ROYAL INSTITUTE
OF TECHNOLOGY

Self-Management for Large-Scale Distributed Systems

AHMAD AL-SHISHTAWY

PhD Thesis
Stockholm, Sweden 2012

TRITA-ICT/ECS AVH 12:04
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-12/04-SE
ISBN 978-91-7501-437-1

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 doktorsexamen i elektronik och datorsystem onsdagen den 26 september 2012 klockan 14.00 i sal E i Forum IT-Universitetet, Kungl Tekniska högskolan, Isajordsgatan 39, Kista.

Swedish Institute of Computer Science
SICS Dissertation Series 57
ISRN SICS-D–57–SE
ISSN 1101-1335.

Tryck: Universitetsservice US AB

Abstract

Autonomic computing aims at making computing systems self-managing by using autonomic managers in order to reduce obstacles caused by management complexity. This thesis presents results of research on self-management for large-scale distributed systems. This research was motivated by the increasing complexity of computing systems and their management.

In the first part, we present our platform, called Niche, for programming self-managing component-based distributed applications. In our work on Niche, we have faced and addressed the following four challenges in achieving self-management in a dynamic environment characterized by volatile resources and high churn: resource discovery, robust and efficient sensing and actuation, management bottleneck, and scale. We present results of our research on addressing the above challenges. Niche implements the autonomic computing architecture, proposed by IBM, in a fully decentralized way. Niche supports a network-transparent view of the system architecture simplifying the design of distributed self-management. Niche provides a concise and expressive API for self-management. The implementation of the platform relies on the scalability and robustness of structured overlay networks. We proceed by presenting a methodology for designing the management part of a distributed self-managing application. We define design steps that include partitioning of management functions and orchestration of multiple autonomic managers.

In the second part, we discuss robustness of management and data consistency, which are necessary in a distributed system. Dealing with the effect of churn on management increases the complexity of the management logic and thus makes its development time consuming and error prone. We propose the abstraction of Robust Management Elements, which are able to heal themselves under continuous churn. Our approach is based on replicating a management element using finite state machine replication with a reconfigurable replica set. Our algorithm automates the reconfiguration (migration) of the replica set in order to tolerate continuous churn. For data consistency, we propose a majority-based distributed key-value store supporting multiple consistency levels that is based on a peer-to-peer network. The store enables the tradeoff between high availability and data consistency. Using majority allows avoiding potential drawbacks of a master-based consistency control, namely, a single-point of failure and a potential performance bottleneck.

In the third part, we investigate self-management for Cloud-based storage systems with the focus on elasticity control using elements of control theory and machine learning. We have conducted research on a number of different designs of an elasticity controller, including a State-Space feedback controller and a controller that combines feedback and feedforward control. We describe our experience in designing an elasticity controller for a Cloud-based key-value store using state-space model that enables to trade-off performance for cost. We describe the steps in designing an elasticity controller. We continue by presenting the design and evaluation of ElastMan, an elasticity controller for Cloud-based elastic key-value stores that combines feedforward and feedback control.

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 deep gratitude to my supervisor Associate Professor Vladimir Vlassov for his vision, ideas, and useful critiques of this research work. With his insightful advice and unsurpassed knowledge that challenged and enriched my thoughts, together with the freedom given to me to pursue independent work, I was smoothly introduced to academia and research and kept focused on my goals. I would like as well to take a chance to thank him for the continuous support, patience and encouragements that have been invaluable on both academic and personal levels.

I feel privileged to have the opportunity to work under the co-supervision of Professor Seif Haridi. His deep knowledge in the divers fields of computer science, fruitful discussions, and enthusiasm have been a tremendous source of inspiration. I am also grateful to Dr. Per Brand for sharing his knowledge and experience with me during my research and for his contributions and feedback to my work.

I acknowledge the help and support given to me by the director of doctoral studies Associate Professor Robert Rönngren and the head of Software and Computer Systems unit Thomas Sjöland. I would like to thank Dr. 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 am truly indebted and thankful to my colleagues and dear friends Tallat Shafaat, Cosmin Arad, Amir Payberah, and Fatemeh Rahimian for their daily support and inspiration through the ups and downs of the years of this work. I am indebted to all my colleagues at KTH and SICS, specially to Dr. Sarunas Girdzijauskas, Dr. Jim Dowling, Dr. Ali Ghodsi, Roberto Roverso, and Niklas Ekström for making the environment at the lab both constructive and fun. I also acknowledge Muhammad Asif Fayyaz, Amir Moulavi, Tareq Jamal Khan, and Lin Bao for the work we did together. I take this opportunity to thank the Grid4All project team, especially Konstantin Popov, Joel Höglund, Dr. Nikos Parlavantzas, and Professor Noel de Palma for being a constant source of help.

This research has been partially funded by the Grid4All FP6 European project; the Complex Service Systems focus project, a part of the ICT-TNG Strategic Research Areas initiative at the KTH; the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research; the RMAC project funded by EIT ICT Labs.

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

Distributed systems such as Peer-to-Peer (P2P) [1], Grid [2], and Cloud [3] systems provide pooling and coordinated use of distributed resources and services. Distributed systems provide platforms to provision large scale distributed applications such as P2P file sharing, multi-tier Web 2.0 applications (e.g., social networks and wikis), and scientific applications (e.g., weather prediction and climate modeling). The increasing popularity and demand for large scale distributed applications came with an increase in the complexity and the management overhead of these applications that posed a challenge obstructing further development [4]. Autonomic computing [5] is an attractive paradigm to tackle the problem of growing software complexity by making software systems and applications self-managing. Self-management, namely self-configuration, self-optimization, self-healing, and self-

protection, can be achieved by using autonomic managers [6]. An autonomic manager continuously monitors software and its execution environment and acts to meet its management objectives such as failure recovery or performance optimization. Managing applications in dynamic environments with dynamic resources and/or load (like community Grids, peer-to-peer systems, and Clouds) is especially challenging due to large scale, complexity, high resource churn (e.g., in P2P systems) and lack of clear management responsibility.

Most distributed systems and applications are built of components using a distributed component models such as the Fractal Component Model [7] and the Kompics Component Framework [8]; 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 and to provide required functionality at an acceptable performance level. Self-* properties can be provided using feedback control loops, known as MAPE-K loops (Monitor, Analyze, Plan , Execute – Knowledge) that come from 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 a robust management abstraction which 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 for distributed systems 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 overlay networks are designed to work in the highly dynamic distributed environment we are targeting. They have certain self-* properties and can tolerate churn. Therefore structured overlay networks 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 a publish/subscribe service.

To better deal with dynamic environments; to improve scalability, robustness, and performance by avoiding management bottlenecks and a single-point-of-failure; we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. Several issues appear 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 and 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.

The growing popularity of Web 2.0 applications, such as wikis, social networks, and blogs, has posed new challenges on the underlying provisioning infrastructure. Many large-scale Web 2.0 applications leverage elastic services, such as elastic key-value stores, that can scale horizontally by adding/removing servers. Voldemort [9], Cassandra [10], and Dynamo [11] are few examples of elastic storage services. Cloud computing [3], with its pay-as-you-go pricing model, provides an attractive environment to provision elastic services as the running cost of such services becomes proportional to the amount of resources needed to handle the current workload.

Managing the resources for Web 2.0 applications, in order to guarantee acceptable performance, is challenging because it is difficult to predict the workload particularly for new applications that can become popular within few days [12, 13]. Furthermore, the performance requirements is usually expressed in terms of upper percentiles which is more difficult to maintain that average performance [11, 14].

The pay-as-you-go pricing model, elasticity, and dynamic workload altogether call for the need for an elasticity controller that automates the provisioning of Cloud resources. The elasticity controller adds more resources under high load in order to meet required service level objectives (SLOs) and releases some resources under low load in order to reduce cost.

1.1 Summary of Research Objectives

Research reported in this thesis aims at enabling and achieving self-management for large-scale distributed systems. In this research, we start by addressing the challenges of enabling self-management for large-scale and/or dynamic distributed systems in order to hide the system complexity and to automate its management, i.e., organization, tuning, healing and protection. We achieve this by implementing the Autonomic Computing Architecture proposed by IBM [6] in a fully decentralized way to match the requirements of large scale distributed systems. The autonomic Computing Architecture consists mainly of touchpoints (sensors and actuators) and autonomic managers that communicate with the managed system (via touchpoints) and with each other to achieve management objectives. We define and present the interaction patterns of multiple autonomic managers and proposing steps for designing self-management in large scale distributed systems. We continue our research by addressing the problems of the robustness of management and data consistency that are unavoidable in a distributed system. We have developed a decentralized algorithm that guarantees the robustness of autonomic managers enabling them to tolerate continuous churn. Our approach is based on replicating the autonomic manager using finite state machine replication with a reconfigurable replica set. Our algorithm automates the reconfiguration (migration) of the replica

set in order to tolerate continuous churn. For data consistency, we propose a design and algorithms for a robust fault-tolerant majority-based key-value store supporting multiple consistency levels that is based on a peer-to-peer network. The store provides an Application Programming Interface (API) consisting of different read and write operations with various data consistency guarantees from which a wide range of applications would be able to choose the operations according to their data consistency, performance and availability requirements. The store uses a majority-based quorum technique to provide strong data consistency guarantees. In the final part of our research we focus on using elements of control theory and machine learning in the management logic of autonomic managers for distributed systems. As a use case, we study the automation of elasticity control for Cloud-based services focusing on Cloud-based stores. We define steps for building elasticity controllers for an elastic Cloud-based service, including system identification and controller design. An elasticity controller automatically resizes an elastic service, in response to changes in workload, in order to meet Service Level Objectives (SLOs) at a reduced cost.

1.2 Main Contributions

As presented in previous section, the main objectives of research presented in this thesis include ways, methods, and mechanisms of enabling and achieving self-management for large-scale distributed systems; robust self-management and data consistency in large-scale distributed systems; and automation of elasticity for Cloud-based storage systems. Along this research objectives the main contributions of the thesis are as follows.

1. A platform called *Niche* that enables the development, deployment, and execution of large scale component based distributed applications in dynamic environments. We have also developed a distributed file storage service, called YASS, to illustrate and evaluate Niche;

2. A design methodology that supports the design of distributed management and defines different interaction patterns between managers. We define design steps, that includes partitioning of management functions and orchestration of multiple autonomic managers;

3. A novel approach and corresponding distributed algorithms to achieve the robustness of management that uses replicated state machines and relies on our proposed algorithms to automate replicated state machine migration in order to tolerate continuous churn. Our approach uses symmetric replication, which is a replica placement scheme used in Structured Overlay Networks (SONs), to decide on the placement of replicas and uses SON to monitor them. The replicated state machine is extended, beyond its main purpose of providing the service, to process monitoring information and to decide on when to migrate.

4. A design and corresponding distributed algorithm for a majority-based key-value store with multiple consistency levels that is intended to be deployed in a large-scale dynamic P2P environment. Our store provides a number of read-/write operations with multiple consistency levels and with semantics similar to Yahoo!'s PNUTS [15]. The store uses the majority-based quorum technique to maintain consistency of replicated data. Our majority-based store provides stronger consistency guarantees than guarantees provided in a classical Distributed Hash Table (DHT) [16] but less expensive than guarantees of Paxos-based replication. Using majority allows avoiding potential drawbacks of a master-based consistency control, namely, a single-point of failure and a potential performance bottleneck. Furthermore, using a majority rather than a single master allows the system to achieve robustness and withstand churn in a dynamic environment. Our mechanism is decentralized and thus allows improving load balancing and scalability.

5. Design steps for building an elasticity controller for a key-value store in a Cloud environment using state-space model. State-space enables us to trade-off performance for cost. We describe the steps in designing the elasticity controller including system identification and controller design. The controller allows the system to automatically scale the amount of resources while meeting performance SLO, in order to reduce SLO violations and the total cost for the provided service.

6. A novel approach to automation of elasticity of Cloud-based services by combining feedforward and feedback control; A design and evaluation of an elasticity controller, called ElastMan, using the proposed approach. ElastMan, an Elasticity Manager for Cloud-based key-value stores, combines and leverages the advantages of both feedback and feedforward control. The feedforward control is used to quickly respond to rapid changes in workload. This enables us to smooth the noisy signal of the 99th percentile of read operation latency and thus use feedback control. The feedback controller is used to handle diurnal workload and to correct errors in the feedforward control due to the noise that is caused mainly by the variable performance of Cloud VMs. The feedback controller uses a scale-independent design by indirectly controlling the number of VMs by controlling the average workload per VM. This enables the controller, given the near-linear scalability of key-value stores, to control stores of various scales.

1.3 Thesis Organization

The thesis is organized into five parts as follows. Part I, which consists of five chapters, presents an overview of the thesis. After the introduction in Chapter 1 that presents motivation, research objectives, and main contributions of the thesis, we lay out the necessary background in Chapter 2 followed by a more detailed

overview of the thesis in Chapter 3. The thesis contributions in Chapter 4. Finally, conclusions and discussion of future work are presented in Chapter 5. Part II deals with the problem of self-management for dynamic large-scale distributed systems, We present the Niche Platform in Chapter 6 and Chapter 7 followed by our design methodology for self-management in Chapter 8. Part III discusses robustness of management and data consistency in large-scale distributed systems. Algorithms for providing strong consistency and robustness of management is presented in Chapter 9. Algorithms for multiple data consistency levels suitable for distributed key-value object stores is presented in Chapter 10. Part IV discusses self-management for Cloud-based storage systems. Chapter 11 discusses and describes the steps in building a controller based on state-space model. Chapter 12 presents ElastMan, an elasticity manager for Cloud-based key-value stores based on combining feedforward and feedback control. Finally, Part V includes the bibliography used throughout the thesis.

Chapter 2

Background

This chapter lays out the necessary background for the thesis. In our research work on enabling and achieving self-management of large scale distributed systems we leverage the self-organizing properties of structured overlay networks, for providing basic services and runtime support, together with component models, for reconfiguration and introspection. The result of this research is an autonomic computing platform suitable for large-scale dynamic distributed environments. In this work, for the management logic we use policy-based management. Our research involves various decentralized algorithms that are necessary in large scale distributed environments in order to avoid hot spots and a single point of failure. An example is our decentralized algorithm for achieving the robustness of management. In our research on self-management for Cloud-based services, in particular key-value stores,

we apply the control theoretic approach to automate elasticity. For the management logic we use elements of control theory, and machine learning techniques. we study their feasibility and performance in controlling the elasticity of Cloud-based key-value stores. Key-value stores play a vital role in many large-scale Web 2.0 applications. These key concepts are described in the rest of this chapter.

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 [5]. Autonomic computing focuses 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. Furthermore, 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 [17] 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. The purpose of autonomic computing is not to replace human administrators 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 [4] 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 (and/or other operational metrics such as energy consumption and cost) at optimal 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 includes for example protection against viruses and intrusion attempts.

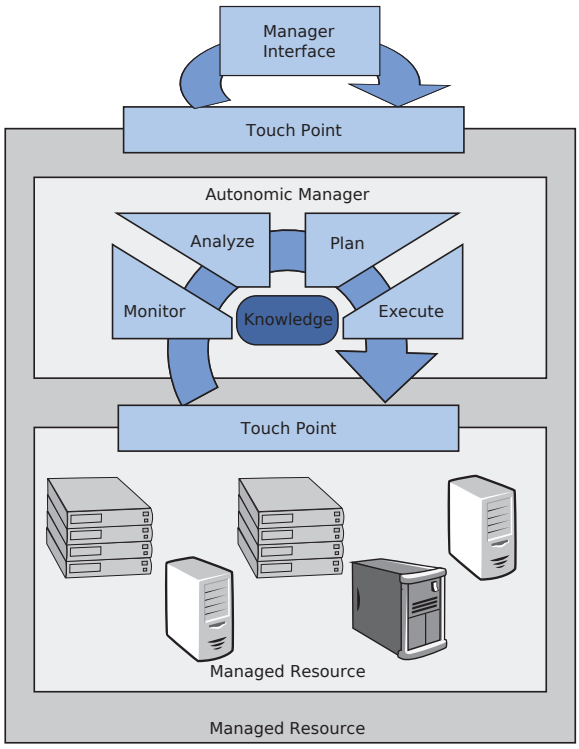

Figure 2.1: A simple autonomic computing architecture with one autonomic manager.

The Autonomic Computing Architecture

The autonomic computing reference architecture proposed by IBM [6] consists of the following five building blocks (see Figure 2.1).

- **Touchpoint** consists of a set of sensors and effectors (actuators) 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 the 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.
- **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.

In our work we propose a design and an implementation of the autonomic computing reference architecture for large-scale distributed systems.

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.

- **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 [18]. Autonomic behaviour of computing systems are achieved through dynamically modifying the structure (compositional adaptation) and thus the behaviour of the system [19, 20] 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.
- **Control Theoretic Approach:** Classical control theory have been successfully applied to solve control problems in computing systems [21] 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 [22]. Challenges beyond classical control theory have also been addressed [23] 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.

- **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 [24–26]. 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 [27, 28].

- **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 [29–32].

In our work on Niche (a distributed component management system), we followed mainly the architectural approach to autonomic computing. We use and extend the Fractal component model (presented in the next section) to dynamically modifying the structure and thus the behaviour of the system. However, there is no clear line dividing these different approaches and they may be combined together in one system. Later, in our research on automation of elasticity for Cloud-based services we used the control theoretic approach to self-management.

2.2 The Fractal Component Model

The Fractal component model [7, 33] is a modular and extensible component model used to design, implement, deploy and reconfigure various systems and applications. 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.

In our work on Niche, we use the Fractal component model to introspect and reconfigure components of a distributed application. We extend the Fractal component model in various ways such as network transparent bindings that enables component mobility and also with component groups and with one-to-any and one-to-all bindings.

2.3 Structured Peer-to-Peer Overlay Networks

Peer-to-peer (P2P) refers to a class of distributed network architectures which are formed of participants (usually called peers or nodes) that reside 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 [1]. Put in other words, each peer plays the role of both client and server. Thus, a P2P network usually does 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 nature 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 established between peers in a certain way according to the P2P network type (topology). 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 divides 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, which is continuously maintained by an algorithm. The remainder of this section will focus on structured P2P networks.

A structured P2P network such as Chord [34], CAN [35], and Pastry [36] maintains a structure of overlay links. Using this structure allows to implement a Distributed Hash Table (DHT). A DHT provides a lookup service similar to hash tables that stores key-value pairs. 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 among 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-organization.

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

In our work on Niche, 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 leverage the scalability and self-organizing properties (e.g., automatic correction of routing tables in order to tolerate joins, leaves, and failures of peers, automatic maintenance of responsibility for DHT buckets) of structured P2P network for providing basic services and runtime support. We used an indexing service to implement network transparent name-based communication and component groups. We used efficient multicast/broadcast for communication and discovery. We used a publish/subscribe service to implement event based communication between management elements.

2.4 Web 2.0 Applications

The growing popularity of Web 2.0 applications, such as wikis, social networks, and blogs, has posed new challenges on the underlying provisioning infrastructure. Web 2.0 applications are data-centric with frequent data access [37]. This poses new challenges on the data-layer of n-tier application servers because the performance of the data-layer is typically governed by strict Service Level Objectives (SLOs) [14] in order to satisfy costumer expectations.

Key-Value Stores

With the rapid increase of Web 2.0 users, the poor scalability of a typical data-layer with ACID [38] properties limited the scalability of Web 2.0 applications. This has led to the development of new data-stores with relaxed consistency guarantees and simpler operations such as Voldemort [9], Cassandra [10], and Dynamo [11]. These storage systems typically provide a simple key-value store with eventual consistency guarantees. The simplified data and consistency models of key-value stores enable them to efficiently scale horizontally by adding more servers and thus serve more clients.

Another problem facing Web 2.0 applications is that a certain service, feature, or topic might suddenly become popular resulting in a spike in the workload [12, 13]. The fact that storage is a stateful service complicates the problem since only a particular subset of servers hosts the data related to the popular item. The subset becomes overloaded while other servers can be underloaded.

These challenges have led to the need for an automated approach, to manage the data-tier, that is capable of quickly and efficiently responding to changes in the workload in order to meet the required SLOs of the storage service.

Cloud Computing and Elastic Services

Cloud computing [3], with its pay-as-you-go pricing model, provides an attractive solution to host the ever-growing number of Web 2.0 applications as the running cost of such services becomes proportional to the amount of resources needed to handle the current workload. This model is attractive, specially for startups, because it is difficult to predict the future load that is going to be imposed on the application and thus the amount of resources (e.g., servers) needed to serve that load. Another reason is the initial investment, in the form of buying the servers, that is avoided with the Cloud pay-as-you-go pricing model. The independence of peak loads for different applications enables Cloud providers to efficiently share the resources among the applications. However, sharing the physical resources among Virtual Machines (VMs) running different applications makes it challenging to model and predict the performance of the VMs [39, 40].

To leverage the Cloud pricing model and to efficiently handle the dynamic Web 2.0 workload, Cloud services (such as a key-value store in the data-tier of a Cloud-

based multi-tier application) are designed to be elastic. An elastic service is designed to be able to scale horizontally at runtime without disrupting the running service. An elastic service can be scaled up (e.g., by the system administrator) in the case of increasing workload by adding more resources in order to meet SLOs. In the case of decreasing load, an elastic service can be scaled down by removing extra resource and thus reducing the cost without violating the SLOs. For stateful services, scaling is usually combined with a rebalancing step necessary to redistribute the data among the new set of servers.

Managing the resources for Web 2.0 applications, in order to guarantee acceptable performance, is challenging because of the gradual (diurnal) and sudden (spikes) variations in the workload [41]. It is difficult to predict the workload particularly for new applications that can become popular within a few days [12, 13]. Furthermore, the performance requirement is usually expressed in terms of upper percentiles (e.g., "99% of reads are performed in less than 10ms within one minute") which is more difficult to maintain than the average performance [11, 14].

Feedback vs. Feedforward Control

The pay-as-you-go pricing model, elasticity, and dynamic workload of Web 2.0 applications altogether call for the need for an elasticity controller that automates the provisioning of Cloud resources. The elasticity controller leverages the horizontal scalability of elastic services by provisioning more resources under high workloads in order to meet required service level objectives (SLOs). The pay-as-you-go pricing model provides an incentive for the elasticity controller to release extra resources when they are not needed once the workload decreases.

In computing systems, a controller [21] or an autonomic manager [5] is a software component that regulates the nonfunctional properties (performance metrics) of a target system. Nonfunctional properties are properties of the system such as the response time or CPU utilization. From the controller perspective these performance metrics are the *system output*. The regulation is achieved by monitoring the target system through a monitoring interface and adapting the system's configurations, such as the number of servers, accordingly through a control interface (*control input*). Controllers can be classified into feedback or feedforward controllers depending on whether or not a controller uses feedback to control a system. Feedback control requires monitoring of the system output whereas the feedforward control does not monitor the system output because it does not use the output to control.

In feedback control, the system's output (e.g., response time) is being monitored. The controller calculates the control error by comparing the current system's output with a desired value set by the system administrators. Depending on the amount and sign of the control error, the controller changes the control input (e.g., number of servers to add or remove) in order to reduce the control error. The main advantage of feedback control is that the controller can adapt to disturbance such as changes in the behaviour of the system or its operating envi-

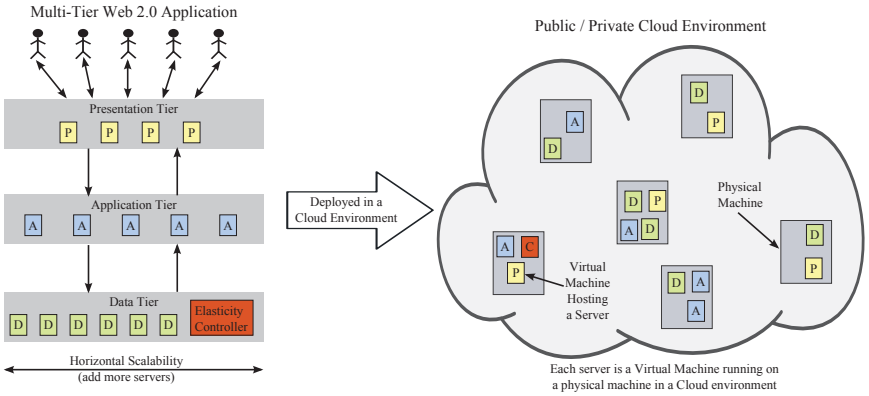

Figure 2.2: Multi-Tier Web 2.0 Application with Elasticity Controller Deployed in a Cloud Environment

ronment. Disadvantages include oscillation, overshoot, and possible instability if the controller is not properly designed. Due to the nonlinearity of most systems, feedback controllers are approximated around linear regions called the operating region. Feedback controllers work properly only around the operating region they were designed for.

In feedforward control, the system's output is not being monitored. Instead the feedforward controller relies on a model of the system that is used to calculate the systems output based on the current system state. For example, given the current request rate and the number of servers, the system model is used to calculate the corresponding response time and act accordingly to meet the desired response time. The major disadvantage of feedforward control is that it is very sensitive to unexpected disturbances that are not accounted for in the system model. This usually results in a relatively complex system model compared to feedback control. The main advantages of feedforward control include being faster than feedback control and avoiding oscillations and overshoot.

Target System

As a part of research presented in this thesis, in our work on self-management for Cloud-based services, we are targeting multi-tier Web 2.0 applications as depicted in the left side of Figure 2.2. We are focusing on managing the data-tier because of its major effect on the performance of Web 2.0 applications, which are mostly data centric [37]. Furthermore, the fact that storage is a stateful service makes it harder to manage as each request can be handled only by a subset of the servers that store replicas of the particular data item in the request.

For the data-tier, we assume horizontally scalable key-value stores due to their

popularity in many large scale Web 2.0 applications such as Facebook and LinkedIn. A typical key-value store provides a simple put/get interface. This simplicity enables key-value stores to efficiently partition the data among multiple servers and thus to scale well to a large number of servers.

The three minimum requirements to manage a key-value store using our approach (described in Section 12.4) are as follows. First, the store must provide a monitoring interface that enables the monitoring of both the workload and the latency of put/get operations. Second, the store must also provide an actuation interface that enables the horizontal scalability by adding or removing service instances.

Third, actuation (adding or removing service instances) must be combined with a rebalance operation, because storage is a stateful service. The rebalance operation redistributes the data among the new set of servers in order to balance the load among them. Many key-value stores, such as Voldemort [9] and Cassandra [10], provide tools to rebalance the data among the service instances. In this work, we focus on the control problem and rely on the built-in capabilities of the storage service to rebalance the load. If the storage does not provide such service, techniques such as rebalancing using fine grained workload statistics proposed by Trushkowsky et al. [14], the Aqueduct online data migration proposed by Lu et al. [42], or the data rebalance controller proposed by Lim et al. [43] can be used.

In this work, we target Web 2.0 applications running in Cloud environments such as Amazon's EC2 [44] or private Clouds. The target environment is depicted on the right side of Figure 2.2. We assume that each service instance runs on its own VM; Each physical server hosts multiple VMs. The Cloud environment hosts multiple such applications (not shown in the figure). Such environment complicates the control problem. This is mainly due to the fact that VMs compete for the shared resources. This high environmental noise makes it difficult to model and predict the performance of VMs [39, 40].

2.5 State of the Art and Related Work 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 [4,17]. System self-management allows reducing management costs and improving management efficiency by removing human administrators 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., [4,5,17,45]. 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 [4]. 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 [46].

The major approach to self-management is to use one or multiple feedback control loops [17, 21], a.k.a. autonomic managers [4], to control different properties of the system based on functional decomposition of management tasks and assigning the tasks to multiple cooperative managers [47–49]. 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) [21]. A manager control loop consists of four stages, known as MAPE: Monitoring, Analyzing, Planning, and Execution [4] (Section 2.1).

Authors of [21] apply the control theoretic approach to design computing systems with feedback loops. The architectural approach to autonomic computing [18] 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 [50]. 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 [51]. Authors of [49] and [48] use multi-criteria utility functions for power-aware performance management. Authors of [52, 53] 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.

Authors of [54] propose a generic gossip protocol for dynamic resource allocation

in a large-scale Cloud environment, which can be instantiated for specific objectives, under CPU and memory constraints. The authors illustrate an instance of the generic protocol that aims at minimizing the power consumption through server consolidation, while satisfying a changing load pattern. The protocol minimizes the power consumption through server consolidation when the system is in underload and uses fair resource allocation in case of overload. The authors advocate for the use of a gossip protocol to efficiently compute a configuration matrix, that determines how Cloud resources are allocated, for large-scale Clouds.

Authors of [55] address the problem of automating the horizontal elasticity of a Cloud-based service in order to meet varying demands on the service while enforcing SLAs. The authors use queuing theory to model a Cloud service. The model is used to build two adaptive proactive controllers that estimate the future load on the service. The authors propose the use of a hybrid controller consisting of the proactive controller for scaling down coupled with a reactive controller for scaling up.

Authors of [56] address the self-management challenges for multi-Cloud architectures. The authors focus on three complementary challenges, namely, predictive elasticity, admission control, and placement (or scheduling) of virtual machines. The authors propose a unified approach for tuning the policies that governs the tools that address each of the aforementioned challenges in order to optimize the overall system behavior.

Policy-based self-management [57–62] allows 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 [63] 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) [64, 65] 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. The PF7 EU-project VISION Cloud [66] aims at improving storage in the Cloud by making it better, easier and more secure. The project addresses several self-management aspects of Cloud-based storage by proposing various solutions such as the *computational storage* that provides a solution for bringing computation to the storage. Computational storage enables a secure execution of computational tasks near the required data as well as autonomous data derivation and transformation.

Several completed research projects, in particular, AutoMate [67], Unity [68], and SELFMAN [17, 69], and also the Grid4All [47, 70, 71] project we participated in, propose frameworks to augment component programming systems with man-

agement 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 Content Delivery Network (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.

