ROYAL INSTITUTE
OF TECHNOLOGY

Enabling and Achieving Self-Management for Large Scale Distributed Systems

Platform and Design Methodology for Self-Management

AHMAD AL-SHISHTAWY

Licentiate Thesis
Stockholm, Sweden 2010

TRITA-ICT/ECS AVH 10:01
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-10/01-SE
ISBN 978-91-7415-589-1

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatesexamen i datalogi fridagen den 9 april 2010 klockan 14.00 i sal D i Forum IT-Universitetet, Kungl Tekniska högskolan, Isajordsgatan 39, Kista.

Tryck: Universitetsservice US AB

Abstract

Autonomic computing is a paradigm that aims at reducing administrative overhead by using autonomic managers to make applications self-managing. To better deal with large-scale dynamic environments; and to improve scalability, robustness, and performance; we advocate for distribution of management functions among several cooperative autonomic managers that coordinate their activities in order to achieve management objectives. Programming autonomic management in turn requires programming environment support and higher level abstractions to become feasible.

In this thesis we present an introductory part and a number of papers that summaries our work in the area of autonomic computing. We focus on enabling and achieving self-management for large scale and/or dynamic distributed applications. We start by presenting our platform, called Niche, for programming self-managing component-based distributed applications. Niche supports a network-transparent view of system architecture simplifying designing application self-* code. Niche provides a concise and expressive API for self-* code. The implementation of the framework relies on scalability and robustness of structured overlay networks. We have also developed a distributed file storage service, called YASS, to illustrate and evaluate Niche.

After introducing Niche we proceed by presenting a methodology and design space for designing the management part of a distributed self-managing application in a distributed manner. We define design steps, that includes partitioning of management functions and orchestration of multiple autonomic managers. We illustrate the proposed design methodology by applying it to the design and development of an improved version of our distributed storage service YASS as a case study.

We continue by presenting a generic policy-based management framework which has been integrated into Niche. Policies are sets of rules that govern the system behaviors and reflect the business goals or system management objectives. The policy based management is introduced to simplify the management and reduce the overhead, by setting up policies to govern system behaviors. A prototype of the framework is presented and two generic policy languages (policy engines and corresponding APIs), namely SPL and XACML, are evaluated using our self-managing file storage application YASS as a case study.

Finally, we present a generic approach to achieve robust services that is based on finite state machine replication with dynamic reconfiguration of replica sets. We contribute a decentralized algorithm that maintains the set of resource hosting service replicas in the presence of churn. We use this approach to implement robust management elements as robust services that can operate despite of churn.

To my family

Acknowledgements

This thesis would not have been possible without the help and support of many people around me, only a proportion of which I have space to acknowledge here.

I would like to start by expressing my gratitude to my supervisor, Prof. Vladimir Vlassov, for his continuous support, ideas, patience, and encouragement that have been invaluable on both academic and personal levels. His insightful advice and unsurpassed knowledge kept me focused on my goals.

I am grateful to Per Brand for sharing his knowledge and experience with me during my research and for his contributions and feedback in fine-tuning my work till its final state. I also feel privileged to have the opportunity to work under the supervision of Prof. Seif Haridi. His deep knowledge in many fields of computer science, fruitful discussions, and enthusiasm have been a tremendous source of inspiration. I acknowledge the help and support given to me by Prof. Thomas Sjöland, the head of software and computer systems unit at KTH. I would like to thank Sverker Janson, the director of computer systems laboratory at SICS, for his precious advices and guidance to improve my research quality and orient me to the right direction.

I would also like to acknowledge the Grid4All European project that partially funded this thesis. I take this opportunity to thank the Grid4All team, specially Konstantin Popov and Joel Höglund for being a constant source of help.

I am indebted to all my colleagues at KTH and SICS, specially to Tallat Shafaat, Cosmin Arad, Ali Ghodsi, Amir Payberah, and Fatemeh Rahimian for making the environment at the lab both constructive and fun.

Finally, I owe my deepest gratitude to my wife Marwa and to my daughters Yara and Awan for their love and support at all times. I am most grateful to my parents for helping me to be where I am now.

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Thesis Overview

Chapter 1

Introduction

Grid, Cloud and P2P systems provide pooling and coordinated use of distributed resources and services. Most P2P systems have self-management properties that make them able to operate in the presence of resource churn (join, leave, and failure). The self-management capability hides management complexity, reduces the cost of ownership (administration and maintenance) of P2P systems. On the other hand, most Grid systems are built with an assumption of a stable and rather static Grid infrastructure that in most of cases is managed by system administrators. The complexity and management overheads of Grids makes it difficult for IT-inexperienced users to deploy and to use Grids in order to take advantages of resource sharing in dynamic Virtual Organizations (VOs) similar to P2P user communities. In this research we address the challenge of enabling self-management in large-scale and/or dynamic distributed systems, e.g. domestic Grids, in order to hide the system complexity and to automate its management, i.e. organization, tuning, healing and protection.

Most distributed systems and applications are built of distributed components using a distributed component model such as the Grid Component Model (GCM); therefore we believe that self-management should be enabled on the level of components in order to support distributed component models for development of large scale dynamic distributed systems and applications. These distributed applications need to manage themselves by having some self-* properties (i.e. self-configuration, self-healing, self-protection, self-optimization) in order to survive in a highly dynamic distributed environment. All self-* properties are based on feedback control loops, known as MAPE-K loop (monitor, analyze, plan , execute – knowledge) that come form the field of Autonomic Computing. The first step towards self-management in large-scale distributed systems is to provide distributed sensing and actuating services that are self-managing by themselves. Another important step is to provide robust management abstraction that can be used to construct MAPE-K loops. These services and abstractions should provide strong guarantees in the quality of service under churn and system evolution.

The core of our approach to self-management is based on leveraging the self-organizing properties of structured overlay networks, for providing basic services and runtime support, together with component models, for reconfiguration and introspection. The end result is an autonomic computing platform suitable for large-scale dynamic distributed environments. Structured P2P systems are designed to work in the highly dynamic distributed environment we are targeting. They have self-* properties and can tolerate churn. Therefore structured P2P systems can be used as a base to support self-management in a distributed system, e.g. as a communication medium (for message passing, broadcast, and routing), lookup (distributed hashtables and name based communication), and publish/subscribe service.

To better deal with dynamic environments; to improve scalability, robustness, and performance; we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. Several issues appears when trying to enable self-management for large scale complex distributed systems that do not appear in centralized and cluster based systems. These issues include long network delays and the difficulty of maintaining global knowledge of the system. These problems affect the observability/controllability of the control system and may prevent us from directly applying classical control theory to build control loops. Another important issue is the coordination between multiple autonomic managers to avoid conflicts and oscillations. Autonomic managers must also be replicated in dynamic environments to tolerate failures.

1.1 Main Contributions

The main contributions of the thesis are:

- First, a platform called *Niche* that enables the development, deployment, and execution of large scale component based distributed applications in dynamic environments;
- Second, a design methodology that supports the design of distributed management and defines different interaction patterns between managers;
- Third, a framework for using policy management with Niche. We also evaluate the use of two policy languages versus hard coded policies;
- Finally, an algorithm to automate the reconfiguration of nodes hosting a replicated state machine in order to tolerate resource churn. The algorithm is based on SON algorithms and service migration techniques. The algorithm is used to implement robust management elements as self-healing replicated state machine.

1.2 Thesis Organization

The thesis is organized into three parts as follows. Part I is organized into five chapters including this chapter. Chapter 2 lays out the necessary background for the thesis. Chapter 3 introduces our platform "Niche" for enabling self-management. Thesis contribution is presented in Chapter 3.3, followed by the conclusions and future work in Chapter 5. Part II includes three research papers that where produced during the thesis work. Finally, Part III presents a technical report.

Chapter 2

Background

This chapter lays out the necessary background for the thesis. The core of our approach to self-management is based on leveraging the self-organizing properties of structured overlay networks, for providing basic services and runtime support, together with component models, for reconfiguration and introspection. The end result is an autonomic computing platform suitable for large-scale dynamic distributed environments. These key concepts are described below.

2.1 Autonomic Computing

In 2001, Paul Horn from IBM coined the term *autonomic computing* to mark the start of a new paradigm of computing [1]. Autonomic computing focus on tackling the problem of growing software complexity. This problem poses a great challenge for both science and industry because the increasing complexity of computing systems makes it more difficult for the IT staff to deploy, manage and maintain such systems. This dramatically increases the cost of management. Further more, if not properly and timely managed, the performance of the system may drop or the system may even fail. Another drawback of increasing complexity is that it forces us to focus more on handling management issues instead of improving the system itself and moving forward towards new innovative applications.

Autonomic computing was inspired from the autonomic nervous system that continuously regulates and protect our bodies subconsciously [2] leaving us free to focus on other work. Similarly, an autonomic system should be aware of its environment and continuously monitor itself and adapt accordingly with minimal human involvement. Human managers should only specify higher level policies that define the general behaviour of the system. This will reduce the cost of management, improve performance, and enable the development of new innovative applications. Thus purpose of autonomic computing is not to replace humans entirely but rather to enable systems to adjust and adapt themselves automatically to reflect evolving policies defined by humans.

Properties of Self-Managing Systems

IBM proposed main properties that any self-managing system should have [3] to be an autonomic system. These properties are usually referred to as *self-** properties. The four main properties are:

- **Self-configuration:** An autonomic system should be able to configure itself based on the current environment and available resources. The system should also be able to continuously reconfigure itself and adapt to changes.
- **Self-optimization:** The system should continuously monitor itself and try to tune itself and keep performance at optimum levels.
- **Self-healing:** Failures should be detected by the system. After detection, the system should be able to recover from the failure and fix itself.
- **Self-protection:** The system should be able to protect itself from malicious use. This include protection against viruses, distributed network attacks, and intrusion attempts.

The Autonomic Computing Architecture

The autonomic computing reference architecture proposed by IBM [4] consists of the following five building blocks (see Figure 2.1).

- **Touchpoint:** consists of a set of sensors and effectors used by autonomic managers to interact with managed resources (get status and perform operations). Touchpoints are components in the system that implement a uniform management interface that hides the heterogeneity of managed resources. A managed resource must be exposed through touchpoints to be manageable. Sensors provide information about the state of the resource. Effectors provide a set of operations that can be used to modify the state of resources.
- **Autonomic Manager:** is the key building block in the architecture. Autonomic managers are used to implement the self-management behaviour of the system. This is achieved through a control loop that consists of four main stages: monitor, analyze, plan, and execute. The control loop interacts with the managed resource through the exposed touchpoints.
- **Knowledge Source:** is used to share knowledge (e.g. architecture information, monitoring history, policies, and management data such as change plans) between autonomic managers.
- **Enterprise Service Bus:** provides connectivity of components in the system.

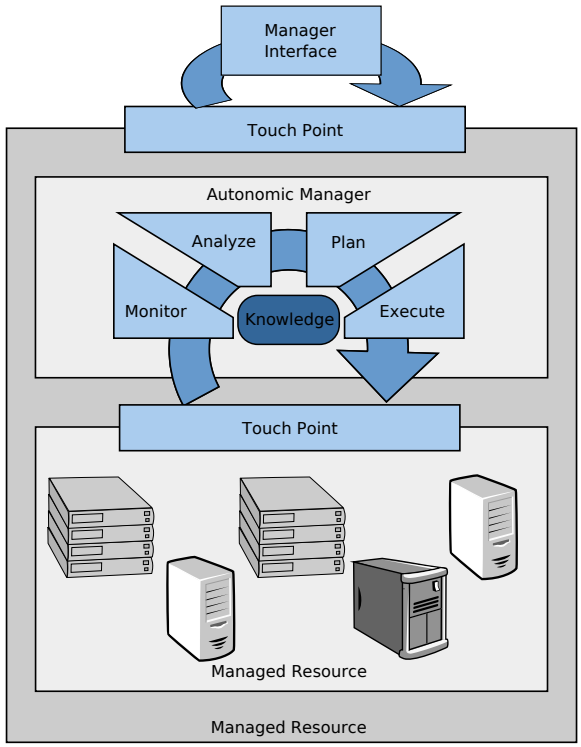

Figure 2.1: A simple autonomic computing architecture with one autonomic manager.

- **Manager Interface:** provides an interface for administrators to interact with the system. This includes the ability to monitor/change the status of the system and to control autonomic managers through policies.

Approaches to Autonomic Computing

Recent research in both academia and industry have adopted different approaches to achieve autonomic behaviour in computing systems. The most popular approaches are described below:

- **Control Theoretic Approach:** Classical control theory have been successfully applied to solve control problems in computing systems [5] such as load balancing, throughput regulation, and power management. Control theory concepts and techniques are being used to guide the development of autonomic managers for modern self-managing systems [6]. Challenges beyond

classical control theory have also been addressed [7] such as use of proactive control (model predictive control) to cope with network delays and uncertain operating environments and also multivariable optimization in the discrete domain.

- **Architectural Approach:** This approach advocates for composing autonomic systems out of components. It is closely related to service oriented architectures. Properties of components including required interfaces, expected behaviours, interaction establishment, and design patterns are described [8]. Autonomic behaviour of computing systems are achieved through dynamically modifying the structure (compositional adaptation) and thus the behaviour of the system [9, 10] in response to changes in the environment or user behaviour. Management in this approach is done at the level of components and interactions between them.
- **Emergence-based Approach:** This approach is inspired from nature where complex structures or behaviours emerge from relatively simple interactions. Examples range from the forming of sand dunes to swarming that is found in many animals. In computing systems, the overall autonomic behaviour of the system at the macro-level is not directly programmed but emerges from the, relatively simple, behavior of various sub systems at the micro-level [11–13]. This approach is highly decentralized. Subsystems make decisions autonomously based on their local knowledge and view of the system. Communication is usually simple, asynchronous, and used to exchanging data between subsystems.
- **Agent-based Approach:** Unlike traditional management approaches, that are usually centralized or hierarchical, agent-based approach for management is decentralized. This is suitable for large-scale computing systems that are distributed with many complex interactions. Agents in a multi-agent system collaborate, coordinate, and negotiate with each other forming a society or an organization to solve a problem of a distributed nature [14, 15].
- **Legacy Systems:** Research in this branch tries to add self-managing behaviours to already existing (legacy) systems. Research includes techniques for monitoring and actuating legacy systems as well as defining requirements for systems to be controllable [16–19].

In our work we followed mainly the architectural approach to autonomic computing. However, there is no clear line dividing these different approaches and they may be combined together in one system.

2.2 The Fractal Component Model

The Fractal component model [20, 21] is a modular and extensible component model that is used to design, implement, deploy and reconfigure various systems and appli-

cations. Fractal is programming language and execution model independent. The main goal of the Fractal component model is to reduce the development, deployment and maintenance costs of complex software systems. This is achieved mainly through separation of concerns that appears at different levels namely: separation of interface and implementation, component oriented programming, and inversion of control. The separation of interface and implementation separates design from implementation. The component oriented programming divides the implementation into smaller separated concerns that are assigned to components. The inversion of control separate the functional and management concerns.

A component in Fractal consists of two parts: the membrane and the content. The membrane is responsible for the non functional properties of the component while the content is responsible for the functional properties. A fractal component can be accessed through interfaces. There are three types of interfaces: client, server, and control interfaces. Client and server interfaces can be linked together through bindings while the control interfaces are used to control and introspect the component. A Fractal component can be a basic of composite component. In the case of a basic component, the content is the direct implementation of its functional properties. The content in a composite component is composed from a finite set of other components. Thus a Fractal application consists of a set of component that interact through composition and bindings.

Fractal enables the management of complex applications by making the software architecture explicit. This is mainly due to the reflexivity of the Fractal component model which means that components have full introspection and intercession capabilities (through control interfaces). The main controllers defined by fractal are attribute control, binding control, content control, and life cycle control.

The model also includes the Fractal architecture description language (Fractal ADL) that is an XML document used to describe the Fractal architecture of applications including component description (interfaces, implementation, membrane, etc.) and relation between components (composition and bindings). The Fractal ADL can also be used to deploy a fractal application where an ADL parser parses the application's ADL file and instantiate the corresponding components and bindings.

2.3 Structured Peer-to-Peer Overlay Networks

Peer-to-peer (P2P) refers to a class of distributed network architectures that is formed between participants (usually called nodes or peers) on the edge of the Internet. P2P is becoming more popular as edge devices are becoming more powerful in terms of network connectivity, storage, and processing power. A common feature to all P2P networks is that the participants form a community of peers where a peer in the community shares some resource (e.g. storage, bandwidth, or processing power) with others and in return it can use the resources shared by others [22]. Put in other words, each peer plays the role of both client and server. Thus, P2P

networks usually dose not need a central server and operates in a decentralised way. Another important feature is that peers also play the role of routers and participate in routing messages between peers in the overlay.

P2P networks are scalable and robust. The fact that each peer plays the role of both client and server has a major effect in allowing P2P networks to scale to large number of peers. This is because, unlike traditional client server model, adding more peers increases the capacity of the system (e.g. adding more storage and bandwidth). Another important factor that helps P2P to scale is that peers act as a router. Thus each peer needs only to know about a subset of other peers. The decentralised feature of P2P networks improve their robustness. There is no single point of failure and P2P networks are designed to tolerate peers joining, leaving and failing at any time they will.

Peers in a P2P network usually form an overlay network on top of the physical network topology. An overlay consists of virtual links that are formed between peers in a certain way according to the P2P network type. A virtual link between any two peers in the overlay may be implemented by several links in the physical network. The overlay is usually used for communication, indexing, and peer discovery. The way links in the overlay are formed divide P2P networks into two main classes: unstructured and structured networks. Overlay links between peers in an unstructured P2P network are formed randomly without any algorithm to organize the structure. On the other hand, overlay links between peers in a structured P2P network follow a fixed structure and is continuously maintained by an algorithm. The remainder of this section will focus on structured P2P networks.

Structured P2P network such as Chord [23], Can [24], and Pastry [25] maintain a structure of overlay links. Using this structure allow peers to implement a distributed hash table (DHT). DHTs provide a lookup service similar to hash tables that consists of a (key, value) pair. Given a key, any peer can efficiently retrieve the associated value by routing a request to the responsible peer. The responsibility of maintaining the mapping between (key, value) pairs and the routing information is distributed between the peers in such a way that peer join/leave/failure cause minimal disruption to the lookup service. This maintenance is automatic and does not require human involvement. This feature is known as self-management.

More complex service can be built on top of DHTs. Such services include name based communication, efficient multicast/broadcast, publish subscribe service, and distributed file systems.

In our work we used structured overlay networks and services built on top of it as a communication medium between different components in the system (functional components and management elements). We used indexing service to implement network transparent name based communication and component groups. We used efficient multicast/broadcast for communication and discovery. We used publish/-subscribe service to implement event based communication between management elements.

2.4 State of the Art in Self-Management for Large Scale Distributed Systems

There is the need to reduce the cost of software ownership, i.e. the cost of the administration, management, maintenance, and optimization of software systems and also networked environments such as Grids, Clouds, and P2P systems. This need is caused by the inevitable increase in complexity and scale of software systems and networked environments, which are becoming too complicated to be directly managed by humans. For many such systems manual management is difficult, costly, inefficient and error-prone.

A large-scale system may consists of thousands of elements to be monitored and controlled, and have a large number of parameters to be tuned in order to optimize system performance and power, to improve resource utilization and to handle faults while providing services according to SLAs. The best way to handle the increases in system complexity, administration and operation costs is to design autonomic systems that are able to manage themselves like the autonomic nervous system regulates and protects the human body [2, 3]. System self-management allows reducing management costs and improving management efficiency by removing humans from most of (low-level) system management mechanisms, so that the main duty of humans is to define policies for autonomic management rather than to manage the mechanisms that implement the policies.

The increasing complexity of software systems and networked environments motivates autonomic system research in both, academia and industry, e.g. [1–3, 26]. Major computer and software vendors have launched R&D initiatives in the field of autonomic computing.

The main goal of autonomic system research is to automate most of system management functions that include configuration management, fault management, performance management, power management, security management, cost management, and SLA management. Self-management objectives are typically classified into four categories: self-configuration, self-healing, self-optimization, and self-protection [3]. Major self-management objectives in large-scale systems, such as Clouds, include repairing on failures, improving resources utilization, performance optimization, power optimization, change (upgrade) management. Autonomic SLA management is also included in the list of self-management tasks. Currently, it is very important to make self-management power-aware, i.e. to minimize energy consumption while meeting service level objectives [27].

The major approach to self-management is to use one or multiple feedback control loops [2, 5], a.k.a. autonomic managers [3], to control different properties of the system based on functional decomposition of management tasks and assigning the tasks to multiple cooperative managers [28–30]. Each manager has a specific management objective (e.g. power optimization or performance optimization), which can be of one or a combination of three kinds: regulatory control (e.g. maintain server utilization at a certain level), optimization (e.g. power and performance

optimizations), disturbance rejection (e.g. provide operation while upgrading the system) [5]. A manager control loop consists of four stages, known as MAPE: Monitoring, Analyzing, Planning, and Execution [3].

Authors of [5] apply the control theoretic approach to design computing systems with feedback loops. The architectural approach to autonomic computing [8] suggests specifying interfaces, behavioral requirements, and interaction patterns for architectural elements, e.g. components. The approach has been shown to be useful for e.g. autonomous repair management [31]. The analyzing and planning stages of a control loop can be implemented using utility functions to make management decisions, e.g. to achieve efficient resource allocation [32]. Authors of [30] and [29] use multi-criteria utility functions for power-aware performance management. Authors of [33, 34] use a model-predictive control technique, namely a limited look-ahead control (LLC), combined with a rule-based managers, to optimize the system performance based on its forecast behavior over a look-ahead horizon.

