


Figure 3: Evaluation of a typical scenario with multiple nodes detecting the merger for various *network sizes* and *fanouts*.

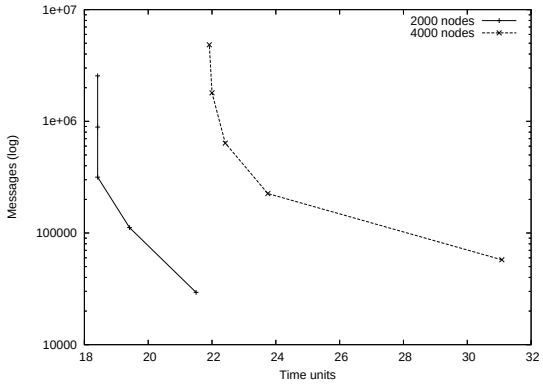


Figure 4: Evaluation of a typical scenario with multiple nodes detecting the merger for various *network sizes* and *fanouts*.

fanouts. 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. Thus, if only one node detects the merger, with churn, there is a non-zero probability that the rings will not converge. With higher values of F, the algorithm becomes more robust to churn as it creates multiple MLOOKUPs. The results presented in Figure 6 and 7 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. Figure 6 and 7 show how different values of F affect the convergence of the rings under different levels of churn.

Finally, we evaluate the scenario where a node may falsely detect a merger. Figure 5 shows the message complexity of the algorithm in case of a false detection. As can be seen, for lower F values, the message complexity is less. Even for higher fanouts, the number of messages generated are less, 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.

Our simulations show that a fanout value of 4 is good for a system with several thousand nodes, even with respect to churn and false-positives.

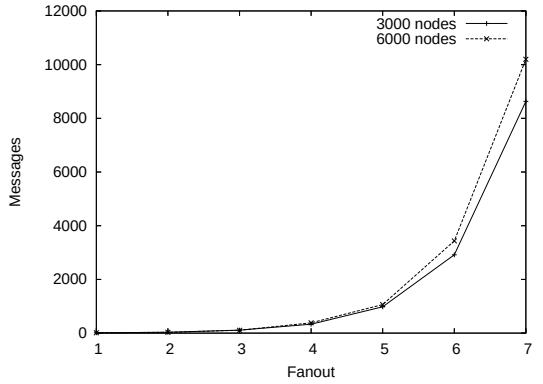


Figure 5: Evaluation of message complexity in case a node falsely detects a merger for various *network sizes* and *fanouts*.

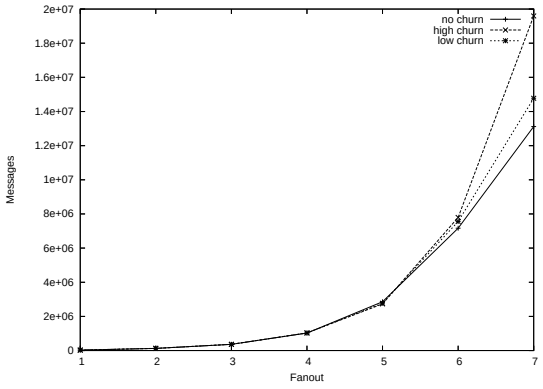


Figure 6: Evaluation of message complexity under churn

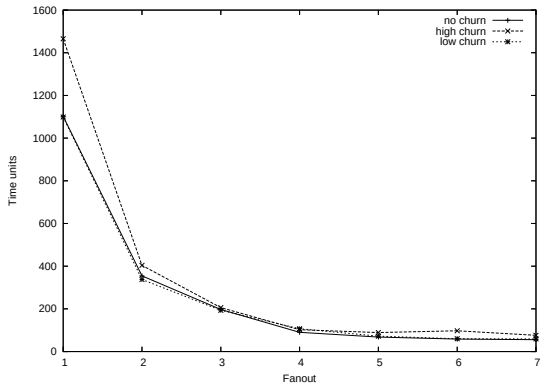


Figure 7: Evaluation of time complexity under churn

5 Related Work

Much work has been done to study the effects of churn on a structured overlay network [22], showing how overlays can cope with massive node joins and failures, thus showing how overlays are resilient to partitions. Datta *et al.* [4] 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. Birman [2] argued that ring-based SONs are inherently ill-suited for dealing with network partitions, while we show how ring-based SONs can be modified to deal with partitions.

