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:	D2.2a Report on Architectural Framework Specification
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:	KTH
Revision:	0.6
Dissemination level:	CO

Contents

1 Executive summary

The main objective of workpackage WP2 is to design and implement the SELFMAN service architecture, a component-based architecture for large, self-managing distributed systems. The architecture relies on the structured overlay networks developed in WP1 for its basic distributed services. The component-based architecture comprises: (1) a reflective component-based computational model, (2) together with its formal semantics, and (3) a component-based architectural framework, comprising abstractions, design patterns, and basic infrastructure services.

This deliverable is responsible for specifying the architectural framework that embodies the SELFMAN service architecture. It includes

- an advanced component model (such as Fractal [1]) with associated execution supports and tools and services participating in an autonomic infrastructure:
- services used for large distributed systems monitoring (probes, sensors), probes, events filtering, aggregation, composition, diagnosis, decision making, reconfiguration, etc;
- transactional dynamic reconfiguration support for component-based systems providing ACID properties where consistency is defined based on architectural constraints;
- an Event-Condition-Action rules mechanism providing a basic mechanism for decision-making in autonomic systems, where rules are implemented as hierarchical Fractal [1] components;
- an architecture for monitoring large-scale distributed systems allowing for flexible topologies of domains including hierarchical and overlapping domains so as to support the different needs of multiple management applications;
- an event-driven component-oriented middleware for building self-managing and self-organizing distributed applications, providing basic mechanisms for scalable connection-oriented communication, failure detection, overlay name-based routing, group communication and a distributed hash table abstraction.
- the P2PKit SON service architecture adapted to the new P2PS relaxed-ring topology described in deliverable D1.1.

2 Contractors contributing to the Deliverable

FT R&D(P4), KTH(P2) and UCL(P1) have contributed to this deliverable.

FT R&D(P4) The contribution of France Telecom to this deliverable is twofold: i) a continuous work on the Fractal [1] component model (notably with INRIA), platforms and tools ; and ii) an anticipation work on a self-management architectural framework.

In the first line of work (basic component model), France Telecom has recently developed the tool 'Fractal ADL Dumper' which allows to store/restore components (with their state) into Fractal ADL XML format. The tool is available as open source in the ObjectWeb/Fractal code base.

In the second line of work, France Telecom has developed some extensions of Fractal that can be seen as core building blocks ("enablers") of a component-based autonomic infrastructure. The extensions concern: transactional reconfiguration of component architectures, an architecture for the aggregation of probes/sensors, an architecture for large scale management of distributed systems, an active rules (Event-Condition-Action) mechanism, a new version of the CLIF load injector and performance analysis and reporting.

It is worth noting that these works are framed by the development of the M2M use case in WP5 (deliverable D5.1 on user requirements) - hence they are seen as contributions both to WP2 and WP5 (user requirements and evaluations). Works on performance analysis and their usage in self-optimization scenarios can also be seen as contributions to WP3 (self-optimization services)

France Telecom contributors to this deliverable are (in alphabetical order): T. Coupaye (Senior Researcher), B. Dillenseger (Senior Researcher), A.Diaconescu (Junior Researcher), A. Harbaoui (PhD Student), N. Jayaprakash (PhD Student), M. Kessis (PhD Student), A. Lefebvre (Senior Researcher), M. Leger (PhD Student).

KTH(P2) The contribution of KTH to this deliverable consists of designing and implementing a component-based event-driven middleware architecture for building Internet-scale, dynamic, distributed applications. The middleware provides basic mechanism for self-managing and self-organizing distributed applications, such as: scalable connection-oriented communication, failure detection, overlay name-based routing, group communication and a distributed hash table abstraction. Our middleware implements the Distributed k-ary System [3] structured overlay network.

The middleware is architectured as event-driven components executed by an adaptive number of threads, automatically accommodating multicore machines. Components interact by exchanging events through a publish-subscribe interface.

KTH contributors to this deliverable are (in alphabetical order): Cosmin Arad (PhD Student), Seif Haridi (Senior Researcher), Roberto Roverso (Research Engineer).

UCL(P1) The contribution of UCL to this deliverable consists of adapting the service architecture of the P2PKit structured overlay network to the new relaxed-ring topology of P2PS. This is ongoing work that will be published during the next period of the project.

UCL contributors to this deliverable are (in alphabetical order): Yves Jaradin (PhD Student), Boriss Mejias (PhD Student), Peter Van Roy (Senior Researcher).

3 Results

3.1 Fractal-based autonomic infrastructure

We consider here that an autonomic system is composed of an autonomic infrastructure superimposed on a target component-based system seen as an overlay network. The autonomic infrastructure is responsible for implementing a control loop, i.e. instrumenting the components of the target system for monitoring, detecting, filtering, aggregating, composing and notifying events, diagnosing the system based on these events, and making decisions to determine what and how corrective actions need to be executed, and finally, executing the corrective actions on the target component-based system. We are targeting as much as possible a component-based implementation of autonomic control loops so as to envision the autonomic management of autonomic infrastructures.

France Telecom has produced 4 extensions of the Fractal [1] component framework (devised by the ObjectWeb open source consortium) participating of an autonomic component-based infrastructure:

- a transactional support for components dynamic reconfigurations. The mechanism includes support for concurrency and recovery (including state store/restore). The ACID properties, in which the consistency is defined based on architectural constraints (both intrinsic to the Fractal component model and application specific constraints), for which a constraint language has been defined, ensures the reliability and correctness of systems reconfigurations.

- a distributed architecture that allows for the filtering and aggregation of data coming from probes/sensors in which composite probes (or probes aggregators) are themselves implemented as Fractal [1] components. Probes and composite probes can be distributed thanks to Fractal RMI (a dialect of Java RMI specific to Fractal).

- an active rules (or Event-Condition-Action or ECA rules) mechanism for component-based systems which is used as a basic decision making mechanism in an autonomic infrastructure. The work focuses more precisely on an extensible component-based architecture, in which rules and their constituents (policies, rules, events, conditions, actions, individual and collective behavior of rules, etc.) are themselves implemented as components so as to support dynamicity and extensibility: rules can be added, removed, their scope (the components they concern) and behavior changed at will, including at runtime.

- an architecture for the management of large-scale distributed systems. This middleware sits between resources (or managed elements in IBM autonomic terminology) and management applications (or manager elements in IBM autonomic terminology). It is worth noting in the context of SELFMAN that an autonomic control loop is a particular case of a management application. The middleware is architectured based on the concept of *domain* (implemented as Fractal [1] components) which encapsulate resources (or resource proxies) and provides them with management services (e.g. event management, query facilities). The middleware allows for flexible topologies of domains including hierarchical and overlapping domains so as to support the different needs of multiple management applications (or autonomic managers) and to handle scalability.

3.2 Event-driven component-based middleware architecture

We have considered the basic low-level services needed to support a self-managing and self-organizing large distributed system. These basic services include scalable connection-oriented communication, eventually perfect failure detection [4, 2], overlay name-based routing, group communication and a distributed hash table abstraction. The application nodes are connected in a structured overlay network induced by the Distributed k-ary System [3] DHT topology.

KTH has implemented these basic services in a middleware library architectured as event-driven components that interact through a publish-subscribe interface. Components subscribe to certain input event types, that they can handle and might trigger (publish) other output events while handling their input events. Triggered events are prioritized into three priority classes. We summarize here the event scheduling and the failure detection algorithms:

3.2.1 Event Scheduling

Triggering and execution of events relies on a *publish-subscribe* mechanism. Components *subscribe* for all the event types that they can handle. Whenever a new event is triggered, it is *published* for scheduling and when scheduled, the corresponding event handlers or all components that had subscribed for that event type are executed. Event handlers are executed by the worker threads of a *thread pool* of adjustable size. While being executed, event handlers might trigger other events.

An event subscription contains a reference to the subscriber component instance, a reference to the event handler method, and the event type for which the subscription is made. All event subscriptions are stored in a hash table indexed by event type. In fact, a set of subscriptions is associated to an event type as there can be more than one component subscribing for the same event type. The event subscription table is depicted in Figure 1.

When an event is triggered a new event instance is created and placed on an event queue. The event queue is a priority queue and is used for prioritization of events. Events can have one of three priorities: low, medium, or high. By convention timer expiration events are given high priority, middleware events are given medium priority, and application events are given low priority. In general, high priority events are scheduled before medium and low priority ones and medium priority events are scheduled before low priority ones. However, to avoid starvation of low priority events we implement the following fairness mechanism: not more than f events are consecutively scheduled from a higher priority queue if there exist events in lower priority queues. f is a fairness parameter.

When an event is dequeued for scheduling, its type is looked up in the subscription table and all subscriptions are retrieved. For each subscription in part a *work* item is created and submitted for execution to the thread pool. A work item is a

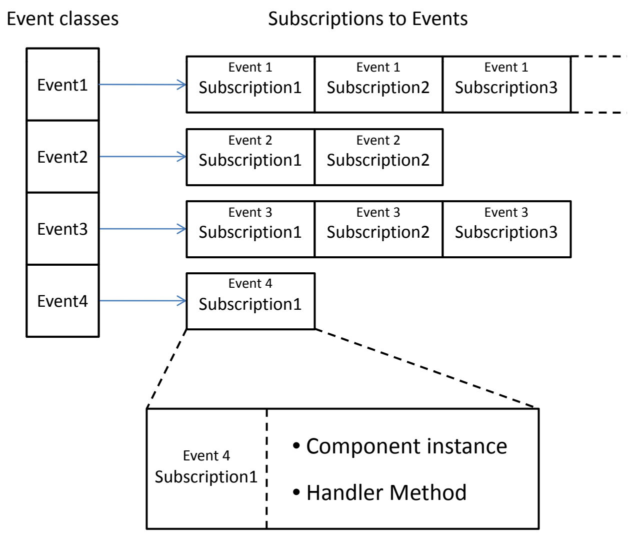

Figure 1: Subscription Table

unit of work that can be executed by a worker thread in the thread pool. It consists of the event instance that needs to be handled and references to the component instance and handler method that need to be executed for handling the event. A worker thread that processes a work item will invoke the handler method on the specified component instance passing it the event instance as an argument.

While invoking an event handler method on some component instance, a worker thread locks that particular component instance. This enforces that one component instance executes only one event handler at a time so the component writer does not have to deal with concurrency. We can say that event handlers execute atomically with respect to each other, or that components are concurrency-safe.

The event scheduling is summarized in Algorithm 1.

3.2.2 Failure Detection

As our middleware nodes are to be deployed over the Internet which behaves as a partially synchronous network [4], we provide an *eventually perfect* [2, 4] failure detector. This failure detector triggers *suspicion* events when it suspects that a peer node has crashed, and *rectification* events when it finds that the suspicion was in fact a false positive. False positives can happen in the Internet where most of the time the message transmission delay is bounded but sometimes, due to congestion, messages or acknowledgments may take longer than expected to arrive, thus resulting into a timeout and triggering a false suspicion.

The failure detector relies on a prediction of round-trip time for each connection in part. As all messages exchanged by the middleware are acknowledged, the RTT can be measured for each sent message. For each connection the average RTT is kept together with the RTT variance. These values are used to compute an expected round-trip timeout (RTTO). $RTTO = E(RTT) + 4 \times VAR(RTT)$. This timeout value is used to set a timer every time a message is sent. If the timer expires before an acknowledgment is received, the peer is suspected to have crashed. If an

Algorithm 1 Event scheduling

```
procedure TRIGGER($e$)                              ▷ Called to trigger event $e$
    $eventQueue$.enqueue($e$);
    schedule();
end procedure

procedure SCHEDULE()
    $e$ := $eventQueue$.dequeue();
    if $e \neq$ nil then
        $subscriptions$ := $subscriptionTable$.get($e.type$);
        for all $sub$ in $subscriptions$ do
            $w$ := makeWork($sub$);
            $workerPool$.executeWork($w$);
        end for
    end if
end procedure

procedure EXECUTEWORK($w$)                          ▷ executed by a worker thread
    lock($w.component$);
    ($w.component$).($w.handler$)($w.event$);
    unlock($w.component$);
end procedure
```

acknowledgment is eventually received, the RTTO is recomputed to adapt to the new RTT. If an acknowledgment is received before the timer expires the timer is just canceled.

In the case when the local peer doesn't actively send messages to the remote peer, the failure detector periodically sends *ping* probes awaiting for *pong* acknowledgments within a timeout of RTTO milliseconds. From the failure detection point of view, pings are equivalent to ordinary messages and pongs are equivalent to message acknowledgments. The local peer waits for γ milliseconds from the time it receives a pong until is sends the next ping. No ping is sent if the remote peer is suspected, but the local peer awaits for the pong to the last sent ping.

As the failure detection mechanism closely relies on the RTTO estimation, computed per each link in part, and on message acknowledgments, it is implemented inside the communication component. Because each connection may have a different expected RTTO we have a failure detector instance for each connection in part. The failure detector is a state machine depicted in Algorithm 2 and Algorithm 3. The state machine is driven by events like: a message is sent by the local peer, an acknowledgment is received, a timer expires, a pong is received. Here is a description of the states of the failure detector:

- INIT if no message is sent a ping is sent after γ ms;
- MSENT a message has been sent and a timer set for RTTO;
- PSENT a ping has been sent and a timer set for RTTO;

- PMSENT a message has been sent while in the PSENT state;
- PSUSPECT no pong was received in the PSENT state and the timer expired;
- MSUSPECT no acknowledgement was received in the MSENT state and the timer expired;
- PSUSPECT_MSENT a message has been sent while in the PSUSPECT state;
- MSUSPECT_MSENT a message has been sent while in the MSUSPECT state.

If the local peer sends a sequence of messages, only the first message is used for failure detection. From the failure detection point of view, all messages sent before the acknowledgement to the first sent message is received are ignored. This behavior relies on the fact that connections are FIFO.

3.2.3 Future Work

As future work, we plan to fit a reflective, hierarchical component model, like Fractal [1], to the DKS architecture, to allow for dynamic software reconfiguration. The Fractal component model is being refined as part of Task T2.1 Component-based computation model. We plan to work together with INRIA(P3) and FT R&D(P4), the two project partners that developed the Fractal component model.

We also plan to implement a transactional database on top our DHT. Initial work on transactional databases on top of DHTs has been done by project partners ZIB(P5) and KTH(P2) and has been delivered in deliverable D3.1a.

3.3 P2PKit relaxed-ring service architecture

We have adapted the P2PKit service architecture to the new P2PS SON with a relaxed ring topology. With the new topology, a node may be able to join the network but unable to ever form part of the *core* ring, for example as a result of firewalling. Such nodes should be avoided as they diminish the performance of the DHT. On the other hand, the situation itself is unavoidable and the ability to work with such restricted nodes is paramount to the usability of the architecture on the Internet.

3.3.1 Structure of P2PKit

The P2PKit architecture is two-tiered. The first tier is made of peers, forming a SON based on P2PS. Peer nodes are expected to have a reasonable stability and the ability to form a perfect ring in the long run. Generic services are installed on peers, in a redundant way.

The second tier is made of clients. Each client connects to a certain number of peers. Particular services can run on clients. Since they are not part of the SON

Algorithm 2 Failure detection algorithm

```
 1: event INIT()
 2:     $state$ := INIT
 3:     timer $TimerR$.start($\gamma$)
 4: end event

 5: event MESSAGESENT($messageId$)
 6:     if $state$ = INIT then
 7:         state := MSENT
 8:         timer $TimerR$.cancel()
 9:         $firstSentMessageId = messageId$
10:         timer $TimerM$.start($RTTO$)
11:     else if $state$ = PSENT then
12:         $state$ := PMSENT
13:     else if $state$ = PSUSPECT then
14:         $state$ := PSUSPECT_MSENT
15:     else if $state$ = MSUSPECT then
16:         $state$ := MSUSPECT_MSENT
17:     else if $state$ = PMSENT or $state$ = PSUSPECT_MSENT then
18:         return                                  ▷ Ignore and wait for Pong
19:     else if $state$ = MSUSPECT_MSENT or $state$ = MSENT then
20:         return                    ▷ Ignore and wait for Ack(firstSentMessageId)
21:     end if
22: end event

23: event ACKRECEIVED($ackId, newRTT$)
24:     if $state$ = INIT then
25:         return         ▷ Ignore, ack of message received during suspecting time
26:     else if $state$ = MSENT and $ackId = firstSentMessageId$ then
27:         timer $TimerM$.cancel()
28:         updateExpectedRTTO($newRTT$)
29:         $state$ := INIT
30:         timer $TimerR$.start($\gamma$)
31:     else if $state$ = MSUSPECT or $state$ = MSUSPECT_MSENT
32:             and $ackId$ = firstSentMessageId then
33:         trigger RECTIFICATIONEVENT($connectedPeer$)
34:         updateExpectedRTTO($newRTT$)
35:         $state$ := INIT
36:         timer $TimerR$.start($\gamma$)
37:     end if
38: end event
```

Algorithm 3 Failure detection algorithm continued

```
1:  event PongReceived($newRTT$)
2:      if $state$ = PSENT or $state$ = PMSENT then
3:          timer $TimerP$.cancel()
4:          updateExpectedRTTO($newRTT$)
5:          $state$ := INIT
6:          timer $TimerR$.start($\gamma$)
7:      else if $state$ = PSUSPECT or $state$ = PSUSPECT_MSENT then
8:          trigger rectificationEvent($connectedPeer$)
9:          updateExpectedRTTO($newRTT$)
10:         $state$ := INIT
11:         timer $TimerR$.start($\gamma$)
12:     end if
13: end event

14: event TimerExpired($TimerR$)
15:     sendto $connectedPeer$.ping()
16:     $state$ := PSENT
17:     timer $TimerP$.start($RTTO$)
18: end event

19: event TimerExpired($TimerP$)
20:     if $state$ = PSENT then
21:         trigger suspicionEvent($connectedPeer$)
22:         $state$ := PSUSPECT
23:     else if $state$ = PMSENT then
24:         trigger suspicionEvent($connectedPeer$)
25:         $state$ := PSUSPECT_MSENT
26:     end if
27: end event

28: event TimerExpired($TimerM$)
29:     if $state$ = MSUSPECT or $state$ = PSUSPECT_MSENT
30:           or $state$ = MSUSPECT_MSENT then
31:         return                          ▷ Ignore, the peer is already suspected
32:     else if $state$ = PMSENT then
33:         return                                      ▷ Ignore, waiting for Pong
34:     else if $state$ = MSENT then
35:         trigger suspicionEvent($connectedPeer$)
36:         $state$ := MSUSPECT
37:     end if
38: end event
```

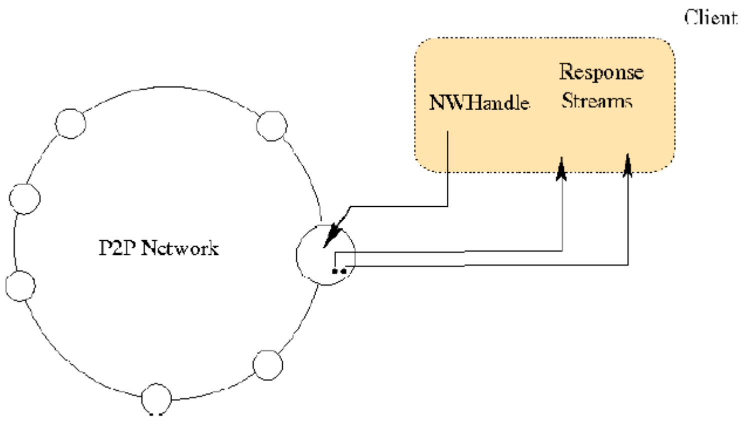

Figure 2: A P2PKit client connects to a P2PS node in order to publish and subscribe to services.

per se, they have much less disruptive power. The only services which rely on a particular client are the ones run by the client itself.

Figure 2 depicts the connections of a P2Pkit client to a P2PS node, which is part of a SON using relaxed-ring topology. It uses a network handler to publish services, and to submit requests. Response streams are used to filter message per every service the client is subscribed to.

The two-tier architecture can be seen as closing the gap between pure SON where stability and performance requirements are low and client-server where the requirements are high for the server and extremely low for the clients. Now we can combine extremely unreliable clients with a traditional SON.

3.3.2 P2PKit services

P2PKit services are active components that interface by message streams. Services can dynamically create new streams and pass references to them to other services. The generic services, installed on all the peers, have also access points. An access point is a well-known name corresponding to a stream on each peer. A service can send messages to an access point, specifying a modality (i.e. sending to just one peer, at least one peer, all the peers, a specifically identified peer or according to the DHT).

Particular services, run by the clients can of course have access to all the resources of the client whereas generic services, being replicated to all the peers are restricted to only use the P2PKit service architecture.

3.3.3 P2PKit as a self-adaptability primitive

Good performance of the P2PKit architecture requires a big stable SON of peers. Since all the local resource access are in the client, peers are really equivalents and an unstable peer can be removed from the SON. On the other hand, a client running on a stable node can launch a peer on the same node to join the SON.

4 Papers and publications

Appendices of the architectural framework specification contains the following documents:

- M. Kessis, P. Déchamboux, C. Roncancio, T. Coupaye, A. Lefebvre. Towards a Flexible Middleware for Autonomous Integrated Management Applications. Published in 2006 at the International Multi-Conference on Computing in the Global Information Technology (ICCGI'06), August 2006.
- M. Leger, T. Coupaye, T. Ledoux. Reliability of Dynamic Reconfigurations in Component-Based Systems. France Telecom Technical Report, February 2007.
- A. Diaconescu, B. Dillenseger. Composite Probes: a Generic Monitoring Framework for Hiearchical Management of Heterogeneous Data. Submitted for publication by France Telecom in April 2007.
- N. Jayaprakash, T. Coupaye, C. Collet, P.-C. David. Flexible Reactive Capabilities in Component-Based Autonomic Sys tems. Submitted for publication by France Telecom in May 2007.
- C. Arad, R. Roverso, A. Ghodsi, S. Haridi. Middleware for Building Internet-scale, Dynamic, Distributed Applications. KTH Technical Report, June 2007.

References

[1] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

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

[4] R. Guerraoui and L. Rondrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag, Heidelberg, Germany, 2006.

A Large Scale Management Infrastructure

Towards a Flexible Middleware for Autonomous Integrated Management Applications

Mehdi Kessis, Pascal Déchamboux*, Claudia Roncancio**, Thierry Coupaye*, Alexandre Lefebvre**
**France Telecom, Research & Development, MAPS/AMS*
28 chemin du Vieux Chêne, 38243 Meylan CEDEX, France
{Firstname.Lastname}@francetelecom.com
***LSR-IMAG*
{Firstname.Lastname}@imag.fr

Abstract

Enterprise and global-scale systems today might have thousands to millions of geographically distributed nodes and this number will increase over time. Managing efficiently such scattered systems becomes increasingly complex and requires powerful management capabilities. Traditional solutions to manage and control them seem to have reached their limits. In recent years, integrated management systems and services as well as autonomic systems have raised much interest in distributed systems and software engineering. This paper discusses the architectural issues facing the design of large-scale distributed management systems. Then it suggests a new flexible and scalable integrated management middleware to handle management problems in large-scale networked and heterogeneous systems.

Key words:
Integrated management, autonomic, component-based architecture.

1. Introduction

In recent years there has been a considerable growth in the use of distributed systems (peer-to-peer networks, clusters, pervasive computing, sensor networks, etc). These systems are scattered in our companies, administrations, homes and even in our pockets. Typically, they consist of large numbers of heterogeneous computing devices connected by communication networks, using various operating systems, resources and services, and user applications running on them. The dependability of our societies on such systems is becoming more and more noticeable. However, they become larger and more complex, heterogeneous and scattered. Traditional solutions to manage and to control them seem to have reached their limits. The size and the complexity of distributed system make it hard or even impossible to manage each of their components. Today, enterprise networks may count tens to thousands of managed resources. Telecommunication operators may manage thousands to millions of resources. For instance, France Telecom is managing more than one million “Livebox” home gateways. Due to the increased cost and complexity of managing such infrastructures manually, distributed computing systems are moving towards more autonomous operation and management.

This paper discusses an integrated management middleware that deals with such complex environments. This middleware plays the role of management broker that can be used by administrators or by management applications (i.e., management processes). It offers them (a) a customized view of the system through resource monitoring functions and (b) resource control management functions. We believe that such an infrastructure may offer a high degree of autonomy through management applications, by automating actions on groups of resources. We note that, in order to be able to automate such processes, a rich information model as well as a powerful programmatic model are needed.

This paper overviews our ongoing research. We aim at investigating new approaches to handle scalability, autonomy and heterogeneity issues in system management. After discussing new management needs and challenges, this paper proposes a middleware architecture to deal with these issues. Our work focuses on the monitoring activity, bringing flexibility and adaptability to the proposed solution.