Policy-based self-management [35–40] allow high-level specification of management objectives in the form of policies that drive autonomic management and can be changed at run-time. Policy-based management can be combined with "hard-coded" management.

There are many research projects focused on or using self-management for software systems and networked environments, including projects performed at the NSF Center for Autonomic Computing [41] and a number of FP6 and FP7 projects funded by European Commission.

For example, the FP7 EU-project RESERVOIR (Resources and Services Virtualization without Barriers) [42, 43] aims at enabling massive scale deployment and management of complex IT services across different administrative domains. In particular, the project develops a technology for distributed management of virtual infrastructures across sites supporting private, public and hybrid cloud architectures.

Several completed and running research projects, in particular, AutoMate [44], Unity [45], and SELFMAN [2, 46], and also the Grid4All [28, 47, 48] project we participated in, propose frameworks to augment component programming systems with management elements. The FP6 projects SELFMAN and Grid4All have taken similar approaches to self-management: both project combine structured overlay networks with component models for the development of an integrated architecture for large-scale self-managing systems. SELFMAN has developed a number of technologies that enable and facilitate development of self-managing systems. Grid4All has developed, in particular, a platform for development, deployment and execution of self-managing applications and services in dynamic environments such as domestic Grids.

There are several industrial solutions (tools, techniques and software suites) for enabling and achieving self-management of enterprise IT systems, e.g. IBM's Tivoli and HP's OpenView, which include different autonomic tools and managers to simplify management, monitoring and automation of complex enterprise-scale IT systems. These solutions are based on functional decomposition of management

performed by multiple cooperative managers with different management objectives (e.g. performance manager, power manager, storage manager, etc.). These tools are specially developed and optimized to be used in IT infrastructure of enterprises and datacenters.

Self-management can be centralized, decentralized, or hybrid (hierarchical). Most of the approaches to self-management are either based on centralized control or assume high availability of macro-scale, precise and up-to-date information about the managed system and its execution environment. The latter assumption is unrealistic for multi-owner highly-dynamic large-scale distributed systems, e.g. P2P systems, community Grids and clouds. Typically, self-management in an enterprise information system, a single-provider CDN or a datacenter cloud is centralized because most of management decisions are made based on the system global (macro-scale) state in order to achieve close to optimal system operation. However, the centralized management it is not scalable and might be not robust.

The area of autonomic computing is still evolving. Still there are many open research issues such as development environments to facilitate development of self-managing applications, efficient monitoring, scalable actuation, and robust management. Our work contributes to state of the art in autonomic computing. In particular, self-management of large-scale and/or dynamic distributed systems.

Chapter 3

Niche: A Platform for Self-Managing Distributed Applications

Niche is a proof of concept prototype that we used in order to evaluate our concepts and approach to self-management that are based on leveraging the self-organizing properties of structured overlay networks, for providing basic services and runtime support, together with component models, for reconfiguration and introspection. The end result is an autonomic computing platform suitable for large-scale dynamic distributed environments. We have designed, developed, and implemented Niche which is a platform for self-management. Niche has been used in this work as an environment to validate and evaluate different aspects of self-management such as monitoring, autonomic managers interactions, and policy based management, as well as to demonstrate our approach by using Niche to develop use cases.

This chapter will present the Niche platform (`http://niche.sics.se`), as system for the development, deployment and execution of self-managing distributed systems, applications and services. Niche has been developed by a joint group of researches and developers at the Royal Institute of Technology (KTH); Swedish Institute of Computer Science (SICS), Stockholm, Sweden; and INRIA, France.

3.1 Niche

Niche implements (in Java) the autonomic computing architecture defined in the IBM autonomic computing initiative, i.e. it allows building MAPE (Monitor, Analyse, Plan and Execute) control loops. Niche includes a component-based programming model (Fractal), API, and an execution environment. Niche, as a programming environment, separates programming of functional and management parts. The functional part is developed using Fractal components and component groups, which are controllable (e.g. can be looked up, moved, rebound, started, stopped,

etc.) and can be monitored by the management part of the application. The management part is programmed using Management Element (ME) abstractions: watchers, aggregators, managers, executors. The sensing and actuation API of Niche connects the functional and management part. MEs monitor and communicate with events, in a publish/subscribe manner. There are built-in events (e.g. component failure event) and application-dependent events (e.g. component load change event). MEs control functional components via the actuation API.

Niche also provides ability to program policy-based management using a policy language, a corresponding API and a policy engine. Current implementation of Niche includes a generic policy-based framework for policy-based management using SPL (Simplified Policy Language) or XACML (eXtensible Access Control Markup Language). The framework includes abstractions (and API) of policies, policy-managers and policy-manager groups. Policy-based management enables self-management under guidelines defined by humans in the form of management policies that can be changed at run-time. With policy-based management it is easier to administrate and maintain management policies. It facilitates development by separating of policy definition and maintenance from application logic. However, our performance evaluation shows that hard-coded management performs better than the policy-based management.

We recommend using policy-based management for high-level policies that require the flexibility of rapidly being changed and manipulated by administrators (easily understood by humans, can be changed on the fly, separate form development code for easier management, etc.). On the other hand low-level relatively static policies and management logic should be hard-coded for performance. It is also important to keep in mind that even when using policy-based management we still have to implement management actuation and monitoring.

Although programming in Niche is on the level of Java, it is both possible and desirable to program management at a higher level (e.g. declaratively). The language support includes the declarative ADL (Architecture Description Language) that is used for describing initial configurations in high-level which is interpreted by Niche at runtime (initial deployment).

Niche has been developed assuming that its run-time environment and applications with Niche might execute in a highly dynamic environment with volatile resources, where resources (computers, VMs) can unpredictably fail or leave. In order to deal with such dynamicity, Niche leverages self-organizing properties of the underlying structured overlay network, including name-based routing (when a direct binding is broken) and the DHT functionality. Niche provides transparent replication of management elements for robustness. For efficiency, Niche directly supports a component group abstraction with group bindings (one-to-all and one-to-any).

The Niche run-time system allows initial deployment of a service or an application on the network of Niche nodes (containers). Niche relies on the underlying overlay services to discover and to allocate resources needed for deployment, and to deploy the application. After deployment, the management part of the applica-

tion can monitor and react on changes in availability of resources by subscribing to resource events fired by Niche containers. All elements of a Niche application – components, bindings, groups, management elements – are identified by unique identifiers (names) that enable location transparency. Niche uses the DHT functionality of the underlying structured overlay network for its lookup service. This is especially important in dynamic environments where components need to be migrated frequently as machines leave and join frequently. Furthermore, each container maintains a cache of name-to-location mappings. Once a name of an element is resolve to its location, the element (its hosting container) is accessed directly rather than by routing messages though the overlay network. If the element moves to a new location, the element name is transparently resolved to the new location.

3.2 Demonstrators

In order to demonstrate Niche and our design methodology (see Chapter 7), we developed two self-managing services (1) YASS: Yet Another Storage Service; and (2) YACS: Yet Another Computing Service. The services can be deployed and provided on computers donated by users of the service or by a service provider. The services can operate even if computers join, leave or fail at any time. Each of the services has self-healing and self-configuration capabilities and can execute on a dynamic overlay network. Self-managing capabilities of services allows the users to minimize the human resources required for the service management. Each of services implements relatively simple self-management algorithms, which can be changed to be more sophisticated, while reusing existing monitoring and actuation code of the services.

YASS (Yet Another Storage Service) is a robust storage service that allows a client to store, read and delete files on a set of computers. The service transparently replicates files in order to achieve high availability of files and to improve access time. The current version of YASS maintains the specified number of file replicas despite of nodes leaving or failing, and it can scale (i.e. increase available storage space) when the total free storage is below a specified threshold. Management tasks include maintenance of file replication degree; maintenance of total storage space and total free space; increasing availability of popular files; releasing extra allocate storage; and balancing the stored files among available resources.

YACS (Yet Another Computing Service) is a robust distributed computing service that allows a client to submit and execute jobs, which are bags of independent tasks, on a network of nodes (computers). YACS guarantees execution of jobs despite of nodes leaving or failing. Furthermore, YACS scales, i.e. changes the number of execution components, when the number of jobs/tasks changes. YACS supports checkpointing that allows restarting execution from the last checkpoint when a worker component fails or leaves.

3.3 Lessons Learned

A middleware, such as Niche, clearly reduces burden from an application developer, because it enables and supports self-management by leveraging self-organizing properties of structured P2P overlays and by providing useful overlay services such as deployment, DHT (can be used for different indexes) and name-based communication. However, it comes at a cost of self-management overhead, in particular, the cost of monitoring and replication of management; though this cost is necessary for the democratic grid (or cloud) that operates on a dynamic environment and requires self-management.

There are four major issues to be addressed when developing a platform such as Niche for self-management of large scale distributed systems: Efficient resource discovery; robust and efficient monitoring and actuation; distribution of management to avoid bottleneck and single-point-of-failure; scale of both the events that happen in the system and the dynamicity of the system (resources and load).

To address these issues when developing Niche we used and applied different solutions and techniques. In particular we leveraged the scalability, robustness, and self-management properties of the structured overlay networks (SONs) as follows.

Resource discovery was the easiest to address, since all resources are members of the Niche overlay, we used efficient broadcast/rangecast to discover resources. This can be further improved using more complex queries that can be implemented on top of SONs.

For monitoring and actuation we used events that are disseminated using publish/subscribe system. This supports resource mobility because sensors/actuators can move with resources and still be able to publish/receive events. Also the Publish/subscribe system can be implemented in an efficient and robust way on top of SONs

In order to better deal with dynamic environments, and also to avoid management bottlenecks and single-point-of-failure, we advocate for a decentralized approach to management. The management functions should be distributed among several cooperative autonomic managers that coordinate their activities (as loosely-coupled as possible) to achieve management objectives. Multiple managers are needed for scalability, robustness, and performance and they are also useful for reflecting separation of concerns. Design steps in developing the management part of a self-managing application include spatial and functional partitioning of management, assignment of management tasks to autonomic managers, and co-ordination of multiple autonomic managers. The design space for multiple management components is large; indirect stigmergy-based interactions, hierarchical management, direct interactions. Co-ordination could use shared management elements.

In dynamic systems the rate of change (join, leaves, failure of resources, change of component load etc.) is high and that it was important to reduce the need for action/communication in the system. This may be open-ended task, but Niche contained many features that directly impact communication. The sensing/actuation infrastructure only delivers events to management elements that directly have sub-

scribed to the event (i.e. avoiding the overhead of keeping management elements up-to-date as to component location). Decentralizing management makes for better scalability. We support component groups and bindings to such groups, to be able to map this useful abstraction to the most (known) efficient communication infrastructure.

Chapter 4

Thesis Contribution

In this chapter, we present a summary if the thesis contribution. We start by listing the publications that where produced during the thesis work. Next, we describe in more details the contributions of the main areas we worked on.

4.1 List of Publications

List of publications included in this thesis

1. A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing* (T. Priol and M. Vanneschi, eds.), pp. 163–174, Springer US, July 2008. Available: `http://dx.doi.org/10.1007/978-0-387-09455-7_12`

2. A. Al-Shishtawy, V. Vlassov, P. Brand, and S. Haridi, "A design methodology for self-management in distributed environments," in *Computational Science and Engineering, 2009. CSE '09. IEEE International Conference on*, vol. 1, (Vancouver, BC, Canada), pp. 430–436, IEEE Computer Society, August 2009. Available: `http://dx.doi.org/10.1109/CSE.2009.301`

3. L. Bao, A. Al-Shishtawy, and V. Vlassov, "Policy based self-management in distributed environments," in *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2009)*, (San Francisco, California), September 2009.

4. A. Al-Shishtawy, M. A. Fayyaz, K. Popov, and V. Vlassov, "Achieving robust self-management for large-scale distributed applications," Tech. Rep. T2010:02, Swedish Institute of Computer Science (SICS), March 2010.

List of publications by the thesis author that are related to this thesis

1. P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy, "The role of overlay services in a self-managing framework for dynamic virtual organizations," in *Making Grids Work* (M. Danelutto, P. Fragopoulou, and V. Getov, eds.), pp. 153–164, Springer US, 2007. Available: `http://dx.doi.org/10.1007/978-0-387-78448-9_12`

2. K. Popov, J. Höglund, A. Al-Shishtawy, N. Parlavantzas, P. Brand, and V. Vlassov, "Design of a self-* application using p2p-based management infrastructure," in *Proceedings of the CoreGRID Integration Workshop 2008. CGIW'08.* (S. Gorlatch, P. Fragopoulou, and T. Priol, eds.), COREGrid, (Crete, GR), pp. 467–479, Crete University Press, April 2008.

3. N. de Palma, K. Popov, V. Vlassov, J. Höglund, A. Al-Shishtawy, and N. Parlavantzas, "A self-management framework for overlay-based applications," in *International Workshop on Collaborative Peer-to-Peer Information Systems (WETICE COPS 2008)*, (Rome, Italy), June 2008.

4. A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Distributed control loop patterns for managing distributed applications," in *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008)*, (Venice, Italy), pp. 260–265, Oct. 2008. Available: `http://dx.doi.org/10.1109/SASOW.2008.57`

4.2 Enabling Self-Management

Our work on enabling self management for large scale distributed systems was published as two book chapters [48, 49], a workshop paper [50], and a poster [51]. The book chapter [48] appears as Chapter 6 in this thesis.

Paper Contribution

The increasing complexity of computing systems, as discussed in Section 2.1, requires a high degree of autonomic management to improve system efficiency and reduce cost of deployment, management, and maintenance. The first step towards achieving autonomic computing systems is to enable self-management, in particular, enable autonomous runtime reconfiguration of systems and applications. By enabling self-management we mean to provide a platform that supports the programming and runtime execution of self managing computing systems. This work is first presented in Chapter 6 of this thesis and extended in the following chapters.

We combined three concepts, autonomic computing, component-based architectures, and structured overlay networks, to develop a platform that enables self-

management of large scale distributed applications. The platform, called Niche, implements the autonomic computing architecture described in Section 2.1.

Niche follows the architectural approach to autonomic computing. In the current implementation, Niche uses the Fractal component model [20]. Fractal simplifies the management of complex applications by making the software architecture explicit. We extended the Fractal component model by introducing the concept of component groups and bindings to groups. This extension results in "one-to-all" and "one-to-any" communication patterns, which support scalable, fault-tolerant and self-healing applications [52]. Groups are first-class entities and they are dynamic. The group membership can change dynamically (e.g. because of churn) affecting neither the source component nor other components of the destination group.

Niche leverages the self-organization properties of structured overlay networks and services built on top them. Self-organization of such networks and services make them attractive for large scale systems and applications. These properties include decentralization, scalability and fault tolerance. The current Niche implementation uses a Chord like structured P2P network called DKS [53]. Niche is build on top of the robust and churn tolerant services that are provided by or implemented using DKS. These services include among others lookup service, DHT, efficient broadcast/multicast, and publish subscribe service. Niche uses these services to provide a network-transparent view of system architecture, which facilitate reasoning about and designing application's management code. In particular, it facilitates migration of components and management elements caused by resource churn. These features make Niche suitable to manage large scale distributed applications deployed in dynamic environments.

Our approach to develop self-managing applications separates application's functional and management parts. We provide a programming model and a corresponding API for developing application-specific management behaviours. Autonomic managers are organized as a network of management elements interacting through events using the underlying publish/subscribe service. We also provide support for sensors and actuators. Niche leverages the introspection and dynamic reconfiguration features of the Fractal component model in order to provide sensors and actuators. Sensors can inform autonomic managers about changes in the application and its environment by generating events. Similarly, autonomic managers can modify the application by triggering events to actuators.

In order to verify and evaluate our approach we used Niche to implement as a use case a robust storage service called YASS. YASS is a storage service that allows users to store, read and delete files on a set of distributed resources. The service transparently replicates the stored files for robustness and scalability. The management part of the first prototype of YASS used two autonomic managers to manage the storage space and the file replicas.

Thesis Author Contribution

This was a joint work between researchers from the Royal Institute of Technology (KTH), the Swedish Institute of Computer Science (SICS), and INRIA. While the initial idea of combining autonomic computing, component-based architectures, and structured overlay networks is not of the thesis author, he played a major role in realizing this idea. In particular the author is a major contributor to:

- Identifying the basic overlay services required by a platform such as Niche to enable self management like name-based communication for network transparency, distributed hash table (DHT), a publish/subscribe mechanism for event dissemination, and resource discovery.
- Identifying the required higher level abstractions to facilitate programming of self managing applications such as name based component bindings, dynamic groups, and the set of network references (SNRs) abstraction that is used to implement them.
- Extending the Fractal component model with component groups and group bindings.
- Identifying the required higher level abstractions to program the management part such as management elements and sensor/actuators abstractions and that communicate through events to construct autonomic managers.
- The design and development the Niche API and platform.
- The design and development of the YASS demonstrator.

4.3 Design Methodology for Self-Management

Our work on control loop interaction patterns and design methodology for self-management was published as a conference paper [28] and a workshop paper [54]. The paper [28] appears as Chapter 7 in this thesis.

Paper Contribution

To better deal with dynamic environments; to improve scalability, robustness, and performance; we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. Multiple managers are needed for scalability, robustness, and performance and also useful for reflecting separation of concerns. Engineering of self-managing distributed applications executed in a dynamic environment requires a methodology for building robust cooperative autonomic managers. This topic is discussed in Chapter 7 of this thesis.

We present a methodology for designing the management part of a distributed self-managing application in a distributed manner. The methodology includes design space and guidelines for different design steps including management decomposition, assignment of management tasks to autonomic managers, and orchestration. For example, management can be decomposed into a number of managers each responsible for a specific self-* property or alternatively application subsystems. These managers are not independent but need to cooperate and coordinate their actions in order to achieve overall management objectives. We identified four patterns for autonomic managers to interact and coordinate their operation. The four patterns are stigmergy, hierarchical management, direct interaction, and sharing of management elements.

We illustrated the proposed design methodology by applying it to design and develop an improved version of the YASS distributed storage service prototype. We applied the four interaction patterns while developing the self-management part of YASS to coordinate the actions of different autonomic managers involved.

Thesis Author Contribution

The author was the main contributor in developing the design methodology. In particular, the interaction patterns between managers that are used to orchestrate and coordinate their activities. The author did the main bulk of the work including writing most of the article. The author also played a major role in applying the methodology to improve the YASS demonstrator and contributed to the implementation of the improved version of YASS.

4.4 Policy Based Self-Management

Our work on policy based self-management was published as a workshop paper [40] and a master thesis [55]. The paper [40] appears as Chapter 8 in this thesis.

Paper Contribution

In Chapter 8, we present a generic policy-based management framework which has been integrated into Niche. Policies are sets of rules which govern the system behaviors and reflect the business goals and objectives. The key idea of policy-based management is to allow IT administrators to define a set of policy rules to govern behaviors of their IT systems, rather than relying on manually managing or ad-hoc mechanics (e.g. writing customized scripts) [56]. The implementation and maintenance of policies are rather difficult, especially if policies are "hard-coded" (embedded) in the management code of a distributed system, and the policy logic is scattered in the system implementation. This makes it difficult to trace and change policies.

The framework introduces a policy manager in the control loop for an autonomic manager. The policy manager first loads a policy file and then, upon receiving

events, the policy manager evaluates the events against the loaded policies and acts accordingly. Using policy managers simplifies the process of maintaining and changing of policies. It may also simplify the development process by separating the application development from the policy development. We also argue for the need for a policy management group that might be needed for improving the scalability and performance of policy based management.

We recommend using policy-based management for high-level policies that require the flexibility of rapidly being changed and manipulated by administrators (easily understood by humans, can be changed on the fly, separate form development code for easier management, etc.). On the other hand low-level relatively static policies and management logic should be hard-coded for performance. It is also important to keep in mind that even when using policy-based management we still have to implement management actuation and monitoring.

A prototype of the framework is presented and evaluated using YASS distributed storage service. We evaluated two generic policy languages (policy engines and corresponding APIs), namely XACML (eXtensible Access Control Markup Language) [57] and SPL (Simplified Policy Language) [37], that we used to implement the policy logic of YASS management which was previously hard coded.

Thesis Author Contribution

The author played a major role in designing the system and introducing a policy manager in the control loop for an autonomic manager. He also suggested the use of SPL as a policy language. The author contributed to the implementation, integration, and evaluation of policy based management into the Niche platform.

4.5 Replication of Management Elements

Our work on replication of management elements was published as a technical report [58] that appears as Chapter 9 in this thesis.

Paper Contribution

To simplify the development of autonomic managers, and thus large scale distributed systems, it is useful to separate the maintenance of MEs from the development of autonomic managers. It is possible to automate the maintenance process and making it a feature of the Niche platform. This can be achieved by providing Robust Management Elements (RMEs) abstraction that developers can use if they need their MEs to be robust. By robust MEs we mean that an ME should: 1) provide transparent mobility against resource join/leave (i.e. be location independent); 2) survive resource failures by being automatically restored on another resource; 3) maintain its state consistent; 4) provide its service with minimal disruption in spite of resource join/leave/fail (high availability).

In this work, as discussed in Chapter 9 of this thesis, we present our approach to achieving RMEs, built on top of *structured overlay networks* [22], by replicating MEs using *replicated state machine* [59, 60] approach. We propose an algorithm that *automatically* maintains and reconfigures the set of resources where the ME replicas are hosted. The reconfiguration take place by *migrating* [61] MEs when needed (e.g. resource failure) to new resources. The decision on when to migrate is decentralized and automated using the symmetric replication [53] replica placement scheme. The contributions of this work are as follows:

- A decentralized algorithm that automates the reconfiguration of the set of nodes that host a replicated state machine to tolerate node churn. The algorithm uses structured overlay networks and the symmetric replication replica placement scheme to detect the need to reconfigure and to select the new set of nodes. Then the algorithm uses service migration to move/restore replicas on the new set of nodes.
- Defines a robust management element as a state machine replicated using the proposed automatic algorithm.
- Construct autonomic manager from a network of distributed RMEs.