There are many projects that use techniques such as control theory, machine learning, empirical modeling, or a combination of them to achieve SLOs at various levels of a multi-tier Web 2.0 application.

For example, Lim et al. [43] proposed the use of two controllers to automate elasticity of a storage. An integral feedback controller is used to keep the average response time at a desired level. A cost-based optimization is used to control the impact of the rebalancing operation, needed to resize the elastic storage, on the response time. The authors also propose the use of proportional thresholding, a technique necessary to avoid oscillations when dealing with discrete systems. The design of the feedback controller relies on the high correlation between CPU utilization and the average response time. Thus, the control problem is transformed into controlling the CPU utilization to indirectly control the average response time. Relying on such strong correlation might not be valid in Cloud environments with variable VM performance nor for controlling using 99th percentile instead of average.

To our best knowledge, Trushkowsky et al. [14] were the first to propose a con-

trol framework for controlling upper percentiles of latency in a stateful distributed system. The authors propose the use of a feedforward model predictive controller to control the upper percentile of latency. The major motivation for using feedforward control is to avoid measuring the noisy upper percentile signal necessary for feedback control. Smoothing the upper percentile signal, in order to use feedback control, may filter out spikes or delay the response to them. The major drawback of using only feedforward is that it is very sensitive to noise such as the variable performance of VMs in the Cloud. The authors relies on replication to reduce the effect of variable VM performance, but in our opinion, this might not be guaranteed to work in all cases. The authors [14] also propose the use of fine grained monitoring to reduce the amount of data transfer during rebalancing. This significantly reduces the disturbance resulting from the rebalance operation. Fine grain monitoring can be integrated with our approach to further improve the performance.

Malkowski et al. [72] focus on controlling all tiers on a multi-tier application due to the dependencies between the tiers. The authors propose the use of an empirical model of the application constructed using detailed measurements of a running application. The controller uses the model to find the best known configuration of the multi-tier application to handle the current load. If no such configuration exists, the controller falls back to another technique such as a feedback controller. Although the empirical model will generally generate better results, it is more difficult to construct.

The area of autonomic computing is still evolving. 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. We address several problems such as automation of elasticity control, robustness of management, distribution of management functionality among cooperative autonomic managers, and the programming of self-managing applications. We provide solutions for these problems in form of distributed algorithms, methodologies, tools, and a platform for self-management in large scale distributed environments.

Chapter 3

Self-Management for Large-Scale Distributed Systems

Autonomic computing aims at making computing systems self-managing by using autonomic managers in order to reduce obstacles caused by management complexity. This chapter summarizes the results of our research on self-management for large-scale distributed systems.

3.1 Enabling and Achieving Self-Management for Large-Scale Distributed Systems

Niche is a proof of concept prototype of a distributed component management platform 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 (SONs), 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. Niche uses name-based routing and DHT functionality of SONs. Self-organizing properties of SONs, which Niche relies on, include automatic handling of join/leave/failure events by automatic correction of routing tables and maintenance of responsibilities for DHT buckets. 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 section presents the Niche platform [73] (`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 researchers and developers at the KTH Royal Institute of Technology; Swedish Institute of Computer Science (SICS), Stockholm, Sweden; and INRIA, France.

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 of a self-managing distributed application. 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 [62]. 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, expressed in a policy language, 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 (faster) than the policy-based management using the policy engine. We recommend using policy-based management with a policy engine 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 join, leave, or fail. In order to deal with such dynamicity, Niche leverages certain properties of the underlying structured P2P overlay network, including name-based routing (when a direct binding is broken), the DHT functionality (for metadata), and self-organizing properties such as automatic correction of routing tables. 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 application 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 resolved 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.

More details about our work on Niche can be found in Chapter 6 and Chapter 7.

We have defined a design methodology for designing the management part of a distributed self-managing application in a distributed manner [47]. 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. The design methodology is presented in more details in Chapter 8.

Demonstrators

In order to demonstrate Niche and our design methodology, 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 service 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.

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 (such as automatic correction of routing tables and maintenance of responsibility for DHT buckets) 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 a community 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: (1) Efficient resource *discovery*; (2) robust and efficient *monitoring* and *actuation*; (3) *distribution* of management to avoid bottleneck and single-point-of-failure; (4) *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 structured overlay networks 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 a 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. Autonomic managers can interact and coordinate their operation in the following four ways: indirect interactions via the managed system (stigmergy); hierarchical interaction (through touch points); direct interaction (via direct bindings); sharing of management elements.

In dynamic systems the rate of changes (joins, leaves, failures of resources, change of component load etc.) is high and therefore 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 subscribed 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.

3.2 Robust Self-Management and Data Consistency in Large-Scale Distributed Systems

Failures in a large scale distributed system is the norm not the exception. Thus the robustness of management and the data consistency are two unavoidable problems that need to be addressed in any autonomic distributed system.

Robust Self-Management

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

Dealing with the effect of churn on management increases the complexity of the management logic and thus makes its development time consuming and error prone. We propose the abstraction of robust management elements (RMEs), which are able to heal themselves under continuous churn. Using RMEs allows the developer to separate the issue of dealing with the effect of churn on management from the management logic. This facilitates the development of robust management by making the developer focus on managing the application while relying on the platform to provide the robustness of management.

An RME 1) is replicated to ensure fault-tolerance; 2) tolerates continuous churn by automatically restoring failed replicas on other nodes; 3) maintains its state consistent among replicas; 4) provides its service with minimal disruption in spite of resource churn (high availability), and 5) is location transparent, i.e., RME clients communicate with it regardless of current location of its replicas. Because we target large-scale distributed environments with no central control, all algorithms of the RME abstraction should be decentralized.

RMEs can be implemented as fault-tolerant long-living services. In Chapter 9 we present a generic approach and an associated algorithm to achieve fault-tolerant long-living services. Our approach is based on replicating a service using finite state machine replication [74] combined with *automatic* reconfiguration of replica set. A replicated state machine guarantees the consistency among replicas as long as a majority of replicas are alive. However, replication by itself is insufficient to

guarantee long-term fault-tolerance under continuous churn, as the number of failed nodes hosting ME replicas, and hence the number of failed replicas, will increase over time, and eventually RME will stop. Therefore, we use *service migration* [75] to enable the reconfiguration of the set of nodes hosting ME replicas (replica set). Using service migration, new nodes can be introduced to replace the failed ones. We propose a decentralized algorithm [76] that will use migration to *automatically* reconfigure the replica set. This will guarantee that RME will tolerate continuous churn. our algorithm uses P2P replica placement schemes to place replicas and uses the P2P overlay to monitor them. The replicated state machine is extended to analyze monitoring data in order to decide on when and where to migrate. We describe how to use our approach to achieve robust management elements. We have done a simulation-based evaluation of our approach. Evaluation results show the feasibility of our approach.

Data Consistency

The emergence of Web 2.0 opened the door to new applications by allowing users to do more than just retrieving of information. Web 2.0 applications facilitate information sharing, and collaboration between users. The wide spread of Web 2.0 applications, such as, wikis, social networks, and media sharing, resulted in a huge amount of user generated data that places great demands and new challenges on storage services. An Internet-scale Web 2.0 application serves a large number of users. This number tends to grow as popularity of the application increases. A system running such application requires a scalable data engine that enables the system to accommodate the growing number of users while maintaining a reasonable performance. Low (acceptable) response time is another important requirement of Web 2.0 applications that needs to be fulfilled despite of uneven load on application servers and geographical distribution of users. Furthermore, the system should be highly available as most of the user requests must be handled even when the system experiences partial failures or has a large number of concurrent requests. As traditional database solutions could not keep up with the increasing scale, new solutions, which can scale horizontally, were proposed, such as, PNUTS [15] and Dynamo [77].

However there is a trade-off between availability and performance on one hand and data consistency on the other. As proved in the CAP theorem [78], for distributed systems only two properties out of the three – Consistency, Availability and Partition-tolerance – can be guaranteed at any given time. For large scale systems, that are geographically distributed, network partition is unavoidable [79]; therefore only one of the two properties, either data consistency or availability, can be guaranteed in such systems. Many Web 2.0 applications deal with one record at a time, and employ only key based data access. Complex querying, data management and ACID transactions of relational data model are not required in such systems. Therefore for such applications a NoSQL key-value store would suffice.

Also Web 2.0 applications can cope with relaxed consistency as, for example, it is acceptable if one's blog entry is not immediately visible for some of the readers.

Another important aspect associated with Web 2.0 applications is the privacy of user data. Several issues lead to increasing concerns of users, such as, where the data is stored, who owns the storage, and how stored data can be used (e.g., for data mining). Typically a Web 2.0 application provider owns datacenters where user data are stored. User data are governed by a privacy policy. However, the provider may change the policy from time to time, and users are forced to accept this if they want to continue using the application. This resulted in many lawsuits during the past few years and a long debate about how to protect user privacy.

Peer-to-Peer (P2P) networks [1] offers an attractive solution to Web 2.0 storage systems. First, because they are scalable, self-organized, and fault-tolerant; second, because they are typically owned by the community, rather than a single organization, thus allow to solve the issue of privacy.

We propose a P2P-based object store with a flexible read/write API [80] allowing the developer of a Web 2.0 application to trade data consistency for availability in order to meet requirements of the application. Our design uses quorum-based voting as a replica control method [81]. Our proposed replication method provides better consistency guarantees than those provided in a classical DHT [16] but yet not as expensive as consistency guarantees of Paxos based replication [82]

Our key-value store is built on top of a DHT using Chord algorithms [83]. Our store benefits from the inherent scalability, fault-tolerance and self-management properties of a DHT. However, classical DHTs lack support for strong data consistency required in many applications. Therefore a majority-based quorum technique is employed in our system to provide strong data consistency guarantees. As mentioned in [84], this technique, when used in P2P systems, is probabilistic and may lead to inconsistencies. Nevertheless, as proved in [84], the probability of getting consistent data using this technique is very high (more than 99%). This guarantee is enough for many Web 2.0 applications that can tolerate relaxed consistency.

To evaluate our approach, we have implemented a prototype of our key-value store and measured its performance by simulating the network using real traces of Internet latencies.

This work is presented in more details in Chapter 10.

3.3 Self-Management for Cloud-Based Storage Systems: Automation of Elasticity

Many large-scale Web 2.0 applications leverage elastic services, such as elastic key-value stores, that can scale horizontally by adding/removing servers. Voldemort [9], Cassandra [10], and Dynamo [11] are few examples of elastic storage services.

Cloud computing [3], with its pay-as-you-go pricing model, provides an attractive environment to provision elastic services as the running cost of such services becomes proportional to the amount of resources needed to handle the current work-

load. The independent peak load for different applications enables Cloud providers to efficiently share the resources among them. However, sharing the physical resources among Virtual Machines (VMs) running different applications makes it challenging to model and predict the performance of the VMs [39, 40] in order to control and optimize the performance of Cloud-based services and applications.

Managing the resources for Web 2.0 applications, in order to guarantee acceptable performance, is challenging because of the diurnal and sudden (spikes) variations in the workload [41]. It is difficult to predict the workload particularly for new applications that can become popular within few days [12, 13]. Furthermore, the performance requirements are usually expressed in terms of upper percentiles which is more difficult to maintain than average performance [11, 14].

The pay-as-you-go pricing model, elasticity, and dynamic workload of Web 2.0 applications altogether call for the need for an elasticity controller that automate the provisioning of Cloud resources. The elasticity controller leverages the horizontal scalability of elastic services by provisioning more resources under high workloads in order to meet required SLOs. The pay-as-you-go pricing model provides an incentive for the elasticity controller to release extra resources when they are not needed once the workload decreases.

In our work on self-management for Cloud-based storage systems, presented in Part IV, we focus on the automation of elasticity. To design elasticity controllers we have taken a control theoretic approach (feedback and feedforward control combined with machine learning) which, we believe, is promising for elasticity control. Using this approach to building an elasticity controller requires system identification for feedback control and/or building of a model of a system for feedforward control. In this work, we have conducted research on a number of different designs of the elasticity controller, including a feedback PID controller, a State-Space feedback controller [85], and a controller that combines feedback and feedforward control (Chapter 12). In the last case we have used machine learning to build a model of the system for the feedforward control.

We start by reporting our experience in designing an elasticity controller based on State-Space feedback control for a key-value storage service deployed in a Cloud environment. Automation of elasticity is achieved by providing a feedback controller that continuously monitors the system and automatically changes (increases or decreases) the number of nodes in order to meet Service Level Objectives (SLOs) under high load and to reduce costs under low load. We believe that this approach to automate elasticity has a considerable potential for practical use in many Cloud-based services and Web 2.0 applications including services for social networks, data stores, online storage, live streaming services. Another contribution presented in this work is an open-source simulation framework called EStoreSim (Elastic key-value Store Simulator) that allows developers to simulate an elastic key-value store in a Cloud environment and be able to experiment with different controllers. More details about this work is presented in Chapter 11.

We continue by proposing the design of ElastMan, an *Elast*icity *Man*ager for elastic key-value stores running in Cloud VMs. ElastMan addresses the challenges

of the variable performance of Cloud VMs and stringent performance requirements expressed in terms of upper percentiles by combining both feedforward and feedback control. The feedforward controller monitors the workload and uses a simple model of the service to predict whether the current workload will violate the workload or not and acts accordingly. The feedforward is used to quickly respond to sudden changes (spikes) in workload. The feedback controller monitors the performance of the service and reacts based on the amount of deviation from the desired performance specified in the SLO. The feedback controller is used to correct errors in the model used by the feedforward controller and to handle diurnal workload. Due to the nonlinearities in Cloud services, resulting from the diminishing reward of adding a service instance (VM) with increasing the scale, we propose a scale-independent model used to design feedback controller. Our design leverages the near-linear scalability of elastic service. The feedback controller controls the elasticity (the number of servers running the key-value store) indirectly by controlling the average workload per server. This enables our feedback controller to operate at various scales of the service. Chapter 12 presents more details on ElastMan. The major contributions of our work are as follows.

- We leverage the advantages of both feedforward and feedback control to build an elasticity controller for elastic key-value stores running in Cloud environments.
- We propose a scale-independent feedback controller suitable for horizontally scaling services running at various scales.
- We describe the complete design of ElastMan including various techniques necessary for elastic Cloud-based services.
- We evaluate effectiveness of the core components of ElastMan using the Voldemort [9] key-value store running in a Cloud environment against both diurnal and sudden variations in workload.

Chapter 4

Thesis Contributions

In this chapter, we present a summary of the thesis contributions. We start by listing the publications that where produced during the thesis work. Next, we discuss in more details the author's contributions.

4.1 List of Publications

List of publications included in this thesis (in the order of appearance)

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. Vladimir Vlassov, Ahmad Al-Shishtawy, Per Brand, and Nikos Parlavantzas, "Self-Management with Niche, a Platform for Self-Managing Distributed Applications," In Formal and Practical Aspects of Autonomic Computing and Networking: Specification, Development and Verification (editor: Phan Cong-Vinh), IGI Global, Dec. 2011 (book chapter). Available: `http://dx.doi.org/10.4018/978-1-60960-845-3`

4. Ahmad Al-Shishtawy, Muhammad Asif Fayyaz, Konstantin Popov, and Vladimir Vlassov, "Achieving Robust Self-Management for Large-Scale Distributed Applications," *The Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2010)*, Budapest, Hungary, 2010. Available: `http://dx.doi.org/10.1109/SASO.2010.42`

5. Ahmad Al-Shishtawy, Tareq Jamal Khan, Vladimir Vlassov, "Robust Fault-Tolerant Majority-Based Key-Value Store Supporting Multiple Consistency Levels," IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), pp.589-596, 7-9 Dec. 2011. Available: `http://dx.doi.org/10.1109/ICPADS.2011.110`

6. M. Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov, "State-Space Feedback Control for Elastic Distributed Storage in a Cloud Environment," The 8th International Conference on Autonomic and Autonomous Systems (ICAS 2012), St. Maarten, Netherlands Antilles, March 25-30, 2012. **Best Paper Award**. Available: `http://www.thinkmind.org/download.php?articleid=icas_2012_1_40_20127`

7. Ahmad Al-Shishtawy and Vladimir Vlassov, "ElastMan: Autonomic Elasticity Manager for Cloud-Based Key-Value Stores", Technical Report TRITA-ICT/ECS R 12:01, KTH Royal Institute of Technology, August, 2012. (Submitted to a conference).

List of publications by the thesis author that are related to this thesis (in reverse chronological order)

1. 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 2010)*, Budapest, Hungary, September 27-October 1, 2010. Available: http://dx.doi.org/10.1109/SASOW.2010.72

2. 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

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. 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.

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 *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

4.2 Contributions

In this section, we report the major contributions of the publications presented in this thesis including the author's contribution to each of them.

Niche: A Platform for Self-Managing Distributed Application

Our work on building the Niche platform was published as book chapter [73], two conference papers [71, 86], two workshop papers [62, 87], and a poster [88]. The paper [71] appears as Chapter 6 in this thesis and the book chapter [73] appears as Chapter 7 in this thesis.

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.

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. Niche uses the Fractal component model [33]. We extended the Fractal component model by introducing the concept of component groups and bindings to groups. 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 (such as automatic handling of join/leave/failure events) of such networks and services make them attractive for large scale systems and applications. Other properties include decentralization, scalability and fault tolerance. Niche is built on top of the robust and churn tolerant services that are provided by or implemented using a SON similar to Chord [34]. 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. In Niche, 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. Niche has been evaluated by implementing a number of self-managing demonstrator applications.

Thesis Author Contribution

This was a joint work between researchers from the KTH Royal Institute of Technology, 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. These services include 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 application.

A Design Methodology for Self-Management in Distributed Environments

Our work on control loop interaction patterns and design methodology for self-management was published as a conference paper [47] and a workshop paper [89]. The paper [47] appears as Chapter 8 in this thesis.

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.

We define 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 paper. 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.

Achieving Robust Self-Management for Large-Scale Distributed Applications

Our work on replication of management elements was published as a conference paper [90]. The paper appears as Chapter 9 in the thesis.

To simplify the development of autonomic managers, and thus large scale distributed systems, it is useful to separate the maintenance of Management Elements 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).

We propose an approach to implement RMEs that is based on state machine replication [74] combined with *automatic* reconfiguration of replica set. We have developed a decentralized algorithm that *automatically* reconfigures the set of nodes hosting ME replicas using service migration [75]. Our approach guarantees that RME will tolerate continuous churn as long as a majority of replicas is alive. The contributions of this work are as follows.

- The use of *Structured Overlay Networks* (SONs) [1] to monitor the nodes hosting replicas in order to detect changes that may require reconfiguration. SONs are also used to determine replica location using replica placement schemes such as symmetric replication [91].

- The replicated state machine, beside replicating a service, receives monitoring information and uses it to construct a new configuration and to decide when to migrate.

- A decentralized algorithm that automates the reconfiguration of the replica set in order to tolerate continuous resource churn.

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 this chapter including writing most of the paper. The author also contributed to the implementation and the simulation experiments.

Robust Fault-Tolerant Majority-Based Key-Value Store Supporting Multiple Consistency Levels

Our work on the majority-based key-value stores was published as a conference paper [80]. The paper appears as Chapter 10 in the thesis.

The huge amount of user generated data on the Web places great demands and challenges on storage services. A storage service should be highly available and scalable in order to accommodate the growing number of users while maintaining an acceptable performance and consistency of data despite of uneven load and geographical distribution of users. As traditional database solutions could not keep up with the increasing scale, new solutions, which can scale horizontally, were proposed, such as, PNUTS [15] and Dynamo [77].

According to the CAP theorem [78], only two properties out of the three – consistency, availability and partition-tolerance – can be guaranteed at any given time. As for large systems the network partitioning is unavoidable [79], we propose a P2P-based object store with a flexible read/write API allowing the developer of a Web 2.0 application to trade data consistency for availability in order to meet requirements of the application. Our design uses quorum-based voting as a replica control method [81].

Our key-value store is implemented as a DHT using Chord algorithms [83]. Our store benefits from the inherent scalability, fault-tolerance and self-management properties of a DHT. However, classical DHTs lack support for strong data consistency required in many applications. Therefore a majority-based quorum technique is employed in our system to provide strong data consistency guarantees.

Our proposed replication method provides better consistency guarantees than those provided in a classical DHT [16] but yet not as expensive as consistency guarantees of Paxos based replication [82]

Thesis Author Contribution

The author was the main contributor in proposing the idea and developing associated algorithms. The author proposed porting the PNUTS [15] API to P2P overlay networks in order to improve data consistency guarantees as well as proposing the use quorum-based voting as a replica control method. The author participated and supervised the development of the simulator and evaluation experiments.

State-Space Feedback Control for Elastic Distributed Storage in a Cloud Environment

Our work on State-Space feedback control was published as a conference paper [85]. The paper appears as Chapter 11 in the thesis.

Many large-scale Web 2.0 applications leverage elastic services, such as elastic key-value stores, that can scale horizontally by adding/removing servers. Voldemort [9], Cassandra [10], and Dynamo [11] are few examples of elastic storage services. Efforts are being made to automate elasticity in order to improve system performance under dynamic workloads

We present our experience in designing an elasticity controller based of State-Space feedback control for a key-value storage service deployed in a Cloud environment. Automation of elasticity is achieved by providing a feedback controller that continuously monitors the system and automatically changes (increases or decreases) the number of nodes in order to meet Service Level Objectives (SLOs) under high load and to reduce costs under low load. We believe that this approach to automate elasticity has a considerable potential for practical use in many Cloud-based services and Web 2.0 applications including services for social networks, data stores, online storage, live streaming services.

Another contribution presented in this work is an open-source simulation framework called EStoreSim (Elastic key-value Store Simulator) that allows developers to simulate an elastic key-value store in a Cloud environment and be able to experiment with different controllers.

Thesis Author Contribution

The author played a major role in the discussions and studies of several possible approaches to control elastic services deployed in the Cloud. The author was also a main contributor in the development of the proposed approach of using state-space to control Cloud-based services including system identification and controller design. The author played a major role in writing the paper.

ElastMan: Autonomic Elasticity Manage for Cloud-Based Key-Value Stores

Our most recent work on ElastMan elasticity manager was submitted to a conference and is under evaluation. The paper appears as Chapter 12 in the thesis.

Cloud computing [3], with its pay-as-you-go pricing model, provides an attractive environment to provision elastic services as the running cost of such services becomes proportional to the amount of resources needed to handle the current workload.

Managing the resources for Web 2.0 applications, in order to guarantee acceptable performance, is challenging because of the highly dynamic workload that is composed of both diurnal and sudden (spikes) variations [41].

The pay-as-you-go pricing model, elasticity, and dynamic workload of Web 2.0 applications altogether call for the need for an elasticity controller that automate the provisioning of Cloud resources. The elasticity controller leverages the horizontal scalability of elastic services by provisioning more resources under high workloads in order to meet required service level objectives (SLOs). The pay-as-you-go pricing model provides an incentive for the elasticity controller to release extra resources when they are not needed once the workload decreases.

We present the design and evaluation of ElastMan, an *Elast*icity *Man*ager for elastic key-value stores running in Cloud VMs. ElastMan addresses the challenges of the variable performance of Cloud VMs and stringent performance requirements expressed in terms of upper percentiles by combining feedforward control and feedback control. The feedforward controller monitors the workload and uses a simple model of the service to predict whether the current workload will violate the workload or not and acts accordingly. The feedforward is used to quickly respond to sudden changes (spikes) in workload. The feedback controller monitors the performance of the service and reacts based on the amount of deviation from the desired performance specified in the SLO. The feedback controller is used to correct errors in the model used by the feedforward controller and to handle diurnal changes in workload. The major contributions of our work are the following.

- We leverage the advantages of both feedforward and feedback control to build an elasticity controller for elastic key-value stores running in Cloud environments.
- We propose a scale-independent feedback controller suitable for horizontally scaling services running at various scales.
- We describe the complete design of ElastMan including various techniques necessary for elastic Cloud-based services.
- We evaluate effectiveness of the core components of ElastMan using the Voldemort [9] key-value store running in a Cloud environment against both diurnal and sudden variations in workload.

Thesis Author Contribution

The author was the main contributor in the development of this work. The author's contribution includes:

- Proposed the use of both feedforward control and feedback control in order to efficiently deal with the challenges of controlling elastic key-value stores running in Cloud VMs under strict performance requirements;
- Proposed a scale-independent design of the feedback controller that enables it to work at various scales of the store;
- Full implementation of ElastMan for controlling the elasticity of the Voldemort [9] key-value store.
- Creating a testbed based on Open-Stack [92] and then deploying and evaluating ElastMan;
- The author was the main contributor in writing the paper.

Chapter 5

Conclusions and Future Work

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

5.1 The Niche Platform

A large scale distributed application deployed in dynamic environments requires aggressive support for self-management. The proposed distributed component management system, Niche, enables the development of distributed component based applications with self-management 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 to run in different environment by changing management and also allows management code to be reused in different applications.

Niche leverages the self-organizing 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 enable 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 facilitates 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.

We have defined 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 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 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.

5.2 Robust Self-Management and Data Consistency in Large-Scale Distributed Systems

We have proposed the concept of Robust Management Elements (RMEs) which are able to heal themselves under continuous churn. Using RMEs allows the developer to separate the issue of robustness of management from the actual management mechanisms. This will simplify the construction of robust autonomic managers. We have presented an approach to achieve RMEs which uses replicated state machines and relies on our proposed algorithms to automate replicated state machine migration in order to tolerate churn. Our approach uses symmetric replication, which is a replica placement scheme used in Structured Overlay Networks (SONs) to decide on the placement of replicas and uses SON to monitor them. The replicated state machine is extended, beyond its main purpose of providing the service, to process monitoring information and to decide when to migrate. Although in this thesis we discussed the use of our approach to achieve RMEs, we believe that this approach is generic and can be used to replicate other services.

In order to validate and evaluate our approach, we have developed a prototype of RMEs and conducted various simulation experiments, which have shown the validity and feasibility of our approach. Evaluation has shown that the performance (latency and number of messages) of our approach mostly depends on the replication degree rather than on the overlay size.

For the data consistency, we have presented a majority-based key-value store (architecture, algorithms, and evaluation) intended to be deployed in a large-scale dynamic P2P environment. The reason for us to choose such unreliable environment over datacenters is mainly to reduce costs and improve data privacy. Our store provides a number of read/write operations with multiple consistency levels and with semantics similar to PNUTS [15].

The store uses the majority-based quorum technique to maintain consistency of replicated data. Our majority-based store provides stronger consistency guarantees than guarantees provided in a classical DHT but less expensive than guarantees of Paxos-based replication. Using majority allows avoiding potential drawbacks of a master-based consistency control, namely, a single-point of failure and a potential performance bottleneck. Furthermore, using a majority rather than a single master allows the system to achieve robustness and withstand churn in a dynamic environment. Our mechanism is decentralized and thus allows improving load balancing and scalability.

Evaluation by simulation has shown that the system performs rather well in terms of latency and operation success ratio in the presence of churn.

5.3 Self-Management for Cloud-Based Storage Systems

Elasticity in Cloud computing is an ability of a system to scale up and down (request and release resources) in response to changes in its environment and workload.

Elasticity provides an opportunity to scale up under high workload and to scale down under low workload to reduce the total cost for the system while meeting SLOs. We have presented our experience in designing an elasticity controller for a key-value store in a Cloud environment and described the steps in designing it including system identification and controller design. The controller allows the system to automatically scale the amount of resources while meeting performance SLO, in order to reduce SLO violations and the total cost for the provided service. We also introduced our open source simulation framework (EStoreSim) for Cloud systems that allows to experiment with different controllers and workloads. We have conducted two series of experiments using EStoreSim. Experiments have shown the feasibility of our approach to automate elasticity control of a key-value store in a Cloud using state-space feedback control. We believe that this approach can be used to automate elasticity of other Cloud-based services.

The strict performance requirements posed on the data-tier in a multi-tier Web 2.0 application together with the variable performance of Cloud virtual machines makes it challenging to automate the elasticity control. We presented the design and evaluation of *ElastMan*, an Elasticity Manager for Cloud-based key-value stores that address these challenges.

ElastMan combines and leverages the advantages of both feedback and feedforward control. The feedforward control is used to quickly respond to rapid changes in workload. This enables us to smooth the noisy signal of the 99th percentile of read operation latency and thus use feedback control. The feedback controller is used to handle diurnal workload and to correct errors in the feedforward control due to the noise that is caused mainly by the variable performance of Cloud VMs. The feedback controller uses a scale-independent design by indirectly controlling the number of VMs by controlling the average workload per VM. This enables the controller, given the near-linear scalability of key-value stores, to operate at various scales of the store.

We have implemented and evaluated ElastMan using the Voldemort key-value store running in a Cloud environment based on OpenStack. The results shows that ElastMan can handle both diurnal workload and quickly respond to rapid changes in the workload.

Our evaluation of using control theoretic approach to automation of elasticity, that we have done by simulation as well as by implementing elasticity controllers for the Voldemort [9] key-value store, shows effectiveness and feasibility of using elements of control theory combined with machine learning for automation on elasticity of Cloud-based services.

5.4 Discussion and Future Work

Autonomic computing initiative was started by IBM [5] in 2001 to overcome the problem of growing complexity related to computing systems management that hinders further developments of complex systems and services. The goal was to

make computer systems self-managing in order to reduce obstacles caused by management complexity.

Many solutions have been proposed to achieve this goal. However, most of the proposed solutions aim at reducing management costs in a centralised or clustered environments rather than enabling complex large scale systems and services. Control theory [21] is a 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 and achieve 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. Therefore we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. This leads to the next challenge that 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 prevent 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) [93] and Model Predictive Control (MPC) [94] are two method of process control that has been in use in industry (e.g., NCS is used to control large factories, MPC is used in process industries such as oil refineries). NCSs faces similar problems related to network delays as distributed computing systems. MPC has been in use in the process industries to depict the behaviour of complex dynamical systems. We believe that MPC is very promising for automation of certain functions that require fast reaction (e.g., handling peak loads).