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.* [27] 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 [21, 19, 9]. In contrast, our algorithm is a terminating algorithm that works as a plug-in for an already existing SON.

Montresor *et al.* [25] show how Chord [28] can be created by a gossip-based protocol [14]. However, their algorithm depends on an underlying membership service like Cyclon [30], Scamp [8] or Newscast [15]. Thus the underlying membership service has to first cope with network mergers (a problem worth studying in its own right), whereafter 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.

The problem of network partitions and mergers has been studied in other distributed systems like in distributed databases [5] and distributed file systems [29]. These studies focus on problems created by the partition and merger on the data level, while we focus on the routing level.

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. Our algorithm is quite fast compared to the basic linear solution presented by Datta *et al.* [4]. 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.

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.

References

[1] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols*, pages 353–366, Portland, OR, USA, March 2004. ACM Press.

[2] Ken Birman. Gossip Algorithms and Emergent Shape. Invited talk at the Workshop on Gossip-based Computer Networking at the Lorentz Center, Leiden, Netherlands, December 2006.

[3] E. Brewer. Towards Robust Distributed Systems, invited talk at the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00), 2000.

[4] A. Datta and K. Aberer. 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)*, volume 4124 of *Lecture Notes in Computer Science (LNCS)*, pages 7–22. Springer-Verlag, 2006.

[5] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[7] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. Efficient Broadcast in Structured P2P Netwoks. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 304–314, Berkeley, CA, USA, 2003. Springer-Verlag.

[8] A. J. Ganesh, A.-M. Kermarrec, and L Massoulié. 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)*, volume 2233 of *Lecture Notes in Computer Science (LNCS)*, pages 44–55, London, UK, 2001. Springer-Verlag.

[9] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, December 2006.

[10] A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In *Proceedings of the 15th International Conference, Parallel and Distributed Computing and Systems*, Marina del Rey, CA, USA, November 2003.

[11] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Special Interest Group on Algorithms and Computation Theory News*, 33(2):51–59, 2002.

[12] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM 2003 Symposium on Communication, Architecture, and Protocols*, pages 381–394, New York, NY, USA, 2003. ACM Press.

[13] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[14] M. Jelasity and Ö. Babaoglu. T-man: Gossip-based overlay topology management. In *Proceedings of 3rd Workshop on Engineering Self-Organising Systems (EOSA'05)*, volume 3910 of *Lecture Notes in Computer Science (LNCS)*, pages 1–15. Springer-Verlag, 2005.

[15] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast Computing. Technical Report IR–CS–006, Vrije Universiteit, November 2003.

[16] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *Proceedings of the 2nd Interational Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 98–107, Berkeley, CA, USA, 2003. Springer-Verlag.

[17] B. Leong, B. Liskov, and E. Demaine. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. In *12th International Conference on Networks (ICON'04)*, Singapore, November 2004. IEEE Computer Society.

[18] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005. USENIX.

[19] X. Li, J. Misra, and C. G. Plaxton. Brief Announcement: Concurrent Maintenance of Rings. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, page 376, New York, NY, USA, 2004. ACM Press.

[20] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger. Observations on the Dynamic Evolution of Peer-to-Peer Networks. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, volume 2429 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2002.

[21] N. A. Lynch, D. Malkhi, and D. Ratajczak. 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), pages 295–305, London, UK, 2002. Springer-Verlag.

[22] R. Mahajan, M. Castro, and A. Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 21–32, Berkeley, CA, USA, 2003. Springer-Verlag.

[23] 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.

[24] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[25] A. Montresor, M. Jelasity, and Ö. Babaoglu. Chord on Demand. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*. IEEE Computer Society, August 2005.

[26] A. Rowstron and P. Druschel. 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)*, volume 2218 of *Lecture Notes in Computer Science (LNCS)*, pages 329–350, Heidelberg, Germany, November 2001. Springer-Verlag.

[27] 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.

[28] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[29] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 172–183. ACM Press, December 1995.

[30] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), 2005.

[31] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11. USENIX, November 1994.