Thesis Author Contribution

The author played a major role in the initial discussions and studies of several possible approaches to solve the problem of replicating stateful management elements. The author was also a main contributor in the development of the proposed approach and algorithms presented in the paper including writing most of the article. The author also contributed to the implementation and the simulation experiments.

Chapter 5

Conclusions and Future Work

In this chapter we present and discuss the conclusions for the main topics addressed through this thesis. At the end, we discuss possible future work that can built upon and improve research presented in this thesis.

5.1 Enabling Self-Management

A large scale distributed application deployed in dynamic environments require aggressive support for self-management. The proposed distributed component management system, Niche, enables the development of distributed component based applications with self-* behaviours. Niche simplifies the development of self-managing application by separating functional and management parts of an application and thus making it possible to develop management code separately from application's functional code. This allows the same application may run in different environment by changing management an also allows management code to be reused in different applications.

Niche leverages the self-* properties of the structured overlay network which it is built upon. Niche provides a small set of abstractions that facilitate application management. Name-based binding, component groups, sensors, actuators, and management elements, among others, are useful abstractions that enables the development of network transparent autonomic systems and applications. Network transparency, in particular, is very important in dynamic environments with high level of churn. It enables the migration of components without disturbing existing bindings and groups it also enables the migration of management elements without changing the subscriptions for events. This facilitate the reasoning and development of self-managing applications.

In order to verify and evaluate our approach we used Niche to design a self-managing application, called YASS, to be used in dynamic Grid environments. Our implementation shows the feasibility of the Niche platform. The separation of functional and management code enable us to modify management to suite different

environments and nonfunctional requirements.

5.2 Design Methodology for Self-Management

We have presented the methodology of developing the management part of a self-managing distributed application in distributed dynamic environment. We advocate for multiple managers rather than a single centralized manager that can induce a single point of failure and a potential performance bottleneck in a distributed environment. The proposed methodology includes four major design steps: decomposition, assignment, orchestration, and mapping (distribution). The management part is constructed as a number of cooperative autonomic managers each responsible either for a specific management function (according to functional decomposition of management) or for a part of the application (according to a spatial decomposition).

Distribution of autonomic managers allows distributing the management overhead and increased management performance due to concurrency and better locality. Multiple managers are needed for scalability, robustness, and performance and also useful for reflecting separation of concerns.

We have defined and described different paradigms (patterns) of manager interactions, including indirect interaction by stigmergy, direct interaction, sharing of management elements, and manager hierarchy. In order to illustrate the design steps, we have developed and presented in this paper a self-managing distributed storage service with self-healing, self-configuration and self-optimizing properties provided by corresponding autonomic managers, developed using the distributed component management system Niche. We have shown how the autonomic managers can coordinate their actions, by the four described orchestration paradigms, in order to achieve the overall management objectives.

5.3 Policy based Self-Management

In this work we proposed a policy based framework which facilitates distributed policy decision making and introduces the concept of Policy-Manager-Group that represents a group of policy-based managers formed to balance load among Policy-Managers.

Policy-based management has several advantages over hard-coded management. First, it is easier to administrate and maintain (e.g. change) management policies than to trace the hard-coded management logic scattered across codebase. Second, the separation of policies and application logic (as well as low-level hard-coded management) makes the implementation easier, since the policy author can focus on modeling policies without considering the specific application implementation, while application developers do not have to think about where and how to implement management logic, but rather have to provide hooks to make their system manageable, i.e. to enable self-management. Third, it is easier to share and reuse the same policy across multiple different applications and to change the policy

consistently. Finally, policy-based management allows policy authors and administrators to edit and to change policies on the fly (at runtime).

From our evaluation results, we can observe that the hard-coded management performs better than the policy-based management, which uses a policy engine. Therefore, it could be recommended to use policy-based management in less performance demanding managers with policies or management objectives that need to be changed on the fly (at runtime).

5.4 Replication of Management Elements

In this work we presented an approach to achieve robust management elements (RMEs) which is an abstraction that will improve the construction of autonomic managers in presence of resource churn by improving the availability and robustness of management elements (used to create autonomic managers in Niche). The approach relies on our proposed decentralized algorithm that automates the reconfiguration (migration) of the set of machines that host a replicated state machine to survive resource churn. We used replicated state machine approach to guarantee coherency of replicas.

Although in this paper we discussed the use of our approach to achieve robust management elements, we believe that this approach is generic and might be used to replicate other services built on top of structured overlay networks for the sake of robustness and high availability.

5.5 Discussion and Future Work

Autonomic computing initiative was started by IBM [1] in 2001 to overcome the problem of growing complexity related to computing systems management that prevents further developments of complex systems and services. The goal was to reduce management complexity by making computer systems self-managing.

Many architectures have been proposed to realise this goal. However, most of these architectures aim at reducing management costs in centralised or clustered environments rather than enabling complex large scale systems and services. Process control theory [5] is an important theory that inspired autonomic computing. Closed control loop is an important concept in this theory. A closed loop continuously monitors a systems and acts accordingly to keep the system in the desired state range.

Several problems appear when trying to enable self-management for large-scale and/or dynamic complex distributed systems that do not appear in centralised and cluster based systems. These problems include the absence of global knowledge of the system and network delays. These problems affect the observability/controllability of the control system and may prevent us from directly applying classical control theory.

Another important problem is scalability of management. One challenge is that management may become a bottleneck and cause hot spots. Therefor we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. This lead to the next challenge which is the coordination of multiple autonomic managers to avoid conflicts and oscillations. Multiple autonomic managers are needed in large scale distributed systems to improve scalability and robustness. Another problem is the failure of autonomic managers caused by resource churn. The challenge is to develop an efficient replication algorithm with sufficient guarantees to replicate autonomic managers in order to tolerate failures.

This is an important problem because the characteristics of large scale distributed environments (e.g. dynamicity, unpredictability, unreliable communication) requires continuous and substantial management of applications. However the same characteristics prevents the direct application of classical control theory and thus making it difficult to develop autonomic applications for such environments. The subsequence of this is that most of the applications for large scale distributed environments are simple, specialized, and/or developed in the context of specific use cases such as file sharing, storage services, communication, content distribution, distributed search engines, etc.

Networked Control System (NCS) [62] and Model Predictive Control (MPC) [63] are two method of process control that has been in use in industry (e.g. NCS used to control large factories and MPC used in process industries such as oil refineries). The main idea in NCS is to link different components in the control system (sensors, controllers, and actuators) using communication networks such as Ethernet or wireless network. The main goal was to reduce the complexity and the overall cost by reducing unnecessary wiring and also making it possible to easily modify or upgrade the control system. NCS faces similar problems related to network delays. MPC has been in use in the process industries to depict the behaviour of complex dynamical systems. The goal was to compensate for the impact of non-linearities of variables.

Our future work related to improving management in our distributed component management system, Niche, includes investigating the developing distributed algorithms based on NCS and MPC to increase observability/controllability of applications deployed in large scale distributed environments. We also plan to further develop our design methodology for self management focusing on coordinating multiple managers. This will facilitate the development of complex autonomic applications for such environments.

A major concern that arises is ease of programming of management logic. Research should hence focus on high-level programming abstractions, language support and tools that facilitate development of self-managing applications. We have already started to address this aspect.

There is the issue of coupled control loops, which we did not study. In our scenario multiple managers are directly or indirectly (via stigmergy) interacting with each other and it is not always clear how to avoid undesirable behavior such

as rapid or large oscillations which not only can cause the system to behave non-optimally but also increase management overhead. We found that it is desirable to decentralize management as much as possible, but this probably aggravates the problems with coupled control loops. Every application (or service) programmer should not need to handle coordination of multiple managers (where each manager may be responsible for a specific behavior). Future work should address design of coordination protocols that could be directly used or specialized.

Although some overhead of monitoring for self-management is unavoidable, there are opportunities for research on efficient monitoring and information gathering/aggregating infrastructures to reduce this overhead. While performance is not perhaps always the dominant concern, we believe that this should be a focus point since monitoring infrastructure itself executes on volatile resources.

Replication of management elements is a general way to achieve robustness of self-management. In fact, developers tend to ignore failure and assume that management programs will be robust. They rely mostly on naïve solutions such as stand by servers to protect against the failure of management. However, these naïve solutions are not suitable for large-scale dynamic environments. Even though we have developed and validated a solution (including distributed algorithms) for replication of management elements in Niche, it is reasonable to continue research on efficient management replication mechanisms.

Bibliography

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] P. V. Roy, S. Haridi, A. Reinefeld, J.-B. Stefani, R. Yap, and T. Coupaye, "Self management for large-scale distributed systems: An overview of the self-man project," in *FMCO '07: Software Technologies Concertation on Formal Methods for Components and Objects*, (Amsterdam, The Netherlands), Oct 2007.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan. 2003.

[4] IBM, "An architectural blueprint for autonomic computing, 4th edition." `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`, June 2006.

[5] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[6] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "Self-managing systems: a control theory foundation," in *Proc. 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems ECBS '05*, pp. 441–448, Apr. 4–7, 2005.

[7] S. Abdelwahed, N. Kandasamy, and S. Neema, "Online control for self-management in computing systems," in *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symposium RTAS 2004*, pp. 368–375, May 25–28, 2004.

[8] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An architectural approach to autonomic computing," in *Proc. International Conference on Autonomic Computing*, pp. 2–9, May 17–18, 2004.

[9] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, pp. 56–64, July 2004.

[10] M. Parashar, Z. Li, H. Liu, V. Matossian, and C. Schmidt, *Self-star Properties in Complex Information Systems*, vol. 3460/2005 of *Lecture Notes in Computer Science*, ch. Enabling Autonomic Grid Applications: Requirements, Models and Infrastructure, pp. 273–290. Springer Berlin / Heidelberg, May 2005.

[11] R. J. Anthony, "Emergence: a paradigm for robust and scalable distributed applications," in *Proc. International Conference on Autonomic Computing*, pp. 132–139, May 17–18, 2004.

[12] T. De Wolf, G. Samaey, T. Holvoet, and D. Roose, "Decentralised autonomic computing: Analysing self-organising emergent behaviour using advanced numerical methods," in *Proc. Second International Conference on Autonomic Computing ICAC 2005*, pp. 52–63, June 13–16, 2005.

[13] O. Babaoglu, M. Jelasity, and A. Montresor, *Unconventional Programming Paradigms*, vol. 3566/2005 of *Lecture Notes in Computer Science*, ch. Grassroots Approach to Self-management in Large-Scale Distributed Systems, pp. 286–296. Springer Berlin / Heidelberg, August 2005.

[14] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "A multi-agent systems approach to autonomic computing," in *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, (Washington, DC, USA), pp. 464–471, IEEE Computer Society, 2004.

[15] D. Bonino, A. Bosca, and F. Corno, "An agent based autonomic semantic platform," in *Proc. International Conference on Autonomic Computing*, pp. 189–196, May 17–18, 2004.

[16] G. Kaiser, J. Parekh, P. Gross, and G. Valetto, "Kinesthetics extreme: an external infrastructure for monitoring distributed legacy systems," in *Proc. Autonomic Computing Workshop*, pp. 22–30, June 25, 2003.

[17] C. Karamanolis, M. Karlsson, and X. Zhu, "Designing controllable computer systems," in *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, (Berkeley, CA, USA), pp. 49–54, USENIX Association, 2005.

[18] G. Valetto, G. Kaiser, and D. Phung, "A uniform programming abstraction for effecting autonomic adaptations onto software systems," in *Proc. Second International Conference on Autonomic Computing ICAC 2005*, pp. 286–297, June 13–16, 2005.

[19] M. M. Fuad and M. J. Oudshoorn, "An autonomic architecture for legacy systems," in *Proc. Third IEEE International Workshop on Engineering of Autonomic and Autonomous Systems EASe 2006*, pp. 79–88, Mar. 27–30, 2006.

[20] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The fractal component model," tech. rep., France Telecom R&D and INRIA, Feb. 5 2004.

[21] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[22] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *Communications Surveys & Tutorials, IEEE*, vol. 7, pp. 72–93, Second Quarter 2005.

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

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 161–172, ACM, 2001.

[25] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, (London, UK), pp. 329–350, Springer-Verlag, 2001.

[26] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*, pp. 257–269, 2005.

[27] "The green grid." `http://www.thegreengrid.org/` (Visited on Oct 2009).

[28] A. Al-Shishtawy, V. Vlassov, P. Brand, and S. Haridi, "A design methodology for self-management in distributed environments," in *Computational Science and Engineering, 2009. CSE '09. IEEE International Conference on*, vol. 1, (Vancouver, BC, Canada), pp. 430–436, IEEE Computer Society, August 2009.

[29] R. Das, J. O. Kephart, C. Lefurgy, G. Tesauro, D. W. Levine, and H. Chan, "Autonomic multi-agent management of power and performance in data centers," in *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, (Richland, SC), pp. 107–114, International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[30] J. Kephart, H. Chan, R. Das, D. Levine, G. Tesauro, F. Rawson, and C. Lefurgy, "Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs," in *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, pp. 24–24, June 2007.

[31] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, J.-B. Stefani, N. de Palma, and V. Quema, "Architecture-based autonomous repair management: An application to J2EE clusters," in *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, (Orlando, Florida), pp. 13–24, IEEE, Oct. 2005.

[32] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.

[33] S. Abdelwahed and N. Kandasamy, "A control-based approach to autonomic performance management in computing systems," in *Autonomic Computing: Concepts, Infrastructure, and Applications* (M. Parashar and S. Hariri, eds.), ch. 8, pp. 149–168, CRC Press, 2006.

[34] V. Bhat, M. Parashar, M. Khandekar, N. Kandasamy, and S. Klasky, "A self-managing wide-area data streaming service using model-based online control," in *Grid Computing, 7th IEEE/ACM International Conference on*, pp. 176–183, Sept. 2006.

[35] H. Chan and B. Arnold, "A policy based system to incorporate self-managing behaviors in applications," in *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (New York, NY, USA), pp. 94–95, ACM, 2003.

[36] J. Feng, G. Wasson, and M. Humphrey, "Resource usage policy expression and enforcement in grid computing," in *Grid Computing, 2007 8th IEEE/ACM International Conference on*, pp. 66–73, Sept. 2007.

[37] D. Agrawal, S. Calo, K.-W. Lee, J. Lobo, and T. W. Res., "Issues in designing a policy language for distributed management of it infrastructures," in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium*, pp. 30–39, June 2007.

[38] "Apache imperius." `http://incubator.apache.org/imperius/` (Visited on Oct 2009.

[39] V. Kumar, B. F. Cooper, G. Eisenhauer, and K. Schwan, "imanage: policy-driven self-management for enterprise-scale systems," in *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, (New York, NY, USA), pp. 287–307, Springer-Verlag New York, Inc., 2007.

[40] L. Bao, A. Al-Shishtawy, and V. Vlassov, "Policy based self-management in distributed environments," in *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2009)*, (San Francisco, California), September 2009.

[41] "The center for autonomic computing." `http://www.nsfcac.org/` (Visited Oct 2009).

[42] B. Rochwerger, A. Galis, E. Levy, J. Caceres, D. Breitgand, Y. Wolfsthal, I. Llorente, M. Wusthoff, R. Montero, and E. Elmroth, "Reservoir: Management technologies and requirements for next generation service oriented infrastructures," in *Integrated Network Management, 2009. IM '09. IFIP/IEEE International Symposium on*, pp. 307–310, June 2009.

[43] "Reservoir: Resources and services virtualization without barriers." `http://reservoir.cs.ucl.ac.uk/` (Visited on Oct 2009).

[44] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri, "Automate: Enabling autonomic applications on the grid," *Cluster Computing*, vol. 9, no. 2, pp. 161–174, 2006.

[45] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: Experiences with a prototype autonomic computing system," *Proc. of Autonomic Computing*, pp. 140–147, May 2004.

[46] "Selfman project." `http://www.ist-selfman.org/` (Visited Oct 2009).

[47] "Grid4all project." `http://www.grid4all.eu` (visited Oct 2009).

[48] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing* (T. Priol and M. Vanneschi, eds.), pp. 163–174, Springer US, July 2008.

[49] P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy, "The role of overlay services in a self-managing framework for dynamic virtual organizations," in *Making Grids Work* (M. Danelutto, P. Fragopoulou, and V. Getov, eds.), pp. 153–164, Springer US, 2007.

[50] N. de Palma, K. Popov, V. Vlassov, J. Höglund, A. Al-Shishtawy, and N. Parlavantzas, "A self-management framework for overlay-based applications," in *International Workshop on Collaborative Peer-to-Peer Information Systems (WETICE COPS 2008)*, (Rome, Italy), June 2008.

[51] K. Popov, J. Höglund, A. Al-Shishtawy, N. Parlavantzas, P. Brand, and V. Vlassov, "Design of a self-* application using p2p-based management infrastructure," in *Proceedings of the CoreGRID Integration Workshop (CGIW'08)* (S. Gorlatch, P. Fragopoulou, and T. Priol, eds.), COREGrid, (Crete, GR), pp. 467–479, Crete University Press, April 2008.

[52] P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy, "The role of overlay services in a self-managing framework for dynamic virtual organizations," in *CoreGRID Workshop, Crete, Greece*, June 2007.

[53] A. Ghodsi, *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD thesis, Royal Institute of Technology (KTH), 2006.

[54] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Distributed control loop patterns for managing distributed applications," in *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008)*, (Venice, Italy), pp. 260–265, Oct. 2008.

[55] L. Bao, "Evaluation of approaches to policy-based management in self-managing distributed system," Master's thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology (ICT), 2009.

[56] D. Agrawal, J. Giles, K. Lee, and J. Lobo, "Policy ratification," in *Policies for Distributed Systems and Networks, 2005. Sixth IEEE Int. Workshop* (T. Priol and M. Vanneschi, eds.), pp. 223– 232, June 2005.

[57] "Oasis extensible access control markup language (xacml) tc." `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#expository`.

[58] A. Al-Shishtawy, M. A. Fayyaz, K. Popov, and V. Vlassov, "Achieving robust self-management for large-scale distributed applications," Tech. Rep. T2010:02, Swedish Institute of Computer Science (SICS), March 2010.

[59] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[60] L. Lamport, "Paxos made simple," *SIGACT News*, vol. 32, pp. 51–58, December 2001.

[61] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 103–115, 2006.

[62] S. Tatikoonda, *Control under communication constraints.* PhD thesis, MIT, September 2000.

[63] C. E. Garcíaa, D. M. Prettb, and M. Morari, "Model predictive control: Theory and practice–a survey," *Automatica*, vol. 25, pp. 335–348, May 1989.

Part II

Research Papers

Paper A

Chapter 6

Enabling Self-Management of Component Based Distributed Applications

Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov,
Nikos Parlavantzas, Vladimir Vlassov, and Per Brand

In *From Grids to Service and Pervasive Computing* (T. Priol and M. Vanneschi, eds.), pp. 163–174, Springer US, July 2008.

Enabling Self-Management of Component Based Distributed Applications

Ahmad Al-Shishtawy,[1] Joel Höglund,[2] Konstantin Popov,[2] Nikos Parlavantzas,[3]
Vladimir Vlassov,[1] and Per Brand[2]

[1] Royal Institute of Technology (KTH), Stockholm, Sweden
{ahmadas, vladv}@kth.se

[2] Swedish Institute of Computer Science (SICS), Stockholm, Sweden
{kost, joel, perbrand}@sics.se

[3] INRIA, Grenoble, France
nikolaos.parlavantzas@inria.fr

Abstract

Deploying and managing distributed applications in dynamic Grid environments requires a high degree of autonomous management. Programming autonomous management in turn requires programming environment support and higher level abstractions to become feasible. We present a framework for programming self-managing component-based distributed applications. The framework enables the separation of application's functional and non-functional (self-*) parts. The framework extends the Fractal component model by the component group abstraction and one-to-any and one-to-all bindings between components and groups. The framework supports a network-transparent view of system architecture simplifying designing application self-* code. The framework provides a concise and expressive API for self-* code. The implementation of the framework relies on scalability and robustness of the Niche structured p2p overlay network. We have also developed a distributed file storage service to illustrate and evaluate our framework.

6.1 Introduction

Deployment and run-time management of applications constitute a large part of software's total cost of ownership. These costs increase dramatically for distributed applications that are deployed in dynamic environments such as unreliable networks aggregating heterogeneous, poorly managed resources.

The autonomic computing initiative [1] advocates self-configuring, self-healing, self-optimizing and self-protecting (self-* thereafter) systems as a way to reduce the management costs of such applications. Architecture-based self-* management [2] of component-based applications [3] have been shown useful for self-repair of applications running on clusters [4].

We present a design of a component management platform supporting self-* applications for community based Grids, and illustrate it with an application. Com-

munity based Grids are envisioned to fill the gap between high-quality Grid environments deployed for large-scale scientific and business applications, and existing peer-to-peer systems which are limited to a single application. Our application, a storage service, is intentionally simple from the functional point of view, but it can self-heal, self-configure and self-optimize itself.

Our framework separates application functional and self-* code. We provide a programming model and a matching API for developing application-specific self-* behaviours. The self-* code is organized as a network of *management elements* (MEs) interacting through events. The self-* code *senses* changes in the environment by means of events generated by the management platform or by application specific sensors. The MEs can *actuate* changes in the architecture – add, remove and reconfigure components and bindings between them. Applications using our framework rely on external resource management providing discovery and allocation services.