Our future work includes investigating the developing distributed algorithms based on NCS and MPC to increase observability and 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 standby 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.

A Web 2.0 application is a complex system consisting of multiple components. Controlling the entire system typically involves multiple controllers, with different management objectives, that interact directly or indirectly [47]. In our future work, we plan to investigate the controllers needed to control all tiers of a Web 2.0 application an the orchestration needed between the controllers in order to correctly achieve their goals.

We also plan to complete our implementation and evaluation of ElastMan to include the proportional thresholding technique as proposed by Lim et al. [43] in order to avoid possible oscillations in the feedback control. We also plan to build the feedforward controller for the store in the rebalance mode. We expect that this will enable the system to adapt to changes in workload that might happen during the rebalance operation. Although our work focused on elastic storage services running in Cloud environments, we beleive that it can be applied to other elastic services in the Cloud which is in our future work.

Since ElastMan runs in the Cloud, it is necessary in real implementation to use replication in order to guarantee fault tolerance. One possible way is to use Robust Management Elements [90], that is based on replicated state machines, to replicate ElastMan and guarantee fault tolerance.

Part II

Enabling and Achieving Self-Management for Large-Scale Distributed Systems

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] KTH Royal Institute of Technology, 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 [5] 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 [18] of component-based applications [33] have been shown useful for self-repair of applications running on clusters [50].

We present a design of a component management platform supporting self-* applications for community-based Grids, and illustrate it with an application. Community-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 [33]. 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 [95]. 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 [95] 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.

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

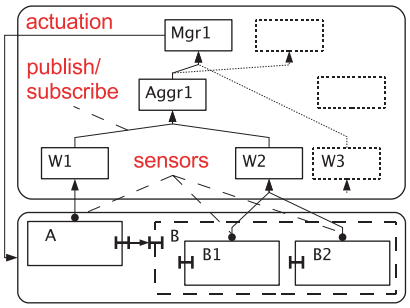

Figure 6.1: Application Architecture.

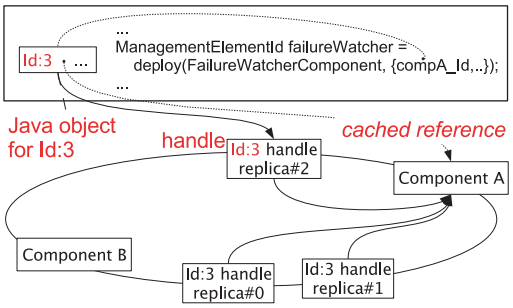

Figure 6.2: Ids and Handlers.

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 [50] 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 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.

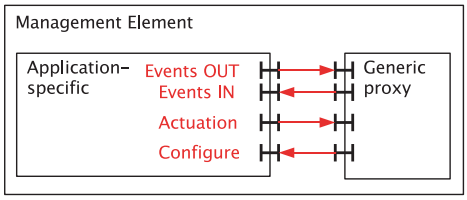

Figure 6.3: Structure of MEs.

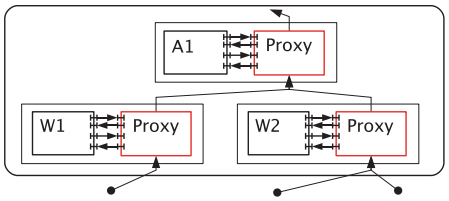

Figure 6.4: Composition of MEs.

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 [95], 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. 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.

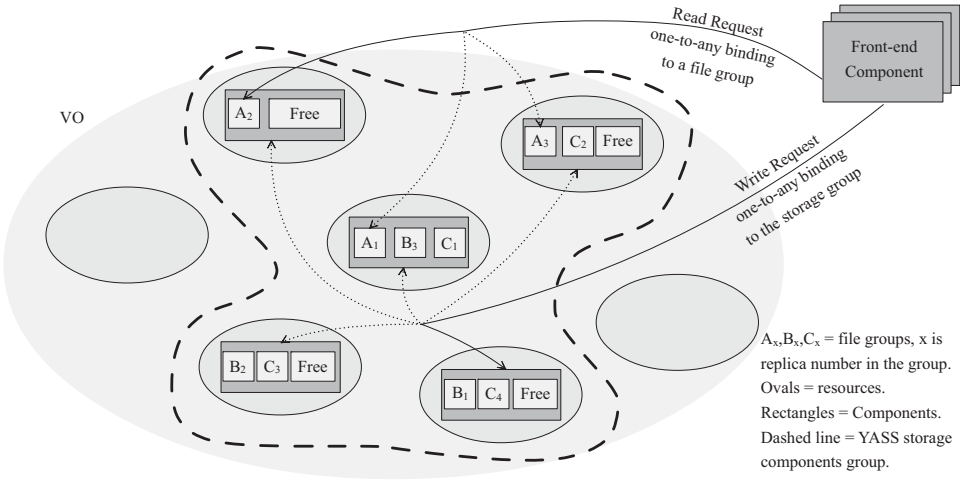

Figure 6.5: YASS Functional Part

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 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.

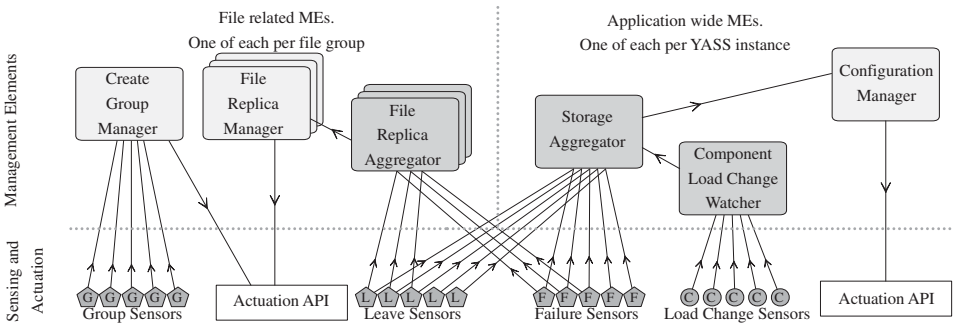

Figure 6.6: YASS Non-Functional Part

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 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

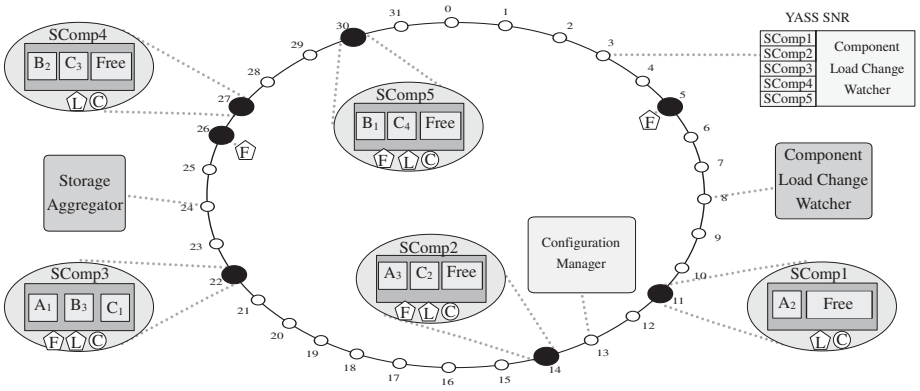

Figure 6.7: Parts of the YASS application deployed on the management infrastructure.

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 available 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

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
```

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.

6.4 Related Work

Our work builds on the technical work on the Jade component-management system [50]. 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.

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
```

Autonomic Management. The vision of autonomic management as presented in [5] 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 [68]. 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 [96] 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 [97], allow dynamic flexibility, but their associated infrastructure lacks support for operation in dynamic environments.

The Grid Component Model, GCM [98], 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 con-

tracts, 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* [99] 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 [100]. 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 [101], which extends the DERMI [102] 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.

Chapter 7

Niche: A Platform for Self-Managing Distributed Application

Vladimir Vlassov, Ahmad Al-Shishtawy,
Per Brand, and Nikos Parlavantzas

In *Formal and Practical Aspects of Autonomic Computing and Networking: Specification, Development, and Verification* (P. Cong-Vinh, ed.), ch. 10, pp. 241–293, IGI Global, 2012. ISBN13: 9781609608453.

Niche: A Platform for Self-Managing Distributed Application

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

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

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

[3]Université Européenne de Bretagne, France
nikolaos.parlavantzas@inria.fr

Abstract

We present Niche, a general-purpose, distributed component management system used to develop, deploy, and execute self-managing distributed applications. Niche consists of both a component-based programming model as well as a distributed runtime environment. It is especially designed for complex distributed applications that run and manage themselves in dynamic and volatile environments.

Self-management in dynamic environments is challenging due to the high rate of system or environmental changes and the corresponding need to frequently reconfigure, heal, and tune the application. The challenges are met partly by making use of an underlying overlay in the platform to provide an efficient, location-independent, and robust sensing and actuation infrastructure, and partly by allowing for maximum decentralization of management.

We describe the overlay services, the execution environment, showing how the challenges in dynamic environments are met. We also describe the programming model and a high-level design methodology for developing decentralized management, illustrated by two application case studies.

7.1 Introduction

Autonomic computing [5] is an attractive paradigm to tackle the problem of growing software complexity by making software systems and applications self-managing. Self-management, namely self-configuration, self-optimization, self-healing, and self-protection, can be achieved by using autonomic managers [6]. An autonomic manager continuously monitors software and its execution environment and acts to meet its management objectives. Managing applications in dynamic environments with dynamic resources and/or load (like community Grids, peer-to-peer systems, and Clouds) is especially challenging due to large scale, complexity, high resource churn (e.g., in P2P systems) and lack of clear management responsibility.

This chapter presents the Niche platform [103] for self-managing distributed applications; we share our practical experience, challenges and issues, and lessons learned when building the Niche platform and developing self-managing demonstrator applications using Niche. We also present a high-level design methodology (including design space and steps) for developing self-managing applications.

Niche is a general-purpose, distributed component management system used to develop, deploy, and execute self-managing distributed applications or services in different kinds of environments, including very dynamic ones with volatile resources. Niche is both a component-based programming model that includes management aspects as well as a distributed runtime environment.

Niche provides a programming environment that is especially designed to enable application developers to design and develop complex distributed applications that will run and manage themselves in dynamic and volatile environments. The volatility may be due to the resources (e.g., low-end edge resources), the varying load, or the action of other applications running on the same infrastructure. The vision is that once the infrastructure-wide Niche runtime environment has been installed, applications that have been developed using Niche, can be installed, and run with virtually no effort. Policies cover such issues as which applications to scale down or stop upon resource contention. After deployment the application manages itself, completely without human intervention, excepting, of course, policy changes. During the application lifetime the application is transparently recovering from failure, and tuning and reconfiguring itself on environmental changes such as resource availability or load. This cannot be done today in volatile environments, i.e., it is beyond the state-of-the-art, except for single machine applications and the most trivial of distributed applications, e.g., client/server.

The rest of this chapter is organized as follows. The next section lays out the necessary background for this work. Then, we discuss challenges for enabling and achieving self-management in a dynamic environment characterized by volatile resources and high resource churn (leaves, failures and joins of computers). Next, we present Niche. We provide some insight into the Niche design ideas and its architecture, programming model and execution environment, followed by a presentation of programming concepts and some insight into the programming of self-managing distributed applications using Niche illustrated with a simple example of a self-healing distributed group service. Next, we present our design methodology (including design space and design steps) for developing a management part of a self-managing distributed application in a decentralized manner, i.e., with multiple interactive autonomic managers. We illustrate our methodology with two demonstrator applications, which are self-managing distributed services developed using Niche. Next, we discuss combining a policy-based management (using a policy language and a policy engine) with hard-coded management logic. Finally, we present some conclusions and our future work.

7.2 Background

The benefits of self-managing applications apply in all kinds of environments, and not only in dynamic ones. The alternative to self-management is management by humans, which is costly, error-prone, and slow. In the well-known IBM Autonomic Computing Initiative [5] the axes of self-management were self-configuration, self-healing, self-tuning and self-protection. Today, there is a considerable body of work in the area, most of it geared to clusters.

However, the more dynamic and volatile the environment, the more often appropriate management actions to heal/tune/reconfigure the application will be needed. In very dynamic environments self-management is not a question of cost but feasibility, as management by humans (even if one could assemble enough of them) will be too slow, and the system will degrade faster than humans can repair it. Any non-trivial distributed application running in such an environment must be self-managing. There are a few distributed applications that are self-managing and can run in dynamic environments, like peer-to-peer file-sharing systems, but they are handcrafted and special-purpose, offering no guidance to designing self-managing distributed applications in general.

Application management in a distributed setting consists of two parts. First, there is the initial deployment and configuration, where individual components are shipped, deployed, and initialized at suitable nodes (or virtual machine instances), then the components are bound to each other as dictated by the application architecture, and the application can start working. Second, there is dynamic reconfiguration when a running application needs to be reconfigured. This is usually due to environmental changes, such as change of load, the state of other applications sharing the same infrastructure, node failure, node leave (either owner rescinding the sharing of his resource, or controlled shutdown), but might also be due to software errors or policy changes. All the tasks in the initial configuration may also be present in dynamic reconfiguration. For instance, increasing the number of nodes in a given tier will involve discovering suitable resources, deploying and initializing components on those resources and binding them appropriately. However, dynamic reconfiguration generally involves more, because firstly, the application is running and disruption must be kept to a minimum, and secondly, management must be able to manipulate running components and existing bindings. In general, in dynamic reconfiguration, there are more constraints on the order in which configuration change actions are taken, compared to initial configuration when the configuration can be built first and components are only activated after this has been completed.

A configuration may be seen as a graph, where the nodes are components and the links are bindings. Components need suitable resources to host them, and we can complete the picture by adding the mapping of components onto physical resources. This is illustrated in Figure 7.1. On the left we show the graph only, the abstract configuration, while on the right the concrete configuration is shown. The bindings that cross resource boundaries will upon use involve remote invocations,

while those that do not can be invoked locally. Reconfiguration may involve a change in the concrete configuration only or in both the abstract and concrete configurations. Note, that we show the more interesting and challenging aspects of reconfiguration; there are also reconfigurations that leave the graph unchanged but only change the way in which components work by changing component attributes.

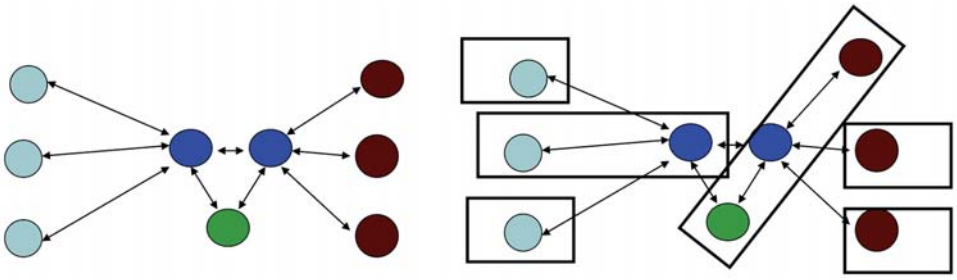

Figure 7.1: Abstract (left) and concrete (right) view of a configuration. Boxes represent nodes or virtual machines, circles represent components.

We now proceed with some examples of dynamic reconfiguration. In these dynamic environments, a resource may announce that it is leaving and a new resource will need to be located and the components currently residing on the resource moved to the new resource. In this case only the concrete configuration is changed. Alternatively, when there is an increase in the number of service components in a service tier this will change the abstract (and concrete) configuration by adding a new node and the appropriate bindings. Another example is when a resource fails. If we disregard the transient broken configuration, where the failed component is no longer present in the configuration and the bindings that existed to it are broken, an identical abstract configuration will eventually be created, differing only in the resource mapping. In general, an application architecture consists of a set of suitable abstract configurations with associated information as to the resource requirements of components. The actual environment will determine which one is best to deploy or to reconfigure towards.

Note that in Figure 7.1 only the top-level components are shown. At a finer level of detail there are many more components, but for our management we can ignore components that are always co-located and bound exclusively to co-located components. Note, that we ignore only those that are always co-located (in all configurations). There are components that might be co-located in some concrete configurations (when a sufficient capable resource is available) but not in others. In Figure 7.1, on the right, a configuration is shown with one machine hosting 3 components; in another concrete configuration they might be mapped to different machines.

We use an architectural approach to self-management, with particular focus

on achieving self-management for dynamic environments, enabling the usage of multiple distributed cooperative autonomic managers for scalability and avoiding a single-point-of failure or contention.

7.3 Related Work

The increasing complexity of software systems and networked environments motivates autonomic system research in both, academia and industry, e.g., [4, 5, 17, 45]. 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 system management functions, including configuration management, fault management, performance management, power management, security management, cost management, SLA management, and SLO management.

There is vast research on building autonomic computing systems using different approaches [45], including control theoretic approach; architectural approach; multi-agent systems; policy-based management; management using utility-functions. For example, authors of [21] apply the control theoretic approach to design computing systems with feedback loops. The architectural approach to autonomic computing [18] suggests specifying interfaces, behavioral requirements, and interaction patterns for architectural elements, e.g., components. The approach has been shown to be useful for autonomous repair management [50]. A reference architecture for autonomic computing is presented in [104]. The authors present patterns for applying their proposed architecture to solve specific problems common to self-managing applications. 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 [51]. Authors of [49] and [48] use multi-objective utility functions for power-aware performance management. Authors of [52] 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 [57–59, 61] allows high-level specification of management objectives in the form of policies that drive autonomic management and can be changed at run time.

Some research is focused on interaction and coordination between multiple autonomic managers. An attempt to analyze and understand how multiple interacting loops can manage a single system has been done in [17] by studying and analyzing existing systems such as biological and software systems. By this study the authors try to understand the rules of a good control loop design. A study of how to compose multiple loops and ensure that they are consistent and complementary is presented in [105]. The authors presented an architecture that supports such compositions.

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 [63] and a number of European projects funded by European Commission such as RESERVOIR, SELFMAN, Grid4All and others.

There are several industrial solutions (tools, techniques and software suites) for enabling and achieving self-management of enterprise IT systems, e.g., IBM 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.

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.

In our work we focus on enabling and achieving self-management for large-scale distributed systems in dynamic environments (dynamic resources and load) using an architectural approach to self-management with multiple distributed cooperative autonomic managers.

7.4 Our Approach

We, like many others, use the feedback control loop approach to achieve self-management. Referring back to Figure 7.1 we can identify the constituent parts of what is needed at runtime.

- **Container:** Each available machine has a container (the boxes in the figure). The container hosts running components and directs actuation (control) commands addressed to a particular component. The container can be told by management to install a new component. Ideally the container can completely isolate and protect components from one another (particularly important when components belonging to different applications are hosted in the same container). This can be achieved by using Virtual Machine technology (currently the containers in Niche do not guarantee this).

- **Sensing:** Management needs to sense or be informed about changes in the application state. Some events are independent of the application type. For example, the failure of a machine (or container) necessarily entails failure of the hosted components, as does the leave of a machine. Other events are application-specific, with a component programmed to report certain events to management (via the management interface of the component). There is a choice with application-independent events (failure and leaves) if the reporting to management is on the level of the container/machine (in which case the

management must make the appropriate mapping to components), or on the level of the individual components.

- **Resource Discovery:** Management needs to sense or be informed about changes in available resources, or alternatively management needs to be able, upon need, to discover free (or underutilized) resources. This could be seen as part of sensing, but note that in general more than a single application is running on the same infrastructure and resource discovery/allocation is an infrastructure-wide service, in contrast to sensing as described above which is directly linked to components in a given application.
- **Actuation:** Management needs to be able to control applications and the components that they are composed of.
- **Management Hosting:** Management needs to be hosted as well. In general the management of a single application is divided into one or more management elements. These management elements are programs that are triggered by some event, perform some planning, and thereafter send the appropriate actuation commands to perform the required reconfiguration.

In a static and constrained environment, these elements of the runtime support may be straightforward or even trivial. For instance, if management is centralized, then the management should know exactly where each application component is hosted, and it is straightforward to send the appropriate command message to a component at its known host. If management is decentralized, it is possible that a component has been moved as a result of the action of another management element without the first management element having been made aware of this. If management never moves, then it is straightforward to find it, and deliver sensing messages to it. If all resources are known statically, then management will always know what resources are potentially available. However, as explained in the next section, to handle dynamic environments we cannot make such simplifying assumptions and the five described elements of the runtime are non-trivial.

The runtime support for management is, of course, only part of the story. Developing the management for a distributed application is a programming task, and a programming model is needed. This will be covered later in the section about the Niche platform.

7.5 Challenges

Achieving self-management in a dynamic environment characterized by volatile resources and high churn (leaves, failures and joins of machines) is challenging. State-of-the-art techniques for self-management in clusters are not suitable. The challenges are:

- **Resource discovery:** Discovering and utilizing free resources;

- **Robust and efficient sensing and actuation:** Churn-tolerant, efficient and robust sensing and actuation infrastructure;
- **Management bottleneck:** Avoiding management bottleneck and single-point-of-failure;
- **Scale**.

In our driving scenarios resources are extremely volatile. This volatility is partly related to churn. There are many scenarios where high churn is expected. In community Grids and other collaborations across the Internet machines may be at any time removed when the owner needs the machine for other purposes. At the edge both the machines and the networks are less reliable.

There are other aspects of volatility. Demanding applications may require more resources than are available in the current infrastructure and additional resources then need to be obtained quickly from an external provider (e.g., Cloud). These new resources need to be integrated with existing resources to allow applications to run over the aggregated resources. Furthermore we do not assume over provisioning within the infrastructure - it may be working close to available capacity so that even smaller changes of load in one application may trigger a reconfiguration as other applications need to be ramped up or down depending on the relative priorities of the applications (according to policy). We see the need for a system-wide infrastructure where volatile resources can efficiently be discovered and utilized. This infrastructure (i.e., the resource discovery service) itself also needs to be self-managing.

The sensing and actuation infrastructure needs to be efficient. The demand for efficiency rules out, at least as the main mechanism, a probing monitoring approach. Instead, the publish/subscribe paradigm needs to be used. The sensing and actuation infrastructure must be robust and churn-tolerant. Sensing events must be delivered (at least once) to subscribing management elements, irrespective of failure events, and irrespective of whether or not the management element has moved. In a dynamic environment it is quite normal for a management element to move from machine to machine during the lifetime of the application as resources leave and join.

It is important that management does not become the bottleneck. For the moment, let us disregard the question of failure of management nodes. The overall management load for a single application depends on both the size of the system (i.e., number of nodes in the configuration graph) and the volatility of the environment. It may well be that a dynamic environment of a few hundred nodes could generate as many events per time unit as a large data centre. The standard mechanism of a single management node will introduce a bottleneck (both in terms of management processing, but also in terms of bandwidth). Decentralization of management is, we believe, the key to solving this problem. Of course, decentralization of management introduces design and synchronization issues. There are issues on how to design management that requires minimal synchronization between the

manager nodes and how to achieve that necessary synchronization. These issues will be discussed later in the section about design methodology.

The issue of failure of management nodes in centralized and decentralized solutions is, on the other hand, not that different. (Of course, with a decentralized approach, only parts of the management fail). If management elements are stateless, fault-recovery is relatively easy. If they are stateful, some form of replication can be used for fault-tolerance, e.g., hot standby in a cluster or state machine replication [90].

Finally, there are many aspects of scale to consider. We have touched upon some of them in the preceding paragraphs, pointing out that we have to take into account the sheer number of environmental sensing events. Clearly the system-wide resource discovery infrastructure needs to scale. But there are other issues to consider regarding scale and efficiency. We have used two approaches in dealing with these issues. The first, keeping in mind our decentralized model of management, is to couple as loosely as possible. In contrast to cluster management systems, not only do we avoid maintaining a centralized system map reflecting the "current state" of the application configuration, we strive for the loosest coupling possible. In particular, management elements only receive event notifications for exactly those events that have been subscribed to. Secondly, we have tried to identify common management patterns, to see if they can be optimized (in terms of number of messages/events or hops) by supporting them directly in the platform as primitives, rather than as programmed abstractions when and if this makes for a difference in messaging or other overhead.

7.6 Niche: A Platform for Self-Managing Distributed Applications

In this section, we present Niche, which is a platform for development, deployment, and execution of component-based self-managing applications. Niche includes a distributed component programming model, APIs, and a runtime system (including a deployment service) that operates on an internal structured overlay network. Niche supports sensing changes in the state of components and an execution environment, and it allows individual components to be found and appropriately manipulated. It deploys both functional and management components and sets up the appropriate sensor and actuation support infrastructure.

Niche has been developed assuming that its runtime environment and applications might execute in a highly dynamic environment with volatile resources, where resources (computers, virtual machines) 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 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).

There are aspects of Niche that are fairly common in autonomic computing. Firstly, Niche supports the feedback control loop paradigm where management logic in a continuous feedback loop senses changes in the environment and component status, reasons about those changes, and then, when needed, actuates, i.e., manipulates components and their bindings. A self-managing application can be divided into a functional part and a management part tied together by sensing and actuation. Secondly, the Niche programming model is based on a component model, called Fractal component model [33], in which components can be monitored and managed. In Fractal, components are bound and interact functionally with each other using two kinds of interfaces: (1) server interfaces offered by the components; (2) and client interfaces used by the components. Components are interconnected by bindings: a client interface of one component is bound to a server interface of another component. Fractal allows nesting of components in composite components and sharing of components. Components have control (management) membranes, with introspection and intercession capabilities. It is through this control membrane that components are started, stopped, configured. It is through this membrane that the components are passivated (as a prelude to component migration), and through which the component can report application-specific events to management (e.g., load). Fractal can be seen as defining a set of capabilities for functional components. It does not force application components to comply, but clearly the capabilities of the programmed components must match the needs of management. For instance, if the component is both stateful and not capable of passivation (or checkpointing) then management will not be able to transparently move the component.

The major novel feature of Niche is that, in order to enable and achieve self-management for large-scale dynamic distributed systems, it combines a suitable component model (Fractal) with a Chord-like structured overlay network to provide a number of robust overlay services. Niche leverages the self-organizing properties of the structured overlay network, e.g., automatic correction of routing tables on node leaves, joins and failures. The Fractal model supports components that can be monitored and managed through component introspection and control interfaces (called controllers in Fractal), e.g., lifecycle, attribute, binding and content controllers. The Niche execution environment provides a number of overlay services, notably, name-based communication, the key-value store (DHT) for lookup services, a controlled broadcast for resource discovery, a publish/subscribe mechanism for event dissemination, and node failure detection. These services are used by Niche to provide higher level 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 interaction. Note that the application programmer does not need to know about the underlying overlay services, this is under the hood, and his/her interaction is through the Niche API.

An important feature of Niche is that all architectural elements such as component interfaces, singleton components, components groups, and management elements, have system-wide unique identifiers. This enables location transparency,

transparent migration and reconfiguration (rebinding) of components and management elements at run time. In Niche, components can be found, monitored and controlled – deployed, created, stopped, rebound, started, etc. 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.

We now proceed to describe both the Niche runtime and, to a lesser extent, the Niche programming model. The Niche programming model will be presented in more detail in the following section interleaved with examples.

Building Management with Niche

Niche implements (in the Java programming language) the autonomic computing reference architecture proposed by IBM in [6], i.e., it allows building MAPE-K (Monitor, Analyze, Plan and Execute; with Knowledge) control loops. An Autonomic Manager in Niche can be organized as a network of Management Elements (MEs) that interact through events, monitor via sensors and act via actuators (e.g., using the actuation API). The ability to distribute MEs among Niche containers enables the construction of decentralized feedback control loops for robustness and performance.

A self-managing application in Niche consists of functional and management parts. Functional components communicate via component bindings, which bind client interfaces to server interfaces; whereas management elements communicate mostly via a publish/subscribe event notification mechanism. 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 of an application can be constructed as a set of interactive or independent control loops each of which monitors some part of the application and reacts on predefined events such as node failures, leaves or joins, component failures, and group membership events; and application-specific events such as component load change events, and low storage capacity events.

In Figure 7.2, we show what an abstract configuration might look like when all management elements are passive in the sense that they are all waiting for some triggering events to take place. The double-headed arrows in the functional part are bindings between components (as the concrete configuration is not shown the bindings may or may not be between different machines). The management elements have references to functional components by name (e.g., component id) or are connected to actuators. The management and functional parts are also “connected”

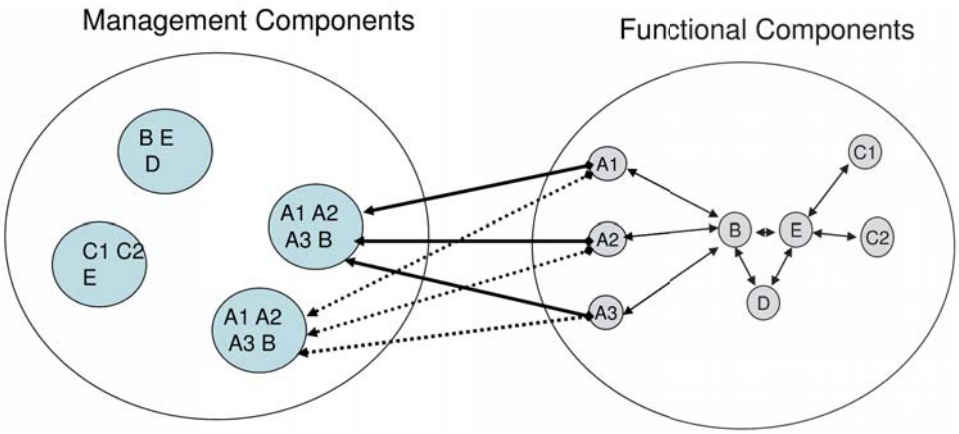

Figure 7.2: Abstract configuration of a self-managing application

by sensors (this is also actually by name, because management, as well as functional components can migrate) In the picture there are sensors from the A group of functional components (A1, A2 and A3) to two management elements (sensors connected to the other management elements are not shown). The management architecture in Figure 7.2 is flat, and later we show how management can be structured hierarchically (see section Development of Self-Managing Applications Using Niche), which is important for larger more complex applications.