C PEPINO: PEer-to-Peer network INspectOr

PEPINO: PEer-to-Peer network INspectOr *

Donatien Grolaux, Boris Mejías, and Peter Van Roy
Université catholique de Louvain, Belgium
firstname.lastname@uclouvain.be

Abstract

PEPINO is a simple and effective peer-to-peer network inspector. It visualises not only meaningful pointers and connections between peers, but also the exchange of messages between them, providing a useful tool for debugging purposes. It can monitor running networks, simulate them and log them in order to reproduce interesting case scenarios. Failures can be explicitly introduced to study fault tolerant algorithms. The graphical representation of the network uses a physical model to attract or repel peers, allowing the user to study the system from different points of view. This demo aims to present the use of PEPINO in the development of a novel relaxed-ring topology for fault tolerant networks, where the representation of the ring based on predecessors may differ from the ring based on successors. We show how PEPINO is also useful for visualising other network topologies such as perfect ring or unstructured networks.

1. Introduction

Developing peer-to-peer systems requires the ability of visualising the network in terms of the designed algorithms. The more dynamic the case-studies become, the harder is to keep track of all interaction between peers. The obvious solution is to use a software to graphically represent the network, and thus, nearly every developer group creates its own network viewer to study their algorithms. Then, why do we present yet another viewer? The reason to do it is because existing tools are so ad-hoc to each network that studying a different network topologies in such tools did not allowed us to visualise our concrete issues.

We developed PEPINO, a PEer-to-Peer network INspectOr that adapts its graphical representation to several network topology such as ring, relaxed-ring, unstructured networks or even client-server. The graphical representation uses a physical model to attract and repel connected nodes depending on the weight of each kind of connection. This is how the viewer is able to dynamically adapt itself to different network architectures, without making any previous assumption on the topology.

Another crucial information for debugging is the exchange of messages between peers. PEPINO displays simultaneously with the representation of the network, the communication between peers and the events triggered by each one of them. Messages and events are annotated with different categories, allowing filters in order to get more meaningful information.

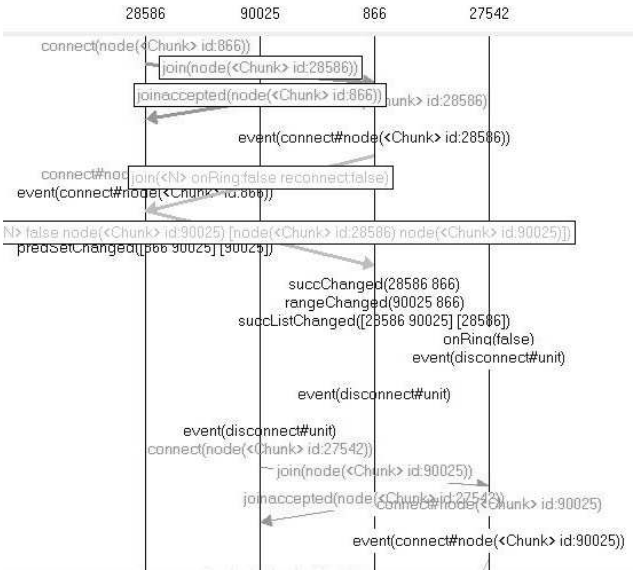


Figure 1. Message exchange between peers

2. PEPINO and P2PS

We use PEPINO in the development of P2PS*v*3 [2], a Chord-like platform to develop fault-tolerant peer-to-peer applications. P2PS uses a network topology based on a relaxed-ring where only the predecessor links form a perfect ring, guaranteeing a correct distribution the responsi-

*This research is mainly funded by EVERGROW (contract number:001935) and SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

bility of the keys. Successors follow a different invariant allowing the presence of branches in the ring. The topology and the algorithms of P2PS provide a network that can survive efficiently to failures of nodes and also to broken links (inaccurate failure detection), which are often ignored.

P2PS is implemented using a software architecture based on tiers, where the lowest tier implements point-to-point communication. The relaxed-ring maintenance is another layer placed upper in the architecture. As we previously mentioned, PEPINO can display messages between peers in different categories. In the particular case of P2PS, every tier is represented by a category. Figure 1 depicts how messages are represented, and how one category is highlighted. The figure is shown in grey scale, but it is possible to distinguish that every category has its own colour. Every category can be enabled or disabled in order to avoid unnecessary verbosity.