Our framework supports an extension of the Fractal component model [3]. We introduce the concept of component groups and bindings to groups. This results in "one-to-all" and "one-to-any" communication patterns, which support scalable, fault-tolerant and self-healing applications [5]. For functional code, a group of components acts as a single entity. Group membership management is provided by the self-* code and is transparent to the functional code. With a one-to-any binding, a component can communicate with a component randomly chosen at run-time from a certain group. With a one-to-all binding, it will communicate with all elements of the group. In either case, the content of the group can change dynamically (e.g. because of churn) affecting neither the source component nor other elements of the destination's group.

The management platform is self-organizing and self-healing upon churn. It is implemented on the Niche overlay network [5] providing for reliable communication and lookup, and for sensing behaviours provided to self-* code.

Our first contribution is a simple yet expressive self-* management framework. The framework supports a network-transparent view of system architecture, which simplifies reasoning about and designing application self-* code. In particular, it facilitates migration of components and management elements caused by resource churn. Our second contribution is the implementation model for our churn-tolerant management platform that leverages the self-* properties of a structured overlay network.

We do not aim at a general model for ensuring coherency and convergence of distributed self-* management. We believe, however, that our framework is general enough for arbitrary self-management control loops. Our example application demonstrates also that these properties are attainable in practice.

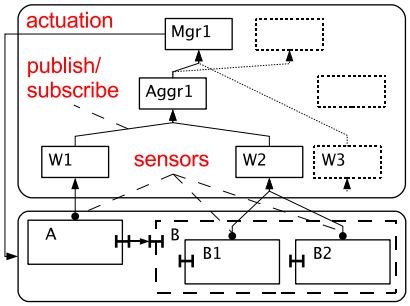

Figure 6.1: Application Architecture.

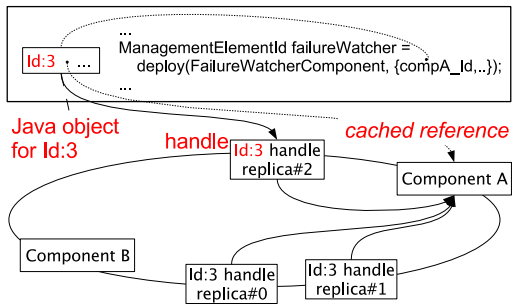

Figure 6.2: Ids and Handlers.

6.2 The Management Framework

An application in the framework consists of a component-based implementation of the application's functional specification (the lower part of Figure 6.1), and an implementation of the application's self-* behaviors (the upper part). The management platform provides for component deployment and communication, and supports sensing of component status.

Self-* code in our management framework consists of *management elements* (MEs), which we subdivide into watchers (`W1`, `W2` .. on Figure 6.1), aggregators (`Aggr1`) and managers (`Mgr1`), depending on their roles in the self-* code. MEs are stateful entities that subscribe to and receive events from *sensors* and other MEs. Sensors are either component-specific and developed by the programmer, or provided by the management framework itself such as component failure sensors. MEs can manipulate the architecture using the management *actuation* API [4] implemented by the framework. The API provides in particular functions to deploy and interconnect components.

Elements of the architecture – components, bindings, MEs, subscriptions, etc. – are identified by unique *identifiers* (IDs). Information about an architecture element is kept in a *handle* that is unique for the given ID, see Figure 6.2. The actuation API is defined in terms of IDs. IDs are introduced by DCMS API calls that deploy components, construct bindings between components and subscriptions between MEs. IDs are specified when operations are to be performed on architecture elements, like deallocating a component. Handles are destroyed (become invalid) as a side effect of destruction operation of their architecture elements. Handles to architecture elements are implemented by *sets of network references* described below. Within a ME, handles are represented by an object that can cache information from the handle. On Figure 6.2, handle object for `id:3` used by the `deploy` actuation API call caches the location of `id:3`.

An ME consists of an application-specific component and an instance of the

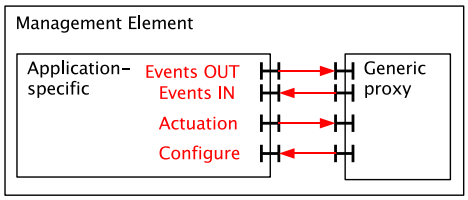

Figure 6.3: Structure of MEs.

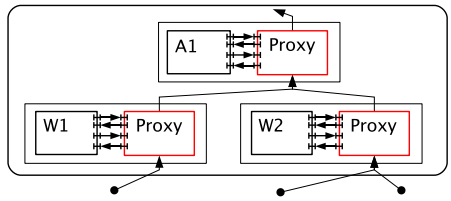

Figure 6.4: Composition of MEs.

generic proxy component, see Figure 6.3. ME proxies provide for communication between MEs, see Figure 6.4, and enable the programmer to control the management architecture transparently to individual MEs. Sensors have a similar two-part structure.

The management framework enables the developer of self-* code to control location of MEs. For every management element the developer can specify a *container* where that element should reside. A container is a first-class entity which sole purpose is to ensure that entities in the container reside on the same physical node. This eliminates network communication latencies between co-located MEs. The container's location can be explicitly defined by a location of a resource that is used to host elements of the architecture, thus eliminating the communication latency and overhead between architecture elements and managers handling them.

A **Set of Network References**, SNR [5], is a primitive data abstraction that is used to associate a *name* with a set of *references*. SNRs are stored under their names on the structured overlay network. SNR references are used to access elements in the system and can be either direct or indirect. Direct references contain the location of an entity, and indirect references refer to other SNRs by names and need to be resolved before use. SNRs can be cached by clients improving access time. The framework recognizes out-of-date references and refreshes cache contents when needed.

Groups are implemented using SNRs containing multiple references. A "one-to-any" or "one-to-all" binding to a group means that when a message is sent through the binding, the group name is resolved to its SNR, and one or more of the group references are used to send the message depending on the type of the binding. SNRs also enable mobility of elements pointed to by the references. MEs can move components between resources, and by updating their references other elements can still find the components by name. A group can grow or shrink transparently from group user point of view. Finally SNRs are used to support sensing through associating watchers with SNRs. Adding a watcher to an SNR will result in sensors being deployed for each element associated with the SNR. Changing the references of an SNR will transparently deploy/undeploy sensors for the corresponding elements.

SNRs can be replicated providing for reliable storage of application architecture.

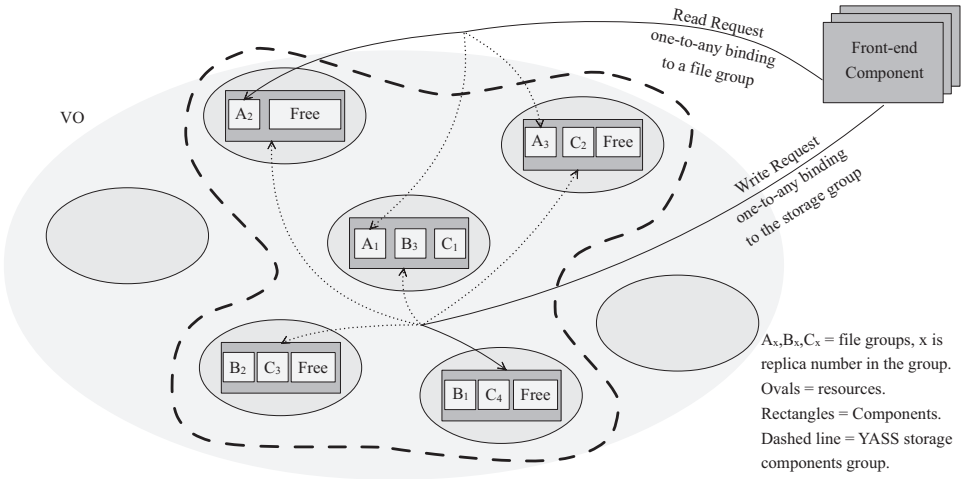

Figure 6.5: YASS Functional Part

The SRN replication provides eventual consistency of SNR replicas, but transient inconsistencies are allowed. Similarly to handling of SNR caching, the framework recognizes out-of-date SNR references and repeats SNR access whenever necessary.

6.3 Implementation and evaluation

We have designed and developed YASS – "yet another storage service" – as a way to refine the requirements of the management framework, to evaluate it and to illustrate its functionality. Our application stores, reads and deletes files on a set of distributed resources. The service replicates files for the sake of robustness and scalability. We target the service for dynamic Grid environments, where resources can join, gracefully leave or fail at any time. YASS automatically maintains the file replication factor upon resource churn, and scales itself based on the load on the service.

Application functional design

A YASS instance consists out of *front-end components* which are deployed on user machines and *storage components* Figure 6.5. Storage components are composed of *file components* representing files. The ovals in Figure 6.5 represent resources contributed to a Virtual Organization (VO). Some of the resources are used to deploy storage components, shown as rectangles.

A user store request is sent to an arbitrary storage component (one-to-any binding) that will find some r different storage components, where r is the file's replication degree, with enough free space to store a file replica. These replicas together

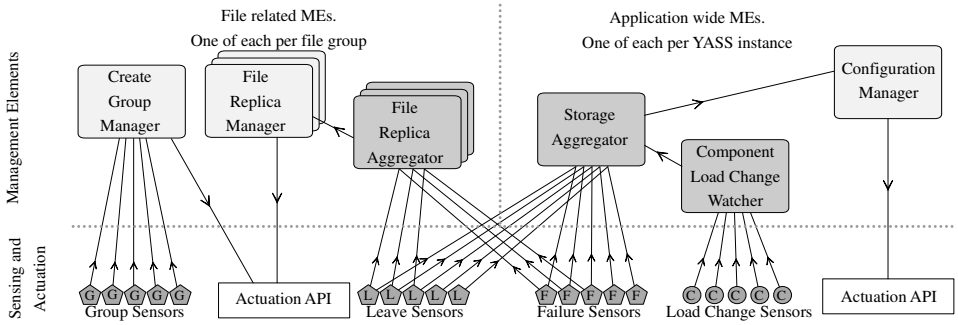

Figure 6.6: YASS Non-Functional Part

will form a *file group* containing the r dynamically created new file components. The user will then use a one-to-all binding to send the file in parallel to the r replicas in the file group. Read requests can be sent to any of the r file components in the group using the one-to-any binding between the front-end and the file group.

Application non-functional design

Configuration of application self-management. The Figure 6.6 shows the architecture of the watchers, aggregators and managers used by the application.

Associated with the group of storage components is a system-wide Storage-aggregator created at service deployment time, which is subscribed to leave- and failure-events which involve any of the storage components. It is also subscribed to a Load-watcher which triggers events in case of high system load. The Storage-aggregator can trigger StorageAvailabilityChange-events, which the Configuration-manager is subscribed to.

When new file-groups are formed by the functional part of the application, the management infrastructure propagates group-creation events to the CreateGroup-manager which initiates a FileReplica-aggregator and a FileReplica-manager for the new group. The new FileReplica-aggregator is subscribed to resource leave- and resource fail-events of the resources associated with the new file group.

Test-cases and initial evaluation

The infrastructure has been initially tested by deploying a YASS instance on a set of nodes. Using one front-end a number of files are stored and replicated. Thereafter a node is stopped, generating one fail-event which is propagated to the Storage-aggregator and to the FileReplica-aggregators of all files present on the stopped node. Below is explained in detail how the self-management acts on these events to restore desired system state.

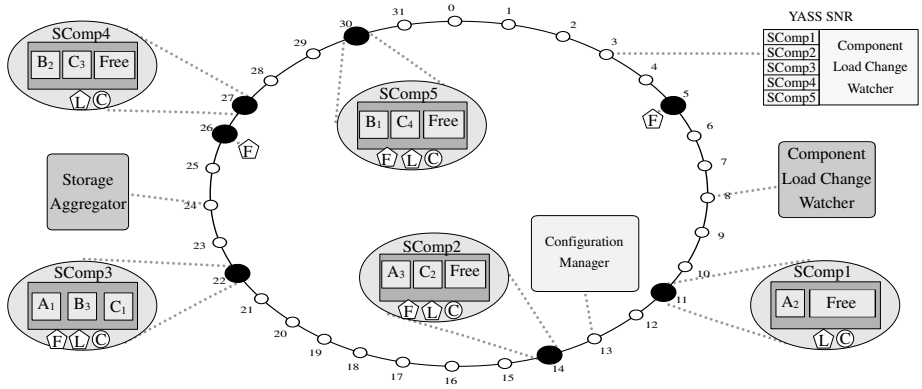

Figure 6.7: Parts of the YASS application deployed on the management infrastructure.

Figure 6.7 shows the management elements associated with the group of storage components. The black circles represent physical nodes in the P2P overlay Id space. Architectural entities (e.g. SNR and MEs) are mapped to ids. Each physical node is responsible for Ids between its predecessor and itself including itself. As there is always a physical node responsible for an id, each entity will be mapped to one of the nodes in the system. For instance the *Configuration Manager* is mapped to id 13, which is the responsibility of the node with id 14 which means it will be executed there.

Application Self-healing. Self-healing is concerned with maintaining the desired replica degree for each stored item. This is achieved as follows for resource leaves and failures:

Resource leave. An infrastructure sensor signals that a resource is about to leave. For each file stored at the leaving resource, the associated FileReplica-aggregator is notified and issues a replicaChange-event which is forwarded to the FileReplica-manager. The FileReplica-manager uses the one-to-any binding of the file-group to issue a FindNewReplica-event to any of the components in the group.

Resource failure. On a resource failure, the FileGroup-aggregator will check if the failed resource previously signaled a ResourceLeave (but did not wait long enough to let the restore replica operation finish). In that case the aggregator will do nothing, since it has already issued a replicaChange event. Otherwise a failure is handled the same way as a leave.

Application Self-configuration. With self-configuration we mean the ability to adapt the system in the face of dynamism, thereby maintaining its capability to meet functional requirements. This is achieved by monitoring the total amount of allocated storage. The Storage-aggregator is initialized with the amount of avail-

Listing 6.1: Pseudocode for parts of the Storage-aggregator

```
upon event ResourceFailure(resource_id) do
   amount_to_subtract = allocated_resources(resource_id)
   total_storage = total_amount - amount_to_subtract
   current_load = update(current_load, total_storage)
   if total_amount < initial_requirement or current_load > high_limit
      then
      trigger(availabilityChangeEvent(total_storage, current_load))
   end
```

able resources at deployment time and updates the state in case of resource leaves or failures. If the total amount of allocated resources drops below given requirements, the Storage-aggregator issues a storageAvailabilityChange-event which is processed by the Configuration-manager. The Configuration-manager will try to find an unused resource (via the external resource management service) to deploy a new storage component, which is added to the group of components. Parts of the Storage-aggregator and Configuration-manager pseudocode is shown in Listing 6.1, demonstrating how the stateful information is kept by the aggregator and updated through sensing events, while the actuation commands are initiated by the manager.

Application Self-optimization. In addition to the two above described test-cases we have also designed but not fully tested application self-optimization. With self-optimization we mean the ability to adapt the system so that it, besides meeting functional requirements, also meets additional non-functional requirements such as efficiency. This is achieved by using the ComponentLoad-watcher to gather information on the total system load, in terms of used storage. The storage components report their load changes, using application specific load sensors. These load-change events are delivered to the Storage-aggregator. The aggregator will be able to determine when the total utilization is critically high, in which case a StorageAvailabilityChange-event is generated and processed by the Configuration-manager in the same way as described in the self-configuration section. If utilization drops below a given threshold, and the amount of allocated resources is above initial requirements, a storageAvailabilityChange-event is generated. In this case the event indicates that the availability is higher than needed, which will cause the Configuration-manager to query the ComponentLoad-watcher for the least loaded storage component, and instruct it to deallocate itself, thereby freeing the resource. Parts of the Configuration-manager pseudocode is shown in Listing 6.2, demonstrating how the number of storage components can be adjusted upon need.

Listing 6.2: Pseudocode for parts of the Configuration-manager

```
upon event availabilityChangeEvent(total_storage, new_load) do
   if total_storage < initial_requirement or new_load > high_limit then
      new_resource =
          resource_discover(component_requirements, compare_criteria)
      new_resource = allocate(new_resource, preferences)
      new_component =
          deploy(storage_component_description, new_resource)
      add_to_group(new_component, component_group)
   elseif total_storage > initial_requirement and new_load < low_limit
     then
     least_loaded_component = component_load_watcher.get_least_loaded()
     least_loaded_resource = least_loaded_component.get_resource()
     trigger(resourceLeaveEvent(least_loaded_resource))
   end
```

6.4 Related Work

Our work builds on the technical work on the Jade component-management system [4]. Jade utilizes the Java RMI, and is limited to cluster environments as it relies on small and bounded communication latencies between nodes.

As the work here suggests a particular implementation model for distributed component based programming, relevant related work can be found in research dealing specifically with autonomic computing in general and in research about component and programming models for distributed systems.

Autonomic Management. The vision of autonomic management as presented in [1] has given rise to a number of proposed solutions to aspects of the problem. Many solutions adds self-management support through the actions of a centralized self-manager. One suggested system which tries to add some support for the self-management of the management system itself is Unity [6]. Following the model proposed by Unity, self-healing and self-configuration are enabled by building applications where each system component is a autonomic element, responsible for its own self-management. Unity assumes cluster-like environments where the application nodes might fail, but the project only partly addresses the issue of self-management of the management infrastructure itself.

Relevant complementary work include work on checkpointing in distributed environments. Here recent work on Cliques [7] can be mentioned, where worker nodes help store checkpoints in a distributed fashion to reduce load on managers which then only deal with group management. Such methods could be introduced in our framework to support stateful applications.

Component Models. Among the proposed component models which target building distributed systems, the traditional ones, such as the Corba Component Model or the standard Enterprise JavaBeans were designed for client-server relationships assuming highly available resources. They provide very limited support for dynamic

reconfiguration. Other component models, such as OpenCOM [8], allow dynamic flexibility, but their associated infrastructure lacks support for operation in dynamic environments.

The Grid Component Model, GCM [9], is a recent component model that specifically targets grid programming. GCM is defined as an extension of Fractal and its features include many-to-many communications with various semantics and autonomic components.

GCM defines simple "autonomic managers" that embody autonomic behaviours and expose generic operations to execute autonomic operations, accept QoS contracts, and to signal QoS violations. However, GCM does not prescribe a particular implementation model and mechanisms to ensure the efficient operation of self-* code in large-scale environments. Thus, GCM can be seen as largely complementary to our work and thanks to the common ancestor, we believe that our results can be exploited within a future GCM implementation. *Behavioural skeletons* [10] aim to model recurring patterns of component assemblies equipped with correct and effective self-management schemes. Behavioural skeletons are being implemented using GCM, but the concept of reusable, domain-specific, self-management structures can be equally applied using our component framework.

GCM also defines collective communications by introducing new kinds of cardinalities for component interfaces: multicast, and gathercast [11]. This enables one-to-n and n-to-one communication. However GCM does not define groups as a first class entities, but only implicitly through bindings, so groups can not be shared and reused. GCM also does not mention how to handle failures and dynamism (churn) and who is responsible to maintain the group. Our one-to-all binding can utilise the multicast service, provided by the underlying P2P overlay, to provide more scalable and efficient implementation in case of large groups. Also our model supports mobility so members of the group can change their location without affecting the group.

A component model designed specifically for structured overlay networks and wide scale deployment is p2pCM [12], which extends the DERMI [13] object middleware platform. The model provides replication of component instances, component lifecycle management and group communication, including anycall functionality to communicate with the closest instance of a component. The model does not offer higher level abstractions such as watchers and event handlers, and the support for self-healing and issues of consistency are only partially addressed.

6.5 Future Work

Currently we are working on the management element wrapper abstraction. This abstraction adds fault-tolerance to the self-* code by enabling ME replication. The goal of the management element wrapper is to provide consistency between the replicated ME in a transparent way and to restore the replication degree if one of the replicas fails. Without this support from the framework, the user can still have

self-* fault-tolerance by explicitly implementing it as a part of the application's non-functional code. The basic idea is that the management element wrapper adds a consistency layer between the replicated ME from one side and the sensors/actuators from the other side. This layer provides a uniform view of the events/actions for both sides.

Currently the we use a simple architecture description language (ADL) only covering application functional behaviours. We hope to extend this to also cover non-functional aspects.

We are also evaluating different aspects of our framework such as the overhead of our management framework in terms of network traffic and the time need execute self-* code. Another important aspect is to analyse the effect of churn on the self-* code.

Finally we would like to evaluate our framework using applications with more complex self-* behaviours.

6.6 Conclusions

The proposed management framework enables development of distributed component based applications with self-* behaviours which are independent from application's functional code, yet can interact with it when necessary. The framework provides a small set of abstractions that facilitate fault-tolerant application management. The framework leverages the self-* properties of the structured overlay network which it is built upon. We used our component management framework to design a self-managing application to be used in dynamic Grid environments. Our implementation shows the feasibility of the framework.

Bibliography

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] J. Hanson, I. Whalley, D. Chess, and J. Kephart, "An architectural approach to autonomic computing," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 2–9, IEEE Computer Society, 2004.

[3] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The fractal component model," tech. rep., France Telecom R&D and INRIA, Feb. 5 2004.

[4] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, J.-B. Stefani, N. de Palma, and V. Quema, "Architecture-based autonomous repair management: An application to J2EE clusters," in *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, (Orlando, Florida), pp. 13–24, IEEE, Oct. 2005.

[5] P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy, "The role of overlay services in a self-managing framework for dynamic virtual organizations," in *CoreGRID Workshop, Crete, Greece*, June 2007.

[6] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: Experiences with a prototype autonomic computing system," *Proc. of Autonomic Computing*, pp. 140–147, May 2004.