The form of a management element is show below, together with a high level description of the features available in the Niche actuation API.

```
loop
     wait SensorEvent
        change internal state   // e.g., for monitoring and aggregation
        analyze/plan
        actuate
```

Actuation is a sequence of invocations (actions) that are listed below (in no specific order). Note that all of the following actions are provided in the Niche actuation API. The list is extensible with user-defined actions.

```
reconfigure existing components    // functional components
                                   //changing concrete configuration only
passivate/move existing components
discover resources     // functional components / changing  configuration.
allocate and deploy new components on a given resource
kill/remove existing components
remove/create bindings
add subscriptions/sensors  // may cause sensors to be installed
remove subscriptions
```

```
discover resources                      // management components
allocate resources and deploy new management elements
trigger events  // for management coordination
```

For implementing the touchpoints (sensors and actuators), Niche leverages the introspection and dynamic reconfiguration features of the Fractal component model in order to provide sensing and actuation API abstractions. Sensors and actuators 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 and component failures, joins, and leaves, as well as modifications in application architecture such as creation of a group.

The application programmer also needs to install/deploy management elements (components). To a large degree this is done in an analogous manner to dealing with functional components. There are two important differences, however. One concerns allocating resources to host management components, and the other concerns connections between management elements. In Niche the application programmer usually lets the Niche runtime find a suitable resource and deploy a management component in one step. Niche reserves a slice of each machine for management activity so that management elements can be placed anywhere (ideally, optimally so as to minimize latency between the management element and its sensors and references). Note that this assumes that the analyze/plan step in management logic are computationally inexpensive. Secondly there are other ways to explicitly share information between management elements, and they are rarely bound to one another (unless they are always co-located). In Figure 7.2, there are no connections between management elements whatsoever, therefore the only coordination that is possible between managers is via stigmergy. Knowledge (as in MAPE-K) in Niche can be shared between MEs using two mechanisms: first, the publish/subscribe mechanism provided by Niche; second, the Niche DHT to store/retrieve information such as references to component group members, name-to-location mappings. In section A Design Methodology for Self-Management in Distributed Environments, we discuss management coordination in more detail in conjunction with design issues involved in the decentralization of management.

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). Currently in Niche such high-level language support includes a declarative ADL (Architecture Description Language) that is used for describing initial configurations at a high-level which is interpreted by Niche at runtime for initial deployment. Policies (supported with a policy language and a corresponding policy engine) can also be used to raise the level of abstraction on management (see section Policy-Based Management).

Execution Environment

The Niche execution environment (see Figure 7.3) is a set of distributed containers (hosting components, groups and management elements) connected via the struc-

tured overlay network, and a number of overlay services including name-based communication, resource discovery, deployment, a lookup service, component group support, the publish/subscribe service for event dissemination including predefined event notification (e.g., component failures). The services allow an application (its management part) to discover and to allocate resources, to deploy the application and reconfigure it at runtime, to monitor and react on changes in the application and in its execution environment, and to locate elements of the application (e.g., components, groups, managers). In this section, we will describe the execution environment. We begin with the aspects of the execution environment that the application programmer needs to be aware of. Thereafter we will describe the mechanisms used to realize the execution environment, and particularly the overlay services. Although the application programmer does not need to understand the underlying mechanisms they are reflected in the performance/fault model. Finally in this section, we describe the performance/fault model and discuss how Niche meets the four challenges discussed in section Challenges.

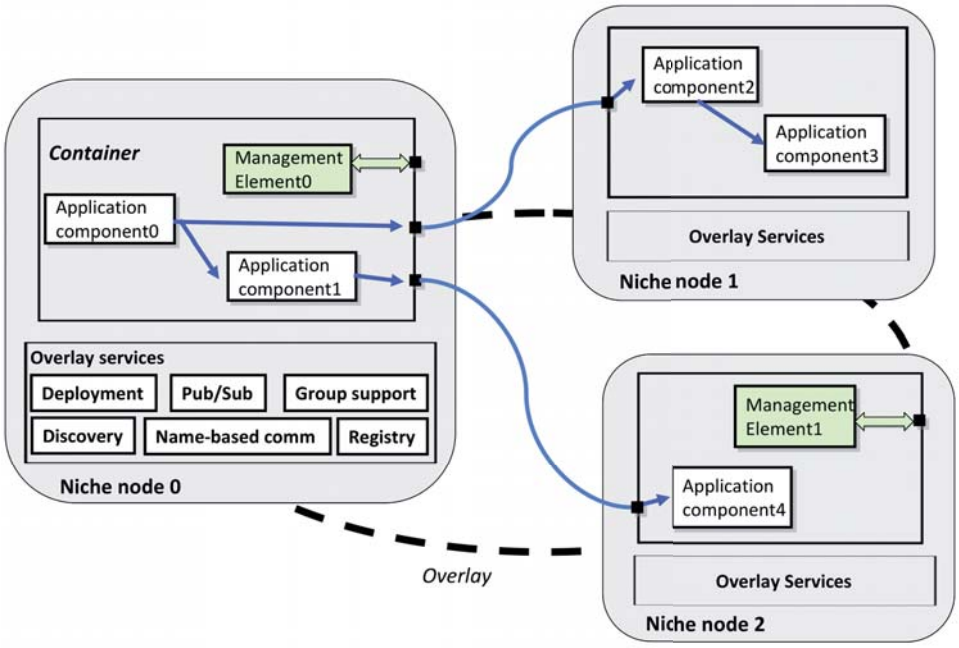

Figure 7.3: Niche architecture

Programmer View

Containers. The Niche runtime environment is a set of distributed containers, called Jade nodes, connected via the Niche structured P2P overlay network. Containers host functional components and management elements of distributed applications executed in Niche. There are two container configurations in the current Niche prototype: (1) the JadeBoot container that bootstraps the system and interprets given ADL (*.fractal) files describing initial configuration of an application on deployment; (2) the JadeNode container, which does not include the ADL interpreter but supports a deployment API to deploy components programmatically.

We use a Webcache PHP application (deployed on an Apache server) to maintain a list of nodes used as access points to join the overlay network. The URL of the Webcache is a part of the configuration information to be provided when installing and configuring the Niche platform. When started, a new Jade node sends an HTTP request to the Webcache to get an address of any of the Jade nodes that can be contacted to join the overlay.

Niche allows a programmer to control the distribution of functional components and management elements among Niche containers, i.e., for every component or/and ME, the programmer can specify the container (by a resource id) where that element should reside (e.g., to co-locate components for efficiency). If a location is not specified, the deployment service of the Niche runtime environment will deploy (or move on failure) an ME on any container selected randomly or in a round-robin manner. Collocation of an ME with a controlled component in the same container allows improving performance of management by monitoring and/or controlling the component locally rather than remotely over the network.

Group Support. Niche provides support for component groups and group bindings. Components can be bound to groups via one-to-any (where a member of the group is chosen at random) or one-to-all bindings. The use of component groups is a fairly common programming pattern. For instance, a tier in a multi-tier application might be modeled as a component group. The application programmer needs to be aware of the fact that component groups are supported directly in the runtime for efficiency reasons (the alternative would be to program a group abstraction).

Resource Discovery and Deployment Service. Niche is an infrastructure that loosely connects available physical resources/containers (computers), and provides for resource discovery. The Niche execution environment is a set of containers (hosting components and managers), which upon joining and leaving the overlay, inform the Niche runtime environment and its applications in a manner completely analogous to peer-to-peer systems (e.g., Chord).

For initial deployment and runtime reconfiguration Niche provides a deployment service (including resource discovery) that can be performed either by the ADL interpreter given an ADL (possibly incomplete) description of architecture of an application to be deployed; or programmatically using a deployment Niche API. ADL-driven deployment of an application does not necessary deploy the entire

application but rather some primary components that in their turn can complete deployment programmatically by executing deployment process logic. A deployment process includes resource discovery, placement and creation of components and component groups, binding component and groups, placement and creation of management elements, subscription to predefined or application-specific events. The deployment service (API) uses the Niche resource discovery service to find resources (Niche containers) with specified properties to deploy components.

All planned removal of resources, like controlled shutdown, should be done by performing a leave action a short time before the resource is removed. It is generally easier for management to perform the necessary reconfiguration on leaves than on failures. Hopefully, management has had the necessary time to successfully move (or kill) the components hosted by the resource by the time the resource is actually removed from the infrastructure (e.g., shut down).

Management Support. In addition to resource discovery and deployment services described above, runtime system support for self-management includes a publish/subscribe service used for monitoring and event-driven management; and a number of server interfaces to manipulate components, groups, and management elements, and to access overlay services (discovery, deployment, and pub/sub).

The publish/subscribe service is used by management elements for publishing and delivering of monitoring and actuation events. The service is accessed though NicheActuatorInterface and TriggerInterface runtime system interfaces described below. The service provides built-in sensors to monitor component and node failures/leaves and group membership changes. The sensors issue corresponding predefined events (e.g., ComponentFailEvent, CreateGroupEvent, MemberAddedEvent, ResourceJoinEvent, ResourceLeaveEvent, ResourceStateChangeEvent), to which MEs can subscribe. A corresponding pub/sub API allows the programmer also to define application-specific sensors and events. The Niche runtime system guarantees event delivery.

The runtime system provides a number of interfaces (available in each container) used by MEs to control the functional part of an application and to access the overlay services (discovery, deployment, pub/sub). The interfaces are automatically bound by the runtime system to corresponding client interfaces of an ME when the management element is deployed and initialized. The set of runtime interfaces includes the following interfaces [103]:

- NicheActuatorInterface (named “actuator”) provides methods to access overlay services, to (un)bind functional components, to manipulate groups, to get access to components in order to monitor and control them (i.e., to register components and MEs with names and to lookup by names). Methods of this interface include, but are not limited to, discover, allocate, deallocate, deploy, redeploy, subscribe, unsubscribe, register, lookup, bind, unbind, create group, remove group, add to group;

- TriggerInterface (named “trigger”) used to trigger events;

- NicheIdRegistry (named "nicheIdRegistry") is an auxiliary low-level interface used to lookup components by system-wide names;

- OverlayAccess (named "overlayAccess") is an auxiliary low-level interface used to obtain access to the runtime system and the NicheActuatorInterface interface.

When developing a management part of an application, the developer should mostly use the first two interfaces. Note that in addition to the above interfaces, the programmer also uses a component and group APIs (Fractal API) to manipulate component and groups for the sake of self-management. Architectural elements (components, groups, MEs) can be located in different Niche containers; therefore invocations of methods of the NicheActuatorInterface interface as well as group and component interfaces can be remote, i.e., cross container boundaries. All architectural elements (components, groups, management elements) of an application are uniquely identified by system-wide IDs assigned on deployment. An element can be registered at the Niche runtime system with a given name to be looked up (and bound with) by its name.

Execution Environment: Internals

Resource Discovery. Niche applications can discover and allocate resources using an overlay-based resource discovery mechanism provided by Niche. Currently the Niche prototype uses a full broadcast (i.e., sends an inquiry to all nodes in the overlay) which scales poorly. However, there are approaches to make broadcast-based discovery more efficient and scalable, such as an incremental controlled broadcast e.g., [106].

Mobility and Location Transparency. The DHT-based lookup (registry) service built into Niche is used to keep information (metadata) on all identifiable architectural elements of an application executed in the Niche environment, such as components, component groups, bindings, management elements, subscriptions. Each architectural element is assigned a system-wide unique identifier (ID) that is used to identify the element in the actuation API. The ID is assigned to the element when the element is created. The ID is used as a key to lookup information about the element in the DHT of the Niche overlay. For most of the element types, the DHT-based lookup service contains location information, e.g., an end-point of a container hosting a given component, or end-points of containers hosting members of a given component group. Being resolved, the location information is cached in the element's handle. If the cached location information is invalid (the element has moved to another container), it will be automatically and transparently updated by the component binding stub via lookup in the DHT. This enables location transparency, transparent migration of component, members of component groups, and management elements at runtime. In order to prevent losing of data on failures of DHT nodes, we use a standard DHT replication mechanism.

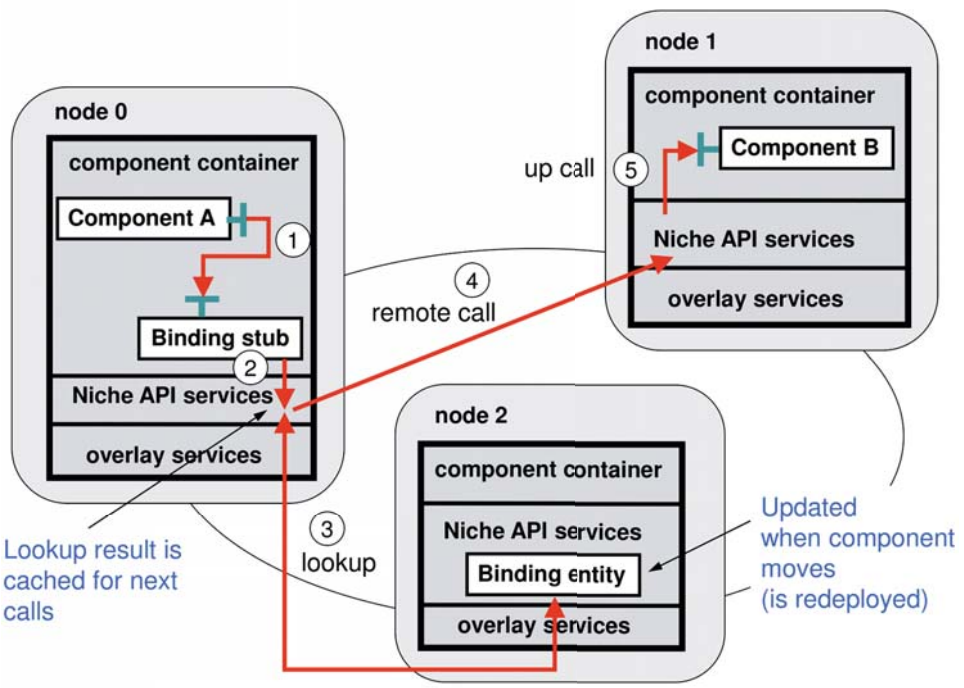

Figure 7.4: Steps of method invocation in Niche

For example, Figure 7.4 depicts steps in executing a (remote) method invocation on a component located in a remote container. Assume a client interface of component A in node 0 is bound to a server interface of component B in node 1; whereas the information about the binding of A to B (i.e., the end-point of B) is stored at node 2. When A makes its first call to B (Step 1), the method call is invoked on the binding stub of B at node 0 (Step 2). The stub performs lookup, using the binding ID as a key, for current location of component B (Step 3). The lookup result, i.e., the end-point reference of B, is cached at node 0 for further calls. When the reference to B is resolved, the stub makes a remote call to the component B using the reference. All further calls to B from node 0 will use the cached end-point reference. If, for any reason, B migrates to another container (not shown in Figure 7.4), the location of B will be updated in the DHT, and the stub of B in node 0 can lookup the new location in the next call to component B. If a node hosting component B fails, a component failure event will be sent to all subscribers, including a manager (if any) responsible for restoring component B in another container. In this case, component A, which is bound to B, does not need to be informed; rebinding of A to the new instance of B is done transparently to A.

Location information is stored in the Niche DHT in the form of a data structure called Set of Network References, SNR, which represents a set of references to identifiable Niche elements (e.g., components, component groups). A component SNR contains one reference, whereas an SNR of a component group contains references to members of the corresponding group. SNRs are stored under their names (used as keys) in the Niche DHT-based key-value store. SNRs are used to find Niche elements by names and can contain either direct or indirect references. A direct reference contains the location of an element; whereas an indirect reference refers to another SNR identified by its name. The indirect reference must be resolved before use. An SNR can be cached by a client in order to improve access time to the referenced element(s). Niche transparently detects out-of-date (invalid) references and refreshes cache contents when needed. Niche supports transparent sensing of elements referenced in an SNR. When a management element is created to control (sense and actuate) functional components referenced by the SNR, the Niche runtime system transparently deploys sensors and actuators for each component. Whenever the references in the SNR are changed, the runtime system transparently (un)deploys sensors and actuators for the corresponding components. For robustness, SNRs are replicated using a DHT replication mechanism. 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 retries SNR access whenever necessary.

Groups are implemented using SNRs containing multiple references. Since a group SNR represents a group, a component bound to the group is actually bound to the group SNR. An invocation through "one-to-any" or "one-to-all" group binding is performed as follows. First, the target group name (the name of the group binding) is resolved to its SNR that contains references to all members of the group. Next, in the case of the one-to-any binding, one of the references is (randomly) selected and the invocation request is sent to the corresponding member of the group. In the case of the one-to-all binding, the invocation request is sent to all members of the group, i.e., to all references in the group SNR. Use of SNRs allows changing the group membership (i.e., growing or shrinking the group) transparently to components bound to the group. Niche supports monitoring of group membership and subscribing to group events issued by group sensors when new members are added or removed from the monitored groups.

Meeting the Challenges

In this section, we discuss how Niche meets the four challenges (see Section Challenges) for self-management in dynamic and volatile environments. The challenges are chiefly concerned with the non-functional properties of the execution environment, so we shall also present the performance/fault model associated with the basic operations of Niche. For most operations the performance model is in terms of network hops, ignoring local computation which is insignificant. Sometimes the number of messages is also taken into account. Clearly, the best that can

be obtained for any remote operation is one or two hops, for asynchronous and synchronous operations, respectively.

Resource Discovery. Niche is an infrastructure that loosely connects available physical resources (computers), and provides for resource discovery by using the structured overlay. Using total broadcast to discover resources means that at most it take O(log N) hops to find the required resource(s) (where N is the number of physical nodes). However, the total number of messages sent is large, O(N). In large systems controlled incremental interval broadcast can be used to decrease the number of messages sent, at the price of greater delay if and when the discovery search needs to be expanded (i.e., when searching for a rare type of available resource). Finally, we note that, often there is actually little net increase in the number of messages, as the resource discovery messages are sent along the same links that continuously need to be probed anyway for overlay self-management.

The use of a structured overlay allows Niche to deal with the first challenge (Resource discovery).

Mobility and Location Transparency. In Niche all the architectural elements are potentially mobile. In much of the Niche actuation API, element identifiers are passed to Niche. An example would be to install a sensor on a given component. Associated with the element identifier is a cached location. If the cached entry is correct, then the action is typically one or two hops, i.e., the minimum. However, due to the action of other management elements the cached location may be invalid in which case a lookup needs to be performed. In the worst case a lookup takes log N hops (where N is the number of physical nodes). What is to be expected depends on the rate of dynamicity of the system. Additionally if the rate of churn is low the overlay can be instrumented so as to decrease the average lookup hops (by increasing the size of routing table at the price of increasing the self-management overhead of the overlay itself).

In our view, the network or location transparency of element identifiers is an important requisite for efficient decentralization of management and directly relates to the second (Robust and efficient sensing and actuation) and third (Management bottleneck) challenges of the previous section. Management elements do not need to be informed when the components that they reference are moved, and neither do sensors need to be informed when the management elements that they reference are moved. For example, in a dynamic environment both a given component and a related management element might be moved (from container to container) many times before the component triggers a high-load event. In this case a DHT-lookup will occur, and the event will reach the management element later than it would be if the location of architectural elements was kept up-to-date, but fewer messages are sent.

Sensing and Actuation. The sensing and actuation services are robust and churn-tolerant, as Niche itself is self-managing. Niche thus meets the second challenge (Robust and efficient sensing and actuation). Niche achieves this by leveraging the self-management properties of an underlying structured overlay. The necessary information to relay events to subscribers (at least once) is stored with redundancy

in the overlay. Upon subscription Niche creates the necessary sensors that serve as the initial detection points. In some cases, sensors can be safely co-located with the entity whose behavior is being monitored (e.g., a component leave event). In other cases, the sensors cannot be co-located. For instance, a crash of a machine will cause all the components (belonging to the same or different applications) being hosted on it to fail. Here the failure sensors need to be located on other nodes. Niche does all this transparently for the developer; the only thing the application developer must do is to use the Niche API to ensure that management elements subscribe to the events that it is programmed to handle, and that components are properly programmed to trigger application-specific events (e.g., load change).

Self-management requires monitoring of the execution environment, components, and component groups. In Niche monitoring is performed by the push rather than pull method for the sake of performance and scalability (the fourth challenge: Scale) using a publish/subscribe event dissemination mechanism. Sensors and management elements can publish predefined (e.g., node failure) and application-specific (e.g., load change) events to be delivered to subscribers (event listeners). Niche provides the publish/subscribe service that allows management elements to publish events and to subscribe to predefined or application-specific events fired by sensors and other MEs. A set of predefined events that can be published by the Niche runtime environment includes resource (node) and component failure/leave events, group change events, component move events, and other events used to notify subscribers (if any) about certain changes in the execution environment and in the architecture of the application. The Niche publish/subscribe API allows the programmer to define application specific events and sensors to issue the events whenever needed. A list of subscribers is maintained in an overlay proxy in the form of an SNR (a Set of Network References described above). The sensor triggers the proxy which then sends the events to subscribers.

Decentralized and Robust Management. Niche allows for maximum decentralization of management. Management can be divided (i.e., parallelized) by aspects (e.g., self-healing, self-tuning), spatially, and hierarchically. Later, we present the design methodology and report on use-case studies of decentralized management. In our view, a single application has many loosely synchronized managers. Niche supports the mobility of management elements. Niche also provides the execution platform for these managers; they typically get assigned to different machines in the Niche overlay. There is some support for optimizing this placement of managers, and some support for replication of managers for fault-tolerance. Thus Niche meets, at least partly, the challenge to avoid the management bottleneck (the third challenge: Management bottleneck). The main reason for the "at least partly" in the last sentence, is that more support for optimal placement of managers, taking into account network locality, will probably be needed (currently Niche recognizes only some special cases, like co-location). A vanilla management replication mechanism is available in the current Niche prototype, and, at the time of writing this chapter, work is ongoing on a robust replicated manager scheme based on the Paxos algorithm, adapted to the Niche overlay [90].

Groups. The fact that Niche provides support for component groups and group bindings contributes to dealing with the fourth challenge (Scale). Supporting component groups directly in the runtime system, rather than as a programming abstraction, allows us to adapt the sensing and actuation infrastructure to minimize messaging overhead and to increase robustness.

7.7 Development of Self-Managing Applications Using Niche

The Niche programming environment enables the development of self-managing applications built of functional components and management elements. Note that the Niche platform [103] uses Java for programming components and management elements.

In this section, we describe in more detail the Niche programming model and exemplify with a Hello World application (singleton and group). The Niche programming model is based on Fractal, a modular and extensible component model intended for designing, implementing, deploying, and reconfiguring complex software systems. Niche borrows the core Fractal concepts, which are components, interfaces, and bindings, and adds new concepts related to group communication, deployment, and management. The following section discusses the main concepts of the Niche programming model and how they are used. Then we describe typical steps of developing a self-managing application illustrated with an example of programming of a self-healing group service.

Niche Programming Concepts

A self-managing application in Niche is built of functional components and management elements. The former constitute the functional part of the application; whereas the latter constitute the management part.

Components are runtime entities that communicate exclusively through named well-defined access points, called interfaces, including control interfaces used for management. Component interfaces are divided into two kinds: client interfaces that emit operation invocations and server interfaces that receive them. Interfaces are connected through communication paths, called bindings. Components and interfaces are named in order to lookup component interfaces by names and bind them.

Components can be primitive or composite, formed by hierarchically assembling other components (called sub-components). This hierarchical composition is a key Fractal feature that helps managing the complexity of understanding and developing component systems.

Another important Fractal feature is its support for extensible reflective facilities, allowing inspection and adaptation of the component structure and behavior. Specifically, each component is made of two parts: the membrane, which embodies

reflective behavior, and the content, which consists of a finite set of sub-components. The membrane exposes an extensible set of control interfaces (called controllers) for reconfiguring internal features of the component and to control its life cycle. The control interfaces are server interfaces that must be implemented by component classes in order to be manageable. In Niche, the control interfaces are used by application-specific management elements (namely, sensors and actuators), and by the Niche runtime environment to monitor and control the components, e.g., to (re)bind, change attributes, and start. Fractal defines the following four basic control interfaces: attribute, biding, content, and life-cycle controllers. The attribute controller (AttributeController) supports configuring named component properties. The binding controller (BindingController) is used to bind and unbind client interfaces to server interfaces, to lookup an interface with a given name, and to list all client interfaces of the component. The content controller (ContentController) supports listing, adding, and removing sub-components. Finally, the life-cycle controller (LifeCycleController) supports starting and stopping the execution of a component and getting the component state.

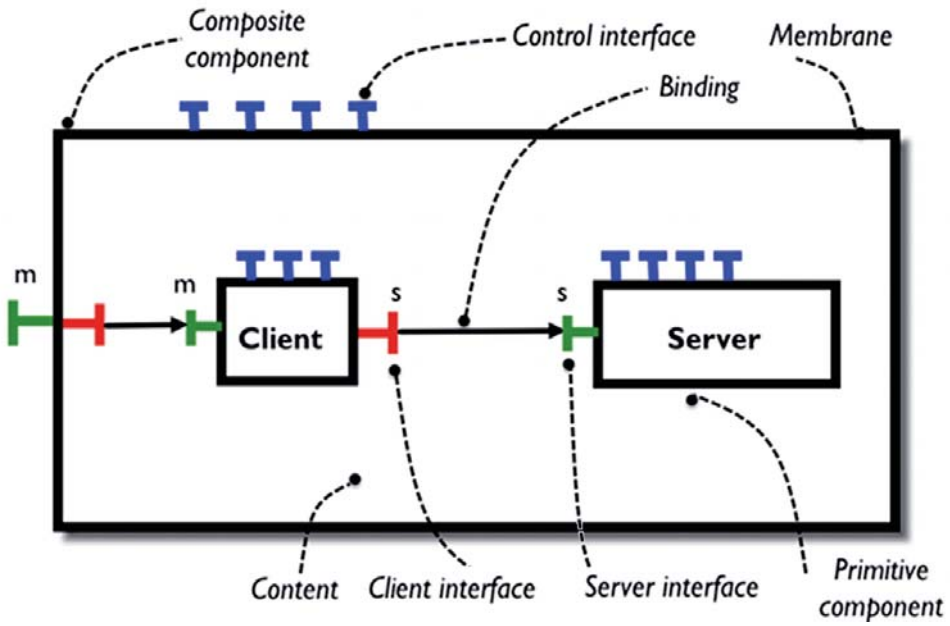

Figure 7.5: A composite Fractal component HelloWorld with two sub-components client and server

The core concepts of the Fractal component model are illustrated in Figure 7.5 that depicts a client-server application HelloWorld, which is a composite Fractal

component containing two sub-components, Client and Server. The client interface of the Client component is bound to the server interface of the Server component. Membranes of components contain control interfaces. Note that on deployment, the composite, the Client, and the Server components can be placed in different containers.

Building a component-based application involves programming primitive components and assembling them into an initial configuration either programmatically, using methods of the NicheActuatorInterface interface of the Niche runtime environment; or declaratively, using an Architecture Description Language (ADL). In the former case, at least one (startup) component must be described in ADL to be initially deployed and started by the ADL interpreter. The startup component can deploy the remaining part of the application by executing a deployment and configuration workflow programmed using the Niche runtime actuation API, which allows the developer to program complex and flexible deployment and configuration workflows. The ADL used by Niche is based on Fractal ADL, an extensible language made of modules, each module defining an abstract syntax for a given architectural concern (e.g., hierarchical containment, deployment). Primitive components are programmed in Java.

Niche extends the Fractal component model with abstractions for group communication (component group, group bindings) as well as abstractions for deployment and resource management (package, node). All these abstractions are described later in this section.