2. Distributed management challenges

Today, large-scale distributed systems management is facing several major challenges. In this study, we focus on four of these challenges: autonomy, scalability, heterogeneity, and administrative isolation.

2.1. Autonomy

Managing efficiently such scattered systems becomes increasingly complex and requires powerful management capabilities. Traditional solutions to manage and control them seem to have reached their limits. In recent years, integrated management systems and services, autonomic systems have raised much interest in distributed systems and software engineering [14]. An autonomic system is capable to repair, configure, heal and protect itself [14]. The emerging field of autonomic distributed computing addresses the challenge of how to design and build distributed computing systems that can manage, heal and optimise themselves. Distributed computing systems are moving towards increasingly autonomous operation and management, in which their interacting components can organise, regulate, repair and optimise themselves without human intervention. [32]. These systems are intended to tackle administration complexity that is out of reach of human administrators, for instance handling a large number of alarms and notifications. Besides, automating management may reduce cost and improve efficiency. To automate management, we need at least three key elements: (a) representation, observation and monitoring capabilities, (b) decision rules and mechanisms and (c) control mechanisms.

2.2. Scalability

Scalability is a major problem for large-scale distributed systems. There is no commonly accepted definition of it [9]. In this paper, we consider the following definition of scalability: *"A scalable system is one that maintains constant, or slowly degrading, overheads and performance as its size increases"* [8]. In the past years, centralised network management has shown inadequacy for efficient management of large heterogeneous networks. As a result, several distributed approaches have been proposed to overcome the problem [21]. This is a main concern of our work because an enterprise network is an order of magnitude less complex than the infrastructure of some service providers, who monitor thousands to millions of resources. There are two key aspects of scalability involved in system management: the size of networks and the number of users. Service providers create extreme demands on both aspects[1]. Management systems should accommodate large numbers of participating nodes and they should allow applications to monitor large numbers of managed resources. Grouping and distributing management operations may improve scalability of the management system.

[1] The international Engineering Consortium (http://www.iec.org), Performance Management of Next Generation Networks.

2.3 Heterogeneity

The information model is a key feature in any management system [2]. It offers a view of managed resources (network, services, applications, etc.) to management applications. Today's networks involve heterogeneous resources. A service failure can be related to network or to application failures. In order to rapidly identify causes of failures and to understand the behaviour of complex managed resources, it is important to be able to describe heterogeneous managed resources and their interactions in the same way. The main object of integrated management [12] is to integrate different types of management (policy, user, network, services) in a unique infrastructure. Such infrastructure offers a complete view of the managed environment. To do so, we need a common description and representation of managed resources and their interactions.

2.4. Administrative Isolation

Traditionally, in large-scale networked systems, elements are grouped in managed domains. *"A domain is a set of objects to which a common management policy applies"* [18]. Each management domain is a logical partition of managed resources and management services. The set of managed objects may include computers, people, privileges, software processes, etc, depending on the purpose for which the domain is defined. In order to deal with large-scale networked and interconnected systems, domain management tools are needed. Such tools offer to the administrator the possibility to create, extend or merge new logical domains from existing primitive domains. Let us consider the France Telecom home gateway example. An example of primitive management domain could be the set of gateways related to a particular DSLAM. A management domain might contain the logical partition of gateways of a particular geographic zone, while another might contain the set of "LB1234" gateway model, in order to update their firmware. Administrator needs automated tools to build and to interact with management domains.

3. Towards a flexible and scalable integrated management middleware

3.1 A component-based model

Flexibility is a required property for managing large-scale networked systems. [11,7]. With a model such as the one proposed by Fractal [1], we believe that we can design, build and dynamically reconfigure component-based management infrastructure.

Fractal is a generic component model focusing on reconfiguration using flexible composition of components. It adopts a recursive view of components that may be nested. A component owns a *membrane* (i.e., a set of controllers), which realizes arbitrary forms of control over the content of the component. Composite components include other sub-components. A component sends and receives invocations through access points called interfaces. Such interactions require communication channels (named *bindings*) between some of the component interfaces. Figure 1 illustrates the architecture of a Fractal component. This composite component contains two sub-components and exposes control interfaces C_1 and C_2 as well as functional interfaces C_S and C_c.

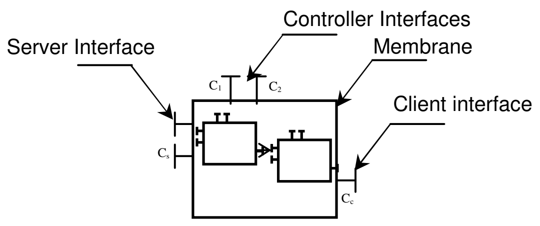

Figure 1 Example of Fractal components

Three main Fractal features are of particular interest:

a) *Component hierarchy*: composite components recursively contain components, ending with primitive components.
b) *Component sharing*: a (sub) component can be contained in several composite components. Typically, this feature can be used to model resources that are intrinsically shared.
c) *Components dynamicity:* bindings between components can be manipulated at runtime and is particularly interesting for management purpose. The model allows the definition of flexible bindings, since bindings may be themselves components.

Such properties are very interesting to design and build management domains. Figure 4 shows an example of mapping between domains and Fractal components. Disjoint domains are represented by disjoint components. Overlapping domains are represented by shared component. Hierarchical domains can be represented by composite domains.

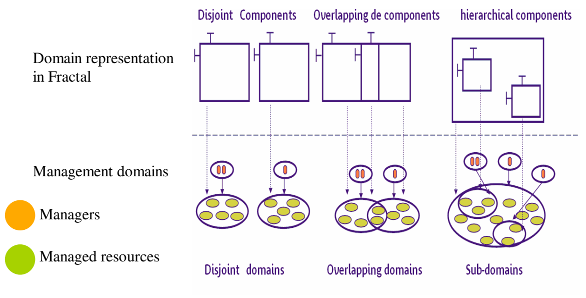

Figure 2 Management Domains in Fractal

3.2. Global management architecture

This section introduces the global architecture of an integrated management middleware that is positioned between management applications and managed resources. Managed resources are the set of physical resources (switches, PC, PDA, Set-Top Box, etc.) and logical resources (all or only a part of the OS, middleware, applications, services, etc.) available in an operator network.

A middleware relying on a flexible overlay network is a promising approach to overcome issues outlined in section 2. Such an approach allows management applications to construct an abstract view of the underlying network infrastructure and to federate different networks (IP, ad-hoc, etc). The middleware we propose builds an overlay network of mediation nodes that support management domains. The overlay network is functionally independent of the network infrastructure. It is formed by mediation nodes, which are interconnected through logical links. Figure 3 illustrates the global architecture of the proposed management infrastructure. Nodes collaborate in order to respond to queries of management applications. Each node represents one or many management domains. A node may contain one or many sub-domains (hierarchical relation). It interacts with several domains (links between nodes). This overlay builds an abstract representation of physical management domains (geographical for example) through management domains that correspond to specific management needs. (e.g., set of gateways model "LB1234"). Each node of the overlay offers a set of management services or tasks (information repository management, resource location service, query service, etc). As it can be noticed, these services cover only non functional management aspects delegated by management applications to the middleware.

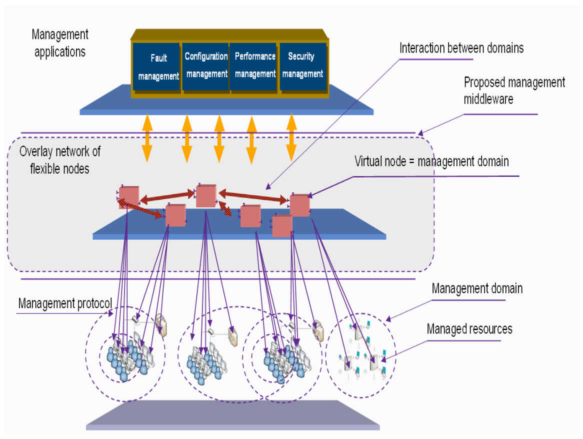

Figure 3 Global architecture of the management middleware

Both nodes structure and their interconnections are managed by the middleware administrator in a transparent way. The middleware hides management applications the complexity of the underlying infrastructure. The resulting overlay network is totally flexible, extensible and dynamically reconfigurable. Nodes and interconnections are dynamically reconfigurable.

To better understand the behaviour of these nodes, let us zoom inside one of them as depicted in Figure 4. Inside a node, a set of components work together in order to achieve a function. In Figure 4, the node offers 4 services: (i) events management service (that handles events and routes them to interested entities), (ii) a CIM repository (representation of the managed infrastructure), (iii) a query management service (for querying CIM repositories) and (iv) a repository for naming resources.

We believe that management applications requires rich information modelling to take into consideration physical and logical interdependent resources. Common Information Model (CIM) [4] is a standard for defining device, network and application characteristics so that system and network administrators and management programs can control heterogeneous devices and applications.

CIM also allows for vendor extensions. The adoption of such a model is key to overall interoperability for information storage and for system management environment.

The proposed middleware is based on two API:

a) *A configuration and deployment API*: It concerns the mediation nodes and management services. Network administrator, can build, deploy and configure the management middleware. After configuration and deployment, the middleware is ready to be requested by management applications.

b) *A mediation API*: This API is used by management applications. It permits them to communicate, indirectly, with managed resources, through our middleware.

Although both interfaces have different concerns, they are both managed by administrators at different level. Furthermore, the proposed middleware should use itself for its own management issues.

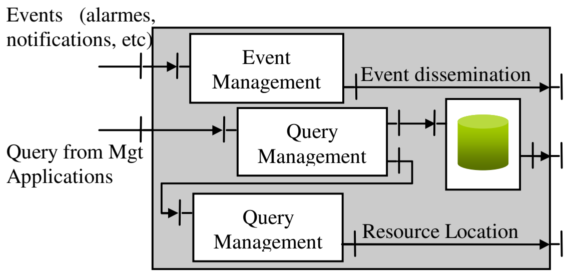

Figure 4 Internal structure of mediation node

4. Discussion

The choice of component-based architectures for managing networks and services have several advantages. V. Wadel et al. [17] consider them when designing management solutions within the world of telecommunication (flexibility, modularity, clear design, etc.).

The management applications are designed in order to be independent from the size of the network or from the number of properties to ensure. The management middleware aims at routing the requests over management information, supporting persistence of this information when necessary, locating resources, etc. It provides mediation nodes whose one of the primary roles is to ensure that its functions whatever are the conditions of the infrastructure that support the management network (i.e., an overlay network). The objective is that this middleware can adapt to such diverse situations as sensors network or as computation grids. This means that flexibility and adaptability are among the main challenges we target. These properties should ensure that our middleware can adapt to the various functional requirements of management applications, and can also adapt its own behaviour to the resources dedicated to its operation. We argue that component-based architecture is a major enabler towards this goal.

The Fractal component model we rely on allows the definition of components as assemblies of components and provides total control over the component used as well as the bindings between them. Hence, at the very end, the overlay can be considered a component composed of other components (e.g., the mediation nodes), managing them and their relationships as well. The overlay can then be configured and reconfigured in order to respond to management needs.

Autonomous management is seen as the only means to deal with large-scale management. This approach will lead to very complex applications to support the processes composing this autonomous management environment.

The information model on which the management applications rely is highly distributed by nature. So should be the middleware for supporting them. We consider that good work has been done for specifying the information model, especially with CIM [4]. Our objective is to be able to organise the management of this information space in a distributed manner while ensuring its safety, scalability, correctness (i.e., in a sense considering the implementation of a reliable distributed CIMOM). We also consider much simpler interfaces to manipulate this information model, especially within our Java implementation context. For example, we consider pure Java objects accessible through technologies such as EJB[2] or JDO[3] for giving access to the CIM repository.

5. Related Works

Several works studied large-scale systems and network management, during these two last decades [6, 10, 20, 8].

CIM/WBEM [6], a DMTF[4] standard, proposes a web-based management architecture. WBEM is built around the CIM model. The main WBEM architecture is composed of three main elements (management applications as client, WBEM server, WBEM providers (probes and actuators)). This architecture follows a flat and centralized model (Manager/Agent model). There is no M to M (Manager to Manager) communication. WBEM servers can be used to manage enterprise environments. However, it does not scale to large distributed environments. The administrative isolation is implemented through the concept of namespace (logical view of CIM instances and classes).

Yalagandula et al proposed SDIMS (Scalable Distributed Information Management System) [5]. It consists of a building block for large-scale distributed services. SDIMS aggregates information about large-scale networked systems and provides detailed views of information (and events) and summary views of global information. It ensures four properties: scalability, flexibility, administrative autonomy and robustness. This work concerns neither resource heterogeneity nor autonomic behavior.

Renesse et al [8] proposed a similar work: Astrolabe. This system gathers, disseminates and aggregates information about zones. A zone is recursively defined to be either a host or a set of non-overlapping zones. It supports scalability through hierarchy (zone hierarchy), flexibility through mobile code, robustness through a randomized peer-to-peer protocol and security through certificates. Each Astrolabe zone has a set of aggregation functions that calculates the attributes for the zone's MIB (SNMP like Management Information Base). Astrolabe is designed under the assumption that MIBs will be relatively small objects, a few hundred or even thousand bytes, not millions which limit its scalability.

Anerousis et al [10] proposed Marvel. This system is a distributed computing environment that allows the creation of scalable management services using intelligent agents and the world-wide web. Marvel builds on top of existing element management agents a hierarchy of servers that aggregate the underlying information in a synchronous or asynchronous fashion. Marvel is based on an information model that generates computed views of management information. These views follow an object-oriented model to store management information. Marvel requires that managed elements be organized into groups. Users can dynamically define these groups based on any factor that makes sense such as location or functionality. The object implementation of Marvel's views is proprietary and not extensible. Besides it does not address autonomy issue.

Recently, Bouchenak et al [15] proposed the JADE framework. It is an environment for implementing autonomic administration software. The main idea of this work consists on modelling the administrated system as a component based software architecture which provides means to configure the environment. A prototype of Jade was developed and used for deployment and fault management of clustered J2EE application. This work provides administrative isolation through composition and sharing relations. This work is based on an ad-hoc information model. The global vision of the proposed work in this paper is coherent and complementary with the autonomic management vision proposed in the European project IST Selfman [3], to which we actively participate.

6. Conclusion and Future works

In this paper we have studied large scale networked heterogeneous systems problem. We have identified four important properties that large-scale management

[2] http://java.sun.com/products/ejb/docs.html

[3] http://java.sun.com/products/jdo/

[4] Distributed Task Force Management; URL: http://www.dmtf.org

systems have to respect: autonomy, scalability, administrative isolation and heterogeneity. We suggest the architecture of a flexible middleware that respects these properties. The proposed middleware is based on the Fractal component model. It offers the administrator the possibility to build a scalable and dynamically reconfigurable overlay network of mediation nodes. Each node represents on or many management domains. The middleware offers the possibility to build, aggregate, select, and query management domains according to administrator needs. All these operations can be done without interrupting management applications activity. The different nodes cooperate to offer management applications several services (event filtering, aggregation, routing, storage management, etc). Scalability is achieved through the support of domain and through the distribution of the management activity.

We believe that the proposed middleware can be a powerful building block for autonomous management applications. A set of management policy can be defined for each management domain that we build. Automatic actions can be assigned to each of them. Scalability is achieved through distribution of the management infrastructure and grouping management operations. Heterogeneity is achieved through CIM local repositories, managed by the node of our overlay network. In these repositories, heterogeneous resources are described in a standard way. Administrative isolation is achieved through the different possible composition relations proposed by the Fractal component model. Autonomy is achieved through the reflexive aspect of the Fractal components.

We are studying the possibility to integrate a data steam management system to handle, in a scalable way, streams of events sent by probes and network equipments. This feature is particularly interesting for large event-based monitoring systems in real-time context. Besides, we are studying the possibility to make some of our node mobile. The ProActive[5] technology, based on the Fractal component model, has already experienced such an approach.

7. References

[1] E. Bruneton, T. Coupaye, and J.-B. Stefani. "Recursive and Dynamic Software Composition with Sharing". Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain, June 10-14, 2002.

[2] J.-P. Martin-Flatin, "Toward Universal Information Models in Enterprise Management", in Proc. VLDB 2001 Workshop on Databases in Telecommunications (DBTel 2001), Rome, Italy, September 2001.

[3] P. Van Roy, A. Ghodsi, JB Stefani, S. Haridi, T. Coupaye, A. Reinefield, E. Winter and R. Yap. " Self management of large-scale distributed systems by combining structured overlay networks and components". Workshop IST NoE CoreGrid Integration, Greece, Nov 2005.

[4] Common Information Model Standard, URL: http://www.dmtf.org/standards/cim/

[5] P. Yalagandula and M. Dahlin, "A scalable distributed information management system ", Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, session Distributed information systems, Pages: 379 – 390, Portland, Oregon, USA, 2004.

[6] Java Specification Request N°48 (JSR 48): WBEM Services Specification, URL: http://www.jcp.org/en/jsr/detail?id=48

[7] J. Won-Ki Hong, J. Kim and J. Park: "A CORBA-Based Quality-of-Service Management Framework for Distributed Multimedia Services and Applications". IEEE Network, Vol. 13, No. 2, (1999) 70-79

[8] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining", ACM Transactions on Computer Systems, Vol. 21, No. 2, May 2003, Pages 164–206.

[9] M. D. Hill, "What is scalability?". ACM SIGARCH Computer Architecture News, December 1990.

[10] N. Anerousis and G. Hjálmtysson, "View-based Management of Services in a Programmable Internetwork". Proc. of the 2000 Network Operations and Management Symposium, Honolulu, HI, April 2000.

[11] G. Goldszmidt, "Distributed Management by Delegation". 1996. Ph.D Thesis – Graduate, School of Arts and Sciences, Columbia University, New York.

[12] H.G. Hegering, S. Abeck and B. Neumair. "Integrated Management of Networked Systems: Concepts, Architectures, and their Operational Application". Morgan Kaufmann Publishers, 1999.

[14] J. O. Kephart and D. M. Chess." The vision of autonomic computing". IEEE Computer, 36(1):41–50, January 2003.

[15] S. Bouchenak, N. de Palma and D. Hagimont, "Autonomic administration of clustered J2EE applications". Proceedings of IFIP/IEEE International Workshop on Self-Managed Systems & Services (SelfMan 2005). 2005.

[16] M. Kahani, H.W. Peter Beadle, "Decentralized Approaches for Network Management", in SIGCOMM, July 1997.

[17] V. Wade, D. Lewis, C. Malbon, T. Richardson, L. Sorensen and C. Stathopoulos, "Component Integration Technologies for Telecoms Management Systems", TCD-CS, Technical Report, Trinity College Dublin Computer Science Department, 1999.

[18] M. Sloman, J-D. Moffett, "Domain model of autonomy". ACM SIGOPS European Workshop 1988

[5] http://www-sop.inria.fr/oasis/ProActive/

B Transactional Reconfiguration of Component-Based Architectures

Reliability of dynamic reconfigurations in component-based software systems

Marc Léger[1], Thierry Coupaye[1], and Thomas Ledoux[2]

[1] France Telecom R&D
28, chemin du Vieux Chêne
F-38243 Meylan
{marc.leger, thierry.coupaye}@orange-ftgroup.com
[2] OBASCO Group, EMN / INRIA, LINA
Ecole des Mines de Nantes
4, rue Alfred Kastler
F-44307 Nantes Cedex 3
thomas.ledoux@emn.fr

Abstract. This article is an analysis based on our experience with the Fractal component model of the need of reliability for dynamic reconfigurations in component based systems. We make a proposal to ensure this reliability, which can applied to concurrent reconfigurations. We started from the definition of ACID properties in the context of component models and we propose to use integrity constraints to define system consistency and transactions for guaranteeing the respect of these constraints at runtime. To deal with concurrency, we have to detect potential conflicts when composing reconfiguration operations.

1 Introduction

Dynamic reconfigurations in component-based software applications [MK96] are central to promissing approaches like autonomic computing [KC03]. There are many motivations to introduce modifications in a system at runtime: correction of security flaws or functional bugs, improvement of systems (e.g., performance optimizations), or adaptions to execution context changes.

Thanks to properties of component models like loosely coupling, reconfigurations can rely on component-based architectures [OMT98]. However, runtime modifications can let the system in an inconsistent state. From a structural point of view, the architecture of the system once reconfigured can be not in conformity with the component model or eventually system specific constraints (e.g. architectural invariants) anymore. From a functional point of view, a reconfiguration must not perturb the execution of the system (i.e., functional and non functional aspects need to be synchronized). Furthermore, in case of concurrent reconfigurations, reconfiguration must be synchronized between themselves.

In this paper, we focus to the reliability of runtime adaptations and we chose to base our work on the Fractal component model [BCL$^+$04] because of its support of dynamic and opened reconfigurations. In our approach, we tried to define each of the ACID properties [TGGL82] in the specific context of component-based systems an show how it can solve this reliability problem during adaptations. These properties are unifying concepts of transactions for distributed computation used essentially for supporting concurrency and recovery. We specify the consistency property by using integrity constraints about system structure and state. An example of a structural constraint at the level of component model is cycle-free component structure. Moreover we must avoid wrong execution flow of reconfiguration operations according to their semantics to ensure the isolation property.

This paper is organized as follows. Section 2 is an overview of dynamic reconfigurations in component models, with a focus on Fractal, and it shows what problems it raises regarding reliability. Then section 3 describes how transactions combined with integrity constraints can be a solution to these problems. Finally section 4 presents some related works before concluding in section 5.

2 The need of reliability for dynamic reconfigurations in component-based systems

2.1 Dynamic reconfigurations in component models

Dynamic reconfigurations allow modifications of a part of a system during its execution without stopping it entirely to keep the system partly available. Actually, maximization of the availability time is essential for some systems like entreprise application servers. Dynamic reconfigurations can involve every manageable element defined in the component model and reified at runtime, they can be:

– structural (e.g., addition or removal of elements like components, interfaces etc. and interconnection modifications with bind unbind operations),
– behavioral (e.g., lifecycle modification used to synchronize component activity with the rest of the system),
– linked to component deployment (e.g., component instantiation, destruction, migration),
– linked to component state (e.g., change of component attribute values),

Fractal [BCL$^+$04] is a recursive component model with sharing and reflexive control. It is based on classic concepts of component (as a runtime entity), interface (an interaction point between components expressing provided and required services) and binding (a communication channel between component interfaces). A component consists of a membrane which can show and control a causaly connected representation of its encapsulated content. An Architecture Description Language (Fractal ADL [Fra]) is used to specify component configurations and there is notably a Java implementation of the model, Julia. Several controllers are defined to control bindings, the hierarchical structure, component lifecycle, attributes and names, but other controllers can be user-defined.

Operations in controllers constitute primitive reconfiguration operations and do either introspection or intercession (modifications) in the system. To compose operations, we consider sequences or parallel executions of intercession operations with conditions expressed by means of introspection operations in component configurations. An example of composite reconfiguration is component hotswap, a mechanism used to update a system where an old version of a component is replaced by a new one. In Fractal, this reconfiguration is composed of a sequence of several primitive reconfiguration operations, it implies to stop the component, unbind all its interfaces, remove it, add the new instantiated component, bind its interfaces and start it (a state transfert operation is used in case of stateful component).

2.2 The reliability problem with dynamic reconfiguring applications

A first problem when modifying a system at runtime is the synchronization between reconfigurations and the functionnal execution of the system. Actually, the part of the system which is modified could be unavailable for functional execution during the reconfiguration time. To take the hotswap example with a stateful component, calls on the old component must be blocked until a a "quiescent state" [KM90] is reached, then the state must be transfered, finally previous calls are forwarded towards the new component.

A second problem at the model level is about consistency violation by reconfigurations. First of all, we must make clear what exactly consistency is for component-based systems. Component models and application models should define what this consistent system is, especially in term of structure. For instance, we may want to add a structural constraint about the number of subcomponents of a composite component. In Fractal, the specification of the component model is not always sufficient and we want to express some integrity constraints on systems. So we must ensure the conformity of the system to the model and constraints after reconfigurations.

The third and last problem we identified is linked to the composition of reconfiguration operations. A prerequisite is the separation of concerns between the functional part and the control part of systems. Then separation between introspection operations and intercession operations must be explicit. Once these operations have been identified, the semantics of reconfiguration operations implies there can be some conflicts between them in case of compostion and for synchronization between several reconfigurations (e.g., in Fractal it is mandatory to unbind all component interfaces before removing the component from its super-component).