[7] D. K. F. Araujo, P. Domingues and L. M. Silva, "Using cliques of nodes to store desktop grid checkpoints," in *Proceedings of CoreGRID Integration Workshop 2008*, pp. 15–26, Apr. 2008.

[8] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama, "A component model for building systems software," in *Proceedings of IASTED Software Engineering and Applications (SEA'04)*, (Cambridge MA, USA), Nov. 2004.

[9] "Basic features of the Grid component model," CoreGRID Deliverable D.PM.04, CoreGRID, EU NoE project FP6-004265, Mar. 2007.

[10] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto, "Behavioural skeletons in gcm: Autonomic management of grid components," in *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, (Washington, DC, USA), pp. 54–63, IEEE Computer Society, 2008.

[11] F. Baude, D. Caromel, L. Henrio, and M. Morel, "Collective interfaces for distributed components," in *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, (Washington, DC, USA), pp. 599–610, IEEE Computer Society, 2007.

[12] C. Pairot, P. García, R. Mondéjar, and A. Gómez-Skarmeta, "p2pCM: A structured peer-to-peer Grid component model," in *International Conference on Computational Science*, pp. 246–249, 2005.

[13] C. Pairot, P. García, and A. Gómez-Skarmeta, "Dermi: A new distributed hash table-based middleware framework," *IEEE Internet Computing*, vol. 08, no. 3, pp. 74–84, 2004.

Paper B

Chapter 7

A Design Methodology for Self-Management in Distributed Environments

Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, and Seif Haridi

In *IEEE International Conference onComputational Science and Engineering, 2009. CSE '09.* , vol. 1, (Vancouver, BC, Canada), pp. 430–436, IEEE Computer Society, August 2009.

A Design Methodology for Self-Management in Distributed Environments

Ahmad Al-Shishtawy[1], Vladimir Vlassov[1], Per Brand[2], and Seif Haridi[1,2]

[1] Royal Institute of Technology (KTH), Stockholm, Sweden
{ahmadas, vladv, haridi}@kth.se

[2] Swedish Institute of Computer Science (SICS), Stockholm, Sweden
{perbrand, seif}@sics.se

Abstract

Autonomic computing is a paradigm that aims at reducing administrative overhead by providing autonomic managers to make applications self-managing. In order to better deal with dynamic environments, for improved performance and scalability, we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. We present a methodology for designing the management part of a distributed self-managing application in a distributed manner. We define design steps, that includes partitioning of management functions and orchestration of multiple autonomic managers. We illustrate the proposed design methodology by applying it to design and development of a distributed storage service as a case study. The storage service prototype has been developed using the distributing component management system *Niche*. Distribution of autonomic managers allows distributing the management overhead and increased management performance due to concurrency and better locality.

7.1 Introduction

Autonomic computing [1] is an attractive paradigm to tackle management overhead of complex applications by making them self-managing. Self-management, namely self-configuration, self-optimization, self-healing, and self-protection (self-* thereafter), is achieved through autonomic managers [2]. An autonomic manager continuously monitors hardware and/or software resources and acts accordingly. Managing applications in dynamic environments (like community Grids and peer-to-peer applications) is specially challenging due to high resource churn and lack of clear management responsibility.

A distributed application requires multiple autonomic managers rather than a single autonomic manager. Multiple managers are needed for scalability, robustness, and performance and also useful for reflecting separation of concerns. Engineering of self-managing distributed applications executed in a dynamic environment requires a methodology for building robust cooperative autonomic managers.

The methodology should include methods for management decomposition, distribution, and orchestration. For example, management can be decomposed into a number of managers each responsible for a specific self-* property or alternatively application subsystems. These managers are not independent but need to cooperate and coordinate their actions in order to achieve overall management objectives.

The major contributions of the paper are as follows. We propose a methodology for designing the management part of a distributed self-managing application in a distributed manner, i.e. with multiple interactive autonomic managers. Decentralization of management and distribution of autonomic managers allows distributing the management overhead, increasing management performance due to concurrency and/or better locality. Decentralization does avoid a single point of failure however it does not necessarily improve robustness. We define design steps, that includes partitioning of management, assignment of management tasks to autonomic managers, and orchestration of multiple autonomic managers. We describe a set of patterns (paradigms) for manager interactions.

We illustrate the proposed design methodology including paradigms of manager interactions by applying it to design and development of a distributed storage service as a case study. The storage service prototype has been developed using the distributing component management system *Niche*[1] [3–5].

The remainder of this paper is organized as follows. Section 7.2 describes Niche and relate it to the autonomic computing architecture. Section 7.3 presents the steps for designing distributed self-managing applications. Section 7.4 focuses on orchestrating multiple autonomic managers. In Section 7.5 we apply the proposed methodology to a distributed file storage as a case study. Related work is discussed in Section 7.6 followed by conclusions and future work in Section 7.7.

7.2 The Distributed Component Management System

The autonomic computing reference architecture proposed by IBM [2] consists of the following five building blocks.

- **Touchpoint:** consists of a set of sensors and effectors used by autonomic managers to interact with managed resources (get status and perform operations). Touchpoints are components in the system that implement a uniform management interface that hides the heterogeneity of managed resources. A managed resource must be exposed through touchpoints to be manageable.
- **Autonomic Manager:** is the key building block in the architecture. Autonomic managers are used to implement the self-management behaviour of the system. This is achieved through a control loop that consists of four main stages: monitor, analyze, plan, and execute. The control loop interacts with the managed resource through the exposed touchpoints.

[1] In our previous work [3, 4] our distributing component management system *Niche* was called DCMS

- **Knowledge Source:** is used to share knowledge (e.g. architecture information and policies) between autonomic managers.

- **Enterprise Service Bus:** provides connectivity of components in the system.

- **Manager Interface:** provides an interface for administrators to interact with the system. This includes the ability to monitor/change the status of the system and to control autonomic managers through policies.

The use-case presented in this paper has been developed using the distributed component management system *Niche* [3, 4]. Niche implements the autonomic computing architecture described above. Niche includes a distributed component programming model, APIs, and a run-time system including deployment service. The main objective of Niche is to enable and to achieve self-management of component-based applications deployed on dynamic distributed environments such as community Grids. A self-managing application in Niche consists of functional and management parts. Functional components communicate via bindings, whereas management components communicate mostly via a publish/subscribe event notification mechanism.

The Niche run-time environment is a network of distributed containers hosting functional and management components. Niche uses a structured overlay network (Niche [4]) as the enterprise service bus. Niche is self-organising on its own and provides overlay services used by Niche such as name-based communication, distributed hash table (DHT) and a publish/subscribe mechanism for event dissemination. These services are used by Niche to provide higher level communication abstractions such as name-based bindings to support component mobility; dynamic component groups; one-to-any and one-to-all bindings, and event based communication.

For implementing the touchpoints, Niche leverages the introspection and dynamic reconfiguration features of the Fractal component model [6] in order to provide sensors and actuation API abstractions. Sensors are special components that can be attached to the application's functional components. There are also built-in sensors in Niche that sense changes in the environment such as resource failures, joins, and leaves, as well as modifications in application architecture such as creation of a group. The actuation API is used to modify the application's functional and management architecture by adding, removing and reconfiguring components, groups, bindings.

The Autonomic Manager (a control loop) in Niche is organized as a network of *Management Elements* (MEs) interacting through events, monitoring via sensors and acting using the actuation API. This enables the construction of distributed control loops. MEs are subdivided into watchers, aggregators, and managers. Watchers are used for monitoring via sensors and can be programmed to find symptoms to be reported to aggregators or directly to managers. Aggregators are

used to aggregate and analyse symptoms and to issue change requests to managers. Managers do planning and execute change requests.

Knowledge in Niche is shared between MEs using two mechanisms: first, using the publish/subscribe mechanism provided by Niche; second, using the Niche DHT to store/retrieve information such as component group members, name-to-location mappings.

7.3 Steps in Designing Distributed Management

A self-managing application can be decomposed into three parts: the functional part, the touchpoints, and the management part. The design process starts by specifying the functional and management requirements for the functional and management parts, respectively. In the case of Niche, the functional part of the application is designed by defining interfaces, components, component groups, and bindings. The management part is designed based on management requirements, by defining autonomic managers (management elements) and the required touchpoints (sensors and effectors).

An Autonomic Manager is a control loop that senses and affects the functional part of the application. For many applications and environments it is desirable to decompose the autonomic manager into a number of cooperating autonomic managers each performing a specific management function or/and controlling a specific part of the application. Decomposition of management can be motivated by different reasons such as follows. It allows avoiding a single point of failure. It may be required to distribute the management overhead among participating resources. Self-managing a complex system may require more than one autonomic manager to simplify design by separation of concerns. Decomposition can also be used to enhance the management performance by running different management tasks concurrently and by placing the autonomic managers closer to the resources they manage.

We define the following iterative steps to be performed when designing and developing the management part of a self-managing distributed application in a distributed manner.

Decomposition: The first step is to divide the management into a number of management tasks. Decomposition can be either functional (e.g. tasks are defined based which self-* properties they implement) or spacial (e.g. tasks are defined based on the structure of the managed application). The major design issue to be considered at this step is granularity of tasks assuming that a task or a group of related tasks can be performed by a single manager.

Assignment: The tasks are then assigned to autonomic managers each of which becomes responsible for one or more management tasks. Assignment can be done based on self-* properties that a task belongs to (according to the

functional decomposition) or based on which part of the application that task is related to (according to the spatial decomposition).

Orchestration: Although autonomic managers can be designed independently, multiple autonomic managers, in the general case, are not independent since they manage the same system and there exist dependencies between management tasks. Therefore they need to interact and coordinate their actions in order to avoid conflicts and interference and to manage the system properly.

Mapping: The set of autonomic managers are then mapped to the resources, i.e. to nodes of the distributed environment. A major issue to be considered at this step is optimized placement of managers and possibly functional components on nodes in order to improve management performance.

In this paper our major focus is on the orchestration of autonomic managers as the most challenging and less studied problem. The actions and objectives of the other stages are more related to classical issues in distributed systems such as partitioning and separation of concerns, and optimal placement of modules in a distributed environment.

7.4 Orchestrating Autonomic Managers

Autonomic managers can interact and coordinate their operation in the following four ways:

Stigmergy

Stigmergy is a way of indirect communication and coordination between agents [7]. Agents make changes in their environment, and these changes are sensed by other agents and cause them to do more actions. Stigmergy was first observed in social insects like ants. In our case agents are autonomic managers and the environment is the managed application.

The stigmergy effect is, in general, unavoidable when you have more than one autonomic manager and can cause undesired behaviour at runtime. Hidden stigmergy makes it challenging to design a self-managing system with multiple autonomic managers. However stigmergy can be part of the design and used as a way of orchestrating autonomic managers (Figure 7.1).

Hierarchical Management

By hierarchical management we mean that some autonomic managers can monitor and control other autonomic managers (Figure 7.2). The lower level autonomic managers are considered as a managed resource for the higher level autonomic manager. Communication between levels take place using touchpoints. Higher level managers can sense and affect lower level managers.

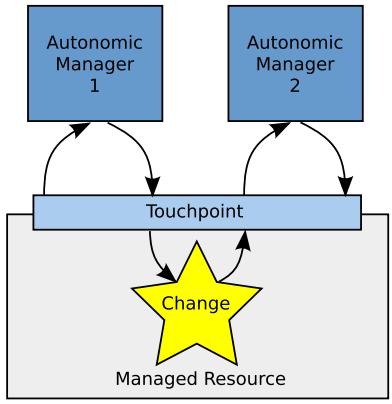

Figure 7.1: The stigmergy effect.

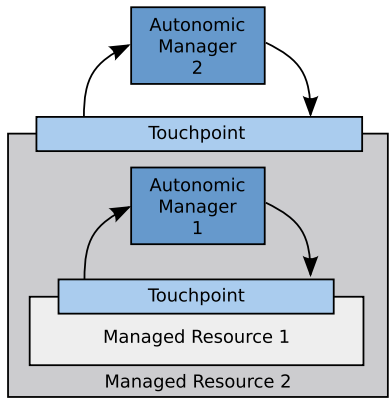

Figure 7.2: Hierarchical management.

Autonomic managers at different levels often operate at different time scales. Lower level autonomic managers are used to manage changes in the system that need immediate actions. Higher level autonomic managers are often slower and used to regulate and orchestrate the system by monitoring global properties and tuning lower level autonomic managers accordingly.

Direct Interaction

Autonomic managers may interact directly with one another. Technically this is achieved by binding the appropriate management elements (typically managers) in the autonomic managers together (Figure 7.3). Cross autonomic manager bindings can be used to coordinate autonomic managers and avoid undesired behaviors such

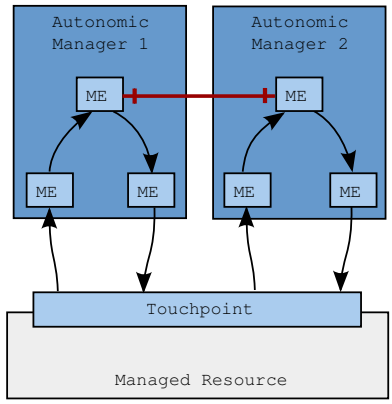

Figure 7.3: Direct interaction.

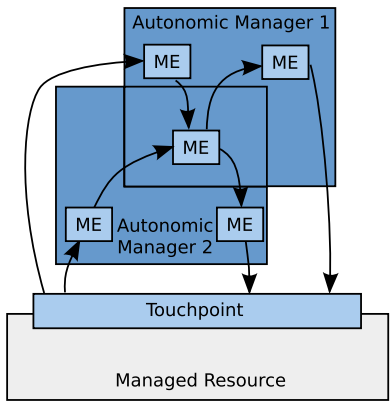

Figure 7.4: Shared Management Elements.

as race conditions or oscillations.

Shared Management Elements

Another way for autonomic managers to communicate and coordinate their actions is by sharing management elements (Figure 7.4). This can be used to share state (knowledge) and to synchronise their actions.

7.5 Case Study: A Distributed Storage Service

In order to illustrate the design methodology, we have developed a storage service called YASS (Yet Another Storage Service) [3], using Niche. The case study illus-

trates how to design a self-managing distributed system monitored and controlled by multiple distributed autonomic managers.

YASS Specification

YASS is a storage service that allows users to store, read and delete files on a set of distributed resources. The service transparently replicates the stored files for robustness and scalability.

Assuming that YASS is to be deployed and provided in a dynamic distributed environment, the following management functions are required in order to make the storage service self-managing in the presence of dynamicity in resources and load: the service should tolerate the resource churn (joins/leaves/failures), optimize usage of resources, and resolve hot-spots. We define the following tasks based on the functional decomposition of management according to self-* properties (namely self-healing, self-configuration, and self-optimization) to be achieved.

- Maintain the file replication degree by restoring the files which were stored on a failed/leaving resource. This function provides the self-healing property of the service so that the service is available despite of the resource churn;
- Maintain the total storage space and total free space to meet QoS requirements by allocating additional resources when needed. This function provides self-configuration of the service;
- Increasing the availability of popular files. This and the next two functions are related to the self-optimization of the service.
- Release excess allocated storage when it is no longer needed.
- Balance the stored files among the allocated resources.

YASS Functional Design

A YASS instance consists of *front-end components* and *storage components* as shown in Figure 7.5. The front-end component provides a user interface that is used to interact with the storage service. Storage components represent the storage capacity available at the resource on which they are deployed.

The storage components are grouped together in a storage group. A user issues commands (store, read, and delete) using the front-end. A store request is sent to an arbitrary storage component (using one-to-any binding between the front-end and the storage group) which in turn will find some r different storage components, where r is the file's replication degree, with enough free space to store a file replica. These replicas together will form a *file group* containing the r storage components that will host the file. The front-end will then use a one-to-all binding to the file group to transfer the file in parallel to the r replicas in the group. A read request is

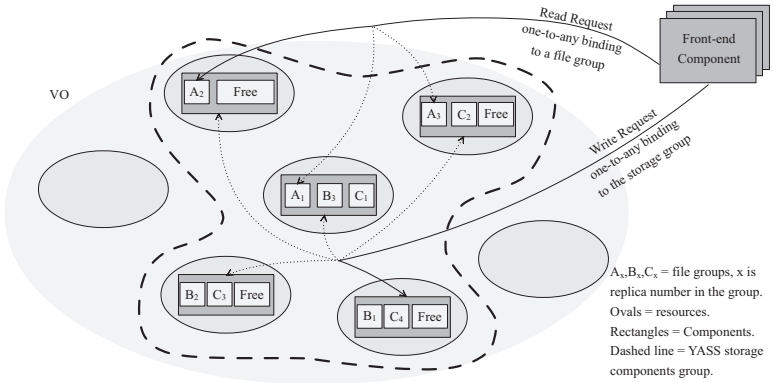

Figure 7.5: YASS Functional Part

sent to any of the r storage components in the group using the one-to-any binding between the front-end and the file group. A delete request is sent to the file group in parallel using a one-to-all binding between the front-end and the file group.

Enabling Management of YASS

Given that the functional part of YASS has been developed, to manage it we need to provide touchpoints. Niche provides basic touchpoints for manipulating the system's architecture and resources, such as sensors of resource failures and component group creation; and effectors for deploying and binding components.

Beside the basic touchpoint the following additional, YASS specific, sensors and effectors are required.

- A load sensor to measure the current free space on a storage component;
- An access frequency sensor to detect popular files;
- A replicate file effector to add one extra replica of a specified file;
- A move file effector to move files for load balancing.

Self-Managing YASS

The following autonomic managers are needed to manage YASS in a dynamic environment. All four orchestration techniques in Section 7.4 are demonstrated.

Replica Autonomic Manager

The replica autonomic manager is responsible for maintaining the desired replication degree for each stored file in spite of resources failing and leaving. This

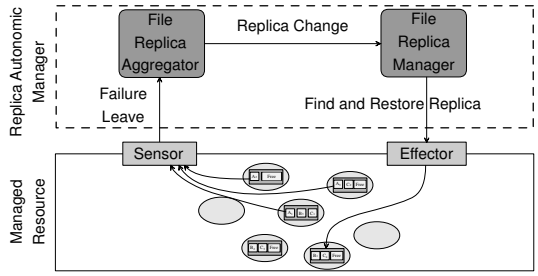

Figure 7.6: Self-healing control loop.

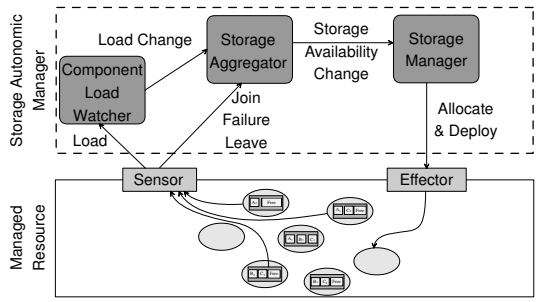

Figure 7.7: Self-configuration control loop.

autonomic manager adds the self-healing property to YASS. The replica autonomic manager consists of two management elements, the File-Replica-Aggregator and the File-Replica-Manager as shown in Figure 7.6.

The File-Replica-Aggregator monitors a file group, containing the subset of storage components that host the file replicas, by subscribing to resource fail or leave events caused by any of the group members. These events are received when a resource, on which a component member in the group is deployed, is about to leave or has failed. The File-Replica-Aggregator responds to these events by triggering a replica change event to the File-Replica-Manager that will issue a find and restore replica command.

Storage Autonomic Manager

The storage autonomic manager is responsible for maintaining the total storage capacity and the total free space in the storage group, in the presence of dynamism, to meet QoS requirements. The dynamism is due either to resources failing/leaving (affecting both the total and free storage space) or file creation/addition/deletion (affecting the free storage space only). The storage autonomic manager reconfigures YASS to restore the total free space and/or the total storage capacity to meet

the requirements. The reconfiguration is done by allocating free resources and deploying additional storage components on them. This autonomic manager adds the self-configuration property to YASS. The storage autonomic manager consists of Component-Load-Watcher, Storage-Aggregator, and Storage-Manager as shown in Figure 7.7.

The Component-Load-Watcher monitors the storage group, containing all storage components, for changes in the total free space available by subscribing to the load sensors events. The Component-Load-Watcher will trigger a load change event when the load is changed by a predefined delta. The Storage-Aggregator is subscribed to the Component-Load-Watcher load change event and the resource fail, leave, and join events (note that the File-Replica-Aggregator also subscribes to the resource failure and leave events). The Storage-Aggregator, by analyzing these events, will be able to estimate the total storage capacity and the total free space. The Storage-Aggregator will trigger a storage availability change event when the total and/or free storage space drops below a predefined thresholds. The Storage-Manager responds to this event by trying to allocate more resources and deploying storage components on them.

Direct Interactions to Coordinate Autonomic Managers

The two autonomic managers, replica autonomic manager and storage autonomic manager, described above seem to be independent. The first manager restores files and the other manager restores storage. But as we will see in the following example it is possible to have a race condition between the two autonomic managers that will cause the replica autonomic manager to fail. For example, when a resource fails the storage autonomic manager may detect that more storage is needed and start allocating resources and deploying storage components. Meanwhile the replica autonomic manager will be restoring the files that were on the failed resource. The replica autonomic manager might fail to restore the files due to space shortage if the storage autonomic manager is slower and does not have time to finish. This may also prevent the users, temporarily, from storing files.

If the replica autonomic manager would have waited for the storage autonomic manager to finish, it would not fail to recreate replicas. We used direct interaction to coordinate the two autonomic managers by binding the File-Replica-Manager to the Storage-Manager.