A management part of a Niche application is programmed using the Management Element (ME) abstractions that include Sensors, Watchers, Aggregators, Managers, Executors and Actuators. Note that the distinction between Watchers, Aggregators, Managers and Executors is an architectural one. From the point of view of the execution environment they are all management elements, and management can be programmed in a flat manner (managers, sensors and actuators only). Figure 7.6 depicts a typical hierarchy of management elements in a Niche application. We distinguish different types of MEs depending on the roles they play in self-management code. Sensors monitor components through interfaces and trigger events to notify appropriate management elements about different application-specific changes in monitored components. There are sensors provided by the Niche runtime environment to monitor component failures/leaves (which in turn may be triggered by container/machine failures and leaves), component groups (changes in membership, group creations), and container failures. Watchers receive notification events from a number of sensors, filter and propagate them to Aggregators, which aggregate the information, detect and report symptoms to Managers. A symptom is an indication of the presence of some abnormality in the functioning of monitored components, groups or environment. Managers analyze the symptoms, make decisions and request Executors to act accordingly. Executors receive commands from managers and issue commands to Actuators, which act on components through control interfaces. Sensors and actuators interact with functional components via control interfaces (e.g., life-cycle and biding controllers), whereas management el-

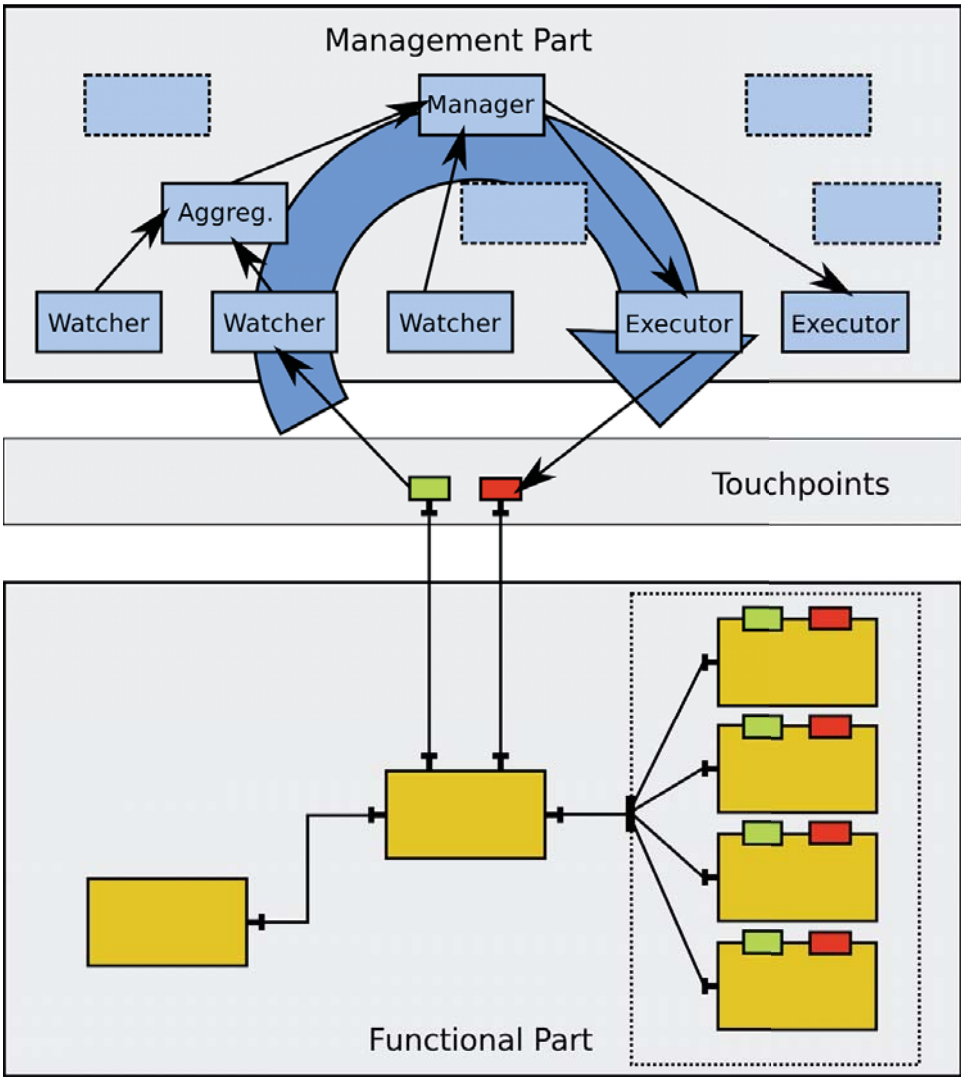

Figure 7.6: Hierarchy of management elements in a Niche application

ements typically communicate by events using the pub/sub service provided by the Niche runtime environment. To manage and to access Niche runtime services, MEs use the NicheActuatorInterface interface bound to the Niche runtime environment which provides useful service and control methods such as discover, allocate, de-allocate, deploy, lookup, bind, unbind, subscribe, and unsubscribe. To publish events, MEs use the TriggerInterface interface of the runtime environment. Both client interfaces, NicheActuatorInterface and TriggerInterface, used by an ME are automatically bound to corresponding server interfaces of the Niche runtime environment when the ME is deployed (created). In order to receive events, an ME must implement the EventHandlerInterface server interface and subscribe to the events of interest.

Development Steps

When developing a self-managing distributed component-based application using Niche, the developer makes the following steps.

1. Development of architecture of the functional and management parts of the application. This step includes the following work: definition and design of functional components (including server and client interfaces) and component groups, assigning names to components and interfaces, definition of component and group bindings, definition and design of management elements including algorithms of event handlers for application-specific management objectives, definition of application-specific monitoring and actuation events, selection of predefined events issued by the Niche runtime environment, definition of event sources and subscriptions.

2. Description of (initial) architecture of functional and management parts in ADL, including components, their interfaces and bindings. Note that it is not necessary to describe the entire configuration in ADL, as components, groups and management elements can be deployed and configured also programmatically using the Niche actuation API rather than the ADL interpreter.

3. Programming of functional and management components. At this stage, the developer defines classes and interfaces of functional and management components, implements server interfaces (functional), event handlers (management), Fractal and Niche control interfaces, e.g., life-cycle and binding controllers.

4. Programming a (startup) component that completes initial deployment and configuration done by the ADL interpreter. An initial part of the application (including the startup component) described in ADL in Step 2 is to be deployed by the ADL interpreter; whereas the remaining part is to be deployed and configured by the programmer-defined startup component using the actuation interface NicheActuatorInterface of the Niche runtime system.

Completion of the deployment might be either trivial if ADL is maximally used in Step 2, or complicated if a rather small part of the application is described in ADL in Step 2. Typically, the startup component is programmed to perform the following actions: bind components deployed by ADL, discover and allocate resources (containers) to deploy components; create, configure and bind components and groups; create and configure management elements and subscribe them to events; and start components.

Programming of Functional Components and Component Groups

This section demonstrates how the above concepts are practically applied in programming the simple client-server HelloWorld application (Figure 7.4) which is a composite component containing two sub-components, Client and Server. The application provides a singleton service that prints a message (the greeting "Hello World!") specified in the client call. In this example, the server component provides a server interface of type Service containing the print method. The client component has a client interface of type Service and a server interface of type Main containing the main method. The client interface of the client component is bound to the server interface of the service component. The composite HelloWorld component provides a server interface that exports the corresponding interface of the client component; its main method is invoked when the application is launched.

Primitive Components

Primitive components are realized as Java classes that implement server interfaces (e.g., Service and Main in the HelloWorld example) as well as any necessary control interfaces (e.g., BindingController). The client component class called ClientImpl, implements the Main interface. Since the client component has a client interface to be bound to the server, the class implements also the BindingController interface, which is the basic control interface for managing bindings. The following code fragment presents the ClientImpl class that implements the Main and the binding controller interfaces. Note that the client interface Service is assigned the name "s".

```
public class ClientImpl implements Main, BindingController {
    // Client interface to be bound to server interface of Server component
    private Service service;
    private String citfName = "s"; // Name of the client interface
    // Implementation of the Main interface
    public void main (final String[] args) {
        // call the service to print the greeting
        service.print ("Hello world!");
     }
    // All methods below belong to the Binding Controller
    // interface with the default implementation
    // Returns names of all client interfaces of the component
    public String[] listFc ( ) {
        return new String[] { citfName };
```

```
    }
    // Returns the interface to which the given client interface is bound
    public Object lookupFc(final String citfName)
                                          throws NoSuchInterfaceException {
        if (!this.citfName.equals(citfName))
           throw new NoSuchInterfaceException(itfName);
        return service;
    }
    // Binds the client interface with the given name
    // to the given server interface
    public void bindFc(final String citfName, final Object sItf)
                                           throws NoSuchInterfaceException {
        if (!this.citfName.equals(citfName))
           throw new NoSuchInterfaceException(itfName);
        service = (Service)sItf;
    }
    // Unbinds the client interface with the given name
    public void unbindFc (final String citfName)
                                            throws NoSuchInterfaceException {
        if (!this.citfName.equals(citfName))
           throw new NoSuchInterfaceException(itfName);
        service = null;
    }
}
```

The server component class, called ServerImpl, implements only the Service interface as shown below.

```
public class ServerImpl implements Service {
public void print (final String msg) {
    for (int i = 0; i < count; ++i)
        System.err.println("Server prints:" + msg);
    }
}
```

Assembling Components

The simplest method to assemble components is through the ADL, which specifies a set of components, their bindings, and their containment relationships, and can be used to automatically deploy a Fractal system. The main concepts of the ADL are component definitions, components, interfaces, and bindings. The ADL description of the HelloWorld application with the singleton service is the following:

```
<definition name="HelloWorld">
    <interface name="m" role="server" signature="Main"/>
    <component name="client">
      <interface name="m" role="server" signature="Main"/>
        <interface name="s" role="client" signature="Service"/>
        <content class="ClientImpl"/>
  </component>
    <component name="server">
        <interface name="s" role="server" signature="Service"/>
        <content class="ServerImpl"/>
```

```
    </component>
    <binding client="this.m" server="client.m" />
    <binding client="client.s" server="server.s" />
</definition>
```

Component Groups and Group Bindings

Niche bindings support communication among components hosted in different machines. Apart from the previously seen, one-to-one bindings, Niche also supports groups and group bindings, which are particularly useful for building decentralized, fault-tolerant applications. Group bindings allow treating a collection of components, the group, as a single entity, and can deliver invocations either to all group members (one-to-all semantics) or to any, randomly-chosen group member (one-to-any semantics). Groups are dynamic in that their membership can change over time (e.g., increase the group size to handle increased load in a tier).

Groups are manipulated through the Niche API, which supports creating groups, binding groups and components, and adding/removing group members. Moreover, the Fractal ADL has been extended to enable describing groups as part of the system architecture.

Figure 7.7 depicts the HelloGroup application, in which the client component is connected to a group of two stateless service components (server1 and server2) using one-to-any invocation semantics. The group of service components provides a service that prints the “Hello World!" greeting by any of the group members on a client request.

The initial configuration of this example application (without management elements) can be described in ADL as follows:

```
<definition name="HelloGroup">
    <interface name="m" role="server" signature="Main"/>
    <component name="client">
        <interface name="m" role="server" signature="Main"/>
        <interface name="s" role="client" signature="Service"/>
        <content class="ClientImpl"/>
    </component>
    <component name="ServiceGroup">
        <interface name="s" role="server" signature="Service"/>
        <interface name="clients" role="client" signature="Service"
                                  cardinality="collection"/>
        <content class="GROUP"/>
  </component>
    <component name="server1">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <component name="server2">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r" />
  <binding client="client.s" server="group.s" bindingType="groupAny"/>
```

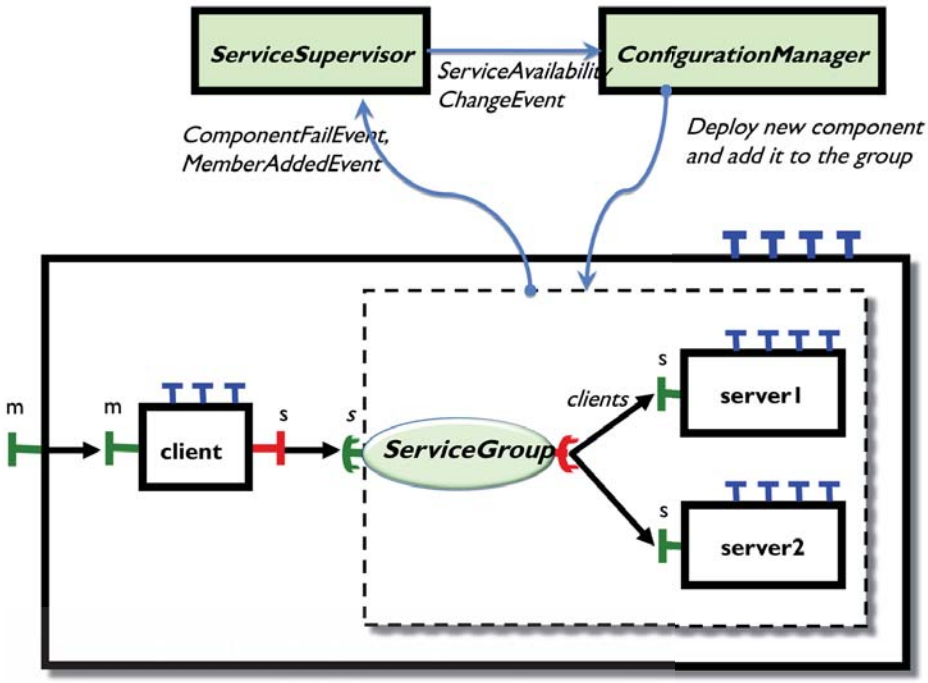

Figure 7.7: HelloGroup application

```
  <binding client="group1.clients1" server="server1.s"/>
  <binding client="group1.clients2" server="server2.s"/>
</definition>
```

As seen in this description, the service group is represented by a special component with content "GROUP". Group membership is then represented as binding the server interfaces of members to the client interfaces of the group. The bindingType attribute represents the invocation semantics (one-to-any in this case). Groups can also be created and bound programmatically using the Niche actuation API (namely the NicheActuatorInterface client interface bound to the Niche runtime system). As an example, the following Java code fragment illustrates group creation performed by a management element.

```
// Code fragment from the StartManager class
// References to the Niche runtime interfaces
// bound on init or via binding controller
private NicheIdRegistry nicheIdRegistry;
private NicheActuatorInterface myActuatorInterface;
...
```

```
// Lookup the client component and all server components by names
ComponentId client =
               (ComponentId) nicheIdRegistry.lookup("HelloGroup _0/client");
ArrayList<ComponentId> servers = new ArrayList();
servers.add((ComponentId) nicheIdRegistry.lookup("HelloGroup _0/server1");
servers.add((ComponentId) nicheIdRegistry.lookup("HelloGroup_0/server2");
// Create a group containing all server components.
GroupId groupTemplate = myActuatorInterface.getGroupTemplate();
groupTemplate.addServerBinding("s", JadeBindInterface.ONE_TO_ANY);
GroupId serviceGroup = myActuatorInterface.createGroup(groupTemplate, servers);
// Bind the client to the group with one-to-any binding
myActuatorInterface.bind(client, "s", serviceGroup,
                         "s", JadeBindInterface.ONE_TO_ANY);
```

Programming of Management Elements

The management part of a Niche application is programmed using the Management Element (ME) abstractions that include Sensors, Watchers, Aggregators, Managers, Executors and Actuators. MEs are typically reactive event-driven components; therefore developing of MEs is mostly programming event handlers, i.e., methods of the EventHandlerInterface server interface that each ME must implement in order to receive sensor events (including user-defined events and predefined events issued by the runtime system) and events from other MEs. The event handler is eventually invoked when a corresponding event is published (generated). The event handlers can be programmed to receive and handle events of different types. A typical management algorithm of an event handler includes, but not necessarily and not limited to, a sequence of conditional if-then(-else or -else-if) control statements (management logic rules) that examine rule conditions (IF clause) based on information retrieved from the received events or/and its internal state (which in turn reflects previous received events as part of monitoring activity); make a management decision and perform management actions and issue events (THEN clause) (see section Policy-Based Management).

When programming an ME class, the programmer must implement the following three server interfaces: the InitInterface interface to initialize an ME instance, the EventHandlerInterface interface to receive and handle events; and the MovableInterface interface to get a checkpoint, when the ME is moved and redeployed for replication or migration (the checkpoint is passed to a new instance through its InitInterface). To perform control actions, to subscribe and publish events, an ME class must include the following two client interfaces: the NicheActuatorInterface interface, named “actuator”; and the TriggerInterface interface, named “trigger”. Both client interfaces are bound to the Niche runtime system when the ME is deployed either through its InitInterface or via the BidingController interface.

When developing the management code of an ME (event handlers) to control the functional part of an application and to subscribe to events, the programmer uses methods of the NicheActuatorInterface client interface that includes a number of actuation methods such as discover, allocate, de-allocate, deploy, create a

component group, add a member to a group, bind, unbind, subscribe, unsubscribe. Note that the programmer can subscribe/unsubscribe to predefined built-in events (e.g., component failure, group membership change) issued by built-in sensors of the Niche runtime system. To publish events, the programmer uses the TriggerInterface client interface of the ME.

For example, Figure 7.7 depicts the HelloGroup application that provides a group service with self-healing capabilities. Feedback control in the application maintains the group size (a specified minimum number of service components) despite node failures, i.e., if any of the components in the group fails, a new service component is created and added to the group so that the group always contain the given number of servers. The self-healing control loop includes the Service Supervisor aggregator that monitors the number of components in the group, and the Configuration manager that is responsible to create and add a new service component on a request from the Service Supervisor. Figure 7.8 depicts a sequence of events and control actions of the management components. Specifically, if one of the service components of the service group fails, the group sensor issues a component failure event received by the Service Supervisor (1), which checks whether the number of components has dropped below a specified threshold (2). If so, the Server Supervisor fires the Service-Availability-Change event received by the Configuration Manager (3), which heals the component, i.e., creates a new instance of the server component and adds it to the group (4). When a new member is added to the group, the Service Supervisor, which keeps track of the number of server components, is notified by the predefined Member-Added-Event issued by the group sensor (5, 6).

The shortened Java code fragment below shows the management logic of the Configuration Manager responsible for healing of a failed server component upon receiving a Service-Availability-Change event issued by the Service Supervisor (steps 3 and 4 in Figure 7.8)

```
// Code fragment from the ConfigurationManager class
public class ConfigurationManager
                    implements EventHandlerInterface, MovableInterface,
    InitInterface, BindingController, LifeCycleController {
private static final String DISCOVER_PREFIX = "dynamic:";
// Reference to the Actuation interface of the Niche runtime
// (automatically bound on deployment).
private NicheActuatorInterface myManagementInterface;
...
// invoked by the runtime system
public void init(NicheActuatorInterface managementInterface) {
        myManagementInterface = managementInterface;
    }

// invoked by the runtime system on deployment
public void init(Serializable[] parameters) {
        initAttributes = parameters;
        componentGroup = (GroupId) initAttributes[0];
        serviceCompProps = initAttributes[1];
```

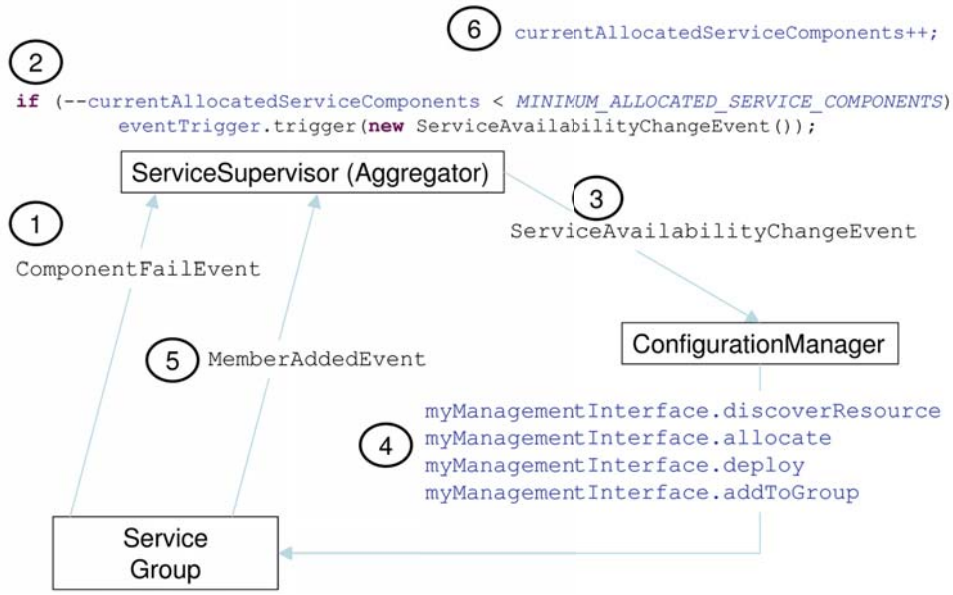

Figure 7.8: Events and actions in the self-healing loop of the HelloGroup application

```
        nodeRequirements = DISCOVER_PREFIX + initAttributes[2];
}
...
 // event handler, invoked on an event
public void eventHandler(Serializable e, int flag) {
        // For any case, check event type,
        // ignore if it is not the event of interest (should not happen)
        if (! (e instanceof ServiceAvailabilityChangeEvent)) return;
        // Find a node that meets the requirements for a server component.
        try {
            newNode =
               myManagementInterface.oneShotDiscoverResource( nodeRequirements);
        } catch (OperationTimedOutException err) {
            ... // Retry later (the code is removed)
        }
        // Allocate resources for a server component at the found node.
        try {
            List allocatedResources =
                                myManagementInterface.allocate(newNode, null);
        } catch (OperationTimedOutException err) {
            ... // Retry later (the code is removed)
        }
         ...
        String deploymentParams = Serialization.serialize(serviceCompProps);
        // Deploy a new server component instance at the allocated node.
```

```
try {
    deployedComponents = myManagementInterface.deploy( allocatedResource,
                                                        deploymentParams );
} catch (OperationTimedOutException err) {
... // Retry later (the code is removed)
}
ComponentId cid = (ComponentId)((Object[])deployedComponents.get(0))[1];
// Add the new server component to the service group and start the server
myManagementInterface.update(componentGroup, cid,
               NicheComponentSupportInterface.ADD_TO_GROUP_AND_START);
}
```

While MEs interact with each other mostly by events, sensors and actuators are programmed to interact with functional components via interface bindings. Interfaces between sensors and components are defined by the programmer, who may choose to use either the push or pull methods of interaction between a sensor and a component. In the case of the push method, the component pushes the sensor to issue an event. In this case, the component's client interface is bound to the corresponding sensor's server interface. In the case of the pull method, a sensor pulls the state from a component. In this case, the sensor's client interface is bound to a corresponding component's server interface. A sensor and a component are auto-bound when the sensor is deployed by a watcher. Actuation (control actions) can be done by MEs either through actuators bound to functional components or directly on components via their control interfaces using the Niche actuation API. Actuators are programmed in a similar way as sensors and are deployed by executors. By analogy to sensors, an actuator can be programmed to interact with a controlled component in the push and/or pull manner. In the former case (push), the actuator pushes a component through component's control interfaces, which can be either application-specific interfaces defined by the programmer or the Fractal control interfaces, e.g., LifeCycleController and AttributeController. In the case of the pull-based actuation, the controlled component checks its actuator for actions to be executed.

Deployment and Resource Management

Niche supports component deployment and resource management through the concepts of component package and node. A component package is a bundle that contains the executables necessary for creating components, the data needed for their correct functioning as well as metadata describing their properties. A node is the physical or virtual machine on which components are deployed and executed. A node provides processing, storage, and communication resources, which are shared among the deployed components.

Niche exposes basic primitives for discovering nodes, allocating resources on those nodes, and deploying components; these primitives are designed to form the basis for external services for deploying components and managing their underlying resources. In the current prototype, component packages are OSGi bundles [107]

and managed resources include CPU time, physical memory, storage space, and network bandwidth. The Fractal ADL has been extended to allow specifying packages and resource constraints on nodes. These extensions are illustrated in the following ADL extract, which refines the client and composite descriptions in the HelloGroup example (added elements are show in Bold).

```
<definition name="HelloGroup">
    <interface name="m" role="server" signature="Main"/>
    <component name="client">
    <interface name="m" role="server" signature="Main"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
    <packages>
              <package name="ClientPackage v1.3" >
                    <property name="local.dir" value="/tmp/j2ee"/>
            </package>
    </packages>
    <virtual-node name="node1" resourceReqs="(&(memory>=1)(CPUSpeed>=1))"/>
    </component>
 <!-- description of other components and bindings  (is not shown) -->
  ...
    <virtual-node name="node1">
</definition>
```

The packages element provides information about the OSGi bundles necessary for creating a component; packages are identified with their unique name in the OSGi bundle repository (e.g., "ClientPackage v1.3"). The virtual-node element describes resource and location requirements of components. At deployment time, each virtual node is mapped to a node (container) that conforms to the given resource requirements specified in the resourceReqs attribute. The necessary bundles are then installed on this node and the associated component is created. In the example, the client and the composite components are co-located at a node with memory larger than 1GB and CPU speed larger than 1Ghz.

Initialization of Management Code

The ADL includes support for initializing the management part of an application in the form of start manager components. Start managers have a predefined definition "StartManagementType" that contains a set of client interfaces corresponding to the Niche API. These interfaces are implicitly bound by the system after start managers are instantiated. The declaration of a start manager is demonstrated in the following ADL extract, which refines the HelloGroup example.

```
<component name="StartManager" definition="org.ow2.jade.StartManagementType">
   <content class=" helloworld.managers.StartManager"/>
 </component>
```

Typically, the start manager contains the code for creating, configuring, and activating the set of management elements that constitute the management part of an

application. In the HelloGroup example, the management part realizes self-healing behavior and relies on an aggregator and a manager, which monitors the server group and maintains its size despite node failures. The start manager implementation (the StartManager class) then contains the code for deploying and configuring the elements of the self-healing loop shown in Figure 7.7 (i.e., ServiceSupervisor and ConfigurationManager). The code is actually located in the implementation of the LifeCycleController interface (startFc operation) of the startup manager, as seen next.

```
// Code fragment from the StartManager class of the HelloGroup application
public class StartManager implements BindingController, LifeCycleController {
// References to the Niche runtime interfaces
// bound on init or via binding controller
private NicheIdRegistry nicheIdRegistry;
private NicheActuatorInterface myActuatorInterface;
...
// Invoked by the Niche runtime system
public void startFc() throws IllegalLifeCycleException {
       ...
    // Lookup client and servers, create service group
    // and bind client to the group (code is not shown)
    GroupId serviceGroup = myActuatorInterface.createGroup(...);
 ...
    // Configure and deploy the Service Supervisor aggregator
  GroupId gid = serviceGroup;
  ManagementDeployParameters params = new ManagementDeployParameters();
    params.describeAggregator( ServiceSupervisor.class.getName(), "SA", null,
                                          new Serializable[] { gid.getId() } );
    NicheId serviceSupervisor =
                     myActuatorInterface.deployManagementElement(params, gid);
    // Subscribe the aggregator to events from group
    myActuatorInterface.subscribe(gid, serviceSupervisor,
                                  ComponentFailEvent.class.getName());
    myActuatorInterface.subscribe(gid, serviceSupervisor,
                                  MemberAddedEvent.class.getName());
    // Configure and deploy the Configuration manager
 String minimumNodeCapacity = "200";
 params = new ManagementDeployParameters();
    params.describeManager(ConfigurationManager.class.getName(), "CM", null,
                           new Serializable[] { gid, fp, minimumNodeCapacity });
    NicheId configurationManager =
                    myActuatorInterface.deployManagementElement( params, gid );
    // Subscribe the manager to events from the aggregator
    myActuatorInterface.subscribe(serviceSupervisor, configurationManager,
                                ServiceAvailabilityChangeEvent.class.getName());
  ...
}
```

Support for Legacy Systems

The Niche self-management framework can be applied to legacy systems by means of a wrapping approach. In this approach, legacy software elements are wrapped as

Fractal components that hide proprietary configuration capabilities behind Fractal control interfaces. The approach has been successfully demonstrated with the Jade management system, which relied also on Fractal and served as a basis for developing Niche [108]. Another example of the use of a "legacy" application (namely the VLC program) in a self-managing application developed using Niche, is the gMovie demo application that performs transcoding of a given movie from one format to another. The description and the code of the gMovie application can be found in [109] and [103].

To briefly illustrate the wrapping approach, consider an enterprise system composed of an application server and a database server. The two servers are wrapped as Fractal components, whose controllers are implemented using legacy configuration mechanisms. For example, the life-cycle controllers are implemented by executing shell scripts for starting or stopping the servers. The attribute controllers are implemented by modifying text entries of configuration files. The connection between the two servers is represented as a binding between the corresponding components. The binding controller of the application server wrapper is then implemented by setting the database host address and port in the application server configuration file.

The wrapping approach produces a layer of Fractal components that enable observing and controlling the legacy software through standard interfaces. This layer can be then complemented with a Niche-based management system (e.g., sensors, actuators, managers), developed according to the described methodology. Of course, the degree of control exposed by the Fractal layer to the management system depends heavily on the legacy system (e.g., it may be impossible to dynamically move software elements). Moreover, the wrapping approach cannot take full advantage of Niche features such as name-based communication and group bindings. The reason is that bindings are only used to represent and manage connections between legacy software elements, not to implement them.

7.8 A Design Methodology for Self-Management in Distributed Environments

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 actuators). Touchpoints enable management of the functional part, i.e., make it manageable.

An Autonomic Manager is a control loop that continuously monitors and affects the functional part of the application when needed. For many applications and en-

vironments 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 avoids 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 decentralized manner given the management requirements and touchpoints.

- **Decomposition:** The first step is to divide the management logic into a number of management tasks. Decomposition can be either functional (e.g., tasks are defined based which self-* properties they implement) or spatial (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. Orchestration of autonomic managers is discussed in the following section.

- **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 section, 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.

Orchestrating Autonomic Managers

Autonomic managers can interact and coordinate their operation in the following four ways as discussed below and illustrated in Figure 7.9: indirect interactions via the managed system (stigmergy); hierarchical interaction (through touch points); direct interaction (via direct bindings); sharing of management elements.