3 A transactional approach to ensure reliable reconfigurations

3.1 ACID properties in the context of dynamic reconfigurations

We think that well-defined transactions associated with structural and behavioral constraints verification is a means to guarantee the reliability of reconfigurations in component models, i.e. to solve problems we identified in the section

2.2. As any reconfiguration operation could lead the system to an inconsistent state, each reconfiguration must always be included in a transaction. In this context, we define the meaning of ACID properties as follows:

- **Atomicity**: either all happen or none happen, that is to say either the system is reconfigured or it is not. A reconfiguration transaction can be a single primitive reconfiguration operation or a more complex operation composed of several operations. Each reconfiguration operation must specify its reversible operation. Thus if a reconfiguration transaction goes badly and is rollbacked, it is possible to come back in a previous stable state by undoing operations. Transactions demarcation is either programmed in the language or automatic (a reconfiguration script corresponds to a transaction).
- **Consistency**: a transaction must be a correct transformation of the system state. So the reconfigured application must be conform to the component model and application specific constraints. That is to say consistency is given by integrity constraints essentially architectural invariants. A reconfiguration transaction can be commited only if the resulting system respects the constraints. Other faults like software and hardware failures (network and machines) are the responsibility of the commit protocol (e.g., 2 phase commit protocol).
- **Isolation**: several reconfiguration transactions are independant and any schedule of reconfiguration operations must be equivalent to their serialization. The scheduling must respect the operation semantics and conflicts. This property relies on the knowledge of the semantics of reconfiguration operations.
- **Durability**: once a reconfiguration completes with success (commit), the new state is persistent. For every transaction, operation are logged in a journal so that reconfigurations can be redone in case of failure. The application state (architecture and component state) is periodically checkpointed basically with ADL dumps and component state is saved in databases. So any component can be recovered in its last stable state resulting from the last successful reconfiguration. However, the only functional state we capture is the state which is well identified in the component model and is saved only at commit time of reconfigurations because we don't want to impose transactions at the functional level.

Only the first problem presented in 2.2 is not completely adressed by our approach because we do not fully modelise the functional execution flow of systems, we relies on the implementation of the component lifecycle operations with interceptors on component interfaces to realize the synchronization. A solution to the synchronization problem is to apply the hotswap protocol proposed in [KM90]. The guarantee we can bring is that the order of operations in the protocol is respected. Among the ACID properties we will especially focus in the following sections on two properties: consistency and isolation.

3.2 Integrity constraints to ensure system consistency

In our proposal, system consistency relies on integrity constraints and we want to express these constraints both at the application and at the model level. An integrity constraints is essentially a predicate which concerns the validity of an assembly of architectural elements but it can also concern component state. Examples of such constraints at the component model level are hierarchical integrity (bindings between components must respect the component hierarchy) or cycle-free structure (a component cannot contain itself to avoid infinite recursion). On the other hand, application specific constraints are used to specify invariants on a given system either on component types or directly on component instances designed by their names. Invariants can concern for example cardinality of sub-components in a super-component, two component interfaces which can never be unbound etc.

In an open world where reconfigurations are not anticipated at compile time, some component models like Fractal are relying on reflexive architectures to dynamically reconfigure systems by means of a runtime mapping between the system which is really executed and its model. So integrity constraints verified on the model will be also valid in the system. We represent the Fractal component model as a typed graph and then each fractal-based application is also a graph which is an instance of this typed graph. The instance graph is a more formal representation of the system provided at runtime by the reflexivity of the component model and is used to navigate in runtime applications. The vertexes are elements from the component model: components, functional interfaces, controllers, attributes and operations. The edges represent relations between the elements: composition links, binding links etc. Then the instance graph must always be well-typed regarding to the typed graph (i.e., conform to the component model) and the instance graph must respect integrity constraints. Therefore contraints at the model level can be specified on the typed graph and others on the instance graph. As the model is extensible and new user-defined controllers can be added, graphs should be also easily extensible in terms of elements and relations.

To express integrity constraints, we propose to use a DSL based on an extension of the query language in Fractal configurations FPath [DL06] to transform it into a real constraint language "à la OCL" [OCL05]. An advantage of the FPath language is that it can navigate both in the ADL and in the runtime system and it is already based on a graph representation of the system during execution. The constraint language must just have introspection capacity without side effects on the system. We want to express invariants, preconditions and postconditions in this language and we want notably to have quantifiers, collection operations and filters. The following basic example is a structural invariant constraint at the application level expressed in FPath (with its XPath 1.0 syntax like) where the component designed by the variable c can never be shared (it can only have one parent at the same time):

```
size(c/parent::*)=1
```

Constraints must be checked both at compile time on the component static configuration and at runtime. We consider checking constraints as far as possible before applying the reconfiguration on the system, eventually by code analysis of a dedicated reconfiguration language like FScript [DL06]. Constraints can also be checked either directly during the execution of the reconfiguration of the real system or by simulation on a local copy of the representation of the system (i.e., the instance graph) so as to limit the effect on the system in case of constraint violation.

3.3 Isolation of reconfigurations to support concurrency

We take the hypothesis that not only application components are distributed but also administrators. Furthermore, reconfiguration initiators are either humans (interactive reconfigurations) or the system itself (the system is able to auto-reconfigure). Concurrency in reconfigurations comes from the fact that one administrator can explicitely want to execute some operations in parallel, or several administrators can reconfigure the same system at the same time. The reconfiguration scheduler can also detect when it can launch parallel reconfiguration tasks to optimize the reconfiguration process.

As seen in section 2.1, reconfiguration operations are composable but all compositions are not valid. In Julia, operation semantics is hidden in controller implementations and so we want to make it explicit and we want eventually to be able to change it and to specify new primitive operations. So we need to express operation semantics in terms of preconditions and postconditions with our constraint language presented in section 3.2. We distinguish two types of conflicts between operations: parallel conflicts and execution dependencies. For two given reconfigurations *R1* and *R2* executed on the same system, a parallel conflict occurs if *R1* and *R2* modify the same manageable elements in the system model (e.g. bind and unbind operations). An execution dependency occurs if *R1* either need *R2* to be executed first (e.g. stop before unbind)or if *R1* cannot be executed after *R2*. That is to say *R2* postconditions cover or not *R1* preconditions.

```
// Example of a precondition for removing a component
operation: void removeSubComponent(Component sub);
preconditions :
// all interfaces of the sub-component are unbound (. is the current node)
not(exists(sub/interface::*[not(bound(.))]));
```

For concurrency management, we propose a pessimistic approach with locking. Our locking algorithm is based on operation semantics to avoid inconsistent operation compositions. We see two different possibilities for the locking algorithm. The first one is to lock directly reconfiguration operations. That is to say, either conflicts between operations are automatically calculated thanks to its preconditions and postconditions or it must define the operations with which it is in conflict. The second one is to use a modified DAG locking algorithm on our instance graph defined in 3.2. Then the lock granularity is defined by the manageable elements in the graph representation and for example a lock acquisition on a component also locks all its interfaces and every operations in each interfaces.

Another approach to locking is to constrain the execution order of reconfiguration operations. We propose to use a simple language inspired of behavior protocols in [PV02] to describe the desired execution order of reconfiguration operations, what we call behavioral reconfiguration constraints. The protocol compliance is checked at runtime by intercepting reconfiguration calls.

4 Related work

Many works on ADLs follow a static approach to check consistency of component-based architectures by compilation but only a few are interested in dynamic analysis of this consistency. We will focus here on other reflective component models which allow non anticipated (also called ad-hoc) reconfigurations.

FORMAware [MBC04] is relatively close to our work. This framework to program component-based application gives the possibility to constrain reconfigurations with architectural style rules. A transaction service manages the reconfiguration by stacking operations. The main difference with our proposal is our integrity constraints are more flexible than styles and they can be applied to every element of our component model. Moreover we define more formally reconfiguration operations to identify conflicts between them, our locking algorithm is then more precise than a simple lock on components and we consider introspection operations as reconfiguration operations.

Plastik [BJC05] is the integration of the OpenCOM component model and the ACME/Armani ADL. As in our solution, architectural invariants can be checked on ADL configuration or at runtime and constraints are expressed at the style level and at the instance level. However, reconfiguration cannot be generic composite reconfigurations with model elements in parameters and the execution, the operation semantics is not explicit and not extensible and the order of reconfiguration operation cannot be constrained as we can do with reconfiguration protocols.

5 Conclusion

Dynamic reconfiguration in component-based systems raises reliability problems, especially in open systems in which they are not anticipated. In this article, we identified the three following global problems based on our experience with the Fractal component model: synchronization between reconfiguration and the functional execution of systems, consistency regarding component and application models, and synchronization between reconfiguration operations. We focused more on the two last problems: the first one concerns conformity at runtime of systems with constraints and models, the second one deals with the validity of composition of reconfiguration operations.

We propose to use integrity constraints to define consistency for dynamic reconfigurations and to include these reconfigurations in transactions. We build a graph representation of our application at runtime thanks to the reflexivity of the Fractal component model and use a constraint language on this graph. Moreover we want to detect execution conflicts between reconfiguration operation in order to be able to compose them with reliability with eventually the specification of reconfiguration protocols. We are currently implementing this proposal in Julia, a Java implementation of the Fractal model.

References

[BCL$^+$04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[BJC05] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In Ronald Morrison and Flávio Oquendo, editors, *EWSA*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[DL06] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, Nantes, France, July 2006.

[Fra] Fractal ADL. *http://fractal.objectweb.org/fractaladl*.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[MBC04] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Supporting adaptable distributed systems with formaware. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pages 320–325, Washington, DC, USA, 2004. IEEE Computer Society.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[OCL05] OCL 2.0 Specification. *http://www.omg.org/docs/ptc/05-06-06.pdf*, 2005.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[TGGL82] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7(3):323–342, 1982.

C Composite Probes

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 3, May-June 2005

Composite Probes: a Generic Monitoring Framework for Hierarchical Management of Heterogeneous Data

Ada Diaconescu and Bruno Dillenseger, Orange Labs

Abstract

System monitoring has become an essential utility for managing software applications. However, as software systems are becoming increasingly complex, analyzing collected monitoring data is becoming progressively more difficult and costly a task. This article presents Composite Probes, a generic monitoring framework for complex system management. Composite Probes provides support for organizing heterogeneous data into hierarchical constructs that process data at different granularity and abstraction levels. Composite Probes represent building blocks that are instantiated, customized and connected to form flexible hierarchies adapted to various application requirements. System administrators configure each instance with specific data-processing functions, including aggregation, filtering and scheduling. A Composite Probes prototype was implemented and successfully tested on different distributed applications.

1 INTRODUCTION

System monitoring has become an essential utility for assisting the development, configuration and runtime administration of software systems. Generic and specific monitoring tools are being employed to test and calibrate software systems offline, as well as to supervise, manage and adapt software systems during runtime. Performance profiling, SLA-compliance verification and autonomic computing are only some of the important industrial and research areas heavily relying on monitoring functions. Existing monitoring utilities collect different data types from various managed entities (e.g. CLIF[1], LeWYS[2], Compas[3], Ganglia[4], or JVMTI[5]). Collected data can range from a system's hardware and software resource consumption, to a system's usage patterns and achieved quality attributes. Monitored data is subsequently analyzed in order to determine various

[1] CLIF: load-injection and monitoring framework for distributes applications (clif.objectweb.org)
[2] LeWYS: monitoring framework for hardware and software resources (lewys.objectweb.org)
[3] COMPAS: framework for performance management in J2EE applications (compas.sourceforge.net)
[4] Ganglia: distributed monitoring for high-performance computing systems (ganglia.sourceforge.net)
[5] JVMTI: JVM™ Tool Interface from Sun Microsystems (java.sun.com/j2se/1.5.0/docs/guide/jvmti)

properties, such as system correctness, performance, overall utilization, state, or SLA-conformance levels. However, as systems are becoming increasingly complex and monitoring tools more sophisticated, progressively larger amounts of heterogeneous data are being collected for analysis and diagnosis operations. Examining accumulated monitoring data is consequently becoming an increasingly complex and costly task, especially when periodically required during system execution. Available monitoring utilities generally provide data in the initially collected format and offer little support for processing this data. Consequently, massive amounts of heterogeneous data are represented at the same abstraction level, in a flat data structure. The responsibility for organizing, aggregating and filtering monitoring data is left entirely to data consumers, or clients. Meanwhile, multiple data-processing tasks are common to most clients, regardless of client-specific management goals. For example, various high-level indicators must be computed from low-level measurements in most complex management applications. There is an emerging need for aggregated data to be readily available, to represent high-level resource measurements or abstract indicators.

This paper proposes a novel monitoring framework called *Composite Probes (CPs)* that aims at extending current monitoring utilities with support for hierarchical data-organization and processing. CPs provides support for constructing flexible monitoring hierarchies from reusable and configurable probes. Such hierarchies can be configured to follow customized data processing and scheduling policies in order to aggregate and filter incoming data, possibly from heterogeneous resources, at multiple abstraction levels. In this manner, CPs instances in a hierarchy provide monitoring data at various granularity and complexity levels, representing basic or abstract resources, fine-grained measures or high-level indices. Namely, a CP can provide measures as different as a system's CPU usage, a cluster's overall resource load, a system's SLA-compliance level, or a an application's state. The main contributions of the CPs framework include:

1. Reusable applicative support for data assembly and event forwarding functions, commonly required for building data association and processing chains. Therefore, CPs helps decrease management costs by preventing replicated efforts, expertise and development work from being conducted at multiple sites.
2. Highly-customizable and extensible data-processing elements, that can be configured to use different aggregation, filtering and scheduling algorithms

Additionally, CPs features important characteristics required for a scaleable, manageable and adaptable monitoring framework:

3. Support for integrating and using low-level monitoring data from heterogeneous resources, at different system levels (e.g., OS, JVM, middleware and application).
4. Uniform data representation and control for all probe types
5. Standard external access via common communication and management protocols
6. Seamless extensibility, via new data-processing algorithms, new scheduling policies, new probe types and additional communication protocols
7. Support for integrating legacy probes and third party data processing functions
8. Inherent scalability of monitoring hierarchies, as large amounts of data can be collected and processed at multiple distributed sites.

2 COMPOSITE PROBES

Framework Overview

Composite Probes (CPs) is a generic monitoring and analysis framework for large-scale, distributed (LSD) applications. The framework's main goal is to provide support for organizing and managing massive amounts of monitoring data collected from heterogeneous resources. CPs aims at extending flat monitoring architectures with flexible hierarchical constructs, in order to facilitate data understanding, provide different information views and reuse data-processing functions.

CPs represent building blocks for constructing flexible and configurable hierarchies, which can process monitoring data at various granularity and abstraction levels. As depicted in Figure 1, in a CPs monitoring hierarchy data obtained from low-level system probes flows upwards through the hierarchy and undergoes incremental processing at each CP instance involved. CPs are classified into two major types, based on their roles and functions (Figure 1 and Figure 2). The first probe type is the *Basic Probe (BP)*, whose role is to extract monitoring data from the managed system resources. BPs represent *leaf* nodes in the hierarchy, meaning that they cannot be further composed of other probes. BPs process collected monitoring data and subsequently forward it to connected *parent* probes. The second probe type is the *Composite Probe (CP)*, whose role is to manage and organize incoming data from multiple data sources, or *child* probes. CPs can contain other CPs or BPs, which constitute the CP's *data sources* (Figure 4). CPs process incoming data and subsequently forward it to connected *parent* probes. Data processing in BPs and CPs involves data aggregation and filtering procedures, as dictated by well-specified scheduling policies. From an external perspective, clients have a uniform view of all probes. Probe access is restricted to probe external interface(s), which are identical for all probe types (Figure 2). For the scope of this paper, the term Composite Probes (CPs) will be used to indicate any of the two probe types, Basic or Composite, unless otherwise specified. The term also refers to the monitoring framework proposed. The exact meaning of the term will be clear from the used context.

All CPs can be identified via a unique *probe Id*. Clients use probe Ids to access any probe in a hierarchy, in order to retrieve monitoring data or send control commands. Communication can be done directly with a targeted probe, or indirectly, via the probe's parents. Indirect communication requires the path to the targeted probe to be provided as a request parameter, which allows the request to be routed to its destination probe.

Two main information flows characterize CPs hierarchies. These are the *data* flow and the *control* flow (Figure 1). In short, the data flow transports monitoring data upwards from lower-level BPs to higher-level CPs. The control flow transports control commands downwards from CPs at higher hierarchical levels to lower-level BPs.

The proposed framework has a modular, configurable and extensible design, which allows its main functions - aggregation, filtering, scheduling and communication - to be individually tuned and modified for each CP instance.

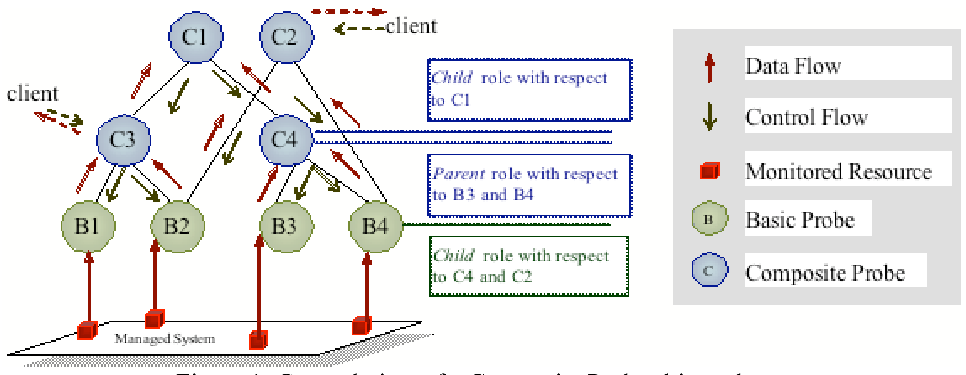
client
client
C1
C2
C3
C4
B1
B2
B3
B4
Child role with respect to C1
Parent role with respect to B3 and B4
Child role with respect to C4 and C2
Managed System
Data Flow
Control Flow
Monitored Resource
Basic Probe
Composite Probe

Figure 1: General view of a Composite Probes hierarchy

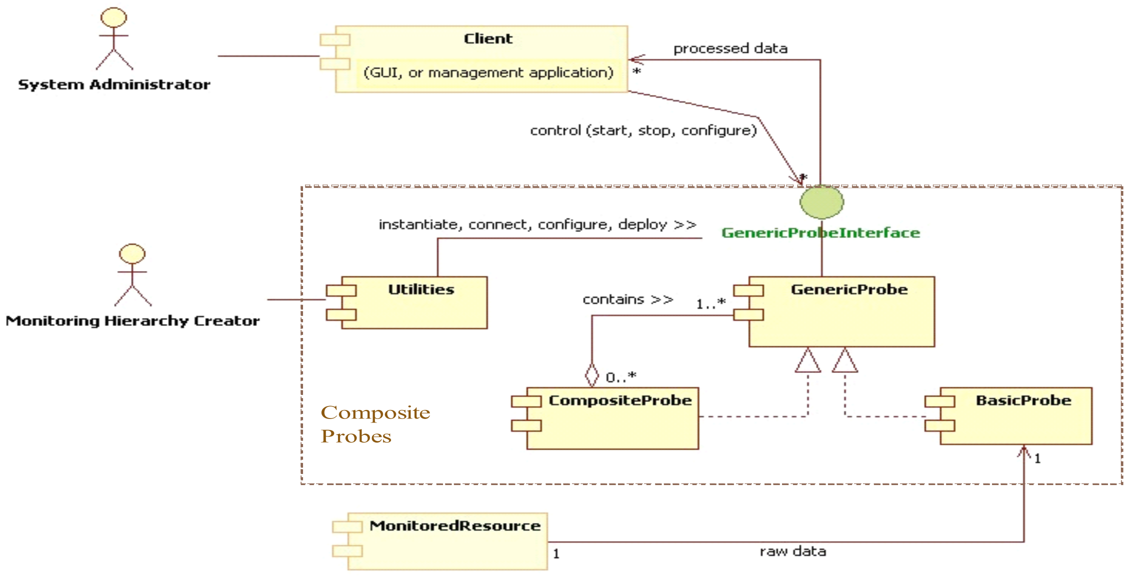

Figure 2: Logical view of the main component types provided in the Composite Probes framework

Example of Composite Probes Hierarchy for Cluster Monitoring

An example scenario in which the CPs use is potentially beneficial involves monitoring a distributed system, such as a computer cluster. Figure 3 provides a simplified example of such clustered system with two interconnected machines. For monitoring this system, two BPs are deployed on each machine for measuring their respective *memory* and *CPU* consumption. In a realistic scenario, low-level monitoring probes from tens or hundreds of clustered machines would produce a significant amount of fine-grained data, difficult to analyze manually during runtime. To alleviate such difficulties, this example shows how a CPs hierarchy is employed to aggregate monitoring data at different abstraction levels, such as overall system and cluster levels. The example CPs hierarchy in Figure 3 organizes monitoring data as follows. Two *system* CPs represent the overall load on each system (i.e. 'system1' and 'system2' probes). At a higher abstraction level, a *cluster* CP aggregates data from all available system CPs and provides cluster-level measurements (i.e. 'cluster' probe). Finally, a *cluster CPU* CP aggregates CPU data from all managed machines and represent the overall CPU load of the entire cluster. Hence, the cluster CPU probe simulates the existence of a single cluster CPU resource, hiding the fact that this abstract resource is aggregated from multiple system-level CPU resources.

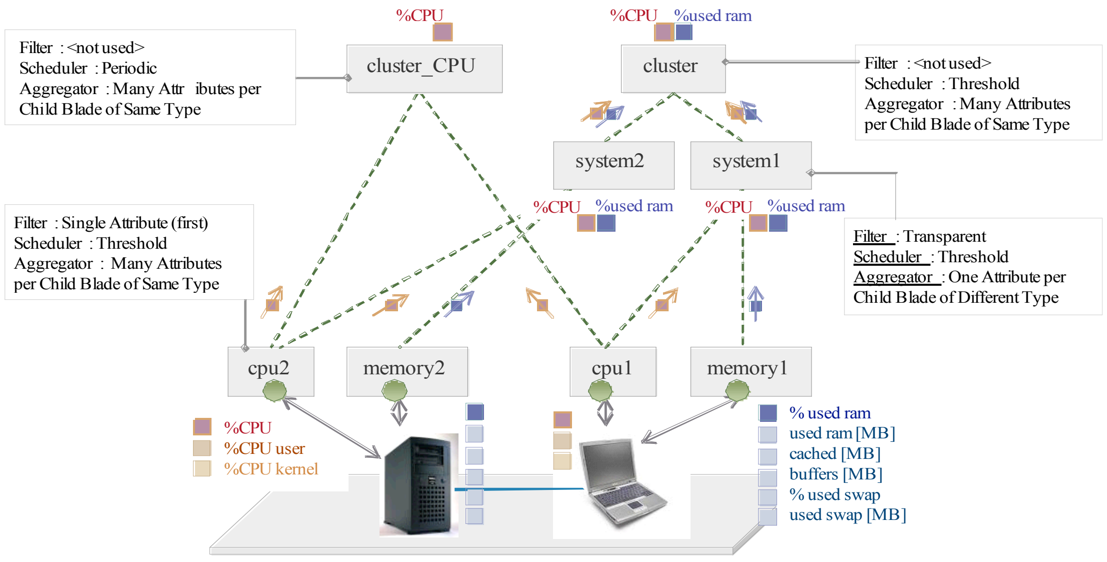

Figure 3: Sample Composite Probes hierarchy for monitoring a clustered system

Monitoring Data Flow

The monitoring *data flow* transports monitoring information through the CPs hierarchy. Data is collected by BPs and propagated up towards CPs at the top of the hierarchy. Figure 4 depicts a simplified view of the data flow through a monitoring probe. In short, a probe receives data from one or multiple *data sources*. Incoming data is collected, processed and stored as *local data*. From an external perspective, local data represents the data that a probe 'monitors'. For BPs, local data is based on actual measures taken from managed resources. For CPs, local data simulates measures taken from an abstract resource that the probe represents. In all cases, local data is subsequently filtered and forwarded to the probe's parents, or *data sinks*. In addition, external clients can access a probe's local data via direct method calls or by listening to the probe’s events (Figure 4).