Before restoring files the File-Replica-Manager informs the Storage-Manager about the amount of storage it needs to restore files. The Storage-Manager checks available storage and informs the File-Replica-Manager that it can proceed if enough space is available or ask it to wait.

The direct coordination used here does not mean that one manager controls the other. For example if there is only one replica left of a file, the File-Replica-Manager may ignore the request to wait from the Storage-Manager and proceed with restoring the file anyway.

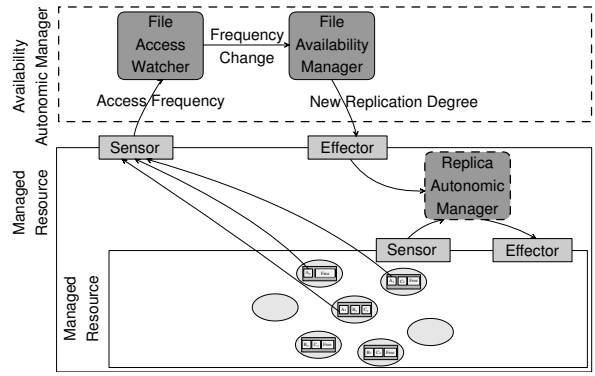

Figure 7.8: Hierarchical management.

Optimising Allocated Storage

Systems should maintain high resource utilization. The storage autonomic manager allocates additional resources if needed to guarantee the ability to store files. However, users might delete files later causing the utilization of the storage space to drop. It is desirable that YASS be able to self-optimize itself by releasing excess resources to improve utilization.

It is possible to design an autonomic manager that will detect low resource utilization, move file replicas stored on a chosen lowly utilized resource, and finally release it. Since the functionality required by this autonomic manager is partially provided by the storage and replica autonomic managers we will try to augment them instead of adding a new autonomic manager, and use stigmergy to coordinate them.

It is easy to modify the storage autonomic manager to detect low storage utilization. The replica manager knows how to restore files. When the utilization of the storage components drops, the storage autonomic manager will detect it and will deallocate some resource. The deallocation of resources will trigger, through stigmergy, another action at the replica autonomic manager. The replica autonomic manager will receive the corresponding resource leave events and will move the files from the leaving resource to other resources.

We believe that this is better than adding another autonomic manager for following two reasons: first, it allows avoiding duplication of functionality; and second, it allows avoiding oscillation between allocation and releasing resources by keeping the decision about the proper amount of storage at one place.

Improving file availability

Popular files should have more replicas in order to increase their availability. A higher level availability autonomic manager can be used to achieve this through

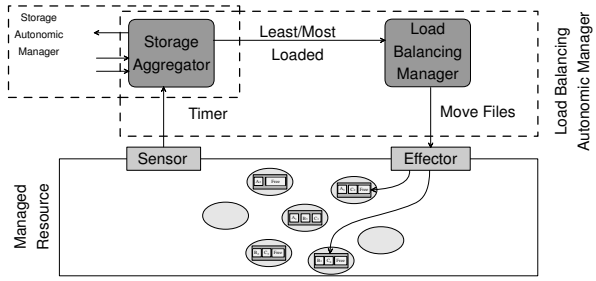

Figure 7.9: Sharing of Management Elements.

regulating the replica autonomic manager. The autonomic manager consists of two management elements. The File-Access-Watcher and File-Availability-Manager shown in Figure 7.8 illustrate hierarchical management.

The File-Access-Watcher monitors the file access frequency. If the popularity of a file changes dramatically it issues a frequency change event. The File-Availability-Manager may decide to change the replication degree of that file. This is achieved by changing the value of the replication degree parameter in the File-Replica-Manager.

Balancing File Storage

A load balancing autonomic manager can be used for self-optimization by trying to lazily balance the stored files among storage components. Since knowledge of current load is available at the Storage-Aggregator, we design the load balancing autonomic manager by sharing the Storage-Aggregator as shown in Figure 7.9.

All autonomic managers we discussed so far are reactive. They receive events and act upon them. Sometimes proactive managers might be also required, such as the one we are discussing. Proactive managers are implemented in Niche using a timer abstraction.

The load balancing autonomic manager is triggered, by a timer, every x time units. The timer event will be received by the shared Storage-Aggregator that will trigger an event containing the most and least loaded storage components. This event will be received by the Load-Balancing-Manager that will move some files from the most to the least loaded storage component.

7.6 Related Work

The vision of autonomic management as presented in [1] has given rise to a number of proposed solutions to aspects of the problem.

An attempt to analyze and understand how multiple interacting loops can manage a single system has been done in [8] by studying and analysing existing systems such as biological and software systems. By this study the authors try to under-

stand the rules of a good control loop design. A study how to compose multiple loops and ensure that they are consistent and complementary is presented in [9]. The authors presented an architecture that supports such compositions.

A reference architecture for autonomic computing is presented in [10]. The authors present patterns for applying their proposed architecture to solve specific problems common to self-managing applications. Behavioural Skeletons is a technique presented in [11] that uses algorithmic skeletons to encapsulate general control loop features that can later be specialized to fit a specific application.

7.7 Conclusions and Future Work

We have presented the methodology of developing the management part of a self-managing distributed application in distributed dynamic environment. We advocate for multiple managers rather than a single centralized manager that can induce a single point of failure and a potential performance bottleneck in a distributed environment. The proposed methodology includes four major design steps: decomposition, assignment, orchestration, and mapping (distribution). The management part is constructed as a number of cooperative autonomic managers each responsible either for a specific management function (according to functional decomposition of management) or for a part of the application (according to a spatial decomposition). We have defined and described different paradigms (patterns) of manager interactions, including indirect interaction by stigmergy, direct interaction, sharing of management elements, and manager hierarchy. In order to illustrate the design steps, we have developed and presented in this paper a self-managing distributed storage service with self-healing, self-configuration and self-optimizing properties provided by corresponding autonomic managers, developed using the distributed component management system Niche. We have shown how the autonomic managers can coordinate their actions, by the four described orchestration paradigms, in order to achieve the overall management objectives.

Dealing with failure of autonomic managers (as opposed to functional parts of the application) is out of the scope of this paper. Clearly, by itself, decentralization of management, might make the application more robust (as some aspects of management continue working, while others stop), but also more fragile due to increased risk of partial failure. In both the centralized and decentralized case, techniques for fault tolerance are needed to insure robustness. Many of these techniques, while ensuring fault recovery do so with some significant delay, in which case a decentralized management architecture may prove advantageous as only some aspects of management are disrupted at any one time.

Our future work includes refinement of the design methodology, further case studies with the focus on orchestration of autonomic managers, investigating robustness of managers by transparent replication of management elements.

Acknowledgements

We would like to thank the Niche research team including Konstantin Popov and Joel Höglund from SICS, and Nikos Parlavantzas from INRIA.

Bibliography

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] IBM, "An architectural blueprint for autonomic computing, 4th edition." `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`, June 2006.

[3] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing* (T. Priol and M. Vanneschi, eds.), pp. 163–174, Springer US, July 2008.

[4] P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy, "The role of overlay services in a self-managing framework for dynamic virtual organizations," in *Making Grids Work* (M. Danelutto, P. Fragopoulou, and V. Getov, eds.), pp. 153–164, Springer US, 2007.

[5] "Niche homepage." `http://niche.sics.se/`.

[6] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The fractal component model," tech. rep., France Telecom R&D and INRIA, Feb. 5 2004.

[7] E. Bonabeau, "Editor's introduction: Stigmergy," *Artificial Life*, vol. 5, no. 2, pp. 95–96, 1999.

[8] P. V. Roy, S. Haridi, A. Reinefeld, J.-B. Stefani, R. Yap, and T. Coupaye, "Self management for large-scale distributed systems: An overview of the self-man project," in *FMCO '07: Software Technologies Concertation on Formal Methods for Components and Objects*, (Amsterdam, The Netherlands), Oct 2007.

[9] S.-W. Cheng, A.-C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste, "An architecture for coordinating multiple self-management systems," in *WICSA '04*, (Washington, DC, USA), p. 243, 2004.

[10] J. W. Sweitzer and C. Draper, "Architecture overview for autonomic computing," in *Autonomic Computing: Concepts, Infrastructure, and Applications* (M. Parashar and S. Hariri, eds.), ch. 5, pp. 71–98, CRC Press, 2006.

[11] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto, "Behavioural skeletons in gcm: Autonomic management of grid components," in *PDP'08*, (Washington, DC, USA), pp. 54–63, 2008.

Paper C

Chapter 8

Policy Based Self-Management in Distributed Environments

Lin Bao, Ahmad Al-Shishtawy, and Vladimir Vlassov

In *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2009)*, (San Francisco, California), September 2009.

Policy Based Self-Management in Distributed Environments

Lin Bao, Ahmad Al-Shishtawy, and Vladimir Vlassov

Royal Institute of Technology (KTH), Stockholm, Sweden
{linb, ahmadas, vladv}@kth.se

Abstract

Currently, increasing costs and escalating complexities are primary issues in the distributed system management. The policy based management is introduced to simplify the management and reduce the overhead, by setting up policies to govern system behaviors. Policies are sets of rules that govern the system behaviors and reflect the business goals or system management objectives.

This paper presents a generic policy-based management framework which has been integrated into an existing distributed component management system, called Niche, that enables and supports self-management. In this framework, programmers can set up more than one Policy-Manager-Group to avoid centralized policy decision making which could become a performance bottleneck. Furthermore, the size of a Policy-Manager-Group, i.e. the number of Policy-Managers in the group, depends on their load, i.e. the number of requests per time unit. In order to achieve good load balancing, a policy request is delivered to one of the policy managers in the group randomly chosen on the fly. A prototype of the framework is presented and two generic policy languages (policy engines and corresponding APIs), namely SPL and XACML, are evaluated using a self-managing file storage application as a case study.

8.1 Introduction

To minimize complexities and overheads of distributed system management, IBM proposed the Autonomic Computing Initiative [1, 2], aiming at developing computing systems which can self-manage themselves. In this work, we address a generic policy-based management framework. Policies are sets of rules which govern the system behaviors and reflect the business goals and objectives. Rules define management actions to be performed under certain conditions and constraints. The key idea of policy-based management is to allow IT administrators to define a set of policy rules to govern behaviors of their IT systems, rather than relying on manually managing or ad-hoc mechanics (e.g. writing customized scripts) [3]. In this way, the complexity of system management can be reduced, and also, the reliability of the system's behavior is improved.

The implementation and maintenance of policies are rather difficult, especially if policies are "hard-coded" (embedded) in the management code of a distributed

system, and the policy logic is scattered in the system implementation. The drawbacks of using "hard-coded" and scattered policy logic are the following: (1) It is hard to trace policies; (2) The application developer has to be involved in implementation of policies; (3) When changing policies, the application has to be rebuilt and redeployed that increases the maintenance overhead. In order to facilitate implementation and maintenance of policies, a language support, including a policy language and a policy evaluation engine, is needed.

This paper presents a generic policy-based management framework which has been integrated into Niche [4, 5], a distributed component management system for development and execution of self-managing distributed applications. The main issues in development of policy-based self-management for a distributed system are programmability, performance and scalability of management. Note that robustness of management can be achieved by replicating management components. Our framework introduces the following key concepts to address above issues: (1) Abstraction of policy that simplifies the modeling and maintenance of policies; (2) Policy Manager Group that allows improving scalability and performance of policy-based management by using multiple managers and achieving good load balance among them; (3) Distributed Policy-Manager-Group Model that allows to avoid centralized policy decision making, which can become a performance bottleneck. We have built a prototype of the policy-based management framework and applied it to a distributed storage service called YASS, Yet Another Storage Service [4, 6] developed using Niche. We have evaluated the performance of policy-based management performed using policy engines, and compared it with the performance of hard-coded management.

The rest of the paper is organized as follows. Section II briefly introduces the Niche platform. In Section III, we describe our policy based management architecture and control loop patterns, and discuss the policy decision making model. We present our policy-based framework prototype and performance evaluation results in Section IV followed by a brief review of some related work in Section V. Finally, Section VI presents some conclusions and directions for our future work.

8.2 Niche: A Distributed Component Management System

Niche [4, 5] is a distributed component management system for development and execution of self-managing distributed systems, services and applications. Niche includes a component-based programming model, a corresponding API, and a runtime execution environment for the development, deployment and execution of self-managing distributed applications. Compared to other existing distributed programming environments, Niche has some features and innovations that facilitate development of distributed systems with robust self-management. In particular, Niche uses a structured overlay network and DHTs that allows increasing the level of distribution transparency in order to enable and to achieve self-management (e.g. component mobility, dynamic reconfiguration) for large-scale distributed systems;

Niche leverages self-organizing properties of the structured overlay network, and provides support for transparent replication of management components in order to improve robustness of management.

Niche separates the programming of functional and management (self-*) parts of a distributed system or application. The functional code is developed using the Fractal component model [7] extended with the concept of component groups and bindings to groups. A Fractal component may contain a client interface (used by the component) and/or a server interface (provided by the component). Components interact through bindings. A binging connects a client interface of one component to a server interface of another component (or component group). The component group concept brings on two communication patterns "one-to-all" and "one-to-any". A component, which is bound to a component group with a one-to-any binding, communicates with any (but only one) component randomly and transparently chosen from the group on the fly. A component, which is bound to a group with a one-to-all binding, communicates with all components in that group at once, i.e. when the component invokes a method on the group interface bound with one-to-all binding, all components, members of the group, receive the invocation. The abstraction of groups and group communication facilitates programming of both functional and self management parts, and allows improving scalability and robustness of management.

The self-* code is organized as a network of distributed management elements (MEs) (Figure 8.1) communicating with each other through events. MEs are subdivided into Watchers (W), Aggregators (Aggr), Managers (Mgr) and Executors, depending on their roles in the self-* code. Watchers monitor the state of the managed application and its environment, and communicate monitored information to Aggregators, which aggregate the information, detect and report symptoms to Managers. Managers analyze the symptoms, make decisions and request Executors to perform management actions.

8.3 Niche Policy Based Management

Architecture

Figure 8.2 shows the conceptual view of policy based management architecture. The main elements are described below.

A Watcher (W) is used to monitor a managed resource[1] or a group of managed resources through sensors that are placed on managed resources. Watchers will collect monitored information and report to an Aggregator.

Aggregators (Aggr) aggregate, filter and analyze the information collected from Watchers or directly from sensors. When a policy decision is possibly needed, the

[1]Further in the paper, we call, for short, *resource* any entity or part of an application and its execution environment, which can be monitored and possibly managed, e.g. component, component group, binging, component container, etc.

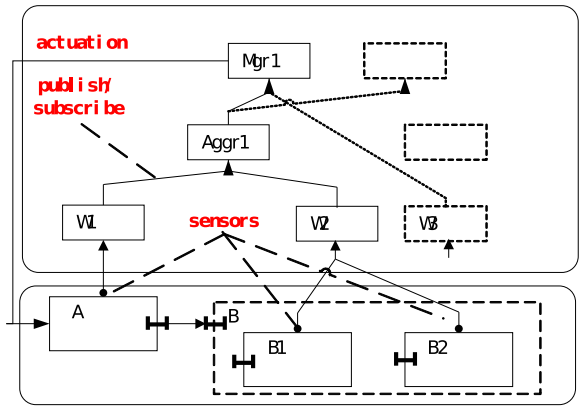

Figure 8.1: Niche Management Elements

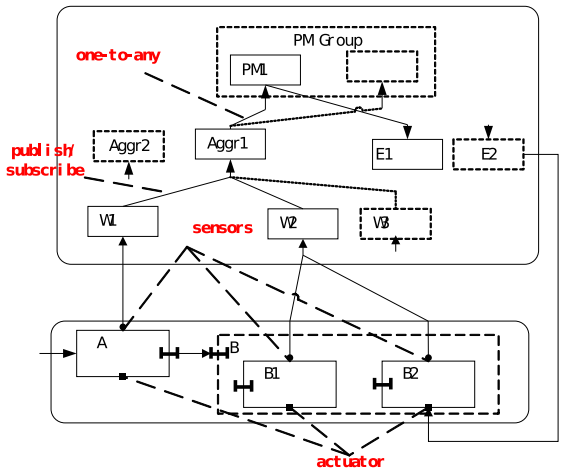

Figure 8.2: Policy Based Management Architecture

aggregator will formulate a policy request event and send it to the Policy-Manager-Group through one-to-any binding.

Policy-Managers (PM) take the responsibility of loading policies from the policy repository, making decisions on policy request events, and delegating the obligations to Executors (E) in charge. Obligations are communicated from Policy-Managers to Executors in the form of policy obligation events.

Niche achieves reliability of management by replicating management elements.

For example, if a Policy-Manager fails when evaluating a request against policies, one of its replicas takes its responsibility and continues with the evaluation.

Executors execute the actions, dictated in policy-obligation-events, on managed resources through actuators deployed on managed resources.

Special Policy-Watchers monitor the policy repositories and policy configuration files. On any change in the policy repositories or policy configuration files (e.g. a policy configuration file has been updated), a Policy-Watcher issues a Policy-Change-Event and sends it to the Policy-Manager-Group through the one-to-all binding. Upon receiving the Policy-Change-Event, all Policy-Managers reload policies. This allows administrators to change policies on the fly.

Policy-Manager-Group is a group of Policy-Managers, which are loaded with the same set of policies. Niche is a distributed component platform. In the distributed system, a single Policy-Manager, governing system behaviors, will be a performance bottleneck, since every request will be forwarded to it. It is allowed in Niche to have more than one Policy-Manager-Group in order to avoid the potential bottleneck with centralized decision making. Furthermore, the size of Policy-Manager-Group, that is, the number of Policy-Managers it consists of, depends on its load, i.e. the intensity of requests (the number of requests per time unit). When a particular Policy-Manager-Group is highly loaded, the number of Policy-Managers is increased in order to reduce burdens of each member. Niche allows changing the group members transparently without affecting components bound to the group.

A Local-Conflicts-Detector checks that the new or modified policy does not conflict with any existing local policy for a given Policy-Manager-Group. There might be several Local-Conflicts-Detectors, one per Policy-Manager-Group. A Global-Conflicts-Detector checks whether the new policy conflicts with other policies in a global system-wise view.

Policy-Based Management Control Loop

Self-management behaviors can be achieved through control loops. A control loop keeps watching states of managed resources and acts accordingly. In policy-based management architecture described above, a control loop is composed of Watchers, Aggregators, a Policy-Manager-Group and Executors (Figure 8.2). Note that the Policy-Manager-Group plays a role of Manager (see Figure 8.1).

Watchers deploy sensors on managed resources to monitor their states, and report changes to Aggregators that communicate policy request events to the Policy-Manager-Group using one-to-any bindings. Upon receiving a policy request event, the randomly chosen Policy-Manager retrieves applicable policies, along with any information required for policy evaluation, and evaluates policies with information available.

Based on rules and actions prescribed in the policy, the Policy-Manager will choose the relevant change plan and delegate to executor in charge. The executor executes the plan on the managed resource through actuators.

Policy-Manager-Group Model

Our framework allows programmers to define one or more Policy-Manager-Groups to govern system behaviors. There are two ways of making decisions in policy management groups: centralized and distributed.

In the centralized model of Policy-Manager-Group, there is only one Policy-Manager-Group formed by all Policy-Managers with common policies. The centralized model is easy to implement, and it needs only one Local-Conflict-Detector and one Policy-Watcher. However, a centralized decision making can become a performance bottleneck in policy based management for a distributed system. Furthermore, management should be distributed, based on spatial and functional partitioning, in order to improve scalability, robustness and performance of management. The distribution of management should match and correspond to architecture of the system being managed, taking into account its structure, location of its components, physical network connectivity, management structure of an organization where the system is used.

In the distributed model of Policy-Manager-Group, each policy manager knows only partial policies of the whole system. Policy managers with common policies form a policy-manager group associated with a Policy-Watcher. There are several advantages of the distributed model. First, it is a more natural way to realize policy based management. For the whole system, global policies are applied to govern system behaviors. For different groups of components, local polices are governing their behaviors based on the hardware platforms and operating systems they are working on. Second, this model is more efficient and scalable. Policy-managers reading and evaluating fewer policies will shorten the evaluation time. However, policy managers from different groups need to coordinate their actions in order to finish policy evaluation when the policy request is unknown to a policy manager, which, in this case, needs to ask another policy manager from a different group. Any form of coordination is a lost to performance. Last, the distributed model of policy-based management is more secure. Not all policies should be exposed to every policy manager. Since some policies contain information on the system parameters, they should be protected against malicious users. Furthermore, both Global-Conflict-Detector and Local-Conflict-Detector are needed to detect whether or not a newly added, changed or deleted policy is in conflict with other policies for the whole system or a given policy-manager-group.

8.4 Niche Policy-based Management Framework Prototype

We have built a prototype of our policy-based management framework for the Niche distributed component management system by using policy engines and corresponding APIs for two policy languages XACML (eXtensible Access Control Markup Language) [8, 9] and SPL (Simplified Policy Language) [10, 11].

We have had several reasons for choosing these two languages for our framework. Each of the languages is supported with a Java-implemented policy engine;

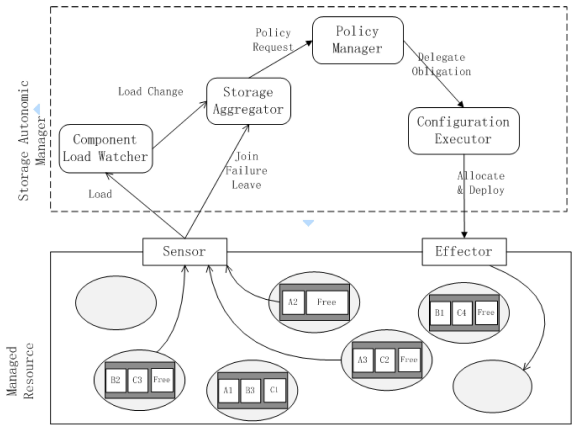

Figure 8.3: YASS self-configuration control loop

this makes it easier to integrate the policy engines into our Java-based Niche platform. Both languages allow defining policy rules (rules with obligations in XACML, or decision statements in SPL) that dictate the management actions to be enforced on managed resources by executors. SPL is intended for management of distributed system. Although XACML was designed for access control rather than for management, its support for obligations can be easily adopted for management of distributed system.