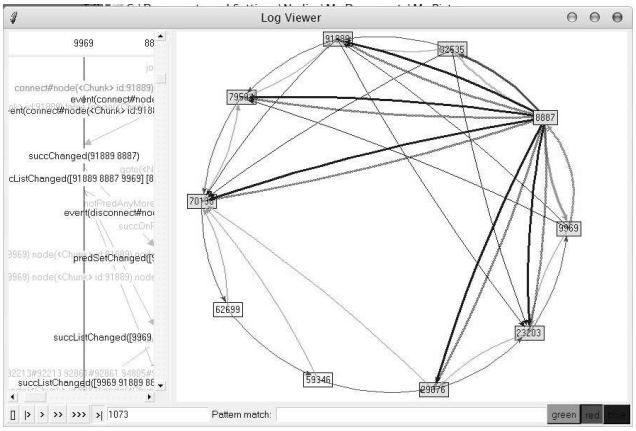


Figure 2. Fingers on a ring network

During the demonstration, different networks will be presented in order to observe their dynamic behaviour. Figure 2 is a screenshot of PEPINO visualising a network using ring topology. On the left side of the screenshot is possible to observe the frame with messages between peers. On the right side, in the graphical representation of the ring, the finger of a particular peer are highlighted. On the bottom right corner, there is a set of buttons allowing the election of the strongest connector between peers for the graphical representation. The underlay physical model will adapt the parameter for attraction or repulsion of nodes according to these settings. Like this, it is possible to observe the network from different points of view. For instance, putting the focus on the predecessors or successor links.

To check how the network reacts to network failures, it is possible to explicitly inject temporary or permanent failures on nodes. Failures can also be injected in the communication channel between peers, which is one way to study inaccuracy of failure detection.

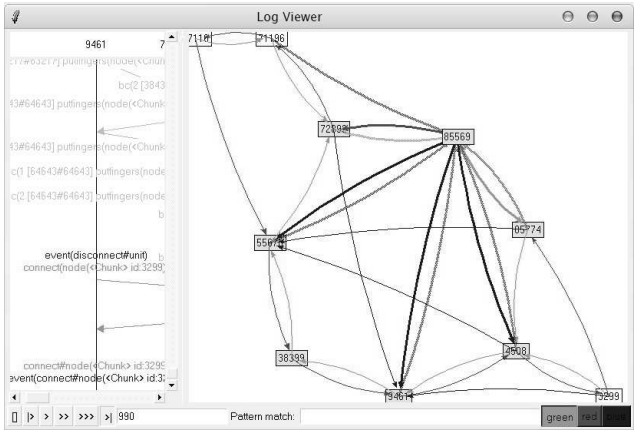


Figure 3. Relaxed-ring topology

Figure 3 depicts the visualisation of the relaxed-ring topology of P2PS, our main focus of interest for the demonstration. The information given by PEPINO helps the developer to understand why branches are created, and how the network recovers from crashed peers and broken links. P2PS implements different algorithms for the election of fingers such as Chord [5], Tango [1] and DKS [3], allowing the comparison between them.

PEPINO is also useful for bug reports. The history of a visualised network can be saved in a log file to be sent to developers. The log can be visualised at different speeds. In figures 2 and 3, a set of arrows can be seen at the bottom left corner. The speed of visualisation can be tuned with those buttons. It is also possible to run the visualisation until a particular event identified by a number, or matching a pattern.

PEPINO is implemented with Mozart [4], and it can be run on Linux, MacOSX and other Unix systems. It also runs on Windows 98/NT/XP.

References

[1] B. Carton and V. Mesaros. Improving the scalability of logarithmic-degree dht-based peer-to-peer networks. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 1060–1067. Springer, 2004.

[2] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. *http://gforge.info.ucl.ac.be/projects/p2ps*, 2007.

[3] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, Dec. 2006.

[4] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org, 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.