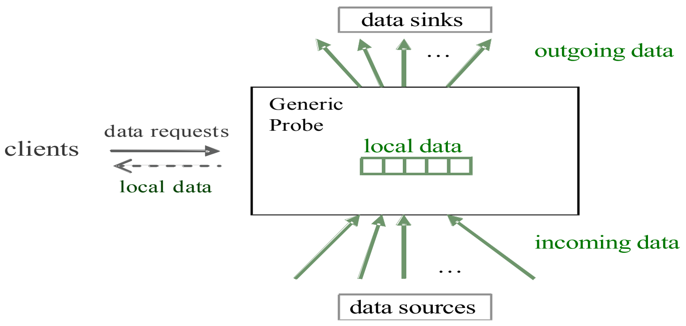

Figure 4: Generic data flow through Composite Probes

Figure 5 details the data flow view for BPs and CPs. The main dissimilarity between the two probe types is in the monitoring data *source*. Namely, BPs receive data from a single data source, which is an instrumented managed resource. The resource instrumentation code is represented in Figure 5 by the *Insert* component; it can be provided by the BPs or reused as existing legacy code. CPs receive data from multiple sources and aggregate this data into summary statistics or meaningful high-level measurements. Statistics are used to summarize a set of observations, in order to communicate concentrated or simplified information to external clients and parent CPs. Possible statistical functions include mean or median functions, standard deviation and variance, minimum or maximum functions. In addition, various aggregation algorithms can be specified for correlating data of different types into meaningful high-level indicators. For example, a system's congestion can be determined based on the individual hardware and software resource loads. Administrators are responsible for selecting or specifying the aggregation functions for each CP in the hierarchy. Aggregated results are stored as probe local data and can optionally be persisted to a storage support.

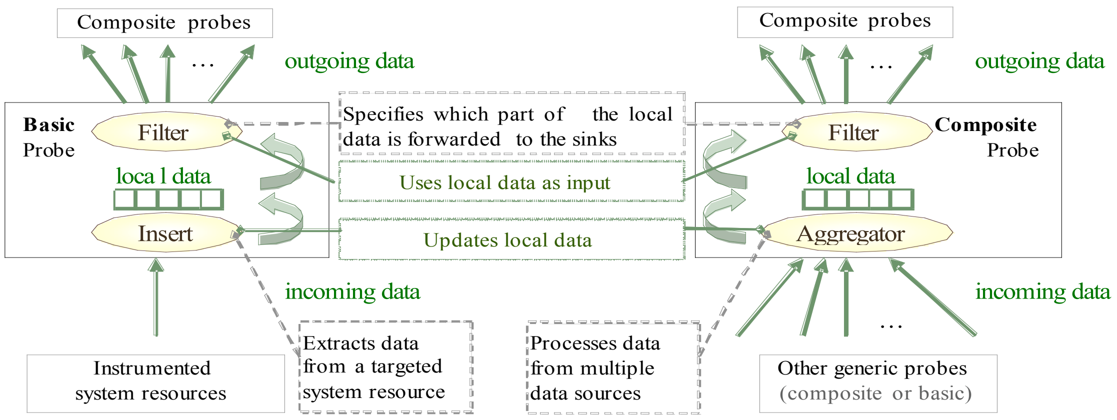
Composite probes
outgoing data
Composite probes
outgoing data
Basic Probe
Filter
loca l data
Insert
Specifies which part of the local data is forwarded to the sinks
Uses local data as input
Updates local data
Filter
Composite Probe
local data
Aggregator
incoming data
incoming data
Instrumented system resources
Extracts data from a targeted system resource
Processes data from multiple data sources
Other generic probes (composite or basic)

Figure 5: Data flow through Basic and Composite Probes

Control Commands

The *control flow* transports control commands for managing the probes' lifecycles. Namely, control commands are used to initialize, start, stop, pause, resume or terminate the execution of one probe or of a probes sub-graph. Namely, commands targeted at a certain probe can be optionally propagated to affect uniformly the probe's sub-graph. This facility simplifies hierarchy control processes by allowing an entire probe tree to be controlled via a single command sent to the tree's root.

Architectural Overview

BPs and CPs provide identical external interfaces and feature very similar internal architectures. The main architectural difference between the two probe types results from their different roles in a monitoring hierarchy. Namely, BPs use a specific *Insert* subcomponent for *extracting* monitoring data from the managed system. Conversely, CPs are in charge of *managing* monitoring data received from multiple lower-level probes. Consequently, CPs do not use an Insert subcomponent themselves, and obtain their data via connections to their child probes instead. Besides the manner in which they obtain their monitoring data, BPs and CPs contain identical subcomponents. Evidently, the way some of these subcomponents inter-communicate is influenced by whether a probe contains an Insert subcomponent or is merely connected to lower-level probes.

An overall view of the BP architecture is presented in Figure 6. The CP architecture is identical, except for the missing Insert subcomponent and additional connections to child probe components. The most important external interfaces include the *Probe Management*, *Data Collector Administration* and *Probe Control* interfaces. System administrators configure probes via their *Probe Management* interfaces. Supported management operations include setting and configuring a probe's aggregator, filter or scheduler functions, as well as connecting or disconnecting a probe from parent or child probes. Administrators use the *Probe Control* interface to perform control operations such as init, start, pause, resume, or stop on a targeted probe or probe sub-graph. Clients have access to a probe's monitoring data via the probe's *Data Collector Administration* interface. Probe interdependencies on external functionalities are represented via client interfaces. The principal client interface is the *Data Collector Write Delegate*, for forwarding data events to a probe's parents. Other client interfaces include the *Probe Response Delegate* and *Supervisor Information*, for sending asynchronous notifications, current probe state or abnormal events to parents.

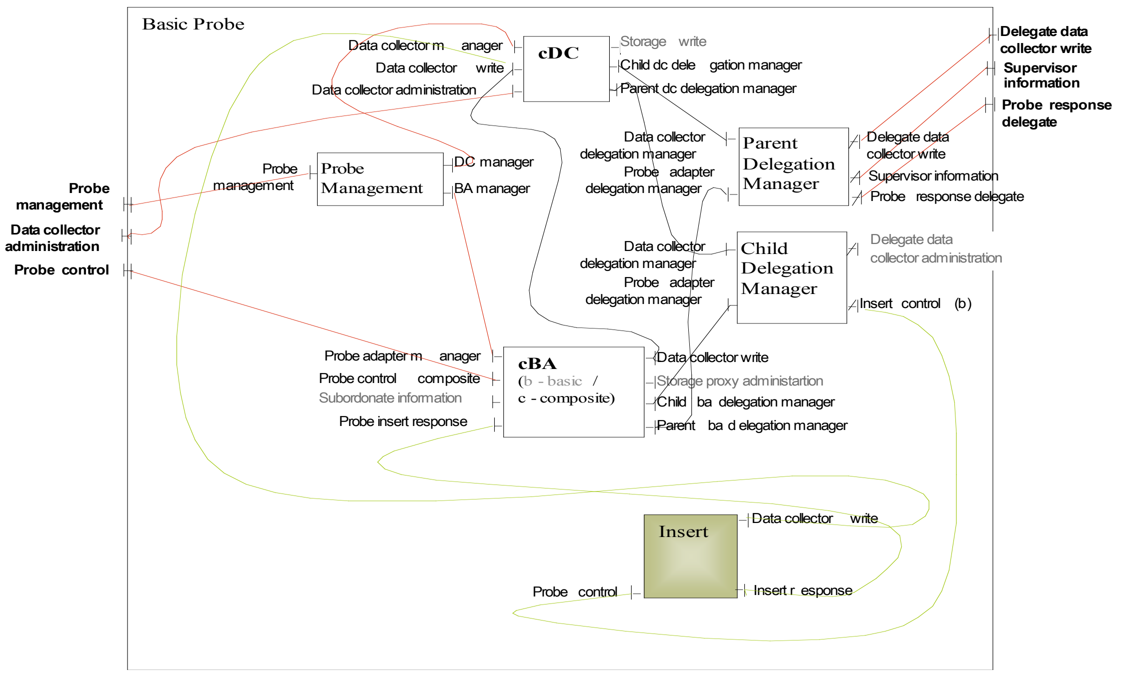
Basic Probe
Data collector manager
Data collector write
Data collector administration
cDC
Storage write
Child dc delegation manager
Parent dc delegation manager
Delegate data collector write
Supervisor information
Probe response delegate
Probe management
Probe Management
DC manager
BA manager
Data collector delegation manager
Probe adapter delegation manager
Parent Delegation Manager
Delegate data collector write
Supervisor information
Probe response delegate
Probe management
Data collector administration
Probe control
Data collector delegation manager
Probe adapter delegation manager
Child Delegation Manager
Delegate data collector administration
Insert control (b)
Probe adapter manager
Probe control composite
Subordonate information
Probe insert response
cBA
(b - basic / c - composite)
Data collector write
Storage proxy administartion
Child ba delegation manager
Parent ba delegation manager
Insert
Data collector write
Probe control
Insert response

Figure 6: Basic Probe architecture (Composite Probe architecture is identical, except for the missing Insert)

Probe subcomponents common to both probe types are described as follows. The *Data Collector (DC)* subcomponent receives monitoring data and processes it according to specific aggregation, filtering and scheduling policies. Processed data is subsequently forwarded to the probe's parents and optionally sent to a storage support for persistent logging, or further processing purposes. The *Blade Adapter (BA)* subcomponent provides asynchronous support for executing the probe's control commands. This prevents clients from being blocked while their control commands are being executed, possibly on a significantly large probe sub-graph. The *Probe Management* subcomponent represents a portal for all management operations, redirecting requests to the specific subcomponents capable of processing them, and shielding external clients from internal details. Finally, the *Child* and *Parent Delegation Manager* subcomponents manage probe communication with child and parent probes, respectively. Their role is to isolate probe inter-connection logic from functional code, and facilitate eventual communication protocol changes.

3 DATA-PROCESSING IN COMPOSITE PROBES

A probe's *Data Collector (DC)* dictates the specific manner in which the probe handles incoming monitoring data and makes it available to clients and parent probes. The principal processes involved in collecting, processing and forwarding incoming data are assigned to specialized DC subcomponents, namely the *Aggregator*, *Filter* and *Scheduler*. Consequently, a DC's processing logic is highly configurable, as the algorithms and policies of each of its subcomponents can be individually specified and configured. In short, the DC Aggregator collects incoming data events over well-specified *intervals*. At the end of each interval, the Aggregator uses the collected data events to calculate and update the probe's local data. Intervals are dictated by the DC's Scheduler. Processed local data is sent to the DC's Filter and the filtered result forwarded to the probe's parent probes. This design enables administrators to create flexible hierarchies with custom data-processing paths by setting the data-processing policies of each instantiated probe.

Data Aggregator

The *Aggregator*'s role is to manage incoming data from multiple data sources. They provide two major functions, namely, collecting incoming data and processing it to calculate local data. The first function stores incoming data events according to a specified policy. For example, data received from identical probe types can be mixed together into a common storage (e.g. the 'cluster' and 'cluster CPU' probes in Figure 3), while data received from sources of different types must be stored separately (the 'system' probe in Figure 3). The second Aggregator function dictates how stored data is processed to calculate local statistics (e.g. maximum or weighted average functions).

Data Filter

The *Filter*'s role is to determine a probe's outgoing data for the probe’s data sinks. A typical filter determines which subset of a probe's local data will be sent as output data. A probe's Filter is unaware of whether the output data will be forwarded to one or multiple data sinks, as all inter-probe communication is managed by probe Delegation Managers.

Interval Scheduler

The *Scheduler*'s role is to dictate a CP's data processing *intervals*. Intervals dictate the exact instants at which a probe's Aggregator calculates local statistics based on the collected monitoring data. Each Scheduler has an associated *Task*, which can be implemented to trigger various data processing and forwarding functions. The Scheduler will trigger the Task's execution at the end of each interval.

4 CURRENT IMPLEMENTATION

Implementation Overview

A CPs prototype was implemented for demonstrating the main capabilities and functions proposed in the framework specification. The prototype implementation is based on Julia[6], a Java implementation of the Fractal[7] component model, and on additional Fractal utilities, including Fractal-ADL[8], Fractal-RMI[9] and Fractal-JMX[10]. Fractal is a hierarchical component model suitable for building complex applications with high modularity and adaptability requirements. The CPs framework is based on an existing monitoring and load-injection application called CLIF[11]. CPs extends CLIF's monitoring architecture and functions in order to add hierarchical data-management functions to the flat data representation initially supported. CPs equally reuses some of the instrumentation code provided in CLIF for monitoring UNIX, Windows and MacOS resources, including CPU, memory and disk utilization. The CPs prototype implements the BP and CP types, and provides several Aggregator, Filter and Scheduler implementations. The framework implementation supports remote client access to CP instances via the RMI[12] and JMX[13] protocols.

Implemented Aggregation Functions

Several Aggregator types are provided in the current CPs implementation. First, a "*Many per Identical Type*" Aggregator was implemented to aggregate multiple parameters from identical data sources. This implies that all data sources send data with identical formats and semantics. Secondly, a '*One per Different Type*' Aggregator was implemented to aggregate a single data item from different source types. This implies that each source sends a single parameter's values, where each parameter is of a different type. A third Aggregator type was implemented to process data on a managed element's state. The '*Component State*' Aggregator receives state events from a single element and calculates

[6] Julia: the reference Java implementation of the Fractal component model (fractal.objectweb.org/julia)
[7] Fractal: component model (fractal.objectweb.org)
[8] FractalADL: Fractal model's Architecture Description Language (fractal.objectweb.org/fractaladl)
[9] FractalRMI: enables remote method calls between Fractal components (fractal.objectweb.org/fractalrmi)
[10] FractalJMX: enables JMX management of Fractal applications (fractal.objectweb.org/fractaljmx)
[11] CLIF: generic Java-based performance testing framework (clif.objectweb.org)
[12] RMI: Java Remote Method Invocation from Sun Microsystems (java.sun.com/javase/technologies/core/basic/rmi)
[13] JMX: Java Management Extensions from Sun (java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement)

its local data as the value of the last event in the interval. Finally, a '*Path State*' Aggregator was implemented to provide the overall state of an application call-path. Each Path State Aggregator collects state data from the multiple elements (e.g. nodes and connections) constituting the managed call path. At the end of each interval, the overall path state is calculated based on the individual element states.

Implemented Filtering Functions

Two filter types are available in the current CPs implementation. First, a '*Single Parameter*' Filter selects a single data element from the input data set and sends it as the output data set. Second, a '*Transparent*' Filter sends the entire, unchanged input data set as the output data set. This filter was provided in order to maintain a uniform data-processing architecture across all CPs.

Implemented Scheduling Policies

Two Scheduler types are available in the current CPs implementation. First, a '*Threshold*' Scheduler determines intervals based on the number of monitoring data events received. Namely, the scheduler dictates the end of an interval each time the number of received events crosses a certain threshold. The second scheduler type, a '*Timer*' Scheduler, determines intervals in a strictly time-based manner. Specifically, it uses a configurable period to determine the end of each interval. Both Scheduler implementations trigger the statistics calculation process in the associated probe Aggregator, at the end of each interval. New statistics are immediately filtered and forwarded to the probe's parents.

5 USAGE SCENARIOS AND PRELIMINARY RESULTS

Two example usage scenarios were tested to show the CPs framework benefits for monitoring large-scale, distributed systems. In the first example, a CPs hierarchy was built for monitoring various resource types in a clustered computer system. In the second example, a different CPs hierarchy was constructed to monitor the state of component-based data-processing applications, at various granularity levels. The goal of the two usage scenarios was to demonstrate the correct functionality of various CPs hierarchies and show the capacity of instantiating and inter-connecting different BPs and CPs, configured with various Aggregator, Filter and Scheduler types. The tested scenarios show the utility of using CPs when managing complex systems. Meanwhile, these initial scenarios did not explicitly test performance and scalability issues, even though such concerns were carefully considered in the framework's architecture specification.

Monitoring the Overall Load of a Clustered System

The CPs framework was used to create a monitoring hierarchy for supervising the resource usage of large-scale distributed systems. The monitoring hierarchy used is presented in the example in Figure 3. The following probe types were created to build the monitoring hierarchy in the presented scenario. First, two BPs were used to monitor the CPU and memory resources on the managed stations. One memory and one CPU probe was instantiated to monitor the respective resources on each machine. The BPs were configured to use Many per Identical Type Aggregators and Single Parameter Filters. Consequently, even though multiple CPU and memory parameters are monitored and made available at the BP level, only the values of one CPU and one memory parameter are forwarded to parent CPs. The percentage memory usage was selected as the unique forwarded measure for memory probes and the CPU percentage usage for CPU probes.

Secondly, a *system load* CP was prepared to represent the overall resource consumption on each machine. Two such system CPs were instantiated and used in the monitoring hierarchy, one for each managed station. Each system load probe collected data from the corresponding CPU and memory BPs on the monitored station. The system CPs were configured to use a One per Different Type Aggregator and a Transparent Filter. This implies that system probes collect one measure from each distinct child probe and forward all calculated values to parent probes. Concretely, a system probe collects two different measures, one from its CPU child probe and one from its child memory probe. Statistics calculated based on these measures are subsequently forwarded to parent probes, in this case, to a *cluster load* CP. In the current scenario, the system CP maintains the different collected measures separate from each other, as shown in Figure 7 (i.e. the 'system' probe in the bottom-right pane) and Figure 8 (i.e. 'system2' probe in the top pane). Nonetheless, the Aggregator used can easily be modified to calculate a unique *system load* measure, based on the separate measures collected. It is up to each system administrator to define the actual function to use for calculating a global system load measure based on discrete resource data (e.g., a max function considers the system load as equal to the load on the most used resource, most likely to become a bottleneck).

Finally, a *cluster load* CP was created to represent the overall resource load on the entire distributed system. A single probe of this type was instantiated and bound to collect data from all system CPs of the managed machines in the cluster. The cluster load CP was configured to use a Many per Identical Type Aggregator. This CP receives measures on the average memory and CPU consumption of the two systems in the cluster. Based on these measures, the Aggregator calculates the average CPU and memory consumption at the overall cluster level, at the end of each monitoring interval. The two selected stations had similar CPU and memory capacities, so that calculating the average resource usage in percentages for the two stations made sense. The Aggregator used can easily be modified to provide a unique *cluster load* measure, based on the separate CPU and memory data collected from the individual systems involved. In addition, a different Aggregator could equally be employed show the total resource usage in the cluster. Care must be taken when creating monitoring hierarchies from probes with different Aggregator and Filter types. System administrators should make sure that calculated measures at higher hierarchical levels make sense with respect to data received from probes in the lower hierarchical levels. All probes were configured to use Threshold Schedulers, meaning that statistics were calculated every time the number of collected monitoring events crossed a certain specified threshold.

Figure 7 and Figure 8 present snapshots of monitoring data obtained via the CPs hierarchy, at various abstraction levels. Figure 7 depicts data from monitoring probes deployed on the first machine and Figure 8 data from the second machine in the cluster.

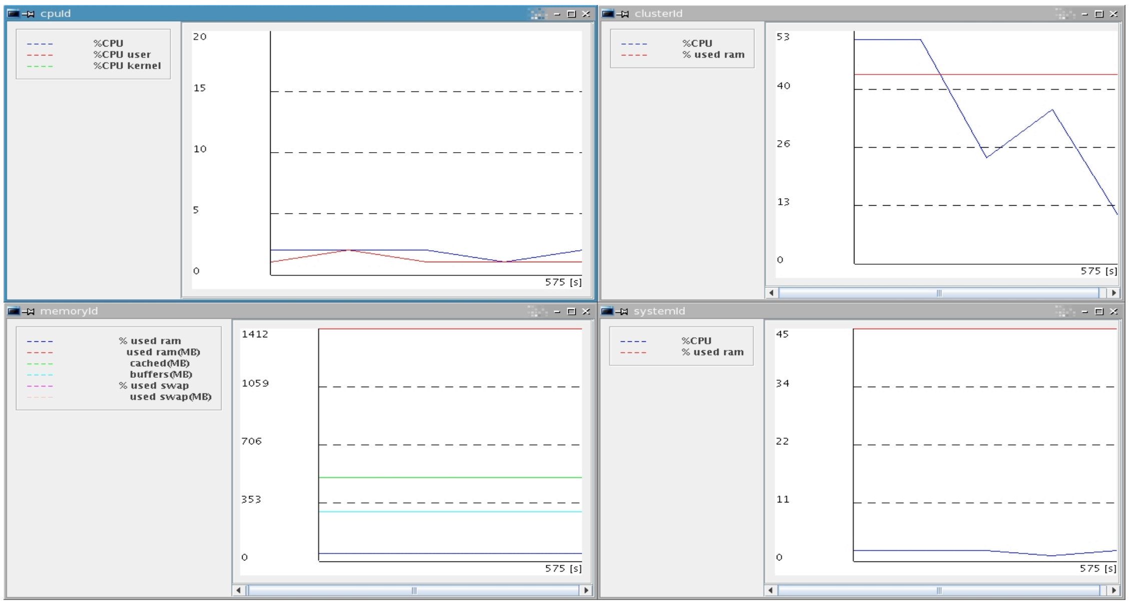

Figure 7: Composite Probes for System 1 and overall Cluster
Top-left: CPU 1 measures; bottom-left: memory 1 measures; top-right: cluster measures showing the average CPU and memory consumption in the entire cluster; Bottom-right: centralized system 1 measures

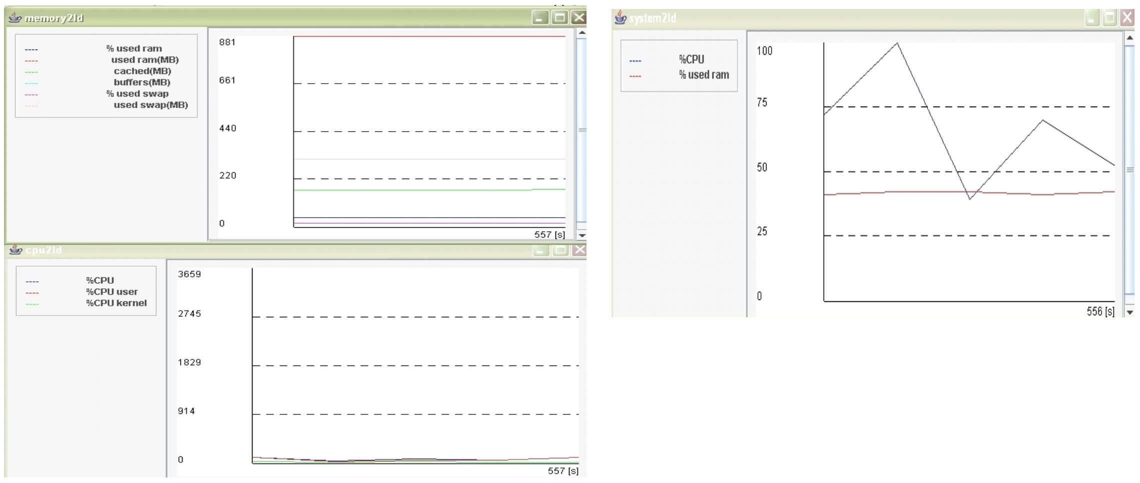

Figure 8: Composite Probes for system 2
Top: overall system 2 measures; middle: memory 2 measures; bottom: CPU 2 measures

As shown in the figures, the CPU, memory and system load probes were deployed on the respective stations they monitored. The cluster load CP was deployed on the first station. Monitoring data from the seven probes in the monitoring hierarchy is presented in the corresponding graphs in Figure 7 and Figure 8. Each graph is labeled with the unique id of the corresponding probe. The graphs show how the system probes provide centralized data on the overall CPU and memory consumption on the machines they represent. In addition, the cluster probe provides average consumption values for the managed cluster resources, based on values received from the constituent system probes. Initial tests were successfully carried out to verify the framework's support for external client access via the RMI and JMX protocols and for dynamic modifications on hierarchies of probes of similar types. Specifically, CPU probes were dynamically added and removed from a cluster CPU probe (Figure 3), without disrupting the monitoring system's execution.

Monitoring the Overall Availability of a Data-Processing Graph System

The CPs framework was used to create a monitoring hierarchy for supervising component-based data-processing applications. This type of applications involves several interconnected *nodes*, or components, that form a graph-like architecture. The application data-processing nodes are possibly distributed across multiple machines. Input data is forwarded between the graph's nodes, following a well-defined data-processing *path*. Each node contains a certain data-processing function, so that the node's output data is the result of the node's function applied on the node's input data. An application can provide several distinct processing paths, where different paths can share multiple nodes.