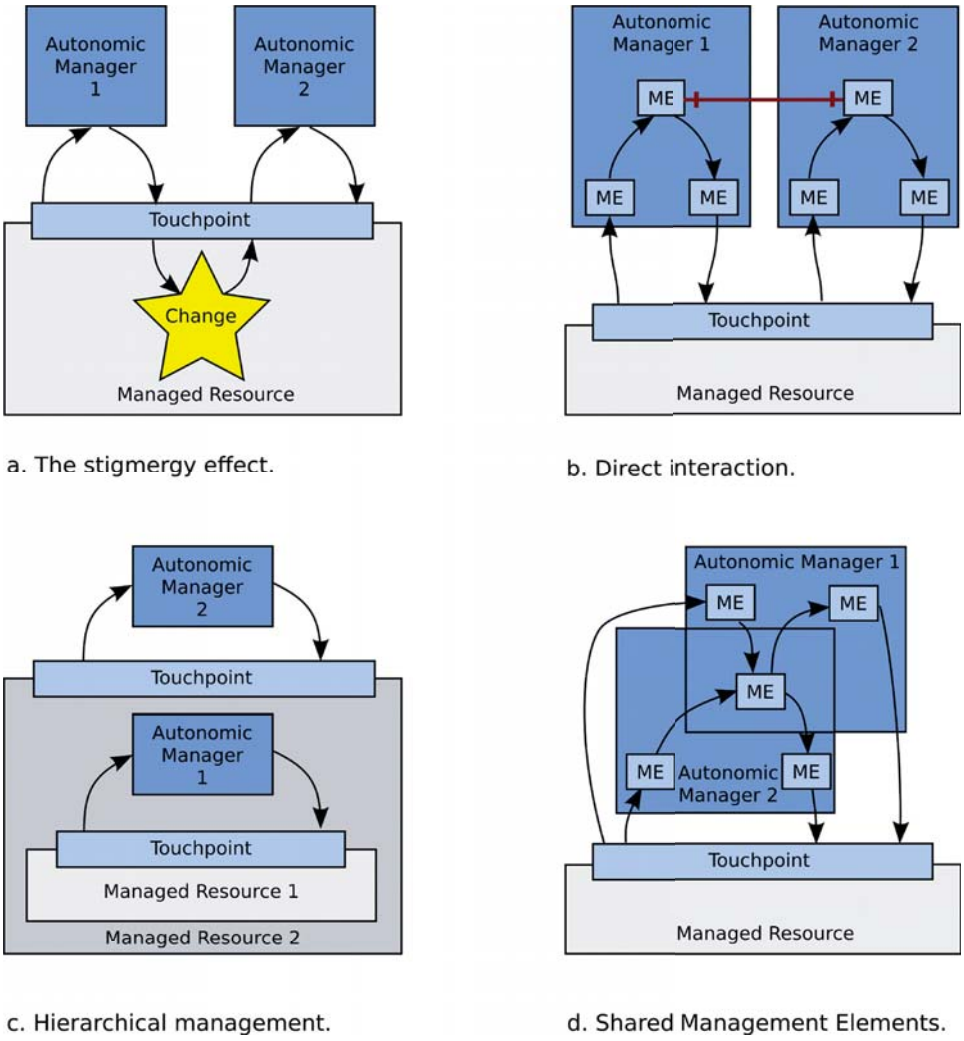

Figure 7.9: Interaction patterns

Stigmergy

Stigmergy is a way of indirect communication and coordination between agents [110]. 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 behavior 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.

Hierarchical Management

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

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

Direct Interaction

Autonomic managers may interact directly with one another. Technically this is achieved by direct communication (via bindings or events) between appropriate management elements in the autonomic managers. Cross autonomic manager bindings can be used to coordinate autonomic managers and avoid undesired behaviors such as race conditions or oscillations.

Shared Management Elements

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

7.9 Demonstrator Applications

In order to demonstrate Niche and our design methodology, we present two self-managing services developed using Niche: (1) a robust storage service called YASS – Yet Another Storage Service; and (2) a robust computing service called YACS

– Yet Another Computing Service. Each of the services has self-healing and self-configuration capabilities and can execute in a dynamic distributed environment, i.e., the services can operate even if computers join, leave or fail at any time. Each of the services implements relatively simple self-management algorithms, which can be extended to be more sophisticated, while reusing existing monitoring and actuation code of the services. The code and documentation of YASS and YACS services can be found at [103].

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 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 nodes leaving or failing. 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.

Demonstrator I: Yet Another Storage Service (YASS)

In order to illustrate our design methodology, we have developed a storage service called YASS (Yet Another Storage Service), using Niche. The case study illustrates 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 7.10. 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.

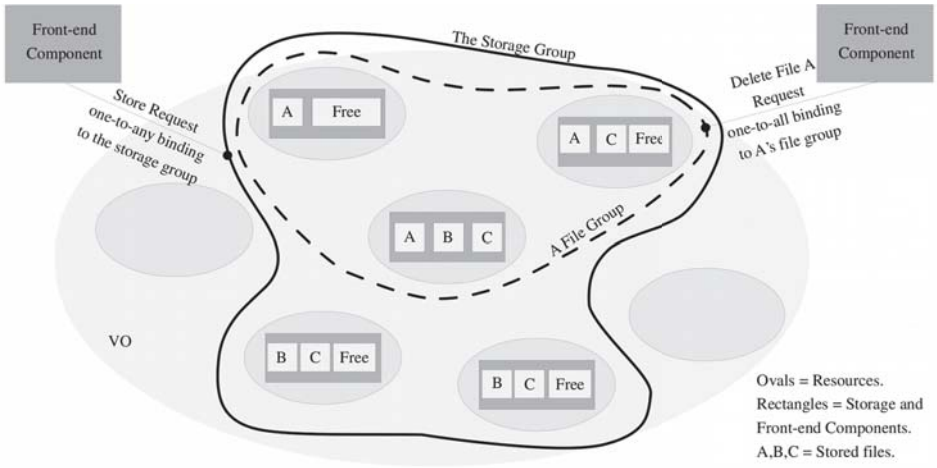

Figure 7.10: YASS functional design

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

These replicas together will form a file group containing the r storage components that will host the file. The front-end will then use a one-to-all binding to the file group to transfer the file in parallel to the r replicas in the group. A read request is sent to any of the r storage components in the group using the one-to-any binding between the front-end and the file group. A delete request is sent to the file group in parallel using a one-to-all binding between the front-end and the file group.

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 for resource failures and component group creation; and actuators for deploying and binding components. Beside the basic touchpoints the following additional, YASS specific, sensors and actuators 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 actuator to add one extra replica of a specified file;
- A move-file actuator 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 described in the previous section on design methodology, are demonstrated below.

Replica Autonomic Manager: The replica autonomic manager is responsible for maintaining the desired replication degree for each stored file in spite of resources failing and leaving. This autonomic manager adds the self-healing property to YASS. The replica autonomic manager consists of two management elements, the File-Replica-Aggregator and the File-Replica-Manager as shown in Figure 7.11. 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).

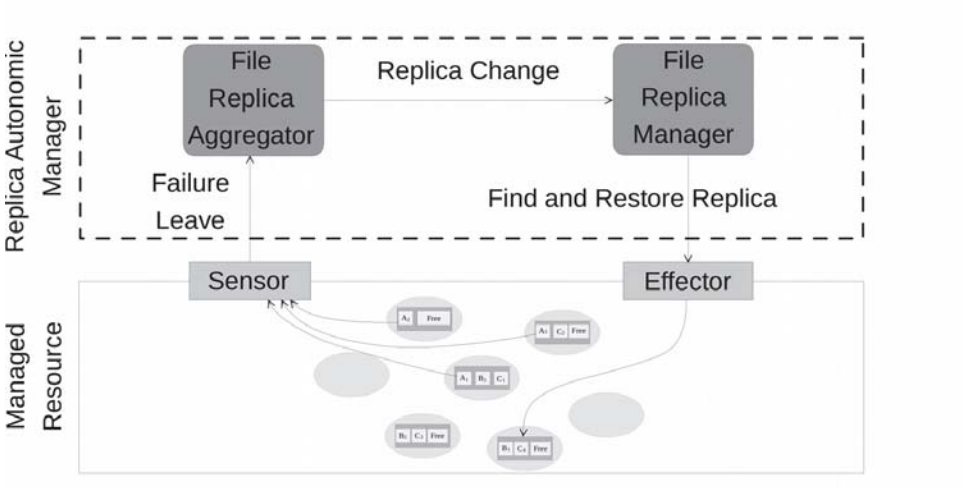

Figure 7.11: Self-healing control loop for restoring file replicas.

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 7.12. 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 threshold. 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 it is possible to have a race condition between the two autonomic managers that will cause the replica autonomic manager to fail.

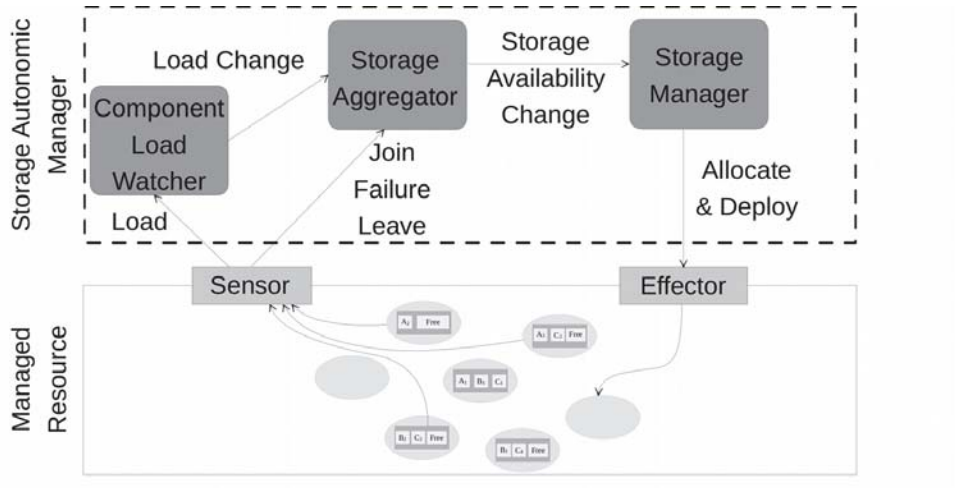

Figure 7.12: Self-configuration control loop for adding storage

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.

Optimizing 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 the following two reasons: first, it allows avoiding duplication of functionality; and second, it allows avoiding oscillation between allocating 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 regulating the replica autonomic manager. The autonomic manager consists of two management elements. The File-Access-Watcher and File-Availability-Manager are shown in Figure 7.13. 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.

Figure 7.13: Hierarchical management used to implement the self-optimization control loop for file availability

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 7.14. 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 in this case. 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.

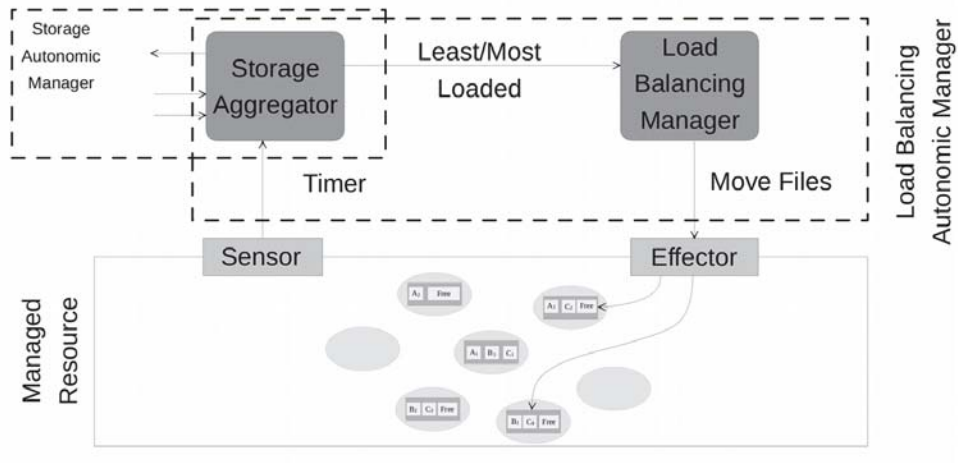

Figure 7.14: Sharing of management elements used to implement the self-optimization control loop for load balancing

Demonstrator II: Yet Another Computing Service (YACS)

This section presents a rough overview of YACS (Yet Another Computing Service) developed using Niche (see [103, 109] for more detail). The major goal in development of YACS was to evaluate the Niche platform and to study design and implementation issues in providing self-management (in particular, self-healing and self-tuning) for a distributed computing service. YACS 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 nodes leaving or failing. YACS supports checkpointing that allows restarting execution from the last checkpoint when a worker component fails or leaves. The YACS includes a checkpoint service that allows the task programmer to perform task checkpointing whenever needed. Furthermore, YACS scales, i.e., changes the number of execution components, whenever the number of jobs/tasks changes. In order to achieve high availability, YACS always maintains a number of free masters and workers so that new jobs can be accepted without delay.

YACS executes jobs, which are collections of tasks, where a task represents instance of work of a particular type that needs to be done. For example, in order to transcode a movie, the movie file can be split into several parts (tasks) to be transcoded independently and in parallel. Tasks are programmed by the user and can be programmed to do just about anything. Tasks can be programmed in any programming language using any programming environment, and placed in a YACS job (bag of independent tasks) using the YACS API.

Figure 7.15 depicts YACS architecture. The functional part of YACS includes

distributed Masters (only one Master is shown in Figure 7.15) and Workers used to execute jobs. A user submits jobs via the YACS Frontend component, which assigns jobs to Masters (one job per Master). A Master finds Workers to execute tasks in the job. When all tasks complete, the user is notified, and results of execution are returned to the user through the YACS frontend. YACS is implemented in Java, and therefore tasks to be executed by YACS can be either programmed in Java by extending the abstract Task class, or wrapped in a Task subclass. The execute method of the Task class has to be implemented to include the task code or the code that invoke the wrapped task. The execute method is invoked by a Worker that performs the task. When the method returns, the Worker sends to its Master an object that holds results and final status of execution. When developing a Task subclass, the programmer can override checkpointing methods to be invoked by the checkpoint service to make a checkpoint or by the Worker to restart the task from its last checkpoint. Checkpoints are stored in files identified by URLs.

There are two management objectives of the YACS management part: (1) self-healing, i.e., to guarantee execution of jobs despite of failures of Masters and Workers, and failures and leaves of Niche containers; (2) self-tuning, i.e., to scale execution (e.g., deploy new Masters and Workers if needed whenever a new Niche container joins the system).

The management elements responsible for self-healing include Master Watchers and Worker Watchers that monitor and control Masters and Workers correspondingly (see Figure 7.15). A Master Watcher deploys a sensor for the Master group it is watching, and subscribes to the component failure events and the state change events that might come from that group. A State Change Event contains a checkpoint (a URL of the checkpoint file) for the job executed by the Master. Master failures are reported by the Component Fail Event that causes the Watcher to find a free Master in the Master group and reassign the failed group to it, or to deploy a new Master instance if there are no free Masters in the group. The job checkpoint is used to restart the job on another Master. A Worker Watcher monitors and controls a group of Workers and responsible for healing Workers and restarting tasks in the case of failures. A Worker Watcher performs in a in a similar way as a Master Watcher described above.

The management elements responsible for self-tuning include Master-, Worker- and Service-Aggregators and the Configuration Manager, which is on top of the management hierarchy. The self-tuning control loop monitors availability of resources (number of Masters and Workers) and adds more resources, i.e., deploys Masters and Workers on available Niche containers upon requests from the Aggregators. The Aggregators collect information about the status of job execution, Master and Workers groups and resources (Niche containers) from Master, Worker and Service Resource Watchers. The Aggregators request the Configuration Manager to deploy and add to the service more Masters and/or Workers when the number of Masters and/or Workers drops (because of failures) below predefined thresholds or when there are not enough Masters and Workers to execute jobs and tasks in parallel.

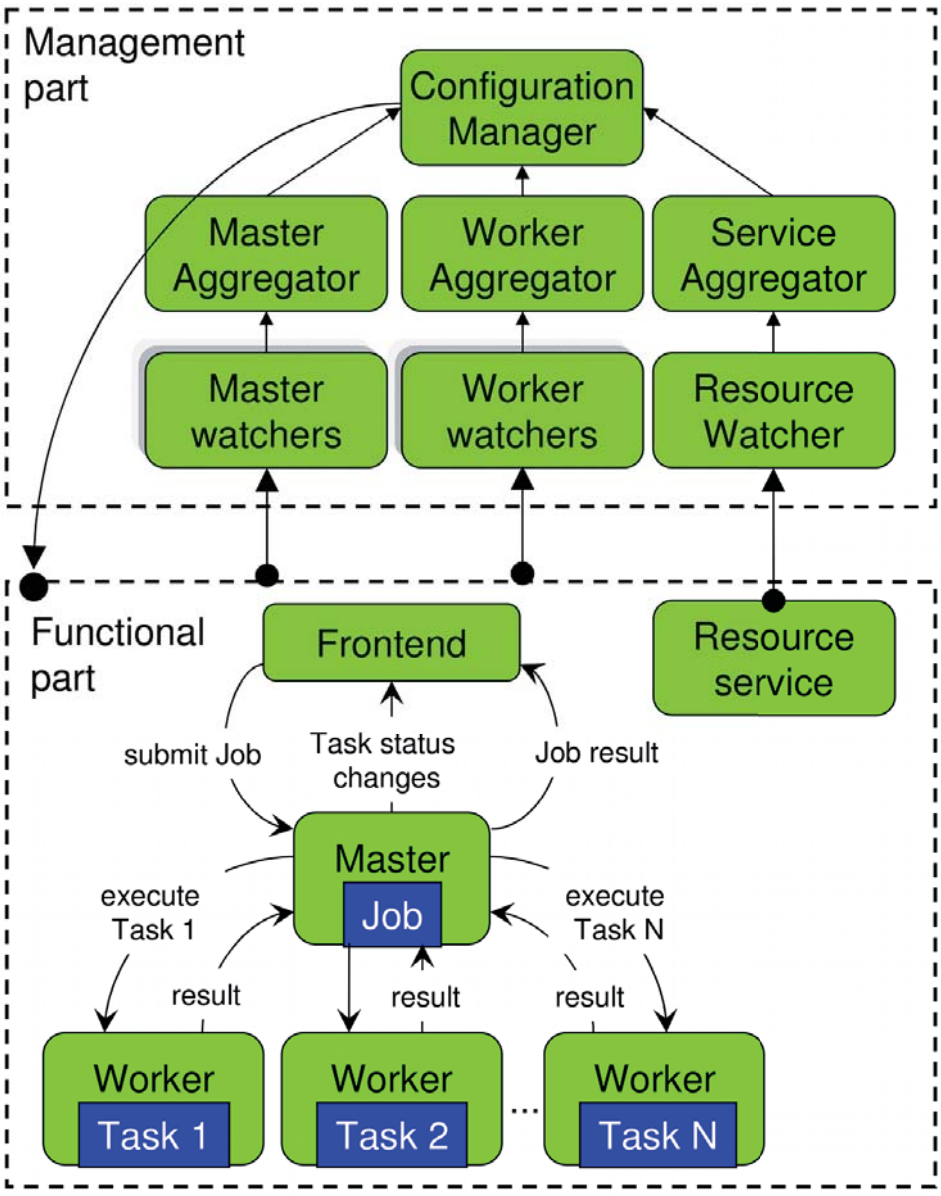

Figure 7.15: Architecture of YACS (yet another computing service)

Evaluation

In order to validate and evaluate the effectiveness of Niche, in terms of efficacy and overheads, the Niche execution environment and both demo applications, YASS (Yet Another Storage Service) and YACS (Yeat Another Computing Services), were tested and evaluated on the Grid5000 testbed (https://www.grid5000.fr/). The performance and overhead of the Niche execution environment was evaluated mostly using specially developed test programs: These confirm the expected performance/fault model presented in section Niche: a Platform for Self-Managing Distributed Applications.

The effectiveness of Niche for developing and executing self-managing applications was validated by YASS, YACS, and, in particular, with the gMovie demo application built on top of YACS. The gMovie application has been developed to validate the functionality and self-* (self-healing and self-configuration) properties of YACS, as well as to validate and evaluate effectiveness and stability of the Niche execution environment. The gMovie application performs transcoding of a given movie from one format to another in parallel on a number of YACS workers. Results of our validation and evaluation indicate that the desired self-* properties, e.g., self-healing in the presence of failures and resource churn can be obtained, and that the programming is not particularly burdensome. Programmers with varying experience were able to learn and understand Niche to the point that they could be productive in a matter of days or weeks. For results of performance evaluation of YACS, the reader is referred ton [109].

7.10 Policy Based Management

So far in our discussion we have shown how to program management logic directly in the management elements using Java (in addition to ADL for initial deployment). However, a part of the analysis and planning phases of the management logic can also be programmed separately using policy languages. Note that currently the developer has to implement the rest of management logic (e.g., actuation workflow) in a programming language (e.g., Java) used to program the management part of a self-managing application.

Policy-based management has been proposed as a practical means to improve and facilitate self-management. Policies are sets of rules which govern the system behaviors and reflect the business goals and objectives. Rules dictate 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) [111]. 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 (e.g., replacement) of policies in a policy-based management are rather difficult, if policies are embedded in the management

logic and programmed in its native language. In this case, policy rules and scattered in the management logic and that makes it difficult to modify the policies, especially at runtime. The major advantages of using a special policy language (and a corresponding policy engine) to program policies are the following:

- All related policy rules can be grouped and defined in policy files. This makes it easier to program and to reason about policy-based management.
- Policy languages are at a higher level than the programming languages used to program management logic. This makes it easier for system administrators to understand and modify policies without the need to interact with system developers.
- When updating policies, the new policies can be applied to the system at run time without the need to stop, rebuild or redeploy the application (or parts of it).

In order to facilitate implementation and maintenance of policies, language support, including a policy language and a policy evaluation engine, is needed. Niche provides ability to program policy-based management using a policy language, a corresponding API and a policy engine [62]. The current implementation of Niche includes a generic policy-based framework for policy-based management using SPL (Simplified Policy Language) [112] or XACML [113]. Both languages allow defining policy rules (rules with obligations in XACML, or decision statements in SPL) that dictate the management actions that are to be enforced on managed resources and applications in certain situations (e.g., on failures). SPL is intended for management of distributed systems; whereas XACML was specially designed for access control rather than for management. Nevertheless, XACML allows for obligations (actions to be performed) conveyed with access decisions (permit/denied/not-applicable); and we have adopted obligations for management.

The policy framework includes abstractions (and corresponding API) of policies, policy-managers and policy-manager groups. A policy is a set of if-then rules that dictate what should be done (e.g., publishing an actuation request) when something has happened (e.g., a symptom that require management actions has been detected). A Policy Manager is a management element that is responsible for loading policies, making decisions based on policies and delegating obligations (actuation requests) to Executors. Niche introduces a policy-manager group abstraction that represents a group of policy-based managers sharing the same set of policies. A policy-manager group can be created for performance or robustness. A Policy Watcher monitors the policy repositories for policy changes and request reloading policies. The Policy Engine evaluates policies and returns decisions (obligations).

Policy-based management enables self-management under guidelines defined by humans in the form of management policies that can be easily 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 due to relatively long policy evaluation latencies of the latter. Based on our evaluation results, we recommend using policy-based management for high-level policies that require the flexibility to be able to be rapidly changed and manipulated by administrators at deployment and runtime. Policies can be easily understood by humans, can be changed on the fly, and separated from development code for easier management.

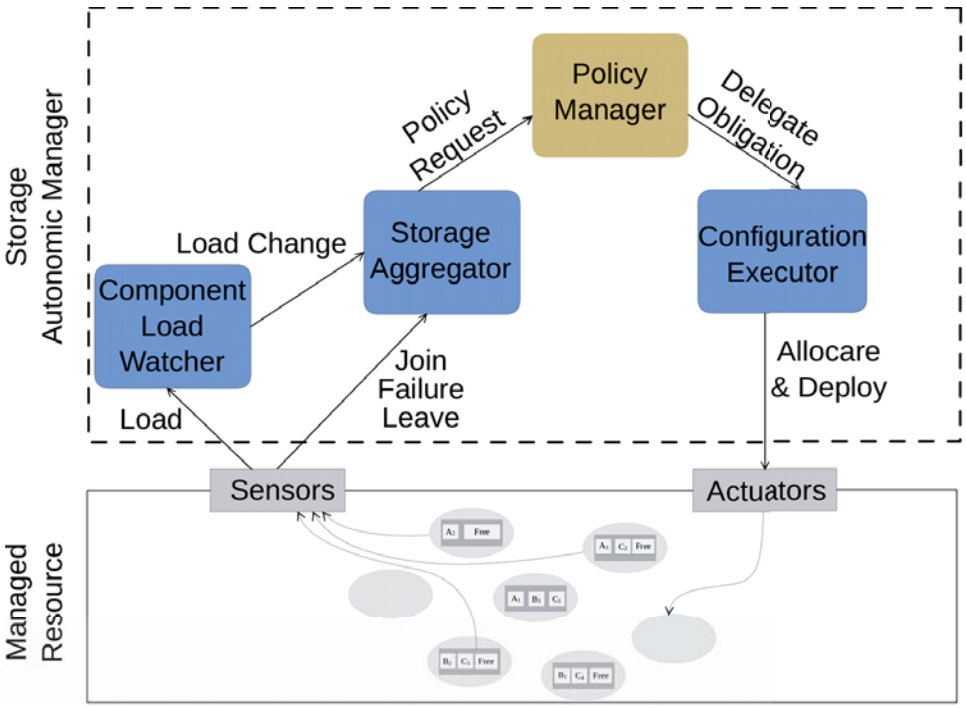

Figure 7.16: YASS self-configuration using policies

Policy based management can be introduced to the management part of an application by adding a policy manager in the control loop. Figure 7.16 depicts an example on how to introduce a policy manager in the Storage Autonomic Manager used in the YASS demonstrator (see Figure 7.12). The policy manager receives monitoring events such as total load in the system. The policy manager then evaluates the policies using the policy engine. An example of a policy used by the Storage Autonomic Manager for releasing extra storage is shown below. The example shows one policy from the policy file written in SPL. When a policy fires (the condition is true) the state of the manager may change and actuation events

may be triggered.

```
...
Policy {
Declaration {
lowloadthreshold = 500;
}
Condition {
storageInfo.totalLoad <= lowloadthreshold
}
Decision {
manager.setTriggeredHighLoad(false) &&
manager.delegateObligation("release storage")
}
}:1;
...
```

7.11 Conclusion

The presented management framework enables the development of distributed component based applications with self-* behaviors 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 robust and efficient application management even in dynamic environments. The framework leverages the self-* properties of the structured overlay network which it is built upon. Our prototype implementation and demonstrators show the feasibility of the framework.

In dynamic environments, such as community Grids or Clouds, self-management presents four challenges. Niche mostly meets these challenges, and presents a programming model and runtime execution service to enable application developers to develop self-managing applications.

The first challenge is that of the efficient and robust resource discovery. This was the most straightforward of the challenges to meet. All resources (containers) are members of the Niche overlay, and resources can be discovered using the overlay.

The second challenge is that of developing a robust and efficient sensing and actuation infrastructure. For efficiency we use a push (i.e., publish/subscribe) rather than a pull mechanism. In Niche all architectural elements (i.e., both functional components and management elements) are potentially mobile. This is necessary in dynamic environments but it means that delivering sensing events and actuation commands is non-trivial. The underlying overlay provides efficient sensing and actuation storing locations in a DHT-like structure, and through replication (as in a peer-to-peer system) sensing and actuation is robust. In terms of messaging all sensing and actuation events are delivered at least once.

The third challenge is to avoid a management bottleneck or single-point-of-failure. We advocate a decentralized approach to management. Management functions (of a single application) should be distributed among several cooperative autonomic managers that coordinate (as loosely-coupled as possible) their activities

to achieve the overall management objectives. While multiple managers are needed for scalability, robustness, and performance, we found that they are also useful for reflecting separation of concerns. We have worked toward a design methodology, and stipulate the design steps to take in developing the management part of a self-managing application including spatial and functional partitioning of management, assignment of management tasks to autonomic managers, and co-ordination of multiple autonomic managers.

The fourth challenge is that of scale, by which we meant that 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 subscribed 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 best (known) efficient communication infrastructure.

7.12 Future Work

Our future work includes issues in the areas of platform improvement, management design, management replication, high-level programming support, coupled control loops, and the relevance of the approach in other domains.

Currently, there are many aspects of the Niche platform that could be improved. This includes better placement of managers, more efficient resource discovery, and improved containers, the limitations of which were mentioned in section on the Niche platform (e.g., enforcing isolation of components).

We believe that in dynamic or large-scale systems that decentralized management is a must. We have taken a few steps in this direction but additional case studies with the focus on the orchestration of multiple autonomic managers for a single application need to be made.

Robustifying management is another concern. Work is ongoing on a Paxos-based replication scheme for management elements. Other complementary approaches will be investigated, as consistent replication schemes are heavyweight.

Currently, the high-level (declarative) language support in Niche is limited. ADLs may be used for initial configuration only. For dynamic reconfiguration the developer needs to use the Niche API directly, which has the disadvantage of being somewhat verbose and error-prone. Workflows could be used to lift the level of abstraction.

There is also 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. Although we did not observe this in our two demonstrators, one might expect problems with coupled control loops in larger and more complex applications. Application programmers should not need to handle coordination of multiple managers (where each manager may be responsible for a specific aspect). Future work might need to address the design of coordination protocols that could be directly used or specialized.

There is another domain, one that we did not target, where scale is also a challenge and decentralization probably necessary. This is the domain of very large (Cloud-scale) applications, involving tens of thousands of machines. Even if the environment is fairly stable the sheer number of involved machines will generate many events, and management might become a bottleneck. It would be of interest to investigate if our approach can, in part of wholly, be useful in that domain.

7.13 Acknowledgments

We thank Konstantin Popov and Joel Höglund (SICS), Noel De Palma (INRIA), Atli Thor Hannesson, Leif Lindbäck, and Lin Bao, for their contribution to development of Niche and self-management demo applications using Niche. This research has been supported in part by the FP6 projects Grid4All (contract IST-2006-034567) and SELFMAN (contract IST-2006-034084) funded by the European Commission. We also thank the anonymous reviewers for their constructive comments.

Chapter 8

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] KTH Royal Institute of Technology, 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.

8.1 Introduction

Autonomic computing [5] 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 [6]. 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] [71, 86, 103].