In order to test and evaluate our framework, we have applied it to YASS, Yet Another Storage Service [4, 6], which is a self-managing storage service with two control loops, one for self-healing (to maintain a specified file replication degree in order to achieve high file availability in presence of node churn) and one for self-configuration (to adjust amount of storage resources according to load changes). For example, the YASS self-configuration control loop consists of Component-Load-Watcher, Storage-Aggregator, Policy-Manager and Configuration-Executor as depicted in Figure 8.3. The Watcher monitors the free storage space in the storage group and reports this information to Storage-Aggregator. The Aggregator computes the total capacity and total free space in the group and informs Policy-Manager when the capacity and/or free space drop below predefined thresholds. The Policy-Manager evaluates the event according to the configuration policy and delegates the management obligations to Executor, which tries to allocate more resources and deploy additional storage components on them in order to increase capacity and/or free space.

In the initial implementation of YASS, all management was coded in Java; whereas in the policy-based implementation, a part of management was expressed in a policy language (XACML or SPL).

		Policy Load	First evaluation	Second evaluation
XACML	MAX	379	36	7
	MIN	168	11	1
	AVG	246.8	18.9	3
SPL	MAX	705	7	7
	MIN	368	3	2
	AVG	487.4	5.7	5.7
Java	AVG	—	≈0	≈0

Table 8.1: Policy Evaluation Result (in milliseconds)

		Policy Re-Load	1st evaluation	2nd evaluation
XACML	MAX	27	4	5
	MIN	23	1	1
	AVG	24.5	3	3
SPL	MAX	62	8	6
	MIN	53	2	2
	AVG	56.3	5.8	5.3

Table 8.2: Policy Reload Result (in milliseconds)

We have used YASS as a use case in order to evaluate expressiveness of different policy languages, XACML and SPL, and the performance of policy-based management compared with hard-coded Java implementation of management. It is worth mentioning that a hard-coded manager, unless specially designed, does not allow changing policies on the fly.

In the current version, for quick prototyping, we set up only one Policy-Manager, which can be a performance bottleneck when the application scales. We have evaluated the performance of our prototype (running YASS) by measuring the average policy evaluation times of XACML and SPL policy managers. We have compared performance of both policy managers with the performance of the hard-coded manager explained above. The evaluation results Table 8.1 show that the hard-coded management implementation performs better (as expected) than the policy-based management implementation. Therefore, it could be recommended to use policy-based management framework to implement less performance-demanding managers with policies or objectives that need to be changed on the fly. The time needed to reload the policy file by both XACML and SPL policy managers is shown in Table 8.2. From these results we have observed that the XACML management implementation is slightly faster than the SPL management implementation; however, on the other hand, in our opinion based on our experience, SPL policies was easier to write and implement than XACML policies.

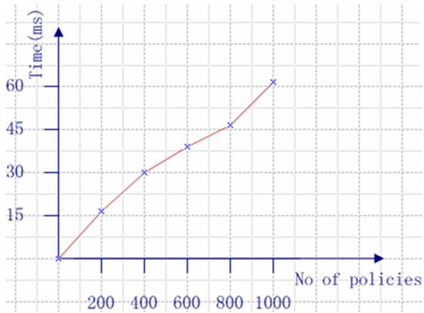

Figure 8.4: XACML policy evaluation results

Scalability Evaluation using Synthetic Policies

The current version of YASS is a simple storage service and its self-management requires a small number of management policies (policy rules) governing the whole application. It is rather difficult to find a large number of real-life policies. To further compare the performance and scalability of management using XACML and SPL policy engines, we have generated dummy synthetic policies in order to increase the size of the policy set, i.e. the number of policies to be evaluated on a management request. In order to force policy engines to evaluate all synthetic policies (rules), we have applied the Permit-Overrides rule combining algorithm for XACML policies, where a permitting rule was the last in evaluation, and the Execute_All_Applicable strategy for SPL policies.

Figure 8.4 shows the XACML preprocessing time versus the number of policies in a one-layered policy. We observe that there is an almost linear correlation between the preprocessing time of XACML and the number of rules. This result demonstrates that the XACML-based implementation is scalable in the preprocessing phase.

Figure 8.5 shows the processing time of SPL versus the number of policies. We observe that there is almost exponential correlation between the processing time of SPL and the number of policies. This result demonstrates that the SPL-based implementation is not scalable in the processing time.

8.5 Related Work

Policy Management for Autonomic Computing (PMAC) [12, 13] provides the policy language and mechanisms needed to create and enforce these policies for managed resources. PMAC is based on a centralized decision maker Autonomic Manager and all policies are stored in a centralized policy repository. Ponder2 [14] is a self-contained, stand-alone policy system for autonomous pervasive environments.

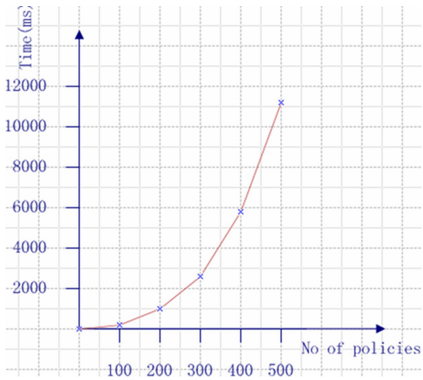

Figure 8.5: SPL policy evaluation results

It eliminates some disadvantages of its predecessor Ponder. First, it supports distributed provision and decision making. Second, it does not depend on a centralized facility, such as LDAP or CIM repositories. Third, it is able to scale to small devices as needed in pervasive systems.

8.6 Conclusions and Future Work

This paper proposed a policy based framework which facilitates distributed policy decision making and introduces the concept of Policy-Manager-Group that represents a group of policy-based managers formed to balance load among Policy-Managers.

Policy-based management has several advantages over hard-coded management. First, it is easier to administrate and maintain (e.g. change) management policies than to trace the hard-coded management logic scattered across codebase. Second, the separation of policies and application logic (as well as low-level hard-coded management) makes the implementation easier, since the policy author can focus on modeling policies without considering the specific application implementation, while application developers do not have to think about where and how to implement management logic, but rather have to provide hooks to make their system manageable, i.e. to enable self-management. Third, it is easier to share and reuse the same policy across multiple different applications and to change the policy consistently. Finally, policy-based management allows policy authors and administrators to edit and to change policies on the fly (at runtime).

From our evaluation results, we can observe that the hard-coded management performs better than the policy-based management, which uses a policy engine. Therefore, it could be recommended to use policy-based management in less performance demanding managers with policies or management objectives that need

to be changed on the fly (at runtime).

Our future work includes implementation of Policy-Manager-Group in the prototype. We also need to define a coordination mechanism for Policy-Manager-Groups, and to find an approach to implement the local conflict detector and the global conflict detector. Finally, we need to specify how to divide the realm of each Policy-Manager-Group governs.

Bibliography

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] IBM, "An architectural blueprint for autonomic computing, 4th edition." `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`, June 2006.

[3] D. Agrawal, J. Giles, K. Lee, and J. Lobo, "Policy ratification," in *Policies for Distributed Systems and Networks, 2005. Sixth IEEE Int. Workshop* (T. Priol and M. Vanneschi, eds.), pp. 223– 232, June 2005.

[4] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing* (T. Priol and M. Vanneschi, eds.), pp. 163–174, Springer US, July 2008.

[5] "Niche homepage." `http://niche.sics.se/`.

[6] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Distributed control loop patterns for managing distributed applications," in *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008)*, (Venice, Italy), pp. 260–265, Oct. 2008.

[7] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The fractal component model," tech. rep., France Telecom R&D and INRIA, Feb. 5 2004.

[8] "Oasis extensible access control markup language (xacml) tc." `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#expository`.

[9] "Sun's xacml programmers guide." `http://sunxacml.sourceforge.net/guide.html`.

[10] "Spl language reference." `http://incubator.apache.org/imperius/docs/spl_reference.html`.

[11] D. Agrawal, S. Calo, K.-W. Lee, J. Lobo, and T. W. Res., "Issues in designing a policy language for distributed management of it infrastructures," in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium*, pp. 30–39, June 2007.

[12] IBM, "Use policy management for autonomic computing." `https://www6.software.ibm.com/developerworks/education/ac-guide/ac-guide-pdf.pdf`, April 2005.

[13] D. Kaminsky, "An introduction to policy for autonomic computing." `http://www.ibm.com/developerworks/autonomic/library/ac-policy.html`, March 2005.

[14] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A policy system for autonomous pervasive environments," in *Autonomic and Autonomous Systems, 2009. ICAS '09. Fifth International Conference*, pp. 330–335, April 2009.

Part III

Technical Report

Paper D

Chapter 9

Achieving Robust Self-Management for Large-Scale Distributed Applications

Ahmad Al-Shishtawy, Muhammad Asif Fayyaz, Konstantin Popov, and Vladimir Vlassov

Technical Report T2010:02, Swedish Institute of Computer Science, March 2010

Achieving Robust Self-Management for Large-Scale Distributed Applications

Ahmad Al-Shishtawy[1,2], Muhammad Asif Fayyaz[1], Konstantin Popov[2], and Vladimir Vlassov[1]

[1] Royal Institute of Technology, Stockholm, Sweden
{ahmadas, mafayyaz, vladv}@kth.se

[2] Swedish Institute of Computer Science, Stockholm, Sweden
{ahmad, kost}@sics.se

March 4, 2010

SICS Technical Report T2010:02
ISSN: 1100-3154

Abstract

Autonomic managers are the main architectural building blocks for constructing self-management capabilities of computing systems and applications. One of the major challenges in developing self-managing applications is robustness of management elements which form autonomic managers. We believe that transparent handling of the effects of resource churn (joins/leaves/-failures) on management should be an essential feature of a platform for self-managing large-scale dynamic distributed applications, because it facilitates the development of robust autonomic managers and hence improves robustness of self-managing applications. This feature can be achieved by providing a robust management element abstraction that hides churn from the programmer.

In this paper, we present a generic approach to achieve robust services that is based on finite state machine replication with dynamic reconfiguration of replica sets. We contribute a decentralized algorithm that maintains the set of nodes hosting service replicas in the presence of churn. We use this approach to implement robust management elements as robust services that can operate despite of churn. Our proposed decentralized algorithm uses peer-to-peer replica placement schemes to automate replicated state machine migration in order to tolerate churn. Our algorithm exploits lookup and failure detection facilities of a structured overlay network for managing the set of active replicas. Using the proposed approach, we can achieve a long running and highly available service, without human intervention, in the presence of resource churn. In order to validate and evaluate our approach, we have implemented a prototype that includes the proposed algorithm.

9.1 Introduction

Autonomic computing [1] is an attractive paradigm to tackle management overhead of complex applications by making them self-managing. Self-management, namely self-configuration, self-optimization, self-healing, and self-protection, is achieved through autonomic managers [2]. An autonomic manager continuously monitors hardware and/or software resources and acts accordingly. Autonomic computing is particularly attractive for large-scale and/or dynamic distributed systems where direct human management might not be feasible.

In our previous work, we have developed a platform called Niche [3, 4] that enables us to build self-managing large-scale distributed systems. Autonomic managers play a major rule in designing self-managing systems [5]. An autonomic manager in Niche consists of a network of management elements (MEs). Each ME is responsible for one or more roles in the construction the Autonomic Manager. These roles are: Monitor, Analyze, Plan, and Execute (the MAPE loop [2]). In Niche, MEs are distributed and interact with each other through events (messages) to form control loops.

Large-scale distributed systems are typically dynamic with resources that may fail or join/leave the system at any time (resource churn). Constructing autonomic managers in dynamic environments with high resource churn is challenging because MEs need to be restored with minimal disruption to the autonomic manager, whenever the resource where MEs execute leaves or fails. This challenge increases if the MEs are stateful because the state needs to be maintained consistent.

We propose a Robust Management Element (RME) abstraction that developers can use if they need their MEs to tolerate resource churn. The RME abstraction allows simplifying the development of robust autonomic managers that can tolerate resource churn, and thus self-managing large-scale distributed systems. This way developers of self-managing systems can focus on the functionality of the management without the need to deal with failures. A Robust Management Element should: 1) be replicated to ensure fault-tolerance; 2) survive continuous resource failures by automatically restoring failed replicas on other nodes; 3) maintain its state consistent among replicas; 4) provide its service with minimal disruption in spite of resource join/leave/fail (high availability). 5) be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location). Because we are targeting large-scale distributed environments with no central control, such as peer-to-peer networks, all algorithms should operate in decentralized fashion.

In this paper, we present our approach to achieving RMEs that is based on finite state machine replication with automatic reconfiguration of replica sets. We replicate MEs on a fixed set of nodes using the *replicated state machine* [6, 7] approach. However, replication by itself is not enough to guarantee long running services in the presence of continuous churn. This is because the number of failed nodes (that host ME replicas) will increase with time. Eventually this will cause the service to stop. Therefor, we use *service migration* [8] to enable the reconfiguration the set

of nodes hosting ME replicas. Using service migration, new nodes can be introduced to replace the failed ones. We propose a decentralized algorithm, based on *Structured Overlay Networks* (SONs) [9], that will use migration to *automatically* reconfigure the set of nodes where the ME replicas are hosted. This will guarantee that the service provided by the RME will tolerate continuous churn. The reconfiguration take place by migrating MEs when needed to new nodes. The major use of SONs in our approach is as follows: first, to maintain location information of the replicas using replica placement schemes such as symmetric replication [10]; second, to detect the failure of replicas and to make a decision to migrate in a decentralized manner; third, to allow clients to locate replicas despite of churn.

The rest of this paper is organised as following: Section 9.2 presents the necessary background required to understand the proposed algorithm. In Section 9.3, we describe our proposed decentralized algorithm to automate the migration process. Followed by applying the algorithm to the Niche platform to achieve RMEs in Section 9.4. Finally, conclusions and future work are discussed in Section 9.5.

9.2 Background

This section presents the necessary background to understand the approach and algorithm presented in this paper, namely: The Niche platform, Symmetric replication scheme, replicated state machines, and an approach to migrate stateful services.

Niche Platform

Niche [3] is a distributed component management system that implements the autonomic computing architecture [2]. Niche includes a distributed component programming model, APIs, and a run-time system including deployment service. The main objective of Niche is to enable and to achieve self-management of component-based applications deployed on a dynamic distributed environments where resources can join, leave, or fail. A self-managing application in Niche consists of functional and management parts. Functional components communicate via interface bindings, whereas management components communicate via a publish/subscribe event notification mechanism.

The Niche run-time environment is a network of distributed containers hosting functional and management components. Niche uses a Chord [9] like structured overlay network (SON) as its communication layer. The SON is self-organising on its own and provides overlay services used by Niche such as name-based communication, distributed hash table (DHT) and a publish/subscribe mechanism for event dissemination. These services are used by Niche to provide higher level communication abstractions such as name-based bindings to support component mobility; dynamic component groups; one-to-any and one-to-all group bindings, and event based communication.

Structured Overlay Networks

We assume the following model of Structured Overlay Networks (SONs) and their APIs. We believe, this model is representative, and in particular it matches the Chord SON. In the model, SON provides the operation to locate items on the network. For example, items can be data items for DHTs, or some compute facilities that are hosted on individual nodes in a SON. We say that the node hosting or providing access to an item is responsible for that item. Both items and nodes posses unique SON identifiers that are assigned from the same name space. The SON automatically and dynamically divides the responsibility between nodes such that there is always a responsible node for every SON identifier. SON provides a 'lookup' operation that returns the address of a node responsible for a given SON identifier. Because of churn, node responsibilities change over time and, thus, 'lookup' can return over time different nodes for the same item. In practical SONs the 'lookup' operation can also occasionally return wrong (inconsistent) results. Further more, SON can notify application software running on a node when the responsibility range of the node changes. When responsibility changes, items need to be moved between nodes accordingly. In Chord-like SONs the identifier space is circular, and nodes are responsible for items with identifiers in the range between the node's identifier and the identifier of the predecessor node. Finally, a SON with a circular identifier space naturally provides for symmetric replication of items on the SON - where replica IDs are placed symmetrically around the identifier space circle.

Symmetric Replication [10] is a scheme used to determine replica placement in SONs. Given an item ID i and a replication degree f, symmetric replication can be used to calculate the IDs of the item's replicas. The ID of the x-th ($1 \leq x \leq f$) replica of the item i is computed according to the following formula:

$$r(i, x) = (i + (x - 1)N/f) \bmod N \tag{9.1}$$

where N is the size of the identifier space.

The IDs of replicas are independent from the nodes present in the system. A *lookup* is used to find the node responsible node for hosting an ID. For the symmetry requirement to always be true, it is required that the replication factor f divides the size of the identifier space N.

Replicated State Machines

A common way to achieve high availability of a service is to replicate it on several nodes. Replicating stateless services is relatively simple and not considered in this paper. A common way to replicate stateful services is to use the replicated state machine approach [6]. In this approach several nodes (replicas) run the same service in order for service to survive node failures.

Using the replicated state machine approach requires the service to be deterministic. A set of deterministic services will have the same state change and produce

the same output given the same sequence of inputs (requests or commands) and initial state. This means that the service should avoid sources of nondeterminism such as using local clocks, random numbers, and multi-threading.

Replicated state machines, given a deterministic service, can use the Paxos [7] algorithm to ensure that all services execute the same input in the same order. The Paxos algorithm relies on a leader election algorithm [11] that will elect one of the replicas as the leader. The leader ensures the order of inputs by assigning client requests to slots. Replicas execute input sequentially i.e. a replica can execute input from slot $n + 1$ only if it had already executed input from slot n.

The Paxos algorithm can tolerate replica failures and still operate correctly as long as the number of failures is less than half of the total number of replicas. This is because Paxos requires that there will always be a *quorum* of alive replicas in the system. The size of the quorum is $(R/2) + 1$, where R is the initial number of replicas in the system. In this paper, we consider only fail-stop model (i.e., a replica will fail only by stopping) and will not consider other models such as Byzantine failures.

Migrating Stateful Services

SMART [8] is a technique for changing the set of nodes where a replicated state machine runs, i.e. migrate the service. The fixed set of nodes, where a replicated state machine runs, is called a *configuration*. Adding and/or removing nodes (replicas) in a configuration will result in a new configuration.

SMART is built on the migration technique outlined in [7]. The idea is to have the current configuration as part of the service state. The migration to a new configuration happens by executing a special request that causes the current configuration to change. This request is like any other request that can modify the state when executed. The change does not happen immediately but scheduled to take effect after α slots. This gives the flexibility to pipeline α concurrent requests to improve performance.

The main advantage of SMART over other migration technique is that it allows to replace non-failed nodes. This enables SMART to rely on an automated service (that may use imperfect failure detector) to maintain the configuration by adding new nodes and removing suspected ones.

An important feature of SMART is the use of configuration-specific replicas. The service migrates from `conf1` to `conf2` by creating a new independent set of replicas in `conf2` that run in parallel with replicas in `conf1`. The replicas in `conf1` are kept long enough to ensure that `conf2` is established. This simplify the migration process and help SMART to overcome problems and limitations of other techniques. This approach can possibly result in many replicas from different configurations to run on the same node. To improve performance, SMART uses a shared execution module that holds the state and is shared among replicas on the same node. The execution module is responsible for modifying the state by executing assigned requests sequentially and producing output. Other that that each

configuration runs its own instance of the Paxos algorithm independently without any sharing. This makes it, from the point of view of the replicated state machine instance, look like as if the Paxos algorithm is running on a static configuration.

Conflicts between configurations are avoided by assigning a non-overlapping range of slots [FirstSlot, LastSlot] to each configuration. The FirstSlot for `conf1` is set to 1. When a configuration change request appears at slot n this will result in setting LastSlot of current configuration to $n + \alpha - 1$ and setting the FirstSlot of the next configuration to $n + \alpha$.

Before a new replica in a new configuration can start working it must acquire a state from another replica that is at least FirstSlot-1. This can be achieved by copying the state from a replica from the previous configuration that has executed LastSlot or from a replica from the current configuration. The replicas from the previous configuration are kept until a majority of the new configuration have initialised their state.

9.3 Automatic Reconfiguration of Replica Sets

In this section we present our approach and associated algorithm to achieve robust services. Our algorithm automates the process of selecting a replica set (configuration) and the decision of migrating to a new configuration in order to tolerate resource churn. This approach, our algorithm together with the replicated state machine technique and migration support, will provide a robust service that can tolerate continuous resource churn and run for long period of time without the need of human intervention.

Our approach was mainly designed to provide Robust Management Elements (RMEs) abstraction that is used to achieve robust self-management. An example is our platform Niche [3, 4] where this technique is applied directly and RMEs are used to build robust autonomic managers. However, we believe that our approach is generic enough and can be used to achieve other robust services. In particular, we believe that our approach is suitable for structured P2P applications that require highly available robust services.

Replicated (finite) state machines (RSM) are identified by a constant SON ID, which we denote as RSMID in the following. RSMIDs permanently identify RSMs regardless of node churn that causes reconfiguration of sets of replicas in RSMs. Clients that send requests to RSM need to know only its RSMID and replication degree. With this information clients can calculate identities of individual replicas according to the symmetric replication scheme, and lookup the nodes currently responsible for the replicas. Most of the nodes found in this way will indeed host up-to-date RSM replicas - but not necessarily all of them because of lookup inconsistency and node churn.