With respect to the performed testing scenarios, the central parameter of interest in the data-processing application scenario was the application *state*. State values were monitored at various granularity levels, such as node, data-processing path, overall node availability and global application availability. The possible node states are *Loaded*, *Initialized*, *Started*, *Stopped* and *Unloaded*. In order to display the nodes' states in graphical formats, the possible node states are mapped to numeric values (i.e. the Loaded state is mapped to value zero, Initialized to 1, Started to 2, Stopped to 3, and Unloaded to 4). A processing path's state depends on the respective states of the path's nodes. More complex scenarios can be envisaged to also consider inter-node connections and execution platform states when determining path states. Currently, a path is considered to be in the Stopped state if *any* of its nodes is in the state Stopped and in the state Started if *all* of its nodes are in the state Started. The *overall node availability* is calculated based on the individual states of all nodes in the application. Similarly, the *overall application availability* is considered based on the individual states of the available paths. Four probe types were used to build the CPs hierarchy for providing state information at the presented abstraction levels (Figure 9-a). With respect to the probes' internal configuration, each probe type was configured with a different Aggregator, depending on the probe's role and functions. Meanwhile, all probe types were configured to use the same Filter and Scheduler types, specifically, Transparent Filters and Timer Schedulers.

The different roles and functionalities assigned to each of the four probe types are described as follows. First, special-purpose BPs where employed to obtain information on the nodes' states. One BP was instantiated for each data-processing node in the monitored system (i.e., probes S1 to S10 in Figure 9-a). The BPs were configured to use State

Aggregators. In this testing scenario, the legacy instrumentation code provided by the data-processing application was used to provide the BP's Insert functionality. This was possible since the managed application already provided JMX-based monitoring events at the node level. Therefore, the BPs were implemented as JMX adapters to the existing instrumentation code, registering as listeners to node *state-change* events. At a higher abstraction level, a second CP type was used for representing the overall node availability in the application (i.e. the 'Ovrl Node States' probe in Figure 9-a). This CP is connected to all BPs (i.e., S1 to S10) and aggregates all individual node states for providing an overall view of the system node availability. A One per Different Type Aggregator was used for this CP type. Consequently, the 'Ovrl Node States' CP currently displays all monitored node states individually, providing a centralized view of the application state at node level. This Aggregator can easily be extended to use the centralized data and calculate the node availability, in percentages, as a unique measure of the system node state.

The remaining CP types are related to the application's data-processing paths. The third CP type monitors and represents the state of individual data-processing paths. A special Aggregator was implemented to calculate a path's state based on the individual states of the nodes in that path. This Aggregator can be extended to consider additionally the states of node inter-connections, or execution platforms. Finally, a fourth CP type was introduced to centralize all path states in the system (i.e. the 'Ovrl Path States' in Figure 9-a), which it collects from the individual path probes (P1, P2, P3 and P4). This CP type measures the global availability of the application, as experienced by external clients. The current implementation uses a One per Different Type Aggregator for this CP, separately maintaining the individual path states. This Aggregator could be extended to calculate the percentage of available paths, as a unique measure of the overall system availability.

The data-processing application depicted in Figure 9-b was instantiated and monitored using the CPs framework. This application consists of 10 data-processing nodes, interconnected to form four distinct data-processing paths. For example, path 1 in Figure 9 consists of the ordered nodes 1, 2, 3, 4 and 5; and path 4 consists of nodes 9, 3 and 10. As such, certain nodes are part of multiple data-processing paths. In the example, paths 1, 2 and 3 all share node 3, while node 10 is only involved in path 4. A direct consequence is that a node's failure can have different effects on the global system functioning, depending on the node's utilization in processing paths. For instance, a failure in node 10 will only disrupt the functioning of path 4, while a disruption in node 3 would affect all application paths and render the entire system unavailable. In this example, the dysfunction of a single node in the system, or 10% node unavailability, can have a considerably different impact on the *overall* system availability. More precisely, 10% unavailability at the node-level can result in 25% application unavailability if node 10 is disrupted and in 100% application unavailability if node 3 is disrupted. A monitoring hierarchy that highlights such information was constructed using the CPs framework, as shown in Figure 9-a. This CP hierarchy allows administrators to obtain system state information at various granularity and abstraction levels, as monitoring data is readily available on each node, path and global node and path availabilities.

Two example scenarios were considered, starting with a fully functioning application, where all nodes were in the 'Started' state. The first use-case scenario shows how stopping node 10 affects the local and overall application availability (Figure 10).

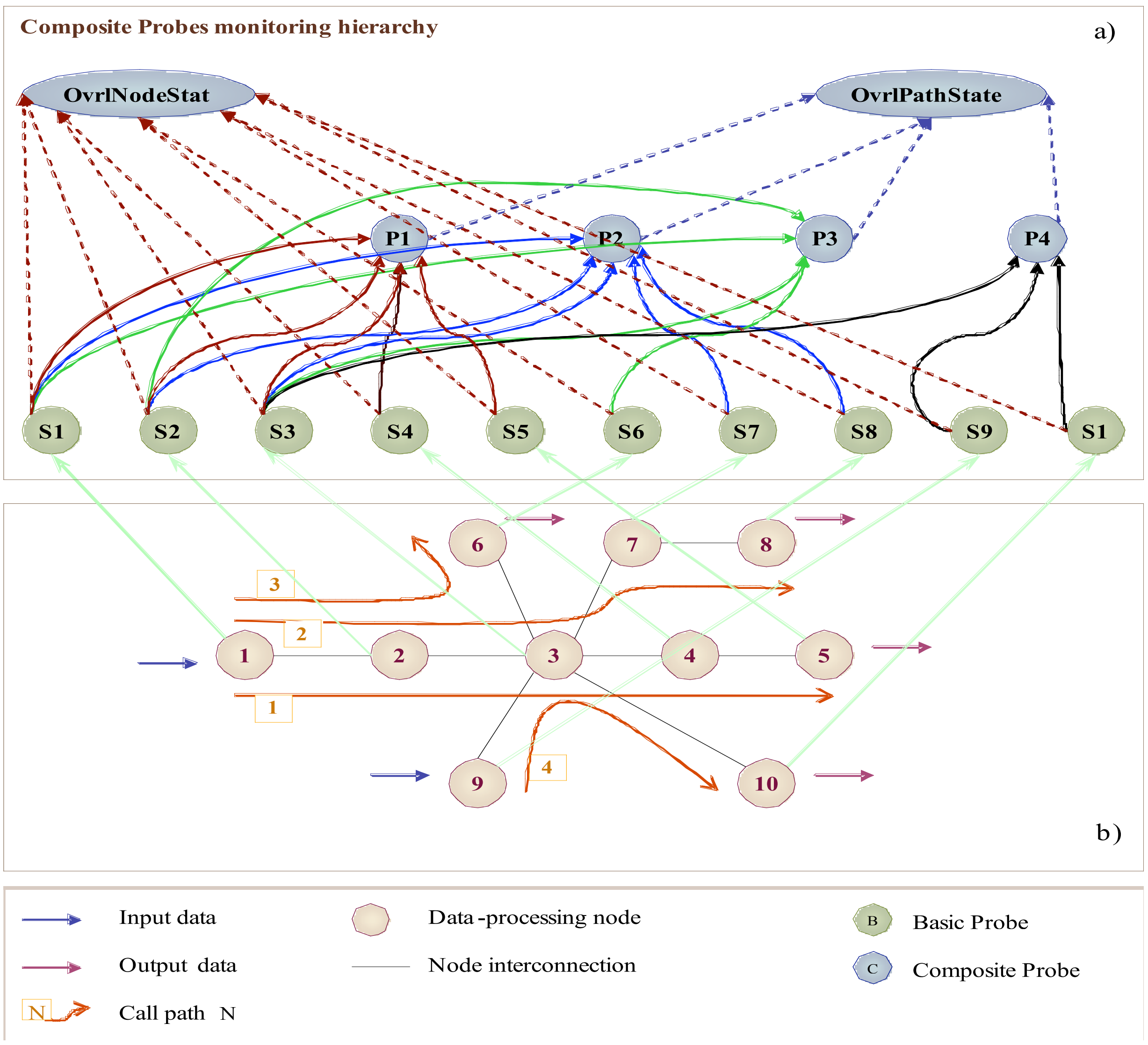

Figure 9: Using Composite Probes to monitor a data-processing application
a) CPs hierarchy, probe types: node BP (S1–S10), path CP (P1–P4), all paths CP (OvrlPathState), all nodes CP (OvrlNodeState); b) Monitored application: ten nodes (1-10) and four paths (1-4)

At the node granularity level, the monitoring data shows node 10 changing its state from Started to Stopped, while all other nodes remain in state Started. Path 4, which uses node 10, consequently changes its state from Started to Stopped, while all other paths remain unaffected. This data is depicted in Figure 10, as follows. The top-right graph corresponds to probe S10, for node 10; the top-left graph corresponds to probe ‘Ovrl Node States’ and shows centralized data on all nodes states; the bottom-right graph corresponds to probe P4 and shows the sate of path 4; the bottom left graph corresponds to probe ‘Ovrl Path States’ and shows centralized data on all paths states. The two overall Aggregators for probes ‘Ovrl Node States’ and ‘Ovrl Path States’ can seamlessly be extended to indicate the overall node availability of 90% (i.e. 9 out of 10 available nodes) and the global application availability of 75% (i.e. 3 out of 4 available paths). Similarly, the second scenario shows how stopping node 3 affects the local and overall application availability (Figure 11). In this case, the node's failure causes the entire application to become unavailable, as all paths are using node 3. At the overall level, this translates in a 90% overall node availability and a 0% global application availability.

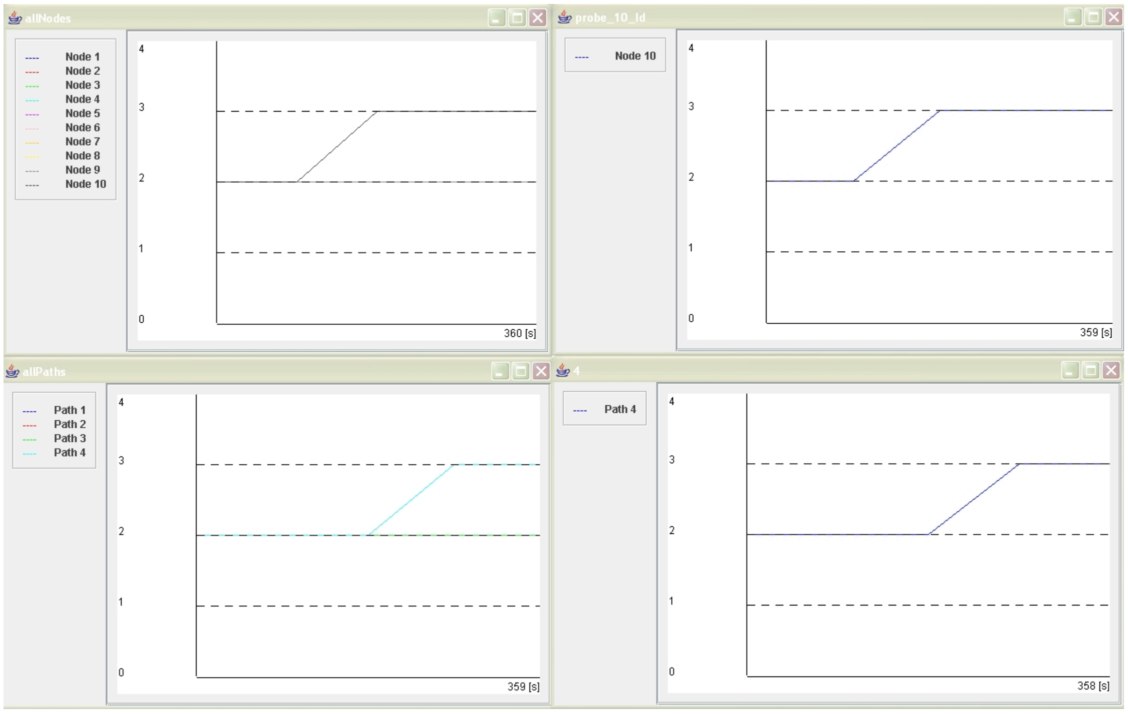

Figure 10: Local and overall impact on application availability when stopping node 10
Global node availability: 90%; Global data-processing application availability: 75%

The two scenarios indicate the important benefits of observing monitoring data at different granularity levels. While in both cases a single node was observed to fail, the actual impact on the overall application availability varied dramatically. Ultimately, from an application management perspective, the global data-processing availability is of major importance, as it is directly experienced by application clients. While significant, such global availability information is difficult to detect by merely following fine-grain measures at the individual node level. In addition, this difficulty increases dramatically with the application's scale and distribution. This example scenario shows how a CPs hierarchy can be used to alleviate such difficulties by providing aggregated monitoring information irrespectively of the system's scale. Similar CPs hierarchies can be equally employed to monitor the state and activity of different types of modular systems, such as component or service-based applications.

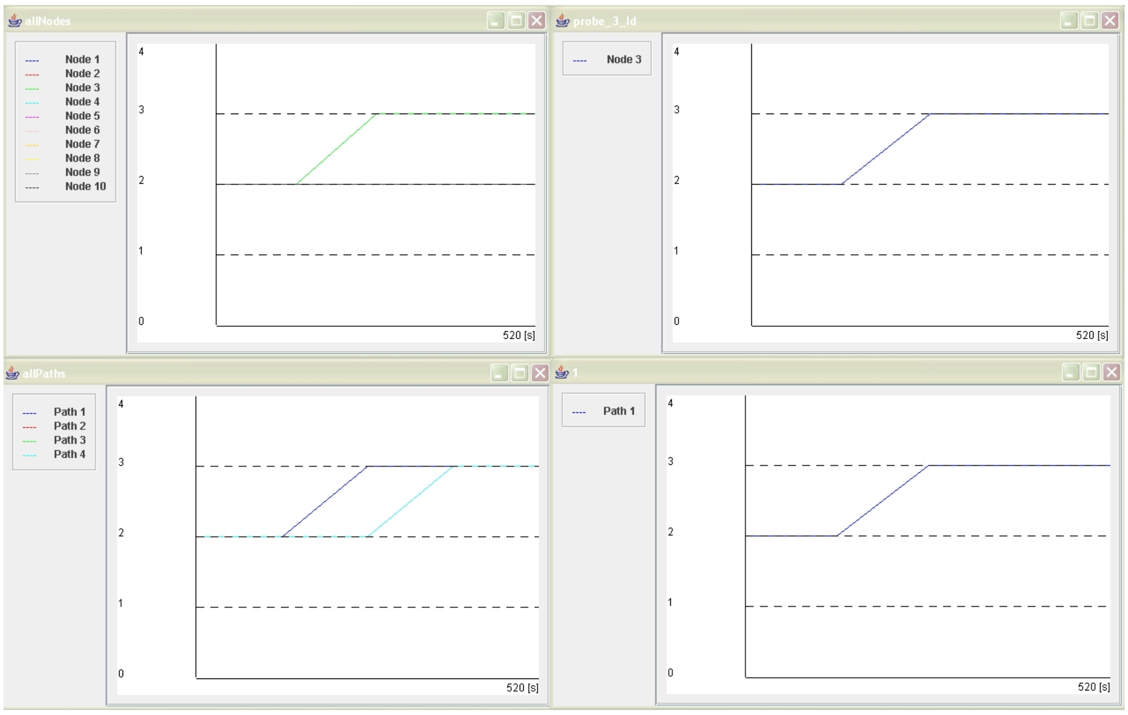

Figure 11: Local and overall impact on application availability when stopping node 3
Global node availability: 90%; Global data-processing application availability: 0%

6 RELATED WORK

Hierarchical monitoring systems, both freeware (e.g. Ganglia [1], Clumon[14], Supermon[15] [2], or Parmon [3]) or industrial (e.g. Big Brother[16], or Cluster Systems Management[17]) are available for monitoring clustered and grid systems. These tools represent mature, scaleable and efficient monitoring solutions for the precise system types they were designed for. On the Other hand, CPs' significant advantage lies in its high flexibility and extensibility features, as it provides support for creating and integrating customised probes and probe hierarchies for a wide range of system types.

A relatively recent project, Test & Performance Tools Platform[18] (TPTP) has many similar goals and characteristics with the proposed CPs framework. TPTP provides frameworks and services for developing test and performance tools, for system evaluation and profiling. The Monitoring Tools Project extends the TPTP platform to provide support for collecting, analyzing, aggregating, and visualizing data in detailed or

[14] Clumon: cluster monitoring system, NCSA (clumon.ncsa.uiuc.edu)
[15] Supermon: high performance cluster monitoring, Los Alamos National Laboratory (supermon.sourceforge.net)
[16] Big Brother : web-based system and network monitoring solution, Big Brother® Software (bb4.com)
[17] Cluster System Management: management of distributed and clustered IBM servers, IBM
[18] Eclipse Test & Performance Tools Platform Project (www.eclipse.org/tptp)

statistical views. Future work will follow the TPTP Project evolution and determine functionalities that can be reused and/or integrated with the CPs framework.

Virtually all research and industry projects in the autonomic computing area involve monitoring and analysis utilities for collecting, organizing and correlating data from the managed domain [4]. Depending on each project's goals, monitoring and analysis are required for creating system models (e.g. [5]), establishing execution contexts (e.g. [6]), evaluating and optimizing system performance (e.g. [6], or [8]), detecting application faults and system failures (e.g. [9], or [10]), or determining execution paths and performance anti-patterns (e.g. [11]). Most existing projects use proprietary solutions for collecting, grouping, aggregating and filtering monitoring data. CPs provides a reusable, scaleable and extensible framework for creating such monitoring and analysis facilities and accessing them via standard protocols.

7 CONCLUSIONS AND FUTURE WORK

Autonomic management systems require complex monitoring and analysis functions, which existing tools do not generally provide. This paper proposes Composite Probes (CPs), a flexible, hierarchical monitoring framework for autonomic management applications. CPs combines Basic Probes (BPs) that extract data from managed resources with highly customizable Composite Probes (CPs) that aggregate and filter data at various abstraction levels. CPs can be seamlessly extended with new instrumentation BPs, data-processing algorithms, scheduling policies and communication protocols. These characteristics make CPs suitable for a wide range of management applications and reusable across a wide range of system types. A CPs prototype was implemented and tested in two system management scenarios. The presented examples demonstrated how the CPs prototype could be used to create special-purpose monitoring hierarchies, combining the available aggregation, filtering and scheduling functions and integrating third-party instrumentation code via JMX. The examples showed CPs' suitability for monitoring dissimilar system types at various abstraction levels. Using CPs, managers have different views on a managed system, such as performance, architectural, or availability views, and browse through CPs hierarchies to determine the exact cause of an observed miss functioning. Future work will focus on integrating CPs with existing monitoring and management frameworks (i.e. CLIF, Jade and Jasmine). CPs will be extended as needed with new BP types, probe adaptors and data-processing functions. Additional support for JMS-based communication is equally envisaged.

8 ACKNOWLEDGEMENTS

The authors would like to thank their colleagues at Orange Labs, Marc Leger, Mehdi Kesis and Thomas Saillard for their valuable research ideas and technical support.

REFERENCES

[1] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience", *Parallel Computing*, Vol. 30, Issue 7, July 2004

[2] M.J. Sottile, R.G. Minnich, "Supermon: a high-speed cluster monitoring system", *IEEE International Conference on Cluster Computing*, pp 39-46, 2002

[3] R. Buyya, "Parmon: a portable and scalable monitoring system for clusters", *Software Practice and Experience*, pp 723-739, 2000

[4] "An Architectural Blueprint for Autonomic Computing", *IBM White Paper*, 2005 www-128.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure,", *IEEE Computer* Vol. 37, Num. 10, pp. 46-54, October 2004

[6] A. Diaconescu, J. Murphy, "Automating the Performance Management of Component-Based Enterprise Systems through the use of Redundancy", *ACM/ IEEE Conference on Automated Software Engineering,* Long Beach, USA, , 2005

[7] A. Diaconescu, A. Mos and J. Murphy, "Automatic Performance Management in Component Based Software Systems", *International Conference on Autonomic Computing*, New York, USA, 2004

[8] S. Bouchenak, N. De Palma, D. Hagimont, S. Krakowiak, and C. Taton, "Autonomic Management of Internet Services: Experience with Self-Optimization", International Conference on Autonomic Computing, Dublin, Ireland, 2006.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services", *International Conference on Dependable Systems and Networks*, pp 595 – 604, 2002

[10] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton, "Autonomic Management of Clustered Applications", *IEEE International Conference on Cluster Computing*, Barcelona, Spain, 2006

[11] T. Parsons, J. Murphy, "Detecting Performance Antipatterns in Systems Built using Contextual Component Frameworks", *Journal of Object Technology*, to appear.

[12] B. Dillenseger, E. Cecchet, "CLIF is a Load Injection Framework", *Middleware Benchmarking workshop*, *OOPSLA*, Anaheim, CA, USA, 2003

About the author(s)

Ada Diaconescu is a research engineer in the Adele/LSR group, University Joseph Fourier, Grenoble, France. The presented work was carried out as part of her postdoctoral work at Orange Labs. She obtained her PhD from the School of Electronic Engineering and Computing at Dublin City University. Her main research interests include autonomic computing and complex systems. Contact her at adadiaconescu@gmail.com. See also adadiaconescu.there-you-are.com

D Event-Condition-Action Rules for Components

Flexible Reactive Capabilities in Component-Based Autonomic Systems

Jayaprakash Nagapraveen[*]
HADAS Group, LIG
Saint Martin d'Hères, France
nagapraveen.jayaprakash@imag.fr

Thierry Coupaye
France Telecom R&D
Grenoble, France
thierry.coupaye@orange-ftgroup.com

Christine Collet
INP Grenoble
LIG Laboratory
Saint Martin d'Hères, France
Christine.Collet@imag.fr

Pierre-Charles David[†]
OBASCO Group, EMN/INRIA
Nantes, France
pcdavid@gmail.com

ABSTRACT

Reactive behaviour, the ability to (r)eact automatically to take corrective actions in response to the occurrence of situations of interest (events) is a key feature in autonomic computing. In active database systems, this behaviour is incorporated by Event-Condition-Action (ECA or active) rules. Our approach consists in defining a mechanism for the integration of these rules in component-based systems to augment them with autonomic properties. The contribution of this article is twofold. First, we propose a rule model, i.e. a rule definition model and a rule execution model, that can be coherently integrated into a component model. Second, we propose a graceful architecture for the integration of active rules into component-based systems in which the rules as well as their semantics (execution model, behaviour) are represented/implemented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. These foundations form the basis of a framework/toolkit which can be seen as a library of components to construct events, conditions, actions, rules and policies (and their execution subcomponents). The framework implementation is extensible: additional components can be added at will to the library to render more elaborate and more specific semantics according to certain applicative requirements.

[*]This work was done while the author was a PhD student at France Telecom R&D and IMAG-LSR
[†]This work has been done while the author was a post-doctoral fellow at France Telecom R&D.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture; D.2.13 [**Software Engineering**]: Reusable Software— ; H.2 [**Database Management**]: Systems

General Terms

Design, Experimentation

Keywords

autonomic systems, component-based architectures, ECA / active rules

1. INTRODUCTION

The overall motivation which underlies the emergence of autonomic computing[9] is based, from an IT industrial point of view, on the observation that the costs related to the software infrastructures (TCO) currently move from costs related to the development and licensing to costs related to the deployment and exploitation. It appears also quite clearly that a "manual administration" of pervasive environments such as "machine to machine" (automotive, home networking, etc.) software infrastructures or grid computing is, in practice, almost impossible. Autonomic computing thus aims basically at, as much as possible, automating the deployment and management (administration) of software systems in order to decrease human interventions and associated costs.

We consider in this work that an autonomic system is composed of an *autonomic infrastructure* superimposed on a *target component-based system.* The upper layer or the autonomic infrastructure, is responsible for implementing a *control loop*, i.e. instrumenting the components of the target system for monitoring, detecting and notifying events, diagnosing the system based on these events, and making decisions to determine what and how corrective actions need to be executed, and finally, executing the corrective actions on the components of the target system.

Our work focuses on the architecture and the behaviour of control loops. We propose to use Event-Condition-Action (ECA or active) rules[12], a mechanism widely used in active database systems to provide a reactive behaviour (an elaborated form of *triggers* as found in most commercial DBMS). The fundamental idea is to "extract" the reactive functionality of active database systems[8], and to "adapt and inject" it in component-based systems so as to provide them with autonomic capabilities.

Our objective is to realise a modular and extensible framework/toolkit for the construction of ECA rule-based autonomic architectures. In this framework, the rules as well as their semantics (execution model, behaviour) are represented/implemented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. In its basic version, the framework is a library formed of basic components (and subcomponents) which permits to construct basic rules. The framework is extensible, i.e, additional components can be added at will to the library to render more elaborate and more specific semantics according to certain applicative requirements.

The main goal of this article is to introduce the design of this framework (called Fractal ECA) through an illustrative example used throughout the paper. Due to space constraints, only the elements of the framework required for the comprehension and implementation of the example are discussed.