The remainder of this paper is organized as follows. Section 8.2 describes Niche and relate it to the autonomic computing architecture. Section 8.3 presents the steps for designing distributed self-managing applications. Section 8.4 focuses on orchestrating multiple autonomic managers. In Section 8.5 we apply the proposed methodology to a distributed file storage as a case study. Related work is discussed in Section 11.9 followed by conclusions and future work in Section 8.7.

8.2 The Distributed Component Management System

The autonomic computing reference architecture proposed by IBM [6] 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 [71, 86] 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* [71, 86]. 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 [86]) 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 [33] 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.

8.3 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.

8.4 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 [110]. 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 8.1).

Hierarchical Management

By hierarchical management we mean that some autonomic managers can monitor and control other autonomic managers (Figure 8.2). 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.

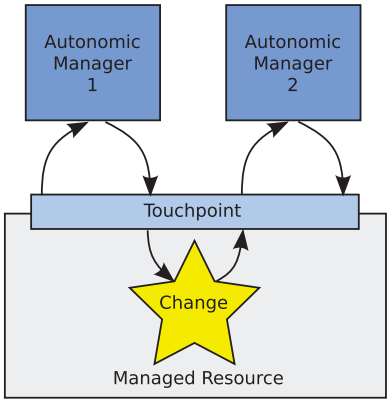

Figure 8.1: The stigmergy effect.

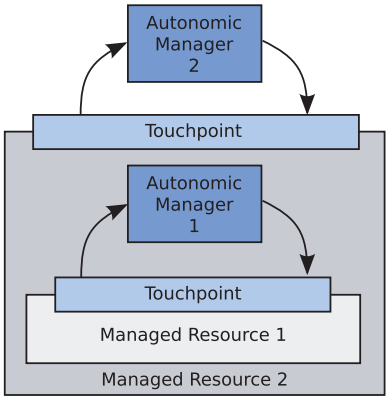

Figure 8.2: Hierarchical management.

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 8.3). Cross autonomic manager bindings can be used to coordinate autonomic managers and avoid undesired behaviors such

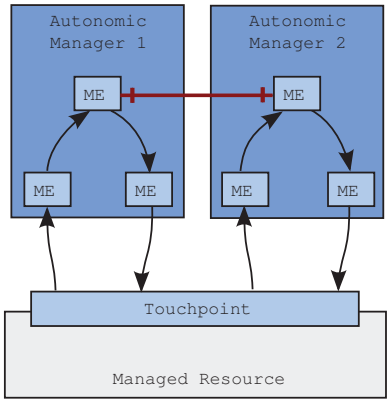

Figure 8.3: Direct interaction.

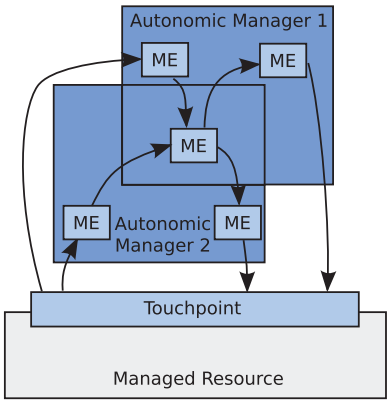

Figure 8.4: Shared Management Elements.

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 8.4). This can be used to share state (knowledge) and to synchronise their actions.

8.5 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) [71], 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 8.5. 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

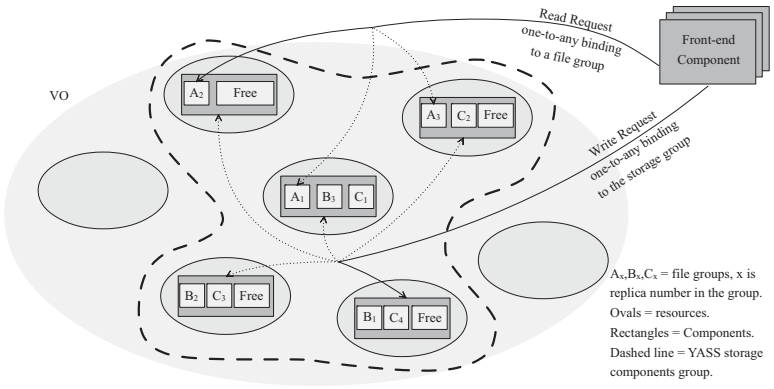

Figure 8.5: YASS Functional Part

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 8.4 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

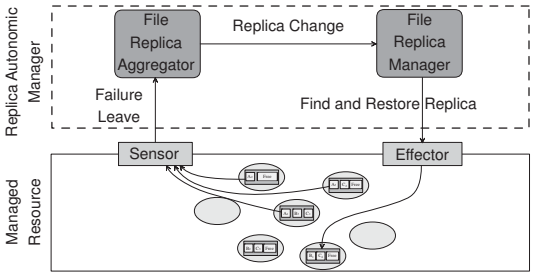

Figure 8.6: Self-healing control loop.

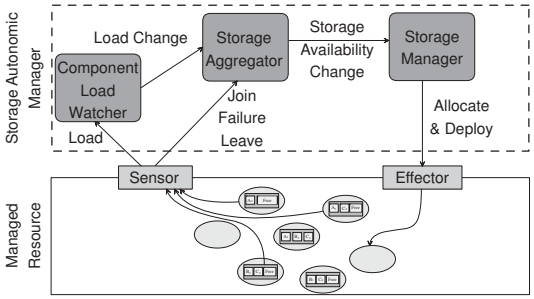

Figure 8.7: Self-configuration control loop.

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 8.6.

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 8.7.

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.

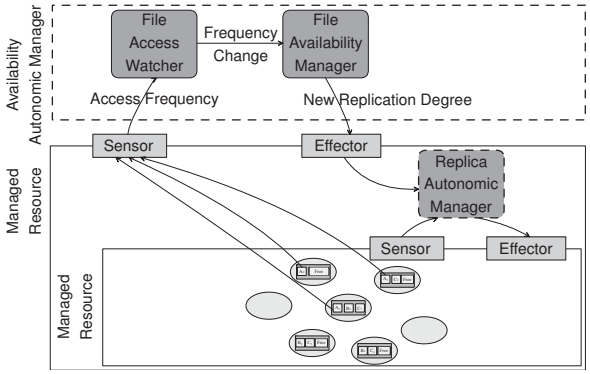

Figure 8.8: Hierarchical management.

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

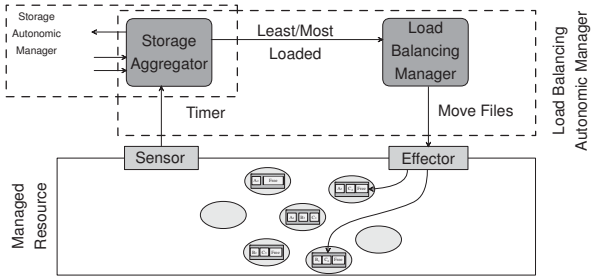

Figure 8.9: Sharing of Management Elements.

regulating the replica autonomic manager. The autonomic manager consists of two management elements. The File-Access-Watcher and File-Availability-Manager shown in Figure 8.8 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 8.9.

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.

8.6 Related Work

The vision of autonomic management as presented in [5] 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 [17] 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 [105]. The authors presented an architecture that supports such compositions.

A reference architecture for autonomic computing is presented in [104]. 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 [99] that uses algorithmic skeletons to encapsulate general control loop features that can later be specialized to fit a specific application.

8.7 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.

Part III

Robust Self-Management and Data Consistency in Large-Scale Distributed Systems

Chapter 9

Achieving Robust Self-Management for Large-Scale Distributed Applications

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

In 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 31–40, Budapest, Hungary, October 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] KTH Royal Institute of Technology, Stockholm, Sweden
{ahmadas, mafayyaz, vladv}@kth.se

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

Abstract

Achieving self-management can be challenging, particularly in dynamic environments with resource churn (joins/leaves/failures). Dealing with the effect of churn on management increases the complexity of the management logic and thus makes its development time consuming and error prone. We propose the abstraction of robust management elements (RMEs), which are able to heal themselves under continuous churn. Using RMEs allows the developer to separate the issue of dealing with the effect of churn on management from the management logic. This facilitates the development of robust management by making the developer focus on managing the application while relying on the platform to provide the robustness of management. RMEs can be implemented as fault-tolerant long-living services.

We present a generic approach and an associated algorithm to achieve fault-tolerant long-living services. Our approach is based on replicating a service using finite state machine replication with a reconfigurable replica set. Our algorithm automates the reconfiguration (migration) of the replica set in order to tolerate continuous churn. The algorithm uses P2P replica placement schemes to place replicas and uses the P2P overlay to monitor them. The replicated state machine is extended to analyze monitoring data in order to decide on when and where to migrate. We describe how to use our approach to achieve robust management elements. We present a simulation-based evaluation of our approach which shows its feasibility.

9.1 Introduction

Autonomic computing [5] is a paradigm to deal with management overhead of complex systems by making them self-managing. Self-management can be achieved through autonomic managers [6] that monitor the system and act accordingly. In our previous work, we have developed a platform called Niche [71,103] that enables one to build self-managing large-scale distributed systems. An autonomic manager in Niche consists of a network of management elements (MEs) each of which can be

responsible for one or more roles of the MAPE loop [6]: Monitor, Analyze, Plan, and Execute. MEs are distributed and interact with each other through events.

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

We propose the Robust Management Element (RME) abstraction that allows simplifying the development of robust autonomic managers that can tolerate resource churn, and thus self-managing large-scale distributed systems. With RMEs, developers of self-managing systems can assume that management elements never fail. An RME 1) is replicated to ensure fault-tolerance; 2) tolerates continuous churn by automatically restoring failed replicas on other nodes; 3) maintains its state consistent among replicas; 4) provides its service with minimal disruption in spite of resource churn (high availability), and 5) is location transparent, i.e., RME clients communicate with it regardless of current location of its replicas. Because we target large-scale distributed environments with no central control, all algorithms of the RME abstraction should be decentralized.

In this paper, we present our approach to implement RMEs which is based on state machine replication [74] combined with *automatic* reconfiguration of replica set. Replication by itself is insufficient to guarantee long-term fault-tolerance under continuous churn, as the number of failed nodes hosting ME replicas, and hence the number of failed replicas, will increase over time, and eventually RME will stop. Therefore, we use *service migration* [75] to enable the reconfiguration of 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 that will use migration to *automatically* reconfigure the set of nodes hosting ME replicas. This will guarantee that RME will tolerate continuous churn.

The major contributions of this paper are:

- The use of *Structured Overlay Networks* (SONs) [1] to monitor the nodes hosting replicas in order to detect changes that may require reconfiguration. SONs are also used to determine replica location using replica placement schemes such as symmetric replication [91].
- The replicated state machine, beside replicating a service, receives monitoring information and uses it to construct a new configuration and to decide when to migrate.
- A decentralized algorithm that automates the reconfiguration of the replica set in order to tolerate continuous resource churn.

The remainder of the paper is organised as follows. Section 9.2 presents necessary background. In Section 9.3 we describe our decentralized algorithm to automate the reconfiguration process. Section 9.4 describes how our approach can

be applied to achieve RMEs in Niche. In Section 9.5 we discuss our experimental results. Related work is discussed in Section 11.9. Finally, Section 10.6 presents conclusions and our future work.

9.2 Background

This section presents the necessary background to our approach and algorithms presented in this paper, namely: The Niche platform, SON and symmetric replication, replicated state machines, and an approach to migrate stateful services.

Niche Platform

Niche [71] is a distributed component management system that implements the autonomic computing architecture [6]. Niche includes a programming model, APIs, and a runtime system. The main objective of Niche is to enable and to achieve self-management of component-based applications deployed in a dynamic distributed environment 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 runtime environment is a network of containers hosting functional and management components. Niche uses a Chord [1]-like structured overlay network (SON) as its communication layer. The SON is self-organising on its own and provides overlay services such as address lookup, Distributed Hash Table (DHT) and a publish/subscribe mechanism for event dissemination. Niche provides 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 and Symmetric Replication

Structured Overlay Networks (SONs) are known for their self-organisation and resilience under churn [114]. We assume the following model of SONs and their APIs. In the model, SON provides the `lookup` 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 identifier space. The SON automatically and dynamically divides the responsibility between nodes such that for every SON identifier there is always a responsible node. The `lookup` operation 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 due to churn.

Furthermore, 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, every node is responsible for items with identifiers in the range between the identifier of its predecessor and its own identifier. Such a SON naturally provides for symmetric replication of items on the SON, where replica identifiers are placed symmetrically around the identifier space circle.

Symmetric Replication [91] is a scheme used to determine replica placement in SONs. Given an item identifier i, a replication degree f, and the size of the identifier space N, symmetric replication is used to calculate the identifiers of the item's replicas. The identifier of the x-th ($1 \leq x \leq f$) replica of the item i is computed as follows:

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

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 is not considered here. A common way to replicate stateful services is to use the replicated state machine approach [74]. Using this 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 and the same initial state. This implies that sources of nondeterminism, such as local clocks, random numbers, and multi-threading, should be avoided.

Replicated state machines can use the Paxos [115] consensus algorithm to ensure that all service replicas execute the same input requests in the same order. Paxos relies on a leader election algorithm, such as [116], to elect one of the replicas as the leader. The leader determines the order of requests by proposing slot numbers for requests. Paxos assigns requests to slots. Several requests can be processed by Paxos concurrently. Replicas execute an assigned request after all requests assigned to previous slots have been executed. Paxos 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.

Migrating Stateful Services

SMART [75] is a technique for changing the set of nodes where a replicated state machine runs, i.e., for migrating the service to a new set of nodes. A fixed set of nodes, where a replicated state machine runs, is called a *configuration*.

SMART is built on the migration technique proposed by Lamport [115] where the configuration is kept as a part of the service state. Migration to a new configuration proceeds by executing a special state-change request that describes the

configuration change. Lamport also proposed to delay the effect of the configuration change (i.e., using the new configuration) for α slots after the state-change request have been executed. This improves performance by allowing to assign concurrently α more requests in the current configuration.

SMART provides a complete treatment of Lamport's idea, but it does not provide a specific algorithm for automatic configuration management. SMART also allows to replace non-failed nodes, enabling configuration management that occasionally removes working nodes due to, e.g., an imperfect failure detector.

The central idea in SMART is the configuration-specific replicas. SMART preforms service migration from a configuration `conf1` to a new configuration `conf2` by creating a new independent set of replicas for `conf2` that run, for a while, in parallel with replicas in `conf1`. The first slot of `conf2` is assigned to be the next slot after the last slot of `conf1`. The replicas in `conf1` are kept long enough to ensure that `conf2` is established and replica state is transferred to new nodes. This simplifies the migration process and helps SMART to overcome limitations of other techniques. Nodes that carry replicas in both `conf1` and `conf2` keep a single copy of replica state per node. The state shared by replicas of different configurations is maintained by a so-called *execution module*. Each configuration runs its own instance of the Paxos algorithm independently without sharing. Thus, from the point of view of the replicated state machine instance, it looks like as if the Paxos algorithm is running on a static configuration.

9.3 Automatic Reconfiguration of Replica Sets

In this section we present our approach and associated algorithms to achieve robust services. Our approach automates the process of selecting a replica set (configuration) and the decision of migrating to a new configuration in order to provide a robust service that can tolerate continuous resource churn and run for long periods of time without the need of human intervention. The approach uses the replicated state machine technique, migration support, and the symmetric replication scheme. Our approach was mainly designed to provide the Robust Management Elements (RMEs) abstraction which is used to achieve robust self-management. An example is our platform Niche [71, 103] where this approach can be applied directly and RMEs can be used to build robust autonomic managers. However, we believe that our approach is generic enough to be used to achieve other robust services.

We assume that the environment that will host the Replicated State Machines (RSMs) consists of a number of nodes forming a Structured Overlay Network (SON) that may host multiple RSMs. Each RSM is identified by a constant ID (denoted RSMID) drawn from the SON identifier space. RSMID permanently identifies an RSM regardless of the number of nodes in the system and node churn that causes reconfiguration of the replica set. Given an RSMID and the replication degree, the symmetric replication scheme is used to calculate the SON ID of each replica. The replica SON ID determines the node responsible for hosting the replica. This

responsibility, unlike the replica ID, is not fixed and may changes over time due to churn. Clients that send requests to the RSM need to know only its RSMID and replication degree. With this information clients can calculate identifiers of individual replicas using the symmetric replication scheme, and locate the nodes currently responsible for the replicas using the lookup operation provided by the SON. Most of the nodes found in this way will indeed host the RSM replicas, but not necessarily all of them because of lookup inconsistency and churn.

Fault-tolerant consensus algorithms like Paxos require a fixed set of known replicas that we call configuration. 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. 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, replication degree, and the configuration consisting of all replica addresses, thus newly created replicas perceive each other as a group and can communicate with each other directly withoud relying on the SON. RSMID is also distributed to future RSM clients.

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 consists of two main parts. The first part runs on all nodes of the overlay and is responsible for monitoring and detecting changes in the replica set caused by churn. This part uses several sources of events and information, including SON node failure notifications, SON notifications about change of responsibility, and requests from clients that indicates the absence of a replica. Changes in the replica set (e.g. failure of a node that hosted a replica) will result in a *configuration change request* that is sent to the corresponding RSM. The second part is a special module, called the *management module*, that is dedicated to receive and process monitoring information (the configuration change requests). The module use this information to construct a configuration and also to decide when it is time to migrate (after a predefined number of changes in the configuration). We discuss the algorithm in greater detail in the following.

Configurations and Replica Placement Schemes

All nodes in the system are part of SON as shown in Fig. 9.1. The RSM that represents the service is assigned an $RSMID$ from the SON identifier space of size N. The set of nodes that will form a configuration are selected using the symmetric

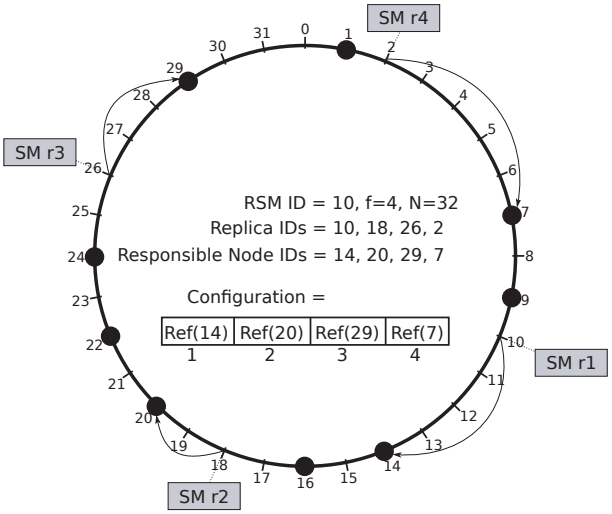

Figure 9.1: Replica Placement Example: Replicas are selected according to the symmetric replication scheme. A Replica is hosted (executed) by the node responsible for its ID (shown by the arrows). A configuration is a fixed set of direct references (IP address and port) to nodes that hosted the replicas 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.

replication scheme [91]. The symmetric replication, given the replication factor f and the $RSMID$, is used to calculate the *Replica IDs* according to equation 10.1. Using the `lookup()` operation, provided by the SON, we can obtain the IDs and direct references (IP address and port) of the responsible nodes. These operations are shown in Algorithm 10.1. The rank of a replica is the parameter x in equation 10.1. A configuration is represented by an array of size f. The array holds direct *references* (IP and port) to the nodes that form the configuration. The array is indexed from 1 to f, and each element contains the reference to the replica with the corresponding rank.

The use of direct references, instead of using lookup operations, as the configuration 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 replication 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

Algorithm 9.1 Helper Procedures

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

7: procedure GetReplicaIDs(RSMID)
8:     for x ← 1, f do ids[x] ← r(RSMID, x)             ▷ See equation 10.1
9:     end for
10:    return ids[ ]
11: end procedure
```

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 9.1, 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 (churn). According to the Paxos, 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 replica may change. For example in Fig.9.1 if node 20 fails then node 22 becomes responsible for identifier 18 and should host `SM_r2`. The algorithms described below automate the migration process by detecting the change and triggering a `ConfChange` request. The `ConfChange` request will be handled by the state machine and will eventually cause it to migrate to a new configuration.

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 architecture of a replica, shown Figure 9.2, uses the shared execution module optimization presented in [75]. This optimization is useful when the same replica participates in multiple configurations. The execution module executes requests. The execution of a request may result in state change, producing output, or both. The execution module should be deterministic. Its outputs and states must depend only on the sequence of input and the initial state. The execution module is also required to support checkpointing which enables state transfer between replicas.

The execution module is divided into two parts: the service specific module and the management module. The service specific module captures the logic of the service and executes all requests except the `ConfChange` request which is handled by

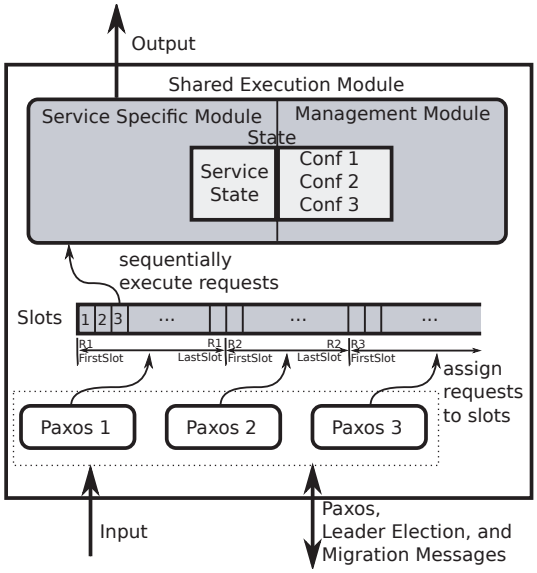

Figure 9.2: State Machine Architecture: Each machine can participate in more than one configuration. A new replica instance is assigned to each configuration. Each configuration is responsible for assigning requests to a none overlapping range of slot. The execution module executes requests sequentially that can change the state and/or produce output.

the management module. The management module maintains a *next configuration* array that it uses to store `ConfChange` requests in the element with the corresponding rank. After a predefined threshold of the number and type (join/leave/failure) of changes, the management module decides that it is time to migrate. It uses the next configuration array to update the current configuration array resulting in a new configuration. After that, the management module passes the new configuration to the migration protocol to actually preform the migration. The reason to split the state into two parts is because the management module is generic and independent of the service and can be reused with different services. This simplifies the development of the service specific module and makes it 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 (determinism and checkpointing).

In a corresponding way, the state of a replica consists of two parts: The first part is internal state of the service specific module which is application specific; The second part consists of the configurations. The remaining parts of the replica, other than the execution module, are responsible to run the replicated state machine algorithms (Paxos and Leader Election) and the migration algorithm (SMART). As

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
```

described in the previous section, each 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, and destroying old configurations after the new configuration is established.

The Paxos algorithm guarantees liveness when a single node acts as a leader, thus it relies on a fault-tolerant leader election algorithm. Our system uses the algorithm described in [116]. This algorithm guarantees progress as long as one of the participating processes can send messages such that every message obtains f timely (i.e., with a pre-defined timeout) responses, where f is a algorithm's constant parameter specifying how many processes are allowed to fail. Note that the f responders may change from one algorithm round to another. This is exactly the same condition on the underlying network that a leader in the Paxos itself relies on for reaching timely consensus. Furthermore, the aforementioned work proposes an extension of the protocol aiming to improve leader stability so that qualified leaders are not arbitrarily demoted which causes significant performance penalty for the Paxos protocol.

Replicated State Machine Maintenance

This section describes 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 by calling `CreateRSM` in Algorithm 9.2. The creating node constructs 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 the message (Algorithm 9.5) starts a state machine (SM) regardless of its responsibility. Note that the initial configuration, due to lookup inconsistency, may contain some incorrect references. This does not cause problems for the RSM because all incorrect references in the configuration will eventually be detected and corrected by our algorithms.

Client Interactions

A client that requires the service provided by the RSM can be on any node in the system. The client needs to know only the $RSMID$ and the replication degree to be able to send requests to the service. Knowing the $RSMID$, the client can determine the current configuration using equation 10.1 and lookup operations (See Algorithm 10.1). In 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`, shown in Algorithm 9.2, that 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. For the client to be able to submit requests, the client's view must overlap, at least at one node, with the actual configuration. Otherwise, the request will fail and the client can retry later. We 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. 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. Clients may also cache the reference to the current leader and use it directly until the leader changes.

Request Execution

The execution of client requests is initiated by receiving a submit request from a client and consists of three steps: checking if the node is responsible for the RSMID in the request, scheduling the request, and executing it. These steps are shown in Algorithm 9.3.

When a node receives a request from a client it will first check, using the RSMID in the request, if it is hosting the replica to which the request is directed to. If this is the case, then the node will submit the request to that replica. The replica will try to schedule the request for execution if the replica believes that it is the leader. Otherwise the replica will forward the request to the leader. The scheduling is

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                                          ▷ Node is hosting the replica
4:         if SM.leader = n then SM.schedule(Req)              ▷ Paxos schedule it
5:         else                                                ▷ forward the request to the leader
6:             sendto SM.leader : SUBMIT(RSMID, rank, Req)
7:         end if
8:     else                                                    ▷ Node is not hosting the replica
9:         if r(RSMID, rank) ∈]n.predecessor, n] then          ▷ I'm responsible
10:            JOINRSM(RSMID, rank)                            ▷ Fix the configuration
11:        else                                                ▷ I'm not responsible
12:            DONOTHING                                       ▷ This is probably due to lookup inconsistency
13:        end if
14:    end if
15: end receipt

16: procedure EXECUTESLOT(req)                                 ▷ The Execution Module
17:     if req.type = ConfChange then                          ▷ The Management Module
18:         nextConf[req.rank] ← req.id
                                                               ▷ Update the candidate for the next configuration
19:         if nextConf.changes = threshold then
20:             newConf ← UPDATE(CurrentConf,NextConf)
21:             SM.migrate(newConf)
                                                               ▷ SMART will set LastSlot and start new configuration
22:         end if
23:     else                                                   ▷ The Service Specific Module handles all other requests
24:         ServiceSpecificModule.Execute(req)
25:     end if
26: end procedure
```

done by assigning the request to a slot that is agreed upon among all replicas in the configuration (using the Paxos algorithm). Meanwhile, the execution module executes scheduled requests sequentially in the order of their slot numbers.

On the other hand, if the node is not hosting a replica with the corresponding RSMID, it will proceed with one of the following two scenarios. In the first scenario, 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 12) because it is not responsible for the target RSM. In the second scenario, it is possible that the client's view is correct but the current configuration contains some incorrect references. In this case, the node that discovers, through the client request, that it was supposed to be hosting a replica will attempt to correct the current configuration by sending a `ConfChange` request replacing the incorrect reference with the reference to itself (Algorithm 9.3 line 10). At execution time, the execution module will direct all requests except the `ConfChange` request to the service specific module for execution. The `ConfChange` will be directed to the management module for processing.

Handling Churn

Algorithm 9.4 contains procedures to maintain the replicated state machine when a node joins, leaves, or fails. When any of these events occur, a new node might

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

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)
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
```

become responsible for hosting a replica. In the case of node join, the new node will send a message to its successor to get information (RSMID and replication degree) about any replicas that the new node should be responsible for. In the case of leave, the leaving node will send a message to its successor containing information about all replicas that it was hosting. In the case of failure, the successor of the failed node needs to discover if the failed node was hosting any replicas. This can be done in a proactive way by checking all intervals (Algorithm 9.4 line 7) that are symmetric to the interval that the failed node was responsible for. One way to achieve this is by using range-cast that can be efficiently implemented on SONs, e.g., using bulk operations [91]. The discovery can also be done lazily using client requests as described in the previous section and Algorithm 9.3 line 10.

In all three cases described above, 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 replica. 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

The proposed approach with corresponding algorithms, described in the previous section, allows meeting the requirements of the Robust Management Element (RME) abstraction specified in Section 9.1. It can be used to implement the RME abstraction in Niche in order to achieve robustness and high availability of autonomic managers in spite of churn. An autonomic manager in Niche is constructed from a set of management elements (MEs). A robust management element can be implemented by wrapping an ordinary ME inside a state machine which is transparantly replicated by the RME support added to the Niche platform. The ME will serve as the service-specific module shown in Figure 9.2. 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. The clients (e.g., sensors) need only to know the RME identifier to be able to use an RME regardless of the location of individual replicas. The RME support in the Niche platform will facilitate development of applications with robust self-management.

9.5 Prototype and Evaluation

In this section, we present a simulation-based performance evaluation of our approach for replicating and maintaining stateful services in various scenarios. In evaluating the performance, we are mainly interested in measuring the *request latency* and the *number of messages* exchanged by our algorithms. The evaluation is divided in three main categories: critical path evaluation, failure recovery evaluation, and evaluation of the overheads associated with the leader election.

To evaluate the performance of our approach and to show the practicality of our algorithms, we built a prototype implementation of Robust Management Elements using the *Kompics* component model [8]. Kompics is a framework for building and evaluating distributed systems in simulation, local execution, and distributed deployments. In order to make network simulation more realistic, we used the King latency dataset, available at [117], that measures the latencies between DNS servers using the King [118] technique. For the underlying SON, we used Chord implementation provided by Kompics. To evaluate the performance of our algorithms in various churn scenarios, we have used the *lifetime-based node failure* model [114, 119] with the shifted Pareto lifetime Distribution.