D Improving the Peer-to-Peer Ring for Building Fault-Tolerant Grids

IMPROVING THE PEER-TO-PEER RING FOR BUILDING FAULT-TOLERANT GRIDS

Boris Mejías and Donatien Grolaux and Peter Van Roy
Université catholique de Louvain, Belgium *
{bmc|ned|pvr}@info.ucl.ac.be

Abstract

Peer-to-peer networks are gaining popularity in order to build Grid systems. Among different approaches, structured overlay networks using ring topology are the most preferred ones. However, one of the main problems of peer-to-peer rings is to guarantee lookup consistency in presence of multiple joins, leaves and failures nodes. Since lookup consistency and fault-tolerance are crucial properties for building Grids or any application, these issues cannot be avoided. We introduce a novel relaxed-ring architecture for fault-tolerant and cost-efficient ring maintenance. Limitations related to failure handling are formally identified, providing strong guarantees to develop applications on top of the relaxed-ring architecture. Besides permanent failures, the paper analyses temporary failures and broken links, which are often ignored.

Keywords: Peer-to-peer, relaxed-ring, fault-tolerance, lookup consistency, ring maintenance.

1. Introduction

Classical approaches for building Grid systems using resource and service discovery are mainly centralised or hierarchical. Since centralised networks present the weakness of having single point of failure, peer-to-peer networks are gaining popularity as an alternative decentralised choice. Building decentralised applications requires several guarantees from the underlay peer-to-peer network. Fault-tolerance and consistent lookup of resources are crucial properties that a peer-to-peer system must provide. Structured overlay network providing a Distributed Hash Table (DHT) using Chord-like ring topology [10]are a popular choice to solve the requirements of efficient routing, lookup consistency and accessibility of all resources. But all these properties are compromised in

*This research is mainly funded by EVERGROW (contract number:001935) and SELFMAN (contract number: 034084), with additional funding by CoreGRID (contract number: 004265).

presence of failure or high churn (multiple peers joining or leaving in very short time).

The benefits of using peer-to-peer systems have been already stated in previous CoreGRID results [12, 11], but the problems related to fault-tolerance has not been deeply addressed. A high level approach is proposed in [2], where the failure detection and self-organisation of the network is entirely delegated to the peer-to-peer system. Since this work addresses these issues precisely at the low level, it can be seen as a complementary result.

Despite the self-organising nature of the ring architecture, its maintenance presents several challenges in order to provide lookup consistency at any time. Chord itself presents temporary inconsistency with massive peers joining the network, even in fault-free systems. A stabilisation protocol must be run periodically to fix these inconsistencies. Existing analyses conclude that the problem comes from the fact that joins and leaves are not atomic operations, and they always need the synchronisation of three peers. Synchronising three peers is hard to guarantee with asynchronous communication, but this is inherent to distributed programming.

Existing solutions [7–8]introduce a locking system in order to provide atomicity of join and leave operations. Locks are also hard to manage in asynchronous systems, and that is why these solutions only work on fault-free systems, which is not realistic. A better solution is provided by DKS [5], simplifying the locking mechanism and proving correctness of the algorithms in absent of failures. 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 locked peers must be suspended in presence of join or leave in order to guarantee consistency. Leaving peers are not allowed to leave the network until they are granted with the relevant locks. Given that, peers crashing can be seen as peers just leaving 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. Then, locks in a distributed system can hardly present a fault-tolerant solution.

We have developed an algorithm that only needs the agreement of two nodes at each stage, which is easier to guarantee given point-to-point communication. This decision leads us to a relaxed-ring topology, simplifying the joining algorithm and becoming fault tolerant to permanent or temporary failures of nodes, and also to broken links, which are often ignored by existing approaches.

The following section describes the relaxed-ring architecture and its guarantees. We continue with further analysis of the topology and its fault tolerant behaviour, ending with conclusions.

2. P2PS's relaxed-ring

The relaxed-ring topology is part of the new version of P2PS [4], which is designed as a modular architecture based on tiers. The whole system is implemented using the Mozart-Oz programming system [9], where the lowest level tier implements point-to-point communication between peers. Some layer upper to this one, we implement the maintenance of the relaxed-ring topology, which is the focus of this paper. This layer can correctly route lookup requests providing consistency. Other layers built on top of this one are in charge of providing efficient routing, reliable message sending, broadcast/multicast primitives and naming services. All these layers provide efficient support for building decentralised systems such as grid based on services architectures like P2PKit [6].

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 providing efficient routing. Ring's key-distribution is formed by integers from 0 to N 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 '$()$' excludes a key from the range and square brackets '$[]$' includes it.