Sample Autonomic Scenario. Let us consider as a target system, a minimal HTTP webserver with the sole functionality of retrieving HTML documents. It is made of two main components, namely a *Request Receiver* and a *Request Processor*. The request receiver component or front-end uses a lower-level scheduler component that creates a new activity (a thread in this case) for each request. The request processor component or back-end analyses the request, logs it and responds to it. Our Comanche web server[1] (shown in the bottom of Figure 4), is a multi-threaded system in Java that follows a prefork model - the parent process forks new child thread for every request. Most of the action takes place in the child threads - figuring out what the requests mean and sending the requested content back to the client. Once the processing is done, the child thread waits for the next one from its parent. Multithreading avoids the server from being idle until an I/O operation is finished. On the other hand, it introduces an overhead on the CPU due to the creation of new threads and to the commutation between existing threads. The right trade-off between these two conflicting factors reflects on the system performance. We propose a simple active rule to maintain this compromise (in complete informal format):

RULE *DoubleThreadPoolSize*
ON each webpage request
IF (no free threads in the thread pool)
DO { Double the thread pool size }

[1]which comes with the Fractal distribution. See http://fractal.objectweb.org.

The above rule's purpose is to extend the capacity of the webserver by increasing the size of the thread pool, that is the number of possible concurrent child threads. A complementary rule can be equally envisaged that reduces the size of the thread pool, when the number of passive threads are high, thus liberating unused ressources.

The overall semantics of an ECA rule is the following: *when an event of type E occurs, if condition C is satisfied then execute action A*. Behind this apparent simplicity hides a great deal of complexity that raises many questions which we shall illustrate based upon our concrete example. Indeed, several questions are raised by the execution of this simple rule, to list a few:

- Is a new instance of a rule triggered for every occurrence of a triggering event (webpage request)? or once for several occurrences of the triggering event?
- When are the condition and action parts executed with respect to the target system execution? before the execution of the triggering operation or after?
- Is the rule executed in the same execution thread as the triggering operation or in a separate one?

For the above questions, several options exist, each representing an execution strategy of the rule. In the sequel, we discuss possible answers to these questions based on our illustrative scenario. More generally, our core work is to provide the rule programmer with an abstract framework for reasoning about rules execution and an actual architectural framework for practically programming rules and their semantics.

The rest of the paper is organized as follows. Section 2 and 3 introduce our first contribution: the proposition of an ECA rule model suited for component-based systems made of a rule definition model (Section 2) together with a rule execution model (Section 3). Section 4 and 5 introduce our second contribution: the proposal of an architectural design that allows for the graceful integration of active rules into component based-systems (Section 4) and some elements about the actual implementation of rules as Fractal components (Section 5). Section 6 discusses related works. Section 7 concludes the article.

2. RULE DEFINITION MODEL

A definition (or knowledge) model specifies how the rules are represented and manipulated. This section describes the rule definition model we propose for component-based autonomic systems.

Event. An event is a happening of interest at a given point in time. It is characterized by an event type, i.e, an expression describing a class of significant occurrences of interest. In our framework, we consider the following event types: i) *applicative* - corresponds to inter-component interactions, ii) *structural* - represents modifications (reconfigurations) of the structure (topology) of the system, like adding or removing a component, or creating new bindings between components, iii) *system-level* - characterises events coming from the external or underlying environment or context of execution (e.g. JVM and OS events). In our illustrative scenario, we have an applicative event type: an operation (method) invocation on a component interface, and more precisely calls on the interface of the frontend component of our web server to request web pages.

Condition. Conditions are optional and express additional constraints on the state of the system that must be satisfied for the action part to be executed. For example, in our autonomic scenario, the condition predicate checks the number of available threads, These kinds of condition expressions (e.g. whether an attribute's value is bigger than a particular value or not) are simple boolean expressions built using logical operators. More complex expressions can be formed based on queries on the structure of the system as well as on its behaviour. A query that selects the various components linked to a particular one is one such example. In such a case, the condition is considered to be true when the query returns a non-empty result.

Action. The corrective (re)actions that the target system can be subjected to are expressed in the action part of the rule. The event and condition parts of the rule serve to analyse the symptoms affecting the system. In our scenario, a method call to the Request Receiver component triggers the rule. Its condition part evaluates if the number of free threads is below a limit. If yes, the action that increases twofold the size of the thread pool is performed. To rectify the anomalies, the action can range from simple parameterizations of component attributes, for example, an increase in the size of a cache or pool, to complex structural reconfiguration operations, which can include addition, removal or replacement of one or several components. Other types of actions can be envisaged, like external notifications, for example, an email or SMS notification to an administrator.

3. RULE EXECUTION MODEL

A rule execution model specifies the behavioural semantics of rules. This section introduces the design of our proposed execution model and discusses the main dimensions of this model on our illustrative scenario.

The entire execution of a single rule is comprised of the following three phases and various states:

1. Triggering and Event Processing Phase R(E): this phase begins with the notification of the event(s) that triggers ("wakes up") the rule. The notification is performed by the entity on which the event occurs. It consists in processing the event(s) based on the various rule execution parameters. The rule goes from the *triggerable* state to the *triggered* state.
2. Condition Evaluation Phase R(C): the second phase of the execution evaluates the condition expression. If the condition is satisfied then the rule transits from the *evaluable* state to *evaluated* state.
3. Action Execution Phase R(A): the last phase of the rule execution corresponds to the execution of the action part of the rule. It takes the rule from the *executable* state to the final state of *executed* state (generally confonded with the initial *triggerable* state), thus inducing a positive feedback change in the system behaviour.

It is worth mentioning that the condition of a triggered rule is not always evaluated immediately (hence the two separate states *triggered* and *evaluable*), and that a triggered rule with a satisfied condition is not always executed immediately (hence the two separate states *evaluated* and *executable*). When and how (e.g. which activity/thread) a rule is processed depends on the various dimensions of the rule execution model. Some of the most important ones are discussed later in the context of our illustrative scenario. Of course, when multiple rules are concerned, which is the case in real autonomic systems, an execution model also specifies when and how rules triggered simultaneously (by same or different events) (a.k.a. *multiple rules*)and rules triggered by other rules (a.k.a. *cascading rules*) are executed. This is handled by *rule execution strategies* (or *policies*) which basically specify the scheduling of rules (e.g. depth-first order, width-first order, flat order, by cycles in sequential or parallel settings). Due to space limitation, this article does not detail these aspects. The reader may refer to [4]. Prior to that and more fundamentally, if rules have an execution model of their own, it has to be stated that the introduction of active rules in a system (be it a database or a component-based system) has also a non negligible impact on the behaviour of that system. Indeed, there exists a dependency between the execution of the system and the execution of rules, for it is the former that triggers the latter and also the two executions are interwoven/intertwined together.

3.1 From active database systems to active component based systems

Active rule execution models in database systems have been extensively studied but cannot be directly applied to component-based systems. First, events that trigger a rule in an active DBMS are query (SQL) statements on a global data schema, so are the condition and action parts. But this it is not the case in component-based systems where we have a variety of events, condition predicates and actions (as defined in the rule knowledge model). In active database systems, all rule operations are performed on a single database, whereas in a component-based system, they may have to be performed on different components. Indeed, in a component-based system, situations of interest can happen on any component of the system. To gain a thorough understanding of the component and its execution environment, we might have to perform additional queries on it or on its neighbours and finally execute the corrective actions elsewhere. So, the distributed characteristic of component-based systems is one of the distinguishing factors.

Execution units and execution points in component-based execution models. Finally, besides the two differences we have just mentioned, a key difference between active database systems and active component-based systems is that execution models in active database systems are based on a central concept, that of *transaction*, which is (generally) inexistant in component-based systems. A transaction in database systems is a sequence of operations that constitutes a unit of concurrency and recovery thanks to the well-known "ACID properties" (*Atomicity, Consistency, Isolation and Durability*). Transaction is a core and foundational concept of active database systems because, thanks to transaction *demarcations* (*start, commit, abort/rollback*), they provide a natural and convenient execution unit for the execution of active rules. An execution unit specifies an interval (between two execution points in a sequential flow or basically between two points in time) during which events can be detected/notified to interested rules and rules can be evaluated and executed. Hence, an execution unit specifies the *granularity* of rules execution. On the one hand, component-based systems generally do not consider transactions. On the other hand, the behaviour of a component-based system generally refers to interaction through interfaces only, thanks to operation (methods in Java) invocation. Hence, we define the execution unit in component-based systems as delimited by the interval between the reception of an operation invocation on a server interface and the emission of a response onto a client interface. For method invocations on a component's functional interfaces (produces applicative events), and operations that modify the structure of the system (produces structural events), we may signal two events: *begin* and *end*. Other forms of events (e.g. system events) can be integrated in the model that by considering their begin and end events are merged (i.e. they both represent the same execution point or point in time).

3.2 Rule execution dimensions

Based on these hypotheses, we consider our approach for defining a rule execution model, similar to the one followed in active database systems[4], where it is defined as a set of *dimensions*, with each dimension being attributed a particular *value*. The differences/issues outlined above, have been addressed in the form of a flexible rule execution model for component-based systems, adapted from rule execution models defined in active DBMS. The sequel discusses rule execution dimensions in the context of the autonomic scenario introduced earlier.

Event Processing Mode. On every webpage request, the Comanche webserver requests the scheduler service for an execution thread. So, if an instance of the thread management rule is triggered for every call, then the system ressources would be spent unnecessary resulting in a lower performance. Ideally, a rule needs to be triggered once at the appropriate moment to rectify the situation. The event processing dimension addresses this issue, with the possibility of triggering a rule for several occurrences of the event type. A rule may handle either only one event at a time or a set of events. This is specified in the *event processing mode*, having an *instance-oriented* semantics for the former, and a *set-oriented* semantics for the latter. In other words, an instance-oriented semantics suggests that a rule will be triggered for every occurrence of a triggering event. Such a kind of event processing strategy is interesting whenever each event has to be treated individually, e.g. when an exception is raised, or on every attack or on every forced entry by a malicious user which requires preventive measures to be triggered in the form of a rule. If several rules are triggered with the same purpose, system resources are bound to get depleted, affecting the performance of the application. Therefore, this strategy is beneficial when a single execution of a rule is enough to resolve the anomaly in the system. We shall opt for a set-oriented value to the rule's event processing mode: as said earlier, a single execution of the rule is sufficient to retain the performance level of the system.

Coupling Modes. Once a rule is triggered, we have to determine when and how it will continue its execution for it should not affect the system's execution. If we consider our thread management rule, should our rule be executed before processing the request or after? Should it be done in the same execution thread or in a separate one? Several options exist for the above. This is taken care of by the *coupling mode* dimension characterized by the couple : < *execution mode, activity mode*>.

The *execution mode* specifies when the condition and action parts of a rule are evaluated and executed with respect to the execution of the triggering operation. The triggering operation is the method invocation on a component's interface which produces events that trigger a rule. Such a rule that is activated and is ready for execution, is called a triggered rule. The commonly supported execution modes include (cf. Figure 1):

- *immediate (or as soon as possible - ASAP):* the triggered rule is processed immediately - its condition is evaluated, if true, the action is executed without any pause.
- *defered:* the triggered rule is processed later, awaiting the end of the triggering operation. The rule is triggered when an event indicating the beginning of the triggering operation is received. But the condition evaluation and action execution of the rule are processed only on receiving an event indicating the end of the triggering operation.
- *delayed:* here, the condition is evaluated immediately after the rule has been triggered, but the action part is executed only on the completion of the triggering operation.

Rules dealing with security issues have generally a high priority, and may typically take the *immediate* value in order to execute instantly before the damage is done. *Defered* rules are typically used in situations in which the action has to be executed on the final state of the system. In this respect, immediate rules might embed pre-conditions, while defered rules might embed post-conditions. *Delayed* rules are intermediary with a condition evaluated at the beginning of the triggering operation (i.e. before the state of the system might be changed by other rules for instance) and the action executed at its end. Our rule concerns the performance of the system. It permits the system to take preventive measures in order to maintain its performance levels. So, it is not so crucial, and can be executed once the initial task is completed, a defered value will be attributed to our rule's execution mode.

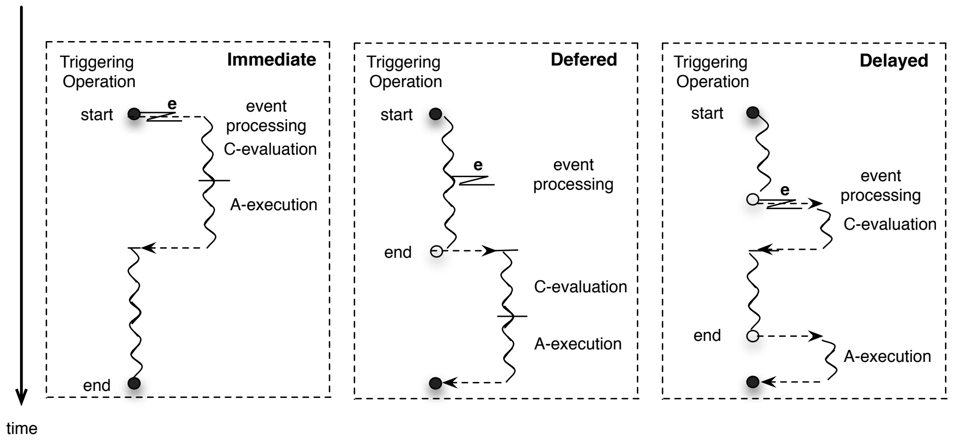

Figure 1: Execution Mode

An execution thread represents a sequential flow of control. The *activity mode* dimension determines whether the triggered rule is executed in the *same* thread as the triggering operation or in a *separate* thread. It is recommended to create a separate flow of control for aspects related to the administration of the system, e.g, logging. Thus, separating the system under control and its administration, so that its normal execution is not too disturbed. On the other hand, some administration scenarios might require the system to be paused, to enable some modification, and to resume later its execution. Depending upon the scenarios, the best method of executing the rule has to be judged and employed. For our thread management rule, we shall follow the conventional choice of executing the rule in a separate thread because there is no risk due to concurrency on the usage of threads in the pool since the rule creates only new threads.

Focus on interactions between a couple of dimensions. Since there exist some dependancies and intricacies among these dimensions, the individual semantics of the dimensions might slightly differ which one could guess at a first glance when considered as a whole. To illustrate such intricacies, we now focus on the interactions between event processing and execution modes.

Prior to presenting the possible combinations for the couple $<$ event processing mode, execution mode $>$ (including the ones that match the above choice in our example) , the following notations are employed in the sequel and in Figure 2:

- b_n: the event corresponding to the beginning of the n_{th} method invocation
- e_n: the event corresponding to the end of the n_{th} operation (method) invocation
- $R_n(E)$: depicts the event processing phase of the n_{th} occurrence of the considered triggered rule R
- $R_n(C)$: depicts the condition evaluation phase of the n_{th} occurrence of the considered triggered rule R
- $R_n(A)$: depicts the action execution phase of the n_{th} occurrence of the considered triggered rule

The events b_n and e_n are representative of the n_{th} occurrence of a triggering event type.

1. $<$ Immediate, Instance $>$: on every event b_n, a rule is triggered and processed immediately in its entirety. All e_n events are ignored.

2. $<$ Immediate, Set $>$: on the event indicating the beginning of the first triggering operation, i.e, on b_1, a rule is triggered and continues processing till its completion. All b_is that occur till the triggered rule completes evaluation, are consumed by the rule, i.e, the triggering operations are taken into consideration by the rule in execution. The next b_i when the rule is in $R_n(A)$ or after, triggers the next rule. Similarly, this rule also consumes all b_i that occurs till it completes evaluation. Two similar rules can coexist when one is in the action execution phase and the other begins execution.

3. $<$ Defered, Instance $>$: on every b_n, a rule is triggered, it processes the event and waits for a complementary e_n event to continue evaluating - $R_n(C)$ and complete execution - $R_n(A)$.

4. $<$ Defered, Set $>$: all b_is trigger each a rule. The rules complete the $R_n(E)$ phase, and wait for an end event e_i. When a e_i event is received, all rules triggered after R_i are discarded and their corresponding triggering operations consumed by the rule R_i. All events e_j where j $>$ i received later, are discarded. For any e_k where k $<$ i, the corresponding rule R_k resumes execution, and consumes all triggering operations that have triggered rules R_j where k $<$ j $<$ i.

5. $<$ Delayed, Instance $>$: on every b_n, a rule is triggered, processes its event part - $R_n(E)$, evaluates its condition part $R_n(C)$ and waits for the e_n event to execute the last part of the rule - $R_n(A)$.

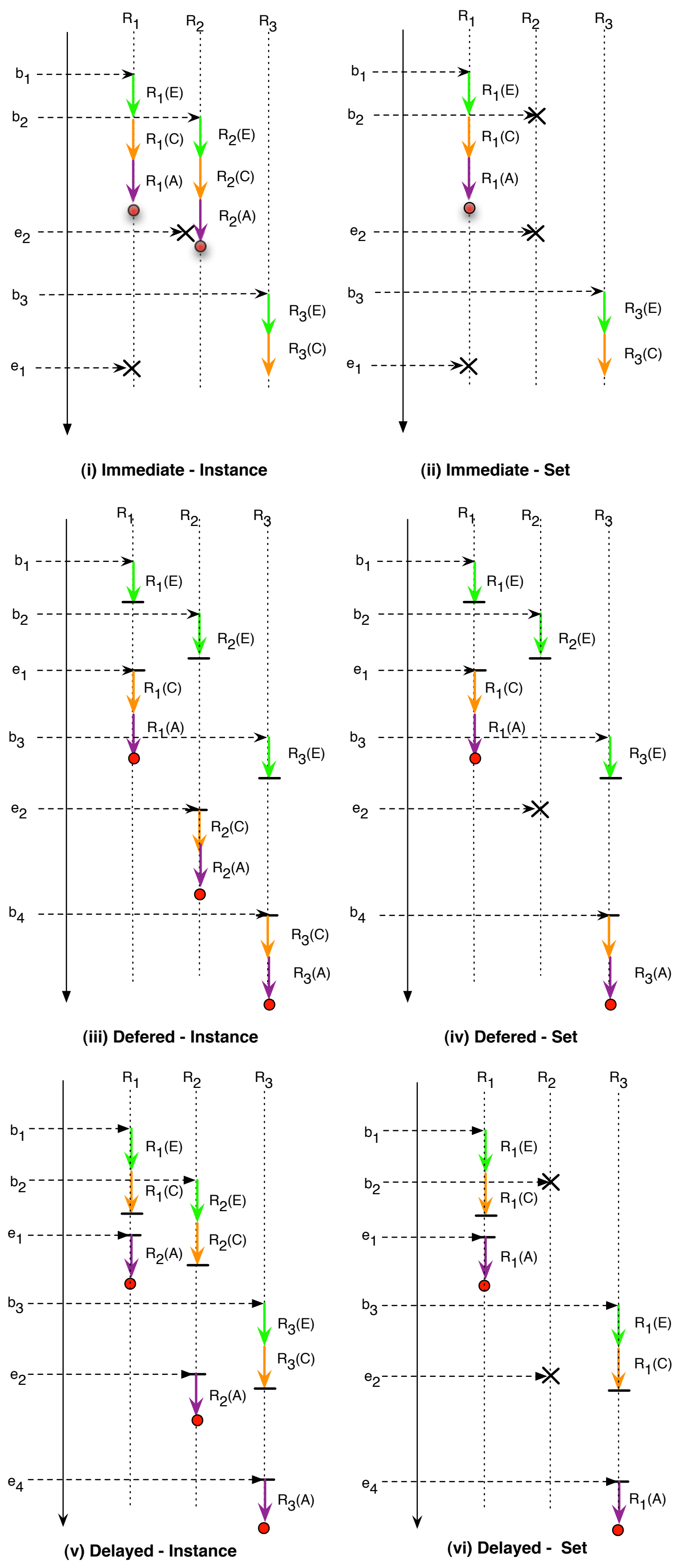

Figure 2: Execution Models

6. < Delayed, Set > : on b_1, i.e, event indicating the start of the triggering operation, a rule R_1 is triggered and stops after condition evaluation. All b_is that occur in this period, are consumed by the R_1. Once the event e_1 is received, rule R_1 resumes execution and completes its last phase $R_1(A)$. All e_is, whose complementary b_is have been consumed by the rule R_1. The next b_i that occurs, a rule is triggered and the same strategy is followed.

Conclusion. To sum up on our example, the rule execution dimensions of the thread management rule take the following values: *Event Processing Mode : set*, *Execution Mode : defered*, *Activity Mode : separate*. In other words, the rule is executed once for a set of events, after the triggering operation's execution returns, in a separate execution thread. In our simple scenario with only one rule and the three dimensions (including one with 3 values) that we consider in this article, we already get 12 combinations of values, i.e. potentially twelve different ways of executing the rule. Note that this does not have a big impact with our sample basic rule but think of a real system with many rules. To the rule programmer facing the complexity of rules semantics (execution model), we propose a architectural framework in which rule semantics is explicitely programmed/embodied into software components.

4. ARCHITECTURAL INTEGRATION OF ECA RULES IN COMPONENT-BASED SYSTEMS

As advocated by IBM in its autonomic computing manifesto [9], a *supervision loop* (or known as *control loop* in control theory terminology [11]) has to be realized in order to provide autonomic behaviour to a target system. On similar lines, we define an autonomic system as composed of an autonomic infrastructure superimposed on a target system, where the autonomic insfrastructure is responsible for implementing the control loop. At the heart of the control loop are reaction mechanisms that, on analysis of the events of interest, determine the action operations needed to achieve the objectives. Our reaction mechanism is formed of active rules, whose structure and execution have been explained thoroughly in the previous section. The thread management rule, defined in the introduction section, is one such active rule. This section details how such rules are architecturally integrated with an underlying target system.

If it is legitimate to work towards adding autonomic behaviour a posteriori to any system and even more to tackle explicitly existing legacy systems, we believe it is likely more advantageous to build explicitly, a priori, the system in a "certain way" to be able to make them autonomous in a flexible and generic way. This "certain way" is the component-based approach - and more precisely the Fractal component model [2], which has, according to us, interesting properties for the realization of autonomic systems. It has been mentioned before that our webserver - Comanche - is implemented as Fractal components.

4.1 Canonical Autonomic Architecture

An architecture for autonomic computing[14] must accomplish some fundamental goals, outlined in IBM's autonomic computing manifesto[9]:

1. It should possess knowledge about itself and about its execution environment in order to be able to detect modifications taking place externally in its environment, or in its behaviour to subsequently undertake corrective actions. It must describe how to compose these components so that the components can cooperate toward the goals of system-wide self-management.

2. It should be adaptable, i.e., its construction should be based on a structuring model which can isolate its constituting elements, and subject them to adaptations - and on operational techniques to actually perform these adaptations (interception, programs transformation, etc.). It should be able to dynamically adapt or reconfigure itself to varying and unpredictable environments without any explicit user intervention.

These key features are present in the Fractal component model, and we believe Fractal/Julia (its implementation in Java) is a suitable substrate framework for autonomic systems development as illustrated in the sequel.

Fractal Component Model. The Fractal[2] initiative aims at supporting component-based development, deployment and management (monitoring and dynamic reconfiguration) of complex software systems, including in particular operating systems and middleware. It includes several extensions coming from research works, for management (e.g. Fractal JMX), security, transactions support, etc. Fractal is also used for developing several middlewares such as Speedo - a Java Data Object implementation, CLIF - a load injection framework, etc[2]. The Fractal component model relies on some classical concepts in CBSE: *components* are runtime entities that conforms to the model, *interfaces* are the only interaction points between components that express dependencies between components in terms of *required/client* and *provided/server* interfaces, *bindings* are communication channels between component interfaces that can be primitive, i.e. local to an address space or composite, i.e. made of components and bindings for distribution or security purposes. Fractal also exhibits more original concepts. A component is the composition of a *membrane* and a *content*. The membrane exercices an *arbitrary reflexive control* over its content (including interception of messages, modification of message parameters, etc.). A membrane is composed of a set of *controllers* that may or may not export control interfaces accessible from outside the considered component. For runtime information on the component system, the control interfaces provide with (meta) information about the its structure and also means to manipulate this structure. The model is *recursive (hierarchical) with sharing* at arbitrary levels. The recursion stops with base components that have an empty content. Base components encapsulate entities in an underlying programming language. A component can be shared by multiple enclosing components. Finally, the model is programming *language independent* and *open*: everything (e.g. controllers, type system) is optional and extensible[3] in the model, which only defines some "standard" API for controlling bindings between components, the hierarchical structure of a component system or the components life-cycle (e.g. start, stop).