Failure-tolerant consensus algorithms like Paxos require a fixed set of known replicas we call configuration in the following. Some of replicas, though, can be temporarily unreachable or down (the crash-recovery model). The SMART protocol

extends the Paxos algorithm to enable explicit reconfiguration of replica sets. Note that RSMIDs cannot be used for neither of the algorithms because the lookup operation can return over time different sets of nodes. In the algorithm we contribute for management of replica sets, individual RSM replicas are mutually identified by their addresses which in particular do not change under churn. Every single replica in a RSM configuration knows addresses of all other replicas in the RSM.

The RSM, its clients and the replica set management algorithm work roughly as follows. First, a dedicated initiator chooses RSMID, performs lookups of nodes responsible for individual replicas and sends to them a request to create RSM replicas. Note the request contains RSMID and all replica addresses (configuration), thus newly created replicas perceive each other as a group and can communicate with each other. RSMID is also distributed to future RSM clients. Whenever clients need to contact a RSM, they resolve the RSMID similar to the initiator and multicast their requests to obtained addresses.

Because of churn, the set of nodes responsible for individual RSM replicas changes over time. In response, our distributed configuration management algorithm creates new replicas on nodes that become responsible for RSM replicas, and eventually deletes unused ones. The algorithm runs on all nodes of the overlay and uses several sources of events and information, including SON node failure notifications, SON notifications about change of responsibility, and messages from clients. We discuss the algorithm in greater detail in the following.

Our algorithm is built on top of Structured Overlay Networks (SONs) because of their self-organising features and resilience under churn [12]. The algorithm exploits lookup and failure detection facilities of SONs for managing the set of active replicas. Replica placement schemes such as symmetric replication [10] is used to maintain location information of the replicas. This is used by the algorithm to select replicas in the replica set and is used by the clients to determine replica locations in order to use the service. Join, leave, and failure events are used to make a decision to migrate in a decentralized manner. Other useful operations, that can be efficiently built on top of SONs, include multi-cast and range-cast. We use these operations to recover from replica failures.

State Machine Architecture

The replicated state machine (RSM) consists of a set of replicas, which forms a configuration. Migration techniques can be used to change the configuration (the replica set). The architecture of a replica (a state machine) that supports migration is shown in Fig. 9.1. The architecture uses the shared execution module optimization presented in [8]. This optimization is useful when the same replica participate in multiple configurations. The execution module captures the logic of the service. The execution module executes requests. The execution of a request may result in state change, producing output, or both. The execution module should be a deterministic program. Its outputs and states must depend only on the sequence of input and the initial state. The execution module is also required to support

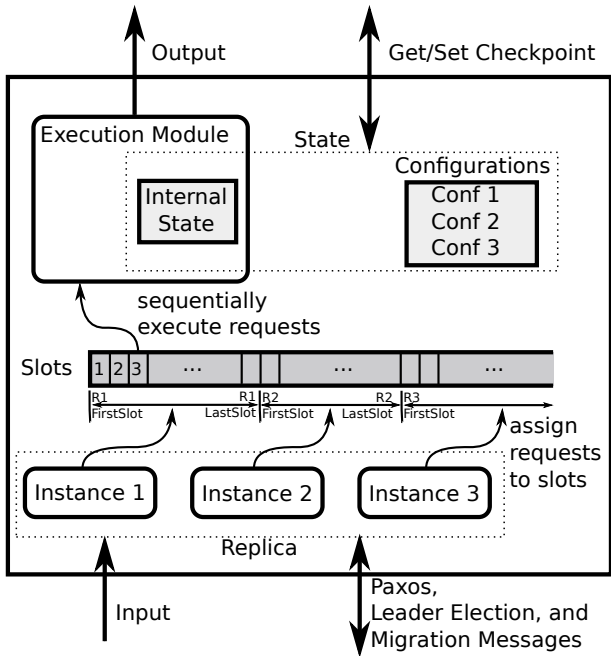

Figure 9.1: State Machine Architecture: Each machine can participate in more than one configuration. A new replica instance is assigned to each configuration. Each configuration is responsible for assigning requests to a none overlapping range of slot. The execution module executes requests sequentially that can change the state and/or produce output.

checkpointing. That is the state can be externally saved and restored. This enables us to transfer states between replicas. The execution module executes all requests except the `ConfChange` request which is handled by the state machine.

The state of a replica consists of two parts: The first part is internal state of the execution module which is application specific; The second part is the configuration. A configuration is represented by an array of size f where f is the replication degree. The array holds direct *references* (IP and port) to the nodes that form the configuration. The reason to split the state in two parts, instead of keeping the configuration in the execution module, is to make the development of the execution module independent from the replication technique. In this way legacy services, that are already developed, can be replicated without modification given that they satisfy execution module constraints.

The remaining parts of the SM, other than the execution module, are responsible to run the replicated state machine algorithms (Paxos and Leader Election) and the migration algorithm (SMART). As described in the previous section, each

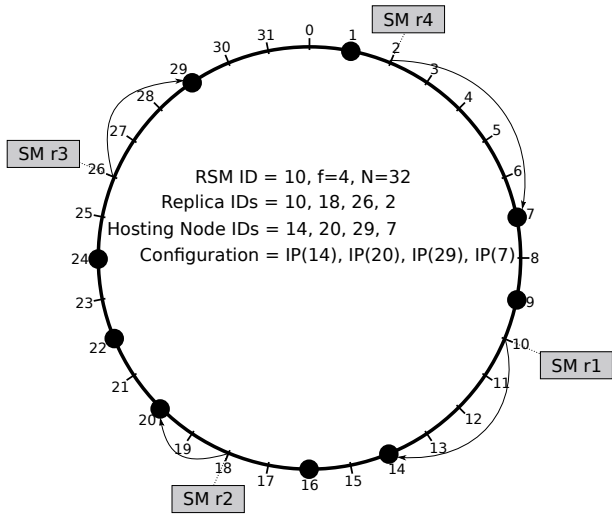

Figure 9.2: Replica Placement Example: Replicas are selected according to the symmetric replication scheme. A Replica is hosted (executed) by the node responsible for its ID (shown by the arrows). A configuration is a fixed set of direct references (IP address and port) to nodes that hosted the replicas at the time of configuration creation. The RSM ID and Replica IDs are fixed and do not change for the entire life time of the service. The Hosted Node IDs and Configuration are only fixed for a single configuration. Black circles represent physical nodes in the system.

configuration is assigned a separate instance of the replicated state machine algorithms. The migration algorithm is responsible for specifying the `FirstSlot` and `LastSlot` for each configuration, starting new configurations when executing `ConfChange` requests, and destroying old configurations after a new configuration is established.

Configurations and Replica Placement Schemes

All nodes in the system are part of a structured overlay network (SON) as shown in Fig. 9.2. The Replicated State Machine that represents the service is assigned a random ID $RSMID$ from the identifier space N. The set of nodes that will form a configuration are selected using the symmetric replication technique [10]. The symmetric replication, given the replication factor f and the $RSMID$, is used to calculate the *Replica IDs* according to equation 9.1. Using the `lookup()` operation, provided by the SON, we can obtain the IDs and direct references (IP and port) of the responsible nodes. These operations are shown in Algorithm 9.1.

The use of direct references, instead of using lookup operations, as the con-

Algorithm 9.1 Helper Procedures

```
1: procedure GetConf(RSMID)
2:     ids[ ] ← GetReplicaIDs(RSMID)
                                              ▷ Replica Item IDs
3:     for i ← 1, f do
4:         refs[i] ← Lookup(ids[i])
5:     end for
6:     return refs[ ]
7: end procedure

8: procedure GetReplicaIDs(RSMID)
9:     for x ← 1, f do
10:        ids[x] ← r(RSMID, x)                ▷ See equation 9.1
11:    end for
12:    return ids[ ]
13: end procedure
```

figuration is important for our approach to work for two reasons. First reason is that we can not rely on the lookup operation because of the lookup inconsistency problem. The lookup operation, used to find the node responsible for an ID, may return incorrect references. These incorrect references will have the same effect in the replicatino algorithm as node failures even though the nodes might be alive. Thus the incorrect references will reduce the fault tolerance of the replication service. Second reason is that the migration algorithm requires that both the new and the previous configurations coexist until the new configuration is established. Relying on lookup operation for `replica_IDs` may not be possible. For example, in Figure 9.2, when a node with $ID = 5$ joins the overlay it becomes responsible for the replica `SM_r4` with $ID = 2$. A correct `lookup(2)` will always return 5. Because of this, the node 7, from the previous configuration, will never be reached using the lookup operation. This can also reduce the fault tolerance of the service and prevent the migration in the case of large number of joins.

Nodes in the system may join, leave, or fail at any time. According to the Paxos requirements, a configuration can survive the failure of less than half of the nodes in the configuration. In other words, $f/2 + 1$ nodes must be alive for the algorithm to work. This must hold independently for each configuration. After a new configuration is established, it is safe to destroy instances of older configurations.

Due to churn, the responsible node for a certain SM may change. For example in Fig.9.2 if node 20 fails then node 22 will become responsible for identifier 18 and should host `SM_r2`. Our algorithm, described in the remainder of this section, will automate migration process by triggering `ConfChange` requests when churn changes responsibilities. This will guarantee that the service provided by the RSM will tolerate churn.

Algorithm 9.2 Replicated State Machine API

```
1: procedure CreateRSM(RSMID)
                                        ▷ Creates a new replicated state machine
2:     Conf[ ] ← GetConf(RSMID)
                                                          ▷ Hosting Node REFs
3:     for i ← 1, f do
4:         sendto Conf[i] : InitSM(RSMID, i, Conf)
5:     end for
6: end procedure

7: procedure JoinRSM(RSMID, rank)
8:     SubmitReq(RSMID, ConfChange(rank, MyRef))
   ▷ The new configuration will be submitted and assigned a slot to be executed
9: end procedure

10: procedure SubmitReq(RSMID, req)
                                                ▷ Used by clients to submit requests
11:     Conf[ ] ← GetConf(RSMID)
                                    ▷ Conf is from the view of the requesting node
12:     for i ← 1, f do
13:         sendto Conf[i] : Submit(RSMID, i, Req)
14:     end for
15: end procedure
```

Replicated State Machine Maintenance

This section will describe the algorithms used to create a replicated state machine and to automate the migration process in order to survive resource churn.

State Machine Creation

A new RSM can be created by any node in the SON by calling `CreateRSM` shown in Algorithm 9.2. The creating node construct the configuration using symmetric replication and lookup operations. The node then sends an `InitSM` message to all nodes in the configuration. Any node that receives an `Init SM` message (Algorithm 9.5) will start a state machine (SM) regardless of its responsibility. Note that the initial configuration, due to lookup inconsistency, may contain some incorrect nodes. This does not cause problems for the replication algorithm. Using migration, the configuration will eventually be corrected.

Client Interactions

A client can be any node in the system that requires the service provided by the RSM. The client need only to know the $RSMID$ to be able to send requests to

the service. Knowing the $RSMID$, the client can calculate the current configuration using equation 9.1 and lookup operations (See Algorithm 9.1). This way we avoid the need for an external configuration repository that points to nodes hosting the replicas in the current configuration. The client submits requests by calling `SubmitReq` as shown in Algorithm 9.2. The method simply sends the request to all replicas in the current configuration. Due to lookup inconsistency, that can happen either at the client side or the RSM side, the client's view of the configuration and the actual configuration may differ. We assume that the client's view overlaps, at least at one node, with the actual configuration for the client to be able to submit requests. Otherwise, the request will fail and the client need to try again later after the system heals itself. We also assume that each request is uniquely stamped and that duplicate requests are filtered.

In the current algorithm the client submits the request to all nodes in the configuration for efficiency. It is possible to optimise the number of messages by submitting the request only to one node in the configuration that will forward it to the current leader. The trade off is that sending to all nodes increases the probability of the request reaching the RSM. This reduces the negative effects of lookup inconsistencies and churn on the availability of the service.

It may happen, due to lookup inconsistency, that the configuration calculated by the client contains some incorrect references. In this case, a incorrectly referenced node ignores client requests (Algorithm 9.3 line 13) when it finds out that it is not responsible for the target RSM.

On the other hand, it is possible that the configuration was created with some incorrect references. In this case, the node that discovers that it was supposed to be in the configuration will attempt to correct the configuration by replacing the incorrect reference with the reference to itself (Algorithm 9.3 line 11).

Request Execution

The execution is initiated by receiving a submit request from a client. This will result in scheduling the request for execution by assigning it to a slot that is agreed upon among all SMs in the configuration (using the Paxos algorithm). Meanwhile, scheduled requests are executed sequentially in the order of their slot numbers. These steps are shown in Algorithm 9.3.

When a node receives a request from a client it will first check if it is hosting an SM, which the request is directed to. If this is the case, then the node will try to schedule the request if the node believes that it is the leader. Otherwise the node will forward the request to the leader. On the other hand, if the node is not hosting an SM with the RSMID in the request, it will proceed as described in section 9.3, i.e. it ignores the request, if it is not resposible for the target SM, otherwise, it tries to correct the configuration

At execution time, the execution module will execute all requests sequentially except the `ConfChange` request that is handled by the SM. The `ConfChange` request will start the migration protocol presented in [8] and outlined in Section 9.2.

Algorithm 9.3 Execution

```
1: receipt of SUBMIT(RSMID, rank, Req) from m at n
2:     SM ← SMs[RSMID][rank]
3:     if SM ≠ φ then
4:         if SM.leader = n then
5:             SM.submit(Req)
6:         else
7:             sendto SM.leader :
               SUBMIT(RSMID, rank, Req)
                                   ▷ forward the request to the leader
8:         end if
9:     else
10:        if r(RSMID, rank) ∈]n.predecessor, n] then
                                                   ▷ I'm responsible
11:            JOINRSM(RSMID, rank)
12:        else
13:            DONOTHING
                                   ▷ This is probably due to lookup inconsistency
14:        end if
15:    end if
16: end receipt

17: procedure EXECUTESLOT(req)
                                   ▷ This is called when executing the current slot
18:    if req.type = ConfChange then
19:        newConf ← Conf[CurrentConf]
20:        newConf[req.rank] ← req.id
                           ▷ Replaces the previous responsible with the new one
21:        SM.migrate(newConf)
                           ▷ SMART will set LastSlot and start new configuration
22:    else
                                   ▷ Execution module handles all other requests
23:        ExecutionModule.Execute(req)
24:    end if
25: end procedure
```

Handling Churn

Algorithm 9.4 shows how to maintain the replicated state machine in case of node join/leave/failure. When any of these cases happen, a new node may become responsible for hosting a replica. In case of node join, the new node will send a message to its successor to get any replica that now it is responsible for. In case of leave, the leaving node will send a message to its successor containing all

Algorithm 9.4 Churn Handling

```
1: procedure NodeJoin
                                   ▷ Called by SON after the node joined the overlay
2:     sendto successor : PullSMs(]predecessor, myId])
3: end procedure

4: procedure NodeLeave
   sendto successor : NewSMs(SMs)
                                   ▷ Transfer all hosted SMs to Successor
5: end procedure

6: procedure NodeFailure(newPred, oldPred)
                                   ▷ Called by SON when the predecessor fails
7:     I ← ⋃_{x=2}^{f} ]r(newPred, x), r(oldPred, x)]
8:     multicast I : PullSMs(I)
9: end procedure
```

replicas that it was hosting. In the case of failure, the successor of the failed node need to discover if the failed node was hosting any SMs. This is done by checking all intervals (line 7) that are symmetric to the interval that the failed node was responsible for. One way to achieve this is by using interval-cast that can be efficiently implemented on SONs e.g. using bulk operations [10].

All newly discovered replicas are handled by `NewSMs` (Algorithm 9.5). The node will request a configuration change by joining the corresponding RSM for each new SM. Note that the configuration size is fixed to f. A configuration change means replacing reference at position r in the configuration array with the reference of the node requesting the change.

9.4 Robust Management Elements in Niche

In order to validate and evaluate the proposed approach to achieve robust services, we have implemented our proposed algorithm together with the replicated state machine technique and migration support using the Kompics component framework [13]. We intend to integrate the implemented prototype with the Niche platform and use it for building robust management elements for self-managing distributed applications. We have conducted a number of tests to validate the algorithm. We are currently conducting simulation tests, using Kompics simulation facilities, to evaluate the performance of our approach.

The autonomic manager in Niche is constructed from a set on management elements. To achieve robustness and high availability of Autonomic Managers, in spite of churn, we will apply the algorithm described in the previous section to management elements. Replicating management elements and automatically

Algorithm 9.5 SM maintenance (handled by the container)

```
1: receipt of InitSM(RSMID, Rank, Conf) from m at n
2:     new SM
                                   ▷ Creates a new replica of the state machine
3:     SM.ID ← RSMID
4:     SM.Rank ← Rank                                          ▷ 1 ≤ Rank ≤ f
5:     SMs[RSMID][Rank] ← SM
                                   ▷ SMs stores all SM that node n is hosting
6:     SM.Start(Conf)
                                   ▷ This will start the SMART protocol
7: end receipt

8: receipt of PullSMs(Intervals) from m at n
9:     for each SM in SMs do
10:        if r(SM.id, SM.rank) ∈ I then
11:            newSMs.add(SM)
12:        end if
13:    end for
14:    sendto m : NewSMs(newSMs)
                                   ▷ SMs are destroyed later by migration protocol
15: end receipt

16: receipt of NewSMs(NewSMs) from m at n
17:    for each SM in NewSMs do
18:        JoinRSM(SM.id, SM.rank)
19:    end for
20: end receipt
```

maintaining them will result in what we call Robust Management Element (RME). An RME will:

- be replicated to ensure fault-tolerance. This is achieved by the replicated state machine algorithm.
- survive continuous resource failures by automatically restoring failed replicas on other nodes. This is achieved using our proposed algorithm that will automatically migrate the RME replicas to new nods when needed.
- maintain its state consistent among replicas. This is guaranteed by the replicated state machine algorithm and the migration mechanism used.
- provide its service with minimal disruption in spite of resource join/leave/fail (high availability). This is due to replication. In case of churn, remaining replicas can still provide the service.

- be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location). The clients need only to know the `RME_ID` to be able to use an RME regardless of the location of individual replicas.

The RMEs are implemented by wrapping ordinary MEs inside a state machine. The ME will serve as the execution module shown in Figure 9.1. However, to be able to use this approach, the ME must follow the same constraints as the execution module. That is the ME must be deterministic and provide checkpointing.

Typically, in replicated state machine approach, a client sends a request that is executed by the replicated state machine and gets a result back. In our case, to implement feedback loops, we have two kinds of clients from the point of view of an RMS. A set of sending client C_s that submit requests to the RME and a set of receiving clients C_r that receive results from the RME. The C_s includes sensors and/or other (R)MEs and the C_r includes actuators and/or other (R)MEs.

To simplify the creation of control loops that are formed from RMEs we use a publish/subscribe mechanism. The publish/subscribe system handles requests/responses to link different stages to form a control loop.

9.5 Conclusions and Future Work

In this paper, we presented an approach to achieve robust management elements that will simplify the construction of autonomic managers. The approach uses replicated state machines and relies on our proposed algorithm to automate replicated state machine migration in order to tolerate churn. The algorithm uses symmetric replication, which is a replication scheme used in structured overlay networks, to decide on the placement of replicas and to detect when to migrate. Although in this paper we discussed the use of our approach to achieve robust management elements, we believe that this approach might be used to replicate other services in structured overlay networks in general.

In order to validate and evaluate our approach, we have implemented a prototype that includes the proposed algorithms. We are currently conducting simulation tests to evaluate the performance of our approach. In our future work, we will integrate the implemented prototype with the Niche platform to support robust management elements in self-managing distributed applications. We also intend to optimise the algorithm in order to reduce the amount of messages and we will investigate the effect of the publish/subscribe system used to construct control loops and try to optimise it. Finally, we will try to apply our approach to other problems in the field of distributed computing.

Bibliography

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] IBM, "An architectural blueprint for autonomic computing, 4th edition." `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`, June 2006.

[3] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing* (T. Priol and M. Vanneschi, eds.), pp. 163–174, Springer US, July 2008.

[4] "Niche homepage." `http://niche.sics.se/`.

[5] A. Al-Shishtawy, V. Vlassov, P. Brand, and S. Haridi, "A design methodology for self-management in distributed environments," in *Computational Science and Engineering, 2009. CSE '09. IEEE International Conference on*, vol. 1, (Vancouver, BC, Canada), pp. 430–436, IEEE Computer Society, August 2009.

[6] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[7] L. Lamport, "Paxos made simple," *SIGACT News*, vol. 32, pp. 51–58, December 2001.

[8] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 103–115, 2006.

[9] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *Communications Surveys & Tutorials, IEEE*, vol. 7, pp. 72–93, Second Quarter 2005.

[10] A. Ghodsi, *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD thesis, Royal Institute of Technology (KTH), 2006.

[11] D. Malkhi, F. Oprea, and L. Zhou, "Omega meets paxos: Leader election and stability without eventual timely links," in *Proc. of the 19th Int. Symp. on Distributed Computing (DISC'05)*, pp. 199–213, Springer-Verlag, July 2005.

[12] J. S. Kong, J. S. Bridgewater, and V. P. Roychowdhury, "Resilience of structured p2p systems under churn: The reachable component method," *Computer Communications*, vol. 31, pp. 2109–2123, June 2008.

[13] C. Arad, J. Dowling, and S. Haridi, "Building and evaluating p2p systems using the kompics component framework," in *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pp. 93 –94, sept. 2009.