2.1 The relaxed-ring

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. The lookup consistency is guaranteed between every step and 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, because anyway, we have to deal with leaves due to network failures.

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 plus 1, 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 k is the root of a branch, and where the connection between peers h and i is broken.

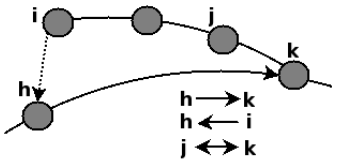


Figure 1. The relaxed-ring architecture

Having the relaxed-ring architecture, we introduce a new principle that modifies the routing mechanism. The principle is that *a peer can never indicate another peer as responsible for a key*. This implies that even when the successor of a peer seems to be the responsible of a key, the request must be forwarded to the successor. Considering the example in figure 1, h may think that k is the responsible for keys in the interval $(h, k]$, but in fact there are three other nodes involved in this range. Note that the forwarding of a lookup request can be directed forward of backward with respect to the key distribution. It has been proved that this modification to the usual routing mechanism does not creates cycles and always converge.

Before starting the description of the algorithms that maintain the relaxed-ring topology, we first 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.*

When a new peer wants to join the ring, first, it gets its own identifier from a random key-generator. At this starting point, the node does not have a successor ($succ$), then, it does not belong to any ring, and it does not know its predecessor ($pred$), so obviously, it does not have responsibilities. 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 several Chord-alike systems. When the responsible of the key contacts the new peer, it begins the join algorithm that will be discussed in the next section.

2.2 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. When peer r replies the lookup request to q, and q send the $join$ message to r triggering the joining process.

The first step is described in algorithm 1, 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.

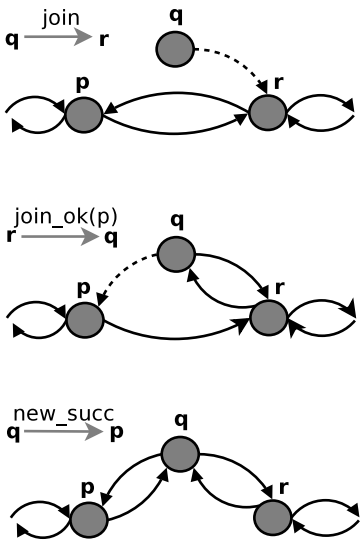


Figure 2. The join algorithm.

If it is possible to perform the join, peer r verifies that peer q is a better predecessor. Function $betterPredecessor$ just checks if the key of the joining peer is in the range of responsibility of the current peer in the case of a regular join. 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 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 initialised. 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. 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 2. 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

Algorithm 1 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
```

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.

Algorithm 2 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
```

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. Note that message $upd_succlist$ is for resilient purposes. It updates the list of successors that will be used for the recovery of a failure detected in the successor. The following theorem states the guarantees of the relaxed ring concerning the join algorithm.

THEOREM 2.1 *The relaxed-ring join algorithm guarantees consistent lookup at any time in presence of multiple joining peers.*

PROOF 1 *Let us assume the contrary. There are two peers p and q responsible for key k. In order to have this situation, p and q must have the same predecessor j, sharing the same range of responsibility. This means that $k \in (j, p]$ and $k \in (j, q]$. 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.

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.

Every node monitors the communication with every peer it is connected to. If a failure is detected, the $crash$ event is triggered as it is described in algorithm 3. 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 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 analogue. 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.

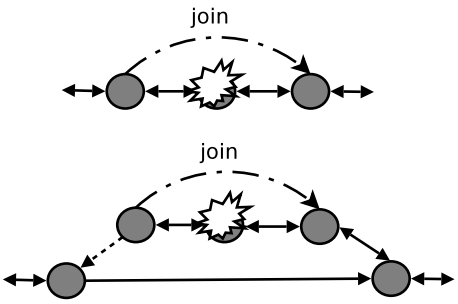


Figure 3. Simple crashes.

When the detected peer p is the predecessor, no recovery mechanism is triggered because p's predecessor will contact the current peer. The algorithm decides a predecessor candidate from the $predlist$ to recover from the case when the tail of a branch is the crashed peer. 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.

The $alive$ event is triggered when a link recovers from a temporary failure. This can be implemented by using watchers or a fault stream per distributed entity [3]. If the peer is alive, it is enough to remove it from the $crashed$ set. This will terminate any pending recovery algorithm.