[2]CLIF, Speedo and other middleware engineered with Fractal are available in open source at http://www.objectweb.org/.

The Julia Implementation. Julia is an execution support for Fractal components written in Java. It is a full-fledged implementation of Fractal that supports the highest conformance level. More fundamentally, Julia is a software framework dedicated to components membrane programming. It is a small run-time library together with bytecode generators that relies on an AOP-like mechanism based on mixins and interceptors. A component membrane in Julia is basically a set of controller and interceptor objects. A mixin mechanism based on lexicographical conventions is used to compose controller classes. Julia comes with a library of mixins and interceptor classes, the component programmer can compose and/or extend. It is worth mentioning that Julia's membranes are particularly suited to insert sensors for observing and actuators for controlling components.

(Re)Configuration Languages. For the configuration and deployment of a Fractal-based system, an Architecture Description Language (ADL), known as Fractal ADL, is used to describe the system architecture. It is XML-based and strongly typed. It describes the interfaces of components (names and signatures), the subcomponents, the bindings between the various components, the initial values of component properties and the implementation of primitive components(e.g., the name of a Java class). All static information on a component is provided by the Fractal ADL. FScript is a scripting language used to describe architectural reconfigurations of Fractal components. FScript includes a special notation called FPath (loosely inspired by XPath) to query, i.e. navigate and select elements from Fractal architectures (components, interfaces...) according to some properties (e.g. which components are connected to this particular component? how many components are bound to this particular component?). FPath is used inside FScript to select the elements to reconfigure, but can be used by itself as a query language for Fractal.

Most of our approach relies on the Fractal component model. Of course, in realistic industrial settings, we cannot assume a whole distributed system to be Fractal-based; but we argue that those non-Fractal parts (legacy components) can be wrapped into Fractal components, and extended with autonomic behaviours. In the same line of thought, we believe that it would be very advantageous to carry out the development of the autonomic infrastructure itself in the form of Fractal components so as to consider the autonomic management of the autonomic infrastructure itself. (Our work

[3]This openness leads to the need for conformance levels and conformance test suites so as to compare distinct implementations of the model.

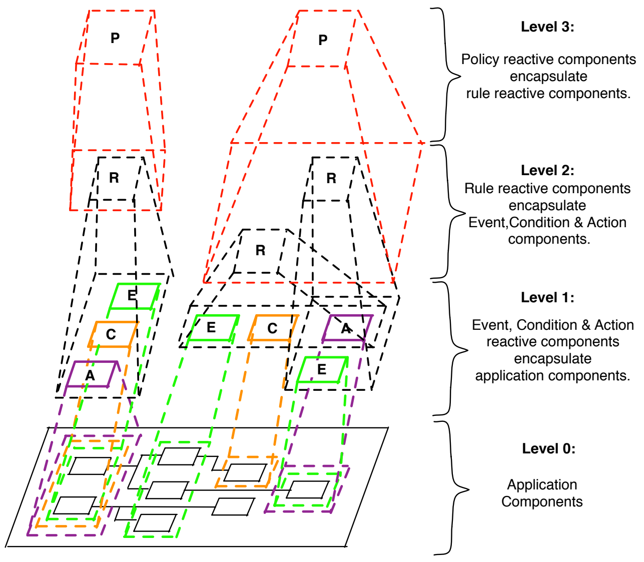
P
P
R
R
R
E
C
A
E
C
A
E
Level 3:
Policy reactive components encapsulate rule reactive components.
Level 2:
Rule reactive components encapsulate Event,Condition & Action components.
Level 1:
Event, Condition & Action reactive components encapsulate application components.
Level 0:
Application Components

Figure 3: Architectural Vision

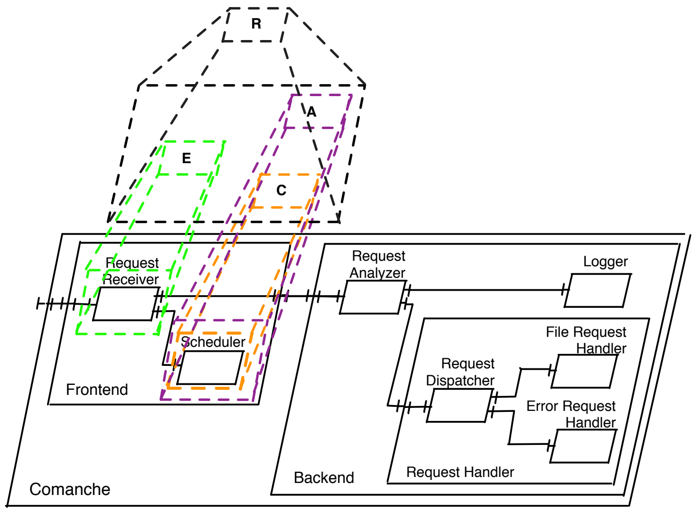

Figure 4: Thread Management Rule

can be considered as the first steps in this direction.)

4.2 Reactive part of the Canonical Autonomic Architecture

The architecture of the autonomic infrastructure is inspired from the fundamental management notion of *domain*[13], which consists in grouping the components on which the various reactive operations can be carried out. More formally, a domain is:

- *a unit of composition* to enable physical or logical partitioning of the application components, and
- *a unit of control* to define the type of control that needs to be carried out on these components.

The similarities between a Fractal component and the concept of domain suggest that a domain can be aptly modelised as a Fractal component. To incorporate reactive behaviour, several types of domains have been defined, each with a particular type of control unit applied onto its composition unit. They are each represented by a Fractal component, known as a *reactive* component. The autonomic infrastructure is formed by these reactive components. The various reactive components with their specific functionalities are listed below and illustrated by Figure 3.

- An *Event* (E) reactive component encapsulates application components where events of interest need to be detected. The membrane of this reactive component is responsible for identifying the application components that would belong to the content, instrumenting them appropriately. Further, on the occurrence of events of interest, they are notified to the appropriate controller inside the membrane of the component which processes them as defined in the rule execution model.
- A *Condition* (C) reactive component contains application components that represent the scope the queries that are to be evaluated. The functionnalities of its membrane include identifying the application components that would be in its jurisdiction, and evaluating queries on them.
- An *Action* (A) reactive component encapsulates application components, on which actions are executed. The type of control enforced by the Action component's membrane involves identifying its content's constituents, and executing some corrective operations on request.
- A *Rule* (R) reactive component coordinates the processing of a rule. It contains (exactly) one instance of the 3 above reactive components, i.e, Event, Condition (optional) and Action. The control applied by its membrane is the execution coordination of these reactive components. It is responsible for the execution of the rule embodying on a particular rule execution model.
- A *Policy* (P) reactive component's sole purpose is to coordinate the execution of the rules based on an execution strategy for a set of rules. The content part of the policy reactive component contains rule components, and its membrane controls the rights to their execution. Only, on explicit notification by the policy membrane, can the rules, once triggered, continue processing.

Figure 3 shows the relationships between the various reactive components. The architecture employs key features of the Fractal Component Model [2], notably: the containment relationship - which can be found in components in the top three levels, where each component has components from the lower level as sub-components, e.g, the Policy component at level 3 contains the Rule components of level 2; and overlapping of reactive domains, thanks to the sharing property, e.g, an application component can belong to several reactive components. Figure 4 represents the implementation of our thread management rule. The Event reactive component encapsulates the request receiver component, because

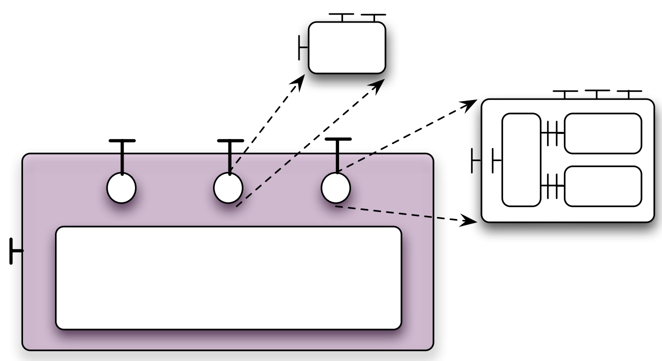

Figure 5: Extended Fractal Component

the event of interest for the rule occurs on it. The condition and action parts encapsulate the scheduler component. For their respective operations, that of verifying whether the thread pool is empty and increasing the size of the thread pool occurs on the same component. Finally, we have the Rule composite that encapsulates the reactive components (Event, Condition & Action) to coordinate their execution. The Policy reactive component is inexistant in the figure since, due to space limitations, only a single rule has been taken as an example to illustrate the framework.

5. AUTONOMIC INFRASTRUCTURE

This section presents the extensions in the prototypal implementation of our proposed framework, that offers the flexibility feature of our autonomic infrastructure.

Each of the reactive components presented above encapsulate either application components or other reactive components, where structurally both these types of components are similar. But the controller unit of each of these reactive components differs from one another because they implement different dimensions of the rule execution model. Therefore, the generic structure of a reactive component is a standard Fractal component - a composite component to be precise, with a flexible membrane formed by a set of newly defined and existing controllers (cf. Figure 5). The Fractal specification contains several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features. Likewise, additional controllers, not defined in the Fractal specification, can also be defined and incorporated in the membrane of a component. This permits the reactive components to have a membrane adapted to its respective rule execution dimensions. The membrane of a reactive component is composed of the following types of controllers : i) standard Fractal controllers as defined in the Fractal specification, ii) standard Fractal controllers represented as classical Fractal components with extended operations and finally iii) new controllers represented as Fractal components.

For instance, the membrane of the Event reactive component is composed of: i) an extended *attribute controller* to specify the parameters of the rule execution dimensions, ii) a *content controller* to add/remove the application components and iii) a *event processing controller* to process the events of interest and notify its enclosing rule component. The event processing controller is a newly defined controller represented as a Fractal component. It is a composite component with corresponding sub-components for the following dimensions: execution mode, activity mode and event processing mode. However, at the time our framework was developed, the set of controllers that constitute the membrane of a component could not be dynamically modified in the Julia implementation, nor could the functionalities of the existing controllers be modified. We have thus extended the Julia implementation to support these necessary features.

6. RELATED WORKS

Decision-making/reaction mechanisms form the core of an autonomic control loop, several systems use rules that specify conditions to be monitored and operations that should be executed when certain conditions are detected. *Production rules* (or *deductive rules*) have the following format - "IF *condition expression* THEN *action list*". These rules can also been extended, as in the DIOS++ framework[10], where an "ELSE" part as been added at the end - "IF *condition expression* THEN *action list* ELSE *action list*". ACEEL [3] uses *adaptation rules* which are couplets of the form "*OnEvent* : *Action*" with the first part defining the triggering event and the latter describing what actions to be performed. Inspired by triggers, we use active rules, which can be considered as a combinaison of the above two broad types of rules: "ON *event expression* IF *condition expression* THEN *action list*". Beyond these syntaxical differences, the main differences between production (or adaptation) rules and active rules concen their execution model. The execution model of production rules is based on the Rete algorithm: events are seen as facts which are added to the knowledge base; rules infer new facts from these facts ; the process stop when a fixed point is reached (no new facts can be inferred). Production or adaptation rule execution models are thus fixed and invariant. By contrast, active rules models are much more powerful and flexible. Also active rules models encompass the connection between the target system and the reaction mechanism while production rules systems do not (inference by production rules is disconnected from the actual execution of the target system).

For incorporating these active rules, the approach followed in SAFRAN[5], K-Components[7] consists in enhancing the computational component model with rule abstractions, where all rules concerning a particular component are injected into to it. Another approach, followed in Autonomia[6] or Automate[1], consists in implementing an autonomic computing infrastructure that acts as a control layer superimposed on the application, that provides the application as well as its individual components with the basic autonomic services to make it autonomic. In our approach, rules are not adhoc features injected into components but are themselves first-class components which can be manipulated as such. Thanks to the domain concept, the architectural connection between the application components and the rules are through containment relationships (hence a rule is not tied to a single component). Enabling thus, easy modification of the rule constituents (Event, Condition & Action).

In summary, to our knowledge, several works have tackled the architectural issues involved in the implementation of a control loop for autonomic features but none make such an explicit and extensive use of component programming for implementing the autonomic features themselves as in our proposition. As a consequence, these approaches often result in ad-hoc and not flexible management of the autonomic features. In most approaches that consider rules of a sort or another (deductive, active, etc.) as the core mechanism for autonomic features, rules execution issues have not been addressed in depth. Their execution strategy/methodology have been taken for granted, and many issues have been left under specified and ambiguous (we only find a brief mention of rule execution model in SAFRAN).

7. CONCLUSION

This article focuses on the architecture and behaviour of autonomic control loops. It proposes to use active rules as a decision-making mechanism, for which we have proposed i) a rule model, which is composed of a rule definition model and a rule execution model, to provide a clear semantics for the integration of rules, and ii) a flexible architecture that permits to dynamically add/delete new rules, to modify the rule definition model as well as the rule execution model of rules. Our rule execution model comprises of a set of dimensions, which we claim is not fully comprehensive, other dimensions can be envisaged. But we do claim that our generic architecture can incorporate new dimensions not yet identified. Further, our autonomic infrastructure takes into consideration the evolution of the target system. We would like to add that our work being positioned on components, i.e, our autonomic infrastructure as well as the underlying target system being component-based, has permitted us to benefit from CBSE properties.

The proposed autonomic infrastructure was developed in a Java implementation of the Fractal component model. The dimensions outlined in the article have been implemented, and experimented on the Fractal-based Comanche webserver. Some other execution dimensions, e.g. those related to the execution of multiple rules, which have not been discussed here due to space constraints have also been implemented.

Several future research directions are envisaged. In order to assess the validity of our proposition, we wish to apply it on more realistic applications. This would eventually permit us to determine the set of execution dimensions that are most relevant for component-based systems. Further, to enrich our proposition, we foresee a formalism for the definition of active rules that would typically be an extension of the Fractal ADL. This extension should be quite straitforward, for Fractal ADL is actually modular and extensible. On a longer term, we shall study the problem of interference between the behaviour of a control loop (i.e. the rules in our case) and the target system and the stability of the global system (target system and rules).

Acknowledgments. This work is partially supported by the French RNTL Selfware project and the European IST Selfman project. The authors thank A. Lefebvre and B. Dillenseger for their careful reading and comments.

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Autonomic Computing Workshop*, pages 48–57, 2003.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[3] D. Chefrour. Developing component based adaptive applications in mobile environments. In *SAC '05: Proc of the 2005 ACM symposium on Applied computing*, pages 1146–1150, New York, NY, USA, 2005. ACM Press.

[4] T. Coupaye and C. Collet. Detailed sketch of a parametric execution model for active database systems. Technical report, LSR - IMAG Laboratory, University of Grenoble. France, 1997.

[5] P.-C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *5th International Symposium on Software Composition (SC'06)*, Lecture Notes in Computer Science, Vienna, Austria, march 2006. Springer-Verlag.

[6] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In *Proc of the 2003 IEEE Int'l Conf on Performance, Computing, and Communications*, pages 61–68, 2003.

[7] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of the Third Int'l Conf on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[8] S. Gatziu, A. Koschel, G. von Bültzingsloewen, and H. Fritschi. Unbundling active functionality. *SIGMOD Record*, 27(1), Mar. 1998.

[9] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM Corporations, October 2001.

[10] H. Liu and M. Parashar. Dios++: A framework for rule-based autonomic management of distributed scientific applications. In *Euro-Par*, pages 66–73, 2003.

[11] M. Kokar and K. Baclawski and Y. Eracar. Control Theory Based Foundations of Self Controlling Software. *IEEE Intelligent Systems*, 14(3):37–45, 1999.

[12] N. W. Paton, F. Schneider, and D. Gries, editors. *Active Rules in Database Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[13] M. Sloman and K. Twidle. Domains: a framework for structuring management policy. *Network and distributed systems management*, pages 433–453, 1994.

[14] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An Architectural Approach to Autonomic Computing. In *Int'l Conf in Autonomic Computing*, pages 2–9, New York, NY, 2004.

E Middleware for Building Internet-scale, Dynamic, Distributed Applications

Middleware for Building Internet-scale, Dynamic, Distributed Applications

Cosmin Arad, Roberto Roverso, Ali Ghodsi, Seif Haridi
{cosmin, roberto, ali, seif}@sics.se

June 25, 2007

Abstract

This report presents the design and implementation of a middleware for building large-scale, dynamic, and self-organizing distributed applications for the Internet. First, we identify the challenges that are faced when building this type of applications and the constraints imposed on the middleware that is to support them. We derive a set of essential services that are to be provided by our middleware in order to facilitate the development of distributed applications. These services include *scalable communication*, *failure detection*, *name-based overlay routing*, *group communication* and a *distributed hash table* abstraction. We present the *event-based component-oriented* architecture of the system, discussing the design choices that we made in order to meet the aforementioned challenges and constraints while providing the essential services for distributed applications. We describe in detail the event scheduling mechanism, the communication and failure detection, as well as the interface to applications and other miscellaneous services.

1 Introduction

Internet-scale distributed applications and services such as wide-area storage systems [12, 6, 15], content distribution networks [5, 9], media streaming systems [17, 14, 3] or peer-to-peer and GRID computing and resource sharing systems [1] have motivated considerable advancements in the research on large-scale distributed systems in the last few years. Typically, cooperating computer nodes that form these distributed systems are organized in an overlay network operating over the Internet.

Building real implementations of this type of systems poses a common set of challenges. First, given the nature of the provided services, these applications should accommodate a large number of users and participating machines. Thus, their implementation should be *scalable* in terms of the size of the network and the communication, storage, and computational load they are subjected to.

Second, these applications should operate in an environment of dynamic membership, where nodes are constantly joining, leaving the network or failing. Hence, the applications need to be *fault-tolerant*. Nodes should accurately detect the failure of neighboring nodes and act accordingly. Moreover given the constraints imposed by a possible structure of the overlay network, these applications should accommodate dynamic peer connectivity, being able to continuously maintain certain connections and garbage collect others.

In order to facilitate the development and quick prototyping of new large-scale, dynamic, and self-organizing distributed applications, we decided to build a middleware library that provides basic reusable services for this type of applications and encapsulates solutions to the aforementioned set of challenges.

DKS, the middleware that we have implemented provides the following services: a Distributed Hash Table (DHT) indexing service, that allows for storing and retrieving key-value pairs, an overlay network allowing for reliable name-based routing of messages, and group communication services. In addition, we provide lower level essential services like failure detection, timers, efficient object-based message marshalling, and web-based testing support.

Implementing a middleware library brings certain constraints. Large-scale systems imply a certain degree of heterogeneity in the performance and capacity of participating machines. In order to accommodate this heterogeneity the library should be as lightweight as possible, thus it should have a low memory footprint and should use a number of threads that accommodates the number of processing cores available on the machine where the middleware is executed. Moreover, it should be extensible and easily integrated with applications.

In the following sections we discuss the services provided by DKS and then we describe the DKS system architecture.

2 Middleware Services

We have derived a set of middleware services that we believe are essential to any large-scale distributed application, and which, if available, facilitate the quick implementation and deployment of new distributed algorithms without reinventing the wheel.

First of all, application nodes need a reliable and efficient communication infrastructure for point-to-point communication and name-based routing. Second, they need to detect the failure of peer nodes in a timely manner and take fail-over measures. Extending these basic abstractions, we believe many applications would benefit from the use of a distributed index provided by a DHT service or from a group communication abstraction. Last, but not least, DKS provides efficient message serialization, timers handling and built-in support for application testing. We now look at each of these services in turn.

2.1 Reliable Name-based Communication and Routing

The application nodes running the DKS middleware form a Structured Overlay Network (SON). Each node has a reference comprised of its overlay network identifier and the underlay network address, namely its IP address and TCP port number, that other nodes can use to open new connections to it.

DKS provides both point-to-point message passing and message routing through the overlay network. It hides the connection management from the application. Messages can be sent to overlay network identifiers, the sender not having to know the underlay network address of the receiver. For point-to-point communication, new temporary connections are established automatically if needed. Endpoints of each connection negotiate whether any of them needs the connection to be permanent or temporary. Temporary connections are automatically garbage collected and closed if not used for a certain period of time. Permanent connections are established between overlay network neighbors, for instance, but the application can chose to make a connection permanent should it be used for a longer time, to avoid connection establishment trashing.

The overlay network topology is induced by the Distributed k-ary System [10] DHT. The topology is maintained automatically and it is used for name-based routing of messages. The name in this case is the overlay network identifier of the destination node.

2.2 Failure Detection

In order to be able to tolerate failures, applications need to detect them first. A node failure detection service is thus crucial for a distributed application. Due to the possibility of network congestion and message loss in the Internet, no bound on transmission delay can be guaranteed thus it is impossible to implement a strongly accurate [4, 11] failure detector using predefined message acknowledgment timeouts. Therefore, DKS provides an *eventually perfect* failure detector, which adapts its timeouts, hence its accuracy, to the variation of network latency, for each connection, thus for each neighboring node in part. At times it can falsely suspect alive nodes to have failed, due to temporary increased network latency, but eventually it adapts and resumes accurate failure detection.

2.3 Distributed Hash Tables

Distributed Hash Tables (DHTs) are an essential component of robust large-scale distributed systems. They provide a directory/index service in the form of a hash table abstraction, which distributed applications can use to reliably store various kind of meta-data. Data items in the DHT are replicated to keep them available as nodes join and leave the system.

The basic DHT operations are storing <key, value> pairs and retrieving the values associated to a key. DKS also provides bulk operations [10], whereby the storage or retrieval of a set of items is optimized in terms of message complexity.

DKS provides multiple DHT tables, with different characteristics such as replication degree or worst-case routing complexity. We are currently working on providing a transactional database abstraction on top of the DHT.

2.4 Group Communication

Exploiting the structure of the overlay network, DKS provides an efficient overlay broadcast service as well as pseudo-reliable version of it [10]. Broadcast messages reach all nodes in the overlay network in a number of communication steps that is logarithmic in the size of the network, with no redundancy. A broadcast with feedback operation is also provided. This allows any node to aggregate global information from every other node in the system.

2.5 Other Services

DKS provides testing support for overlying applications by means of a built-in web-server instance in each application node. Applications can expose internal state though dynamic web pages published on the server. From a central testing node, a script can automatically web-browse application nodes and assert the validity of their published internal state.

Other services provided by the DKS middleware are efficient object-based message serialization and timers management. Messages are serialized directly to native operating system buffers, in order to minimize byte copying and the serialization service provided by Java is avoided in order to decrease message size. DKS allows overlying applications to register and cancel timers and triggers notifications on timers' expiration.

3 System Architecture

For reasons of modularity, readability and easy maintainability the DKS middleware is structured into *components*. Each component implements one service that it provides to other components through an *event-based* interface. In general, components are event-driven but there are some exceptions. For instance, components that deal with I/O operations or timers are control oriented and have their own thread of control.

Event-driven components are implemented as Java objects. They are comprised of some local state variables and a set of event handlers, which are ordinary Java methods. Each event handler handles events of one type. Events are ordinary Java objects and event types are Java types (classes). An event handler is executed whenever an event of the corresponding type is triggered. Event handlers are executed by the worker threads of a *thread pool* of adjustable size. While being executed, event handlers might trigger other events.

Triggering and execution of events relies on a *publish-subscribe* mechanism. Components *subscribe* for all the event types that they can handle. Whenever a new event is triggered, it is *published* for scheduling and when scheduled, the corresponding event handlers or all components that had subscribed for that event type are executed.

An event subscription contains a reference to the subscriber component instance, a reference to the event handler method, and the event type for which the subscription is made. All event subscriptions are stored in a hashtable indexed by event type. In fact, a set of subscriptions is associated to an event type as there can be more than one component subscribing for the same event type. The event subscription table is depicted in Figure 1.

3.1 Event Scheduling

When an event is triggered a new event instance is created and placed on an event queue. The event queue is a priority queue and is used for prioritization of events. Events can have one of three priorities: low, medium, or high. By convention timer expiration events are given high priority, middleware events are given medium priority, and application events are given low priority. In general, high priority events are scheduled before medium and low priority ones and medium priority events are scheduled before low priority ones. However,

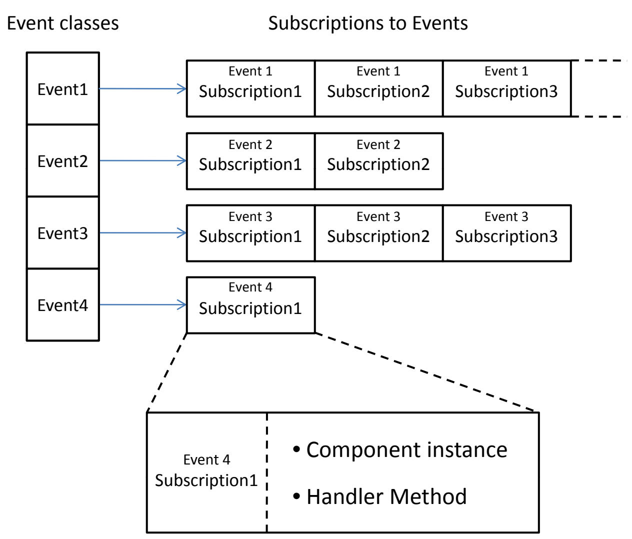

Figure 1: Subscription Table

to avoid starvation of low priority events we implement the following fairness mechanism: not more than f events are consecutively scheduled from a higher priority queue if there exist events in lower priority queues. f is a fairness parameter.