Algorithm 3 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:     else if (p == pred) then
10:        if (predlist ≠ ∅) then
11:            pred_candidate := getLast(predlist)
12:        end if
13:    end if
14: end event

15: upon event ⟨ alive | p ⟩ do
16:    crashed := crashed \ {p}
17: 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. Using the $crashed$ set, function $betterPredecessor$ can check fault status. 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.

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.

Since failures are not detected by all peers at the same time, redirection during recovery of failures may end up in a faulty node. Then, the $goto$ event must be modified such that 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.

Cases hard to handle are broken links and crashes at 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 reaches accuracy. 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.

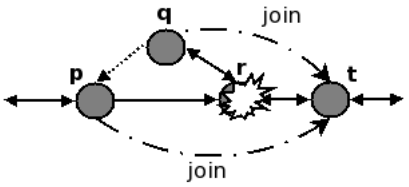


Figure 4. The failure of the root of a branch triggers two recovery events

THEOREM 3.1 *Simultaneous failures of nodes never introduce inconsistent lookup as long as there is no network partition.*

PROOF 2 *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.*

With respect to network partitions, there are two important cases we want to analyse. The crash of a branch's root, and the isolation of a set of nodes from

the rest of the ring. The isolation problem can occur in any system using ring topology, and it can involve consecutive peers or peers distributed all over the ring. Network partitioning introducing temporary uncertainty has been proved by Ghodsi [5], and it is related to the proof provided in [1]about limitations of web services in presence of network partitioning.

Figure 4 depicts a network partition that can occur in the relaxed-ring topology. The proof of theorem 3.1 is based on the fact that per every failure detected, there is 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 3.2 *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 3 *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 3.2 clearly states the limitation of branches in the systems, helping developers to identify the scenarios requiring 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.

4. Conclusion

The amount of Grid systems built on top of peer-to-peer networks is increasing. Since Grid users design their application at a higher level, it is reasonable to assume that failure handling will the delegated to the peer-to-peer system. This is why its crucial to provide a robust fault-tolerant network.

In this paper we have presented a novel relaxed-ring topology for fault-tolerant and self-organising peer-to-peer networks. The topology is derived from the simplification 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, allowing the system to work in realistic scenarios, providing fault-tolerant ring maintenance.

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. Having these guarantees, solving issues related to network partitioning become more addressable.

References

[1] Eric A. Brewer. Towards robust distributed systems (abstract). In PODC âŁ™00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, page 7, New York, NY, USA, 2000. ACM Press.

[2] Denis Caromel, Alexandre di Costanzo, and Christian Delbé . Peer-to-peer and fault- e tolerance: Towards deployment-based technical services. Future Generation Computer Systems, 2007. To appear.

[3] Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed e programming language. In Advanced Topics in Exception Handling Techniques, pages 121-140, 2006.

[4] DistOz Group. P2PS: A peer-to-peer networking library for Mozart-Oz. http://gforge.info.ucl.ac.be/projects/p2ps, 2007.

[5] Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD dissertation, KTH - Royal Institute of Technology, Stockholm, Sweden, December 2006.

[6] Kevin Glynn. P2PKit: A services based architecture for deploying robust peer-to-peer applications. http://p2pkit.info.ucl.ac.be/index.html, 2007.

[7] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Active and concurrent topology maintenance. In DISC, pages 320-334, 2004.

[8] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Concurrent maintenance of rings. Distributed Computing, 19(2):126-148, 2006.

[9] Mozart Community. The Mozart-Oz programming system. http://www.mozart-oz.org.

[10] 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.

[11] Domenico Talia, Paolo Trunfio, Jingdi Zeng, and Mikael Heqvist. A peer-to-peer framework for resource discovery in large-scale grids. In Proc. of the 2nd CoreGRID Integration Workshop, pages 249-260, Krakow, Poland, October 2006.

[12] Paolo Trunfio, Domenico Talia, Paraskevi Fragopoulou, Charis Papadakis, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov, and Seif Haridi. Peer-to-peer models for resource discovery on grids. In Proc. of the 2nd CoreGRID Workshop on Grid and Peer to Peer Systems Architecture, Paris, France, January 2006.