When an event is dequeued for scheduling, its type is looked up in the subscription table and all subscriptions are retrieved. For each subscription in part a *work* item is created and submitted for execution to the thread pool. A work item is a unit of work that can be executed by a worker thread in the thread pool. It consists of the event instance that needs to be handled and references to the component instance and handler method that need to be executed for handling the event. A worker thread that processes a work item will invoke the handler method on the specified component instance passing it the event instance as an argument.

While invoking an event handler method on some component instance, a worker thread locks that particular component instance. This enforces that one component instance executes only one event handler at a time so the component writer does not have to deal with concurrency. We can say that event handlers execute atomically with respect to each other, or that components are concurrency-safe.

The event scheduling is summarized in Algorithm 1.

3.1.1 Event Consumers

Components subscribe to events by type. As a result, a component that subscribes to one event type will handle all events of that type. Some components need to exchange messages with their peer components in other application nodes. Message sending and receiving is handled by a communication component. Whenever the communication component receives a message, it triggers

Algorithm 1 Event scheduling

```
procedure TRIGGER($e$)                              ▷ Called to trigger event $e$
    $eventQueue$.enqueue($e$);
    schedule();
end procedure

procedure SCHEDULE()
    $e$ := $eventQueue$.dequeue();
    if $e \neq$ nil then
        $subscriptions$ := $subscriptionTable$.get($e.type$);
        for all $sub$ in $subscriptions$ do
            $w$ := makeWork($sub$);
            $workerPool$.executeWork($w$);
        end for
    end if
end procedure

procedure EXECUTEWORK($w$)                          ▷ executed by a worker thread
    lock($w.component$);
    ($w.component$).($w.handler$)($w.event$);
    unlock($w.component$);
end procedure
```

a message received event. If all components that handle messages subscribed to this event, many of them would only handle it to find out that it contains a message they are not interested in. Therefore, to avoid this event trashing, we introduce the notion of event consumers.

Certain events, like the ones encapsulating received messages, may have associated with then, a *consumer*. A consumer is a pair containing the component designated to handle the event, together with its event handler method. If an event has an associated list of consumers, these will be scheduled to handle the event, together with other subscribers for the event type.

Currently, this mechanism is only used for events encapsulating messages. Components handling messages register as consumers by specifying a message type and a message handler method. Consumer registrations are kept in a hashtable indexed by message type. Whenever a message is received, based on its type, the list of consumers is retrieved from the consumer registry and all of them are scheduled to handle the message.

The consumers mechanism is a mechanism for message scheduling. Like events, messages have types, and different components handle different types of messages. Registering as a message consumer is the analogue of subscribing for an event type. The alternative approach to message scheduling would be for every message to be an event and use the event scheduling mechanism.

3.1.2 Component Mutual Exclusion

There exist situations where multiple components need to access some shared state. Typically, the shared state resides in one component and needs to be accessed by other components. As we execute components concurrently in the thread pool, we may introduce race conditions on the shared state. To avoid such race conditions, we want to prevent executing a component event handler that is accessing state of a running component and delay its execution until the conflicting running component has finished executing its handler.

When subscribing for an event type, a component A registers an event handler. At the same time it has to state what other component's (say B) state that handler accesses. We assume that all handlers of component B access the state that the handler of component A accesses. Thus, the handler of component A cannot run concurrently with any handler of component B. We say that the handler of component A depends on the handlers of component B and vice-versa. All such dependencies are stored in a dependency table. An example dependency table is depicted in Figure 2. For each event handler we have a dependency set consisting of all the handlers the respective event depends on. These dependency sets are created both ways at handler subscription time.

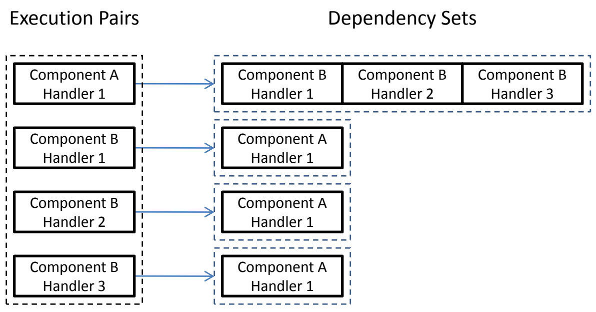

Figure 2: Dependency Table for components A and B

The scheduler maintains a set of running handlers. Whenever a new event handler is to be executed, its dependency set is intersected with the set of running handlers and the new event handler is executed only if this intersection is empty. Otherwise, the coresponding work item is placed in a waiting set. Whenever one of the running handlers finishes executing, the waiting set is inspected for handlers that are now ready to execute. The resulting scheduling algorithm is depicted in Algorithm 2. The `schedule()` funtion is a critical section and not two threads can call `schedule()` concurrently.

3.1.3 Hooks

Certain algorithms implemented by some components have special state transitions that may be of interest to other components. We provide a *hooking* mechanism through which these state-transitions can be notified to interested components.

Instead of breaking the atomicity of the event handlers implementing some protocol that relies on it, we allow other components to hook calls into the control flow of these event handlers. Thus, we label certain points in the flow of the protocol and we call them hooks. To these hooks other components can attach hook handlers. When the protocol execution reaches one of the hooks, it executes all the hook handles attached to it.

Algorithm 2 Event scheduling with component mutual exclusion

```
 1: procedure SCHEDULE()
 2:     for all $w$ in $waitingSet$ do
 3:         concurrentExecuteWork($w$);
 4:     end for
 5:     if $|waitingSet| \neq 0$ then
 6:         return
 7:     end if
 8:     $e := eventQueue$.dequeue();
 9:     if $e \neq$ nil then
10:         $subscriptions := subscriptionTable$.get($e.type$);
11:         for all $sub$ in $subscriptions$ do
12:             $w :=$ makeWork($sub$);
13:             concurrentExecuteWork($w$);
14:         end for
15:     end if
16: end procedure

17: procedure CONCURRENTEXECUTEWORK($w$)
18:     $dependecySet := dependencyTable$.get($<w.component, w.handler>$);
19:     if ($dependecySet \neq \emptyset$) then
20:         if (($dependecySet \cap runningSet) = \emptyset$) then
21:             $runningSet$.add($<w.component, w.handler>$);
22:             $waitingSet$.remove($w$);
23:             $workerPool$.executeWork($w$);
24:         else
25:             $waitingSet$.add($w$);
26:         end if
27:     else
28:         $runningSet$.add($<w.component, w.handler>$);
29:         $waitingSet$.remove($w$);
30:         $workerPool$.executeWork($w$);
31:     end if
32: end procedure

33: procedure EXECUTEWORK($w$)                    ▷ executed by a worker thread
34:     lock($w.component$);
35:     ($w.component$).($w.handler$)($w.event$);
36:     unlock($w.component$);
37:     $runningSet$.remove($<w.component, w.handler>$);
38:     schedule();
39: end procedure
```

Hook handlers are pairs of component instances and methods. They are stored in hashtable indexed by the hook label. Hooks handlers are a case of components directly calling methods into other components. Hence, whenever component A installs a hook handler into a hook contained in event handler H of component B, a mutual exclusion dependency is created between handler H of component B and all event handlers of component A.

One example usage of hooks is in the implementation of the DKS [10] join and leave protocols. These protocols are concerned with maintaining the consistency of the ring so they operate at the routing layer. A DHT component operating at the data layer must be notified when a node has just joined or intends to leave so that it can move data items to the new responsible node.

3.2 Communication

A communication component handles the sending and receiving of messages between middleware nodes over TCP connections. Middleware nodes are addressed by a reference comprising of their Internet address and overlay address. The communication component provides the service of sending a message to a specified node reference by handling the corresponding event and triggers a message received event when a new message is received from a remote node. In providing these basic services the communication component hides the connection management from the other components of the middleware and from the application.

Hidden connection management includes initiating or accepting a new connection that is needed to send a message, periodically garbage collecting not recently used connections, and tie-breaking when two nodes have open two different connections to each other simultaneously, by closing one of them. Every middleware node listens for incoming connections on a TCP port that is part of its node reference.

The communication component also offers explicit control to connection management to other components. Connections are tagged as *permanent* or *temporary*. Permanent connections are never closed while temporary connections are subject to garbage collection. Other components can change the status of a specific connection through specific events. The status of a connection is negotiated with the other peer and a connection can be made temporary only if both end-points agree that they don't need it as a permanent connection. Automatically created connections are initially temporary, but other components can also explicitly create permanent or temporary connections. For instance, a node should have permanent connections to its neighbors in the overlay routing table.

3.2.1 Message Transmission and Reception

Message transmission and reception is done with a selector model rather than with a thread per connection model. This is mainly because as the middleware node is part of an overlay network it may have a considerable number of neighbors and therefore many open connections. Having one thread per connection would lead to a too large number of threads in the system and to considerable context-switching overhead as the Java threads are heavyweight threads. Hence, the communication component contains its own thread that blocks on all pending I/O operation and immediately unblocks and handles I/O operations that become ready. This mechanism enables scalable communication.

Message transmission and reception is done by copying message bytes from memory buffers to socket buffers and vice-versa. The selector thread blocks on a receive operation until some bytes are available in the receive socket buffer. It can also block on a transmit operation if the transmit socket buffer is full.

We use *direct* memory buffers for efficient communication as the Java Virtual Machine make a best effort to perform native I/O operations on direct buffers, avoiding extra byte copying. However, direct buffers have a higher allocation cost than normal buffers. For this reason we pre-allocate a pool of direct buffers at component initialization time. Buffers are acquired from the buffer pool as they are needed and released back thereafter.

Each connection may have an active I/O operation and some state associated with it. When the respective I/O operation becomes ready, the selector continues the operation (by sending or receiving some bytes) and updates its state. Hence, the selector transmit and receive operations are state machines. Whenever all bytes of a message have been received, and are available in a list of buffers, they are passed to a marshaler component for unmarshaling (see Section 3.2.4). Each connection has an associated queue of messages to be sent. These are already marshaled messages and are represented as lists of buffers. Whenever all bytes of a message have been sent, the next message is dequeued, a message header is composed and sent and then the bytes of the message are sent. The communication component guarantees FIFO message transmission which is relied upon by the failure detector and some of the DKS [10] protocols.

Message headers are 9 bytes long and include the message type, the message sequence number and the payload length. All messages are acknowledged. This enables the continuous estimation of the round-trip time (RTT) of each connection which is used for failure detection (see Section 3.3).

When triggering the sending of a message, other components can subscribe to notifications. They can be notified, through a specified event, either when all the bytes of the message have been sent or when the message receipt has been acknowledged.

3.2.2 Connection Closing

Periodically, all temporary connections are inspected for their last used time. If this time falls behind a specified threshold for some connection, the connection is marked for closing. In closing connections we avoid message loss. Immediately closing a connection may result in the loss of messages queued for sending on the other side of the connection. Instead, we send a special *CLOSE* message that instructs the remote peer to flush all its messages queued for sending on the connection at hand. After flushing all messages, the remote peer closes the connection itself.

3.2.3 Double Connection Tie-Breaking

Two peers may attempt to connect to each other at the same time. This may result in two different connections being established between the two peers. Should this happen, both peers detect it on accepting a new connection from the other peer to which they have already connected to. Upon detection the peer with the lowest address closes the connection it has initiated. In doing this it applies the connection closing mechanism described in the previous section.

3.2.4 Message Marshalling

Messages need to be transformed from objects into sequences of bytes for transmission and vice-versa after reception. This is done by a marshaling component intermediating message sending and receiving between other components of the middleware or application and the communication component. Marshaled messages are represented as lists of direct byte buffers which are naturally handled by the communication component. Unmarshaled messages are ordinary Java objects.

Given a sequence of bytes representing a marshaled message, the marshaling component has to instantiate and initialize the correct message object. To keep marshaled message size as little as possible we decided to code message types as integers, as opposed to the Java serialization mechanism which includes a complete class signature and is therefore too verbose.

Each middleware message type is associated a unique integer identifier, which is reserved in a static message type table. Each message class contains a marshal method returning a list of direct byte buffers and an unmarshal method taking a list of direct byte buffers as an argument. The marshal method writes the message fields, in order, as bytes in buffers which it acquires from the buffer pool, while the unmarshal method reads bytes from the provided buffers and initializes the message fields in the reverse order.

Upon marshalling a message the message type is the first integer written in the buffer, followed by the message fields. Upon unmarshaling, the message type is read from the buffer, the message class is looked up in the message type table and a new message object is instantiated. The unmarshal method is then called on this new message object instance.

As certain applications might have their own opaque messages, we have to resort to Java serialization to marshal/unmarshal these messages. This results in larger messages, however we use our efficient serialization mechanism for the bulk of the middleware messages.

3.3 Failure Detection

As our middleware nodes are to be deployed over the Internet which behaves as a partially synchronous network [11], we provide an *eventually perfect* [4, 11] failure detector. This failure detector triggers *suspicion* events when it suspects that a peer node has crashed, and *rectification* events when it finds that the suspicion was in fact a false positive. False positives can happen in the Internet where most of the time the message transmission delay is bounded but sometimes, due to congestion, messages or acknowledgements may take longer than expected to arrive, thus resulting into a timeout and triggering a false suspicion.

The failure detector relies on a prediction of round-trip time for each connection in part. As all messages exchanged by the middleware are acknowledged, the RTT can be measured for each sent message. For each connection the average RTT is kept together with the RTT variance. These values are used to compute an expected round-trip timeout (RTTO). $RTTO = E(RTT) + 4 \times VAR(RTT)$. This timeout value is used to set a timer every time a message is sent. If the timer expires before an acknowledgement is received, the peer is suspected to have crashed. If an acknowledgement is eventually received, the RTTO is recomputed to adapt to the new RTT. If an acknowledgement is received before the timer expires the timer is just canceled.

In the case when the local peer doesn't actively send messages to the remote peer, the failure detector periodically sends *ping* probes awaiting for *pong* acknowledgements within a timeout of RTTO milliseconds. From the failure detection point of view, pings are equivalent to ordinary messages and pongs are equivalent to message acknowledgements. The local peer waits for γ milliseconds from the time it receives a pong until is sends the next ping. No ping is sent if the remote peer is suspected, but the local peer awaits for the pong to the last sent ping.

As the failure detection mechanism closely relies on the RTTO estimation, computed per each link in part, and on message acknowledgements, it is implemented inside the communication component. Because each connection may have a different expected RTTO we have a failure detector instance for each connection in part. The failure detector is a state machine depicted in Algorithm 3 and Algorithm 4. The state machine is driven by events like: a message is sent by the local peer, an acknowledgement is received, a timer expires, a pong is received. Here is a description of the states of the failure detector:

- INIT if no message is sent a ping is sent after γ ms;
- MSENT a message has been sent and a timer set for RTTO;
- PSENT a ping has been sent and a timer set for RTTO;
- PMSENT a message has been sent while in the PSENT state;
- PSUSPECT no pong was received in the PSENT state and the timer expired;
- MSUSPECT no acknowledgement was received in the MSENT state and the timer expired;
- PSUSPECT_MSENT a message has been sent while in the PSUSPECT state;
- MSUSPECT_MSENT a message has been sent while in the MSUSPECT state.

If the local peer sends a sequence of messages, only the first message is used for failure detection. From the failure detection point of view, all messages sent before the acknowledgement to the first sent message is received are ignored. This behavior relies on the fact that connections are FIFO.

3.4 Application Interface

Applications making use of the middleware may interact with it in two different ways. One way, suitable for new applications, is to fit the application to the middleware architecture, that is, to have a component-oriented event-driven application.

Algorithm 3 Failure detection algorithm

```
event INIT()
    $state$ := INIT
    timer $TimerR$.start($\gamma$)
end event

event MESSAGESENT($messageId$)
    if $state$ = INIT then
        state := MSENT
        timer $TimerR$.cancel()
        $firstSentMessageId = messageId$
        timer $TimerM$.start($RTTO$)
    else if $state$ = PSENT then
        $state$ := PMSENT
    else if $state$ = PSUSPECT then
        $state$ := PSUSPECT_MSENT
    else if $state$ = MSUSPECT then
        $state$ := MSUSPECT_MSENT
    else if $state$ = PMSENT or $state$ = PSUSPECT_MSENT then
        return                                  ▷ Ignore and wait for Pong
    else if $state$ = MSUSPECT_MSENT or $state$ = MSENT then
        return                  ▷ Ignore and wait for Ack(firstSentMessageId)
    end if
end event

event ACKRECEIVED($ackId, newRTT$)
    if $state$ = INIT then
        return      ▷ Ignore, ack of message received during suspecting time
    else if $state$ = MSENT and $ackId = firstSentMessageId$ then
        timer $TimerM$.cancel()
        updateExpectedRTTO($newRTT$)
        $state$ := INIT
        timer $TimerR$.start($\gamma$)
    else if $state$ = MSUSPECT or $state$ = MSUSPECT_MSENT
            and $ackId$ = firstSentMessageId then
        trigger RECTIFICATIONEVENT($connectedPeer$)
        updateExpectedRTTO($newRTT$)
        $state$ := INIT
        timer $TimerR$.start($\gamma$)
    end if
end event
```

Algorithm 4 Failure detection algorithm continued

```
 1: event PongReceived($newRTT$)
 2:     if $state$ = PSENT or $state$ = PMSENT then
 3:         timer $TimerP$.cancel()
 4:         updateExpectedRTTO($newRTT$)
 5:         $state$ := INIT
 6:         timer $TimerR$.start($\gamma$)
 7:     else if $state$ = PSUSPECT or $state$ = PSUSPECT_MSENT then
 8:         trigger rectificationEvent($connectedPeer$)
 9:         updateExpectedRTTO($newRTT$)
10:         $state$ := INIT
11:         timer $TimerR$.start($\gamma$)
12:     end if
13: end event

14: event TimerExpired($TimerR$)
15:     sendto $connectedPeer$.ping()
16:     $state$ := PSENT
17:     timer $TimerP$.start($RTTO$)
18: end event

19: event TimerExpired($TimerP$)
20:     if $state$ = PSENT then
21:         trigger suspicionEvent($connectedPeer$)
22:         $state$ := PSUSPECT
23:     else if $state$ = PMSENT then
24:         trigger suspicionEvent($connectedPeer$)
25:         $state$ := PSUSPECT_MSENT
26:     end if
27: end event

28: event TimerExpired($TimerM$)
29:     if $state$ = MSUSPECT or $state$ = PSUSPECT_MSENT
30:             or $state$ = MSUSPECT_MSENT then
31:         return                         ▷ Ignore, the peer is already suspected
32:     else if $state$ = PMSENT then
33:         return                                    ▷ Ignore, waiting for Pong
34:     else if $state$ = MSENT then
35:         trigger suspicionEvent($connectedPeer$)
36:         $state$ := MSUSPECT
37:     end if
38: end event
```

For applications written in a control-oriented manner we provide an interfacing component whose role is to provide blocking calls to the application. Typically, middleware services are used by triggering a request event. When the service operation has been completed a response event is triggered by the middleware. The interfacing component wraps this event-based interface into a blocking call interface, thus every service is made into a call which starts by triggering the corresponding request event and then blocks awaiting for the response event. Handling the response event results in the middleware call returning to the application.

A middle ground between a fully synchronous interface and having a complete event-based application is an asynchronous interface for control-oriented applications. That is, request events are triggered by the application but the application does not have to block waiting for the response event. It can continue to run and can later check whether the response event has been triggered or not. The check can be blocking or non-blocking, that is if the response event has not been triggered yet, the application can either block awaiting it or continue running and check again at a later time. This way, the application can trigger a number of middleware services, whereby the response events are stored in a mailbox that the application checks.

3.5 Miscellaneous Services

Our middleware provides unified timer management and web-based testing support for applications.

3.5.1 Timer Management

A timer component enables other components to start and cancel timers. On starting a timer, a component has to specify an event that the timer component will trigger on timer expiration. The timer component returns an unique timer identifier when it starts a timer. This identifier can later be used by the component starting the timer to cancel it. On timer expiration the event specified when starting the timer is triggered being marked as a high priority event.

There is an unavoidable race between timer expiration and timer cancellation. This may naturally result in a timeout event being triggered even if the corresponding timer was canceled. Components should be aware of this fact and should keep a set of outstanding timers and associated timeout events. When a timer is started it should be added to the set of outstanding timers and when it is canceled it should be removed from this set. Only timeout events that are outstanding should be handled and the others should be ignored. This will prevent any inconsistencies caused by handling both a timer cancellation and a timer event.

The timer component has an associated background thread which is needed by the Java timer mechanism. The motivation for unified timer management is that if various components were to use their own timers that would result in a thread being used for each timer which is a waste of resources. By using the timer component, all timers share a single thread.

3.5.2 Web-based Testing Support

The middleware includes a web-server which serves pages with statistics and state of the middleware components. The web-server does not listen on a separate TCP port so web connections to the middleware are not distinguished from connections initiated by other middleware nodes until a GET HTTP request is received on them instead of a middleware message header. When that happens, the communication component hands over the new connection to the web-server component which replies with the requested page and closes the connection.

By default, the web-server replies with a human readable web-page containing statistics about the open connections, failure detection and status of some of the middleware components. However, every component including application components can publish their own dynamic pages on the web-server. These pages should have an easily parsable format and should contain <variable,value> pairs. Thus the value of certain variables can be automatically asserted from a unit testing framework which can browse the middleware nodes' web-servers and retrieve interesting state information.

3.6 Summary of System Threads

The DKS middleware contains the following threads:

- I/O thread - needed for blocking selection of I/O operations;
- timer thread - background thread needed by the Java timers mechanism;
- n worker threads - executing component event handlers;

4 Conclusion and Future Work

We have presented the architecture of the DKS middleware library, the constraints and deployment environment challenges that motivated our design choices, and described the services that it offers for building large-scale, dynamic, and self-organizing distributed applications. We argue that this set of services benefits most applications of this kind, and permits the rapid prototyping of new ready-deployable applications while avoiding reinventing the wheel.

We are currently evaluating our DKS system implementation using the ModelNet [18, 13] network emulator with a network model built from real Internet measurements [16, 7, 8].

As future work, we plan to enrich the set of services provided by the DKS platform and also to fit a reflective, hierarchical component model, like Fractal [2], to the DKS architecture, to allow for dynamic software reconfiguration. We are working on building a transactional database on top of the DKS DHT. We also work on optimizing the overlay network for latency by providing proximity-aware routing schemes. Finally we plan to add UDP communication support and middlebox[1] traversal support.

[1]NAT or firewall devices.

References

[1] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[3] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 298–313, New York, NY, USA, 2003. ACM Press.

[4] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[5] Bram Cohen. Incentives build robustness in bittorrent. Technical report, bittorrent.org, 2003.

[6] F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 202–215, Chateau Lake Louise, Banff, Canada, October 2001. ACM Press.

[7] DIMES. `http://www.netdimes.org`, 2004-2007.

[8] ETOMIC. `http://www.etomic.org`, 2004-2007.

[9] Michael J. Freedman, Eric Freudenthal, and David Mazi&#232;res. Democratizing content publication with coral. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

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

[11] R. Guerraoui and L. Rondrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Heidelberg, Germany, 2006.

[12] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[13] ModelNet. `http://modelnet.ucsd.edu`, 2002-2007.

[14] J. J. D. Mol, D. H. J. Epema, and H. J. Sips. The orchard algorithm: P2p multicasting without free-riding. In *P2P '06: Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 275–282, Washington, DC, USA, 2006. IEEE Computer Society.

[15] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Chateau Lake Louise, Banff, Canada, October 2001. ACM Press.

[16] Yuval Shavitt and Eran Shir. Dimes: let the internet measure itself. *SIGCOMM Comput. Commun. Rev.*, 35(5):71–74, 2005.

[17] Kunwadee Sripanidkulchai, Aditya Ganjam, Bruce Maggs, and Hui Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 107–120, New York, NY, USA, 2004. ACM Press.

[18] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 271–284, New York, NY, USA, 2002. ACM Press.