Methodology

In the simulation scenarios described below, we assumed one stateful service (a Robust Management Element) and several clients (sensors and actuators). A client

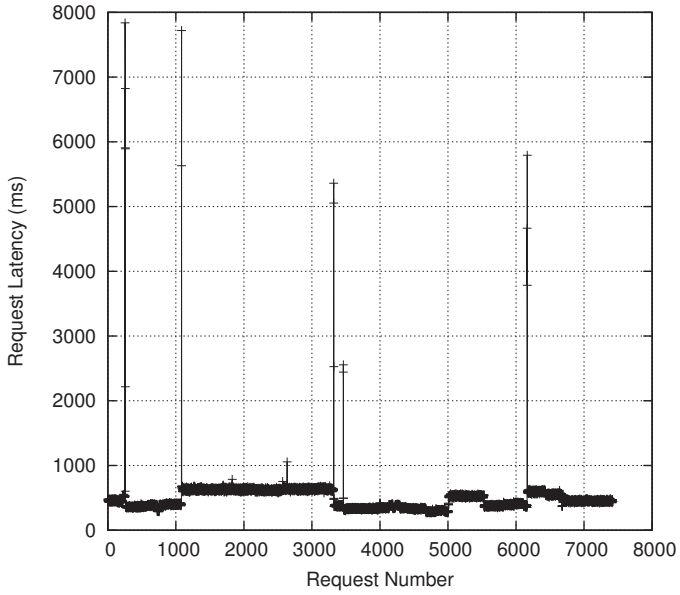

Figure 9.3: Request latency for a single client

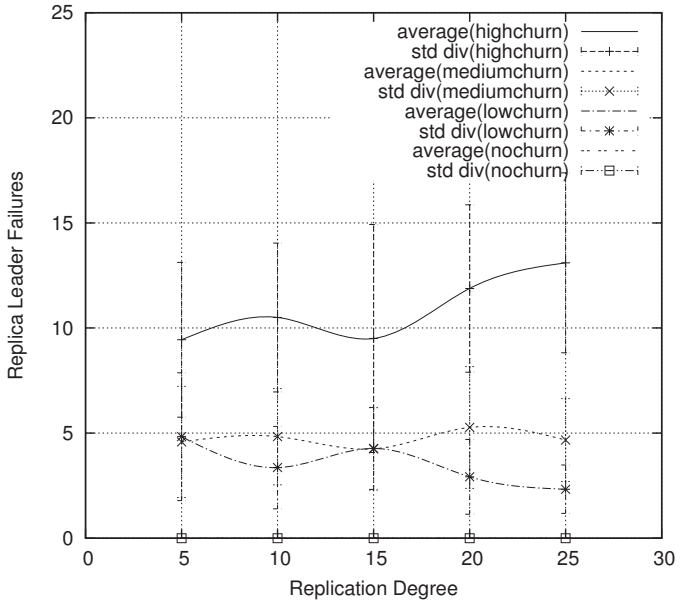

Figure 9.4: Leader failures vs. replication degree

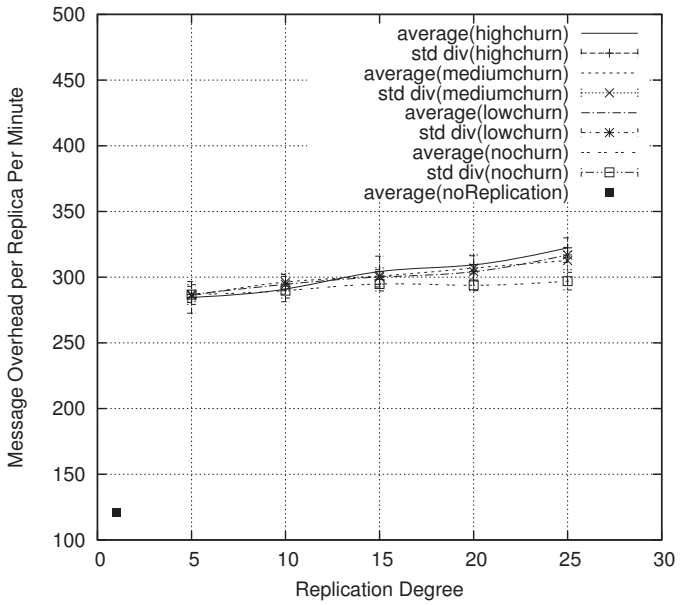

Figure 9.5: Messages/minute vs. replication degree

represents both a sensor and an actuator. The service is replicated using our approach. For simplicity but without losing generality, the service is implemented as an aggregator that accumulates integer values received from clients and replies with the current aggregated value which is the state of the service. A client request (containing a value) represents monitoring information whereas a service response represents an actuation command. Each client repeatedly sends requests to the service. Upon receiving a client request, the service performs all the actions related to the replicated state machine, makes a state transition, and sends the response to the requesting client.

There are various factors in a dynamic distribution environment that can influence the performance of our approach. The input parameters to the simulator include:

- Numeric (architectural) parameters:
  - Overlay size: in the rage of 200 to 600 nodes;
  - Number of services (management elements): 1;
  - Number of clients: 4;
  - Replication degree: varies from 5 to 25;

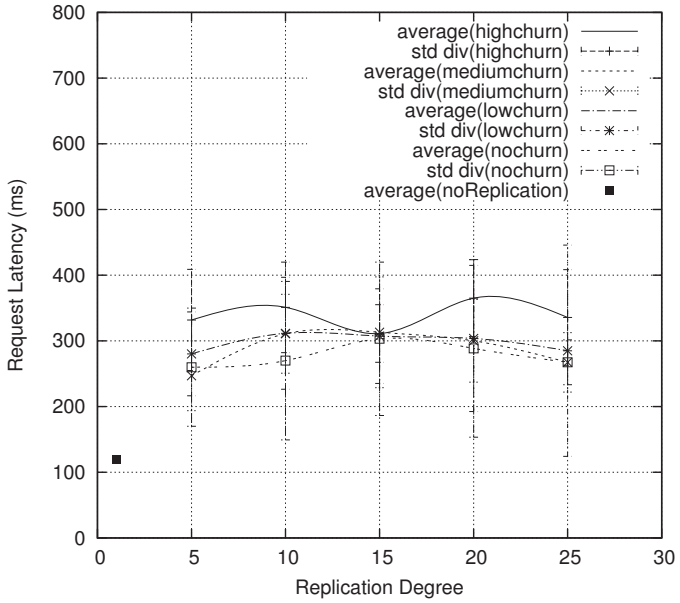

Figure 9.6: Request latency vs. replication degree

  - Failure threshold: this is the number of failures that will cause the RSM to migrate. This can range from 1 to strictly less than half of the number of replicas.

- Timing (operational) parameters

  - Shifted Pareto distribution of client requests with a specified mean time between consecutive requests. In the simulations we used four clients each with mean time between requests of 4 seconds. This gives the total mean time of 1 second between requests from all four clients to the service.
  - Shifted Pareto distribution of node life time with a specified mean to model churn. We modeled three levels of churn: high churn rate (mean life time of 30 minutes), medium churn rate (90 minutes), low churn rate (150 minutes).

In our experiments, we have enabled pipelining of requests (by setting α to 10) as suggested by SMART [75], i.e., up to 10 client requests can be handled concurrently. In all plots, unless otherwise stated, we simulated 8 hours. The plot is the average of 10 independent runs with standard deviation bars.

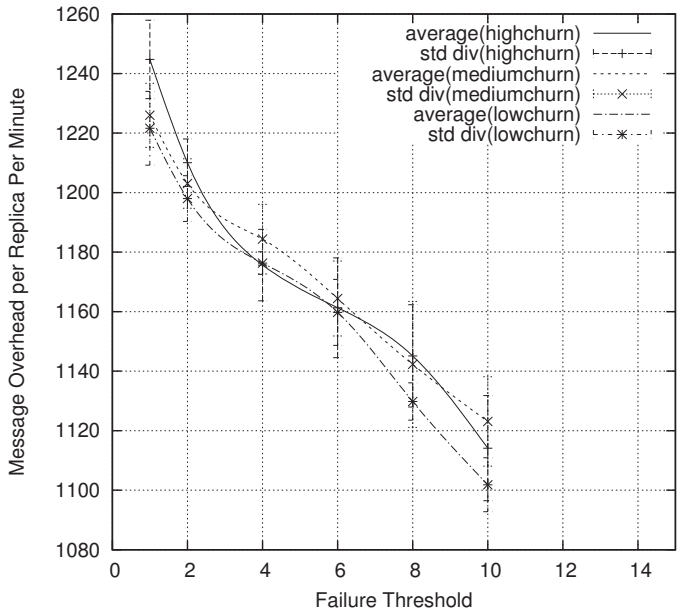

Figure 9.7: Messages per minute vs. failure threshold

In our simulation, we have evaluated how the performance of the proposed algorithms depends on the replication degree (the number of replicas) and the overlay size (the number of physical nodes). The overlay size affects the performance (time and message complexity) of overlay operations [91], namely, lookup and range-cast, used by our algorithms in the following three cases: (1) when creating the initial RSM configuration that is done only once, (2) when looking up the current configuration, and (3) when performing failure recovery. The intensity of configuration lookups depends on the churn rate, and it can be reduced by caching the lookup results (configuration). The intensity of failure recovery depends on the failure rate. Therefore, if the rate of churn (including failures) is lower than the client request rate, the performance of our approach mostly depends on the replication degree rather than on the overlay size. This is because the overlay operations happen relatively rarely. With increasing the overlay size, we expect our approach to scale due to the logarithmic scalability of the overlay operations.

In our experiments, we assumed a reasonable load (the request rate) on the system, and churn rates which are lower than the client request rate. We simulated overlays with hundreds of nodes. Study of systems with larger scales and/or extreme values of load and churn rates is in our future work. We use more than one client in order to get a better average of client-server communication latency. The mean time between requests of 1 sec was selected (via testing experiments) as a reasonable

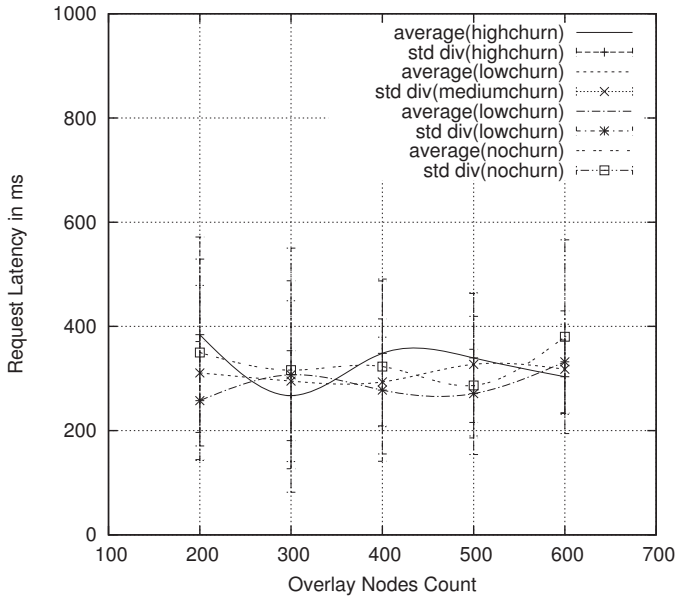

Figure 9.8: Request latency vs. overlay size

workload that does not overload the system.

The baseline in our evaluation is the system with no replication and no churn. We expect that the baseline system has better performance compared to the performance of a system with replication and with/without churn. This is because the replication mechanism as well as migration caused by churn introduce overhead. There are three kinds of overhead in the system: (i) Paxos (which happens upon arrival of requests to RSM), (ii) RSM migration (which happens upon churn), and (iii) the leader election algorithm (which runs continuously). All the overheads cause increase in the number of messages and may cause performance degradation. In our experiments described below we compare performance of different system configurations (overlay size and replication degree) and different churn rates against the baseline system configuration with no replication and no churn (hence no migration).

Simulation Scenarios

Request Critical Path

In this series of experiments we study the effect of various input parameters on the performance (the request latency and the number of messages) of handling client

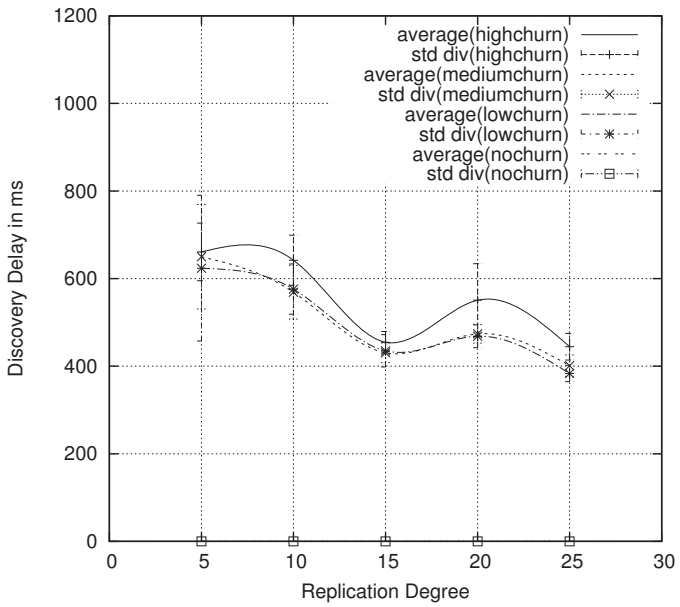

Figure 9.9: Discovery delay vs. replication degree

requests. The request critical path includes sending the request, Paxos, migration, and receiving the reply.

The effect of churn on performance (request latency) is minimal. Figure 9.3 depicts latencies of requests submitted by a single client during 8 hours in a system with a high churn rate. Out of more than 7000 requests, only less than 20 requests were severely affected by churn. The spikes happen when the leader in the Paxos algorithm fails. This is because Paxos can not proceed until a new leader is elected. The average number of leader failures during 8 hours is shown in Figure 9.4. This can help to estimate the number of such spikes that can happen in a system with a specified replication degree and churn rate. The time to detect the failure of the current leader and elect a new leader is maximum 10 seconds according to the leader election parameters used in the simulations. During this time any request that arrives at the RSM will be delayed. If non-leader fails, the RSM is not affected as long as the total number of failed replicas is less than the failure threshold parameter. If the number of failed replicas is at least the value of the failure threshold parameter then a migration will happen. On average a migration takes 300 milliseconds to complete.

Using our approach increases the number of critical path messages needed to handle a request compared to the number of messages in the baseline (Figure 9.5). This is mainly because of the Paxos messages needed to reach consensus for every

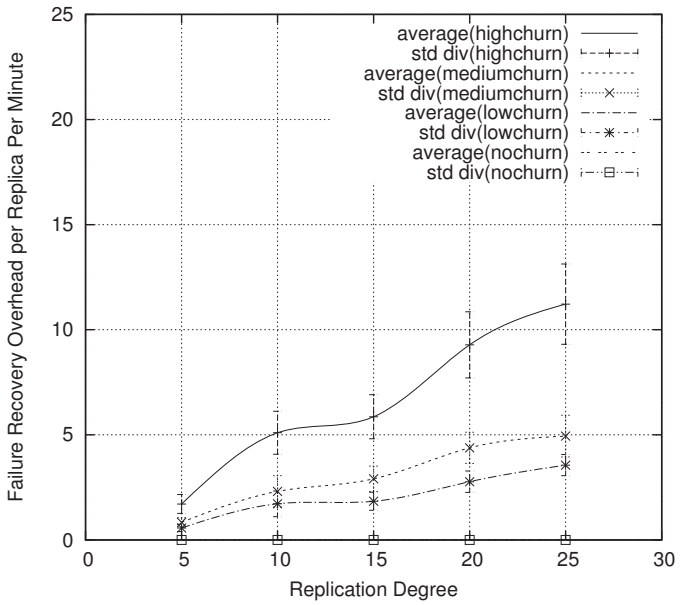

Figure 9.10: Recovery messages vs. replication degree

request. However, increasing the replication degree does not significantly increase the number of messages per replica, as can be seen in Figure 9.5. The slight increase in the number of messages, when the replication degree increases, is due to the increase in the number of migrations. The number of messages is also affected by the churn rate. This is because the higher the churn rate is the higher the migration rate will be. On the other hand, the request latency, as shown in Figure 9.6, is not affected by the replication degree because Paxos requires two phases regardless of the number of replicas. Figure 9.6 also shows the overhead of our approach compared to the baseline.

The average number of critical path messages per request is also affected by the failure threshold parameter as shown in Figure 9.7. A higher failure threshold results in the lower number of messages caused by migration. This is because the higher the failure threshold is, the lower the migration rate will be. For example, with the threshold of 1, the RSM will migrate immediately after one failure; whereas with the threshold of 10, it will wait for 10 replicas to fail before migrating. In this experiment we used 25 replicas. Note that in this case the maximum possible failure threshold is 12. In order to highlight the effect of failure threshold on the message complexity, we increased the request rate from 1 to 4 requests per second.

Our experiments for overlays with hundreds of nodes have shown that the overlay size has minimal or no impact on the request latency, as depicted in Figure 9.8.

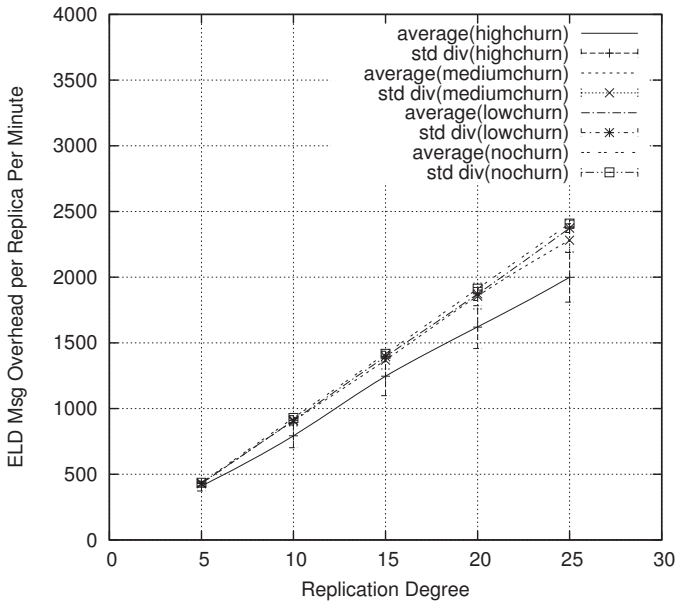

Figure 9.11: Leader election overhead

The request latency deviates when changing the overlay size. One of the possible explanations could be a possible deviations in the average communication latency due to the use of the King latency dataset for network delay. This requires further study and more simulation experiments.

In the above experiments we did not include the lookup performed by clients to discover the configuration. This is because it is not on the critical path, as clients may cache the configuration. For this reason the performance is not affected by the overlay size because all critical path messages are passed over direct links rather that through the overlay.

Failure Recovery

When an overlay node fails, another node (the successor) becomes responsible for any replicas hosted by the failed node. The successor node needs to discover if any replicas were hosted on the failed node. In the simulation experiments we used overlay range-cast to do the discovery. Note that this process is not on the critical path for processing client requests since both can happen in parallel.

Figure 9.9 depicts the discovery delay for various replication degrees. The discovery delay decreases when the number of replicas increases. This is because it is enough to discover only one replica, and it takes shorter time to find a replica in a

system with a higher replication degree, as the probability to find a replica which is close (in terms of link latency and/or overlay hops) to the requesting node is higher. As shown in Figure 9.10, a higher churn rate requires more failure recovery and thus causes higher message overhead.

Other Overheads

Maintaining the SON introduces an overhead in term of messages. We did not count these messages in our evaluation because they vary a lot depending on the type of the overlay and the configuration parameters. One important parameter is the failure detection period that affects the delay between a node failure and the failure notification issued by the SON. This delay is configurable and was not counted when evaluating the fault recovery.

Another source of message overhead is the leader election algorithm. Figure 9.11 shows the average number of leader election messages versus replication degree. The number of messages increases linearly with increasing number of replicas. This overhead is configurable and affects the period between the leader failure and the election of a new leader. In our simulation this period was configured to be maximum 10 seconds. This period is on the critical path and affects the execution of requests as discussed in section 9.5.

9.6 Related Work

For the implementation of the RME abstraction we adopt the replicated state machine approach [74] which is routinely used to build stateful fault-tolerant algorithms and systems. For consensus among replicas on the sequence of input events, our implementation deploys the so-called "Multi-Paxos" [115] version of the Paxos protocol [115, 120] where all proposals from the same leader until its demotion share one single ballot. Other specialized versions of Paxos addressing latency and message complexity (e.g., [121, 122]) can clearly be used instead when appropriate. If input events do not interfere with each other and can be processed in any order yielding the same results and replica state, the Generalized Consensus can be used [123], similarly to relaxing the total order broadcast with generic broadcast [124]. State machine replication can be made tolerant to Byzantine failures [125].

For reconfiguration of the replicate state machine we use the SMART approach [75] which builds on the original idea by Lamport to treat the information about system configuration explicitly as a part of its state [115]. Recently, Lamport also proposed similar extensions to Paxos that enable system reconfiguration by transition through a series of explicit configuration with well-defined policies on proposal numbering [126].

The major alternative way to ensure consistency among replicas is to use a group communication protocol such as Virtual Synchrony [127]. In a Virtual Synchrony system processes (replicas in our case) are organized in groups, and messages sent

by group members arrive to all group members in the same order; the system also notifies all group members about joins and leaves of group members. Between membership changes virtual synchrony systems would use a non-uniform total order broadcast, while membership changes requires fault-tolerant consensus. We could deploy our replica group management protocol with state machine replication using a group communication middleware, but from the practical point of view it appeared to be simpler to implement from scratch a version of reconfigurable Paxos.

9.7 Conclusions and Future Work

We have proposed the concept of Robust Management Elements (RMEs) which are able to heal themselves under continuous churn. Using RMEs allows the developer to separate the issue of robustness of management from the actual management mechanisms. This will simplify the construction of robust autonomic managers. We have presented an approach to achieve RMEs which uses replicated state machines and relies on our proposed algorithms to automate replicated state machine migration in order to tolerate churn. Our approach uses symmetric replication, which is a replica placement scheme used in structured overlay networks to decide on the placement of replicas and uses SON to monitor them. The replicated state machine is used, besides its main purpose of providing the service, to process monitoring information and to decide when to migrate. Although in this paper we discussed the use of our approach to achieve RMEs, we believe that this approach is generic and can be used to replicate other services.

In order to validate and evaluate our approach, we have developed a prototype and conducted various simulation experiments which have shown the validity and feasibility of our approach. Evaluation has shown that the performance (latency and number of messages) of our approach mostly depends on the replication degree rather than on the overlay size.

In our future work, we will evaluate our approach on larger scales and extreme values of load and churn rate. We will optimise the algorithms in order to reduce the amount of messages and improve performance. We intend to implement our approach in the Niche platform to support RMEs in self-managing distributed applications. Finally, we will try to apply our approach to other problems in distributed computing.

Chapter 10

Robust Fault-Tolerant Majority-Based Key-Value Store Supporting Multiple Consistency Levels

Ahmad Al-Shishtawy, Tareq Jamal Khan, and Vladimir Vlassov

In 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS'2011), Tainan, Taiwan, December, 2011.

Robust Fault-Tolerant Majority-Based Key-Value Store Supporting Multiple Consistency Levels

Ahmad Al-Shishtawy[1,2], Tareq Jamal Khan[1], and Vladimir Vlassov[1]

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

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

Abstract

The wide spread of Web 2.0 applications with rapidly growing amounts of user generated data, such as, wikis, social networks, and media sharing, have posed new challenges on the supporting infrastructure, in particular, on storage systems. In order to meet these challenges, Web 2.0 applications have to tradeoff between the high availability and the consistency of their data. Another important issue is the privacy of user generated data that might be caused by organizations that own and control datacenters where user data are stored. We propose a large-scale, robust and fault-tolerant key-value object store that is based on a peer-to-peer network owned and controlled by a community of users. To meet the demands of Web 2.0 applications, the store supports an API consisting of different read and write operations with various data consistency guarantees from which a wide range of web applications would be able to choose the operations according to their data consistency, performance and availability requirements. For evaluation, simulation has been carried out to test the system availability, scalability and fault-tolerance in a dynamic, Internet wide environment.

10.1 Introduction

The emergence of Web 2.0 opened the door to new applications by allowing users to do more than just retrieving of information. Web 2.0 applications facilitate information sharing, and collaboration between users. The wide spread of Web 2.0 applications, such as, wikis, social networks, and media sharing, resulted in a huge amount of user generated data that places great demands and new challenges on storage services. An Internet-scale Web 2.0 application serves a large number of users. This number tends to grow as popularity of the application increases. A system running such application requires a scalable data engine that enables the system to accommodate the growing number of users while maintaining a reasonable performance. Low (acceptable) response time is another important requirement of Web 2.0 applications that needs to be fulfilled despite of uneven load on application servers and geographical distribution of users. Furthermore, the system should

be highly available as most of the user requests must be handled even when the system experiences partial failures or has a large number of concurrent requests. As traditional database solutions could not keep up with the increasing scale, new solutions, which can scale horizontally, were proposed, such as, PNUTS [15] and Dynamo [77].

However there is a trade-off between availability and performance on one hand and data consistency on the other. As proved in the CAP theorem [78], for distributed systems only two properties out of the three – consistency, availability and partition-tolerance – can be guaranteed at any given time. For large scale systems, that are geographically distributed, network partition is unavoidable [79]; therefore only one of the two properties, either data consistency or availability, can be guaranteed in such systems. Many Web 2.0 applications deal with one record at a time, and employ only key-based data access. Complex querying, data management and ACID transactions of relational data model are not required in such systems. Therefore for such applications a NoSQL key-value store would suffice. Also Web 2.0 applications can cope with relaxed consistency as, for example, it is acceptable if one's blog entry is not immediately visible for some of the readers.

Another important aspect associated with Web 2.0 applications is the privacy of user data. Several issues lead to increasing concerns of users, such as, where the data is stored, who owns the storage, and how stored data can be used (e.g., for data mining). Typically a Web 2.0 application provider owns datacenters where user data are stored. User data are governed by a privacy policy. However, the provider may change the policy from time to time, and users are forced to accept this if they want to continue using the application. This resulted in many lawsuits during the past few years and a long debate about how to protect user privacy.

Peer-to-Peer (P2P) networks [1] offers an attractive solution to Web 2.0 storage systems. First, because they are scalable, self-organized, and fault-tolerant; second, because they are typically owned by the community, rather than a single organization, thus allow to solve the issue of privacy.

In this paper, we propose a P2P-based object store with a flexible read/write API allowing the developer of a Web 2.0 application to trade data consistency for availability in order to meet requirements of the application. Our design uses quorum-based voting as a replica control method [81]. Our proposed replication method provides better consistency guarantees than those provided in a classical DHT [16] but yet not as expensive as consistency guarantees of Paxos based replication [82]

Our key-value store is implemented as a Distributed Hash Table (DHT) [16] using Chord algorithms [83]. Our store benefits from the inherent scalability, fault-tolerance and self-management properties of a DHT. However, classical DHTs lack support for strong data consistency required in many applications. Therefore a majority-based quorum technique is employed in our system to provide strong data consistency guarantees. As mentioned in [84], this technique, when used in P2P systems, is probabilistic and may lead to inconsistencies. Nevertheless, as proved in [84], the probability of getting consistent data using this technique is very high

(more than 99%). This guarantee is enough for many Web 2.0 applications that can tolerate relaxed consistency.

To evaluate our approach, we have implemented a prototype of our key-value store and measured its performance by simulating the network using real traces of Internet latencies.

10.2 Related Work

This section presents the necessary background to our approach and algorithms presented in this paper, namely: Peer-to-peer networks, NoSQL data stores, and consistency models.

Peer-to-Peer 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 edges of the Internet. P2P is becoming more popular as edge devices are becoming more powerful in terms of network connectivity, storage, and processing power.

P2P networks are scalable and robust. The fact that each peer plays the role of both client and server allows P2P networks to scale to large number of peers, because adding more peers increases the capacity of the system (such as storage and bandwidth). Another important factor that helps P2P to scale is that peers act as routers. Thus each peer needs only to know about a subset of other peers. The decentralized nature of P2P networks improves their robustness. There is no single point of failure, and P2P networks are designed to tolerate churn (joins, leaves and failures of peers).

Structured P2P network, such as Chord [83], maintains a structure of overlay links. Using this structure allows peers to implement a DHT [16]. Given a key, any peer can efficiently retrieve or store the associated data by routing (in log n hops) a request to the peer responsible for the key. Maintenance of the mapping of keys to peers and of the routing information is distributed among the peers in such a way that churn causes minimal disruption to the lookup service. This maintenance is automatic and does not require human involvement. This feature is known as self-organization.

Symmetric Replication

Symmetric replication scheme [91,128] has been used in our system to replicate data at several nodes. Given a key i, a replication degree f, and the size of the identifier space N, symmetric replication is used to calculate the keys of replicas. The key of the x-th ($1 \leq x \leq f$) replica of the data identified by the key i is computed as follows:

$$r(i, x) = (i + (x - 1)N/f) \bmod N \tag{10.1}$$

The advantage of symmetric replication over successor replication [83] is that each join or leave of a node requires only $O(1)$ messages to be exchanged to restore replicas; whereas the successor replication schema requires O(f) messages.

Consistency Models

In this context, consistency models define rules that help developers to predict the results of read/write operations performed on data objects stored in a distributed data store. Each particular data store supports a consistency model that heavily affects it performance and guaranties. Most relevant consistency models for our discussions are the following.

- *Sequential consistency* offers strong guarantees. All reads and writes appear as if they were executed in a sequential order; hence, every read returns a latest written value.
- *Eventual consistency* guarantees that after sufficiently long period of the absence of new writes, all read operations will return the latest written value.
- *Timeline consistency* is weaker than sequential consistency because it allows a read operation to return a stale value; however, it is stronger than eventual consistency as it guarantees that the returned (stale) value includes all previous updates.

NoSQL Datastores

This section provides some insights into the properties and consistency models of two large-scale data storage systems, Amazon's Dynamo and Yahoo!'s PNUTS.

Dynamo

Amazon's Dynamo [77] is a distributed key-value data store designed to provide a large number of services on the Amazon's service oriented platform with an always-on experience despite of certain failure scenarios such as network partitions and massive server outages. These services also have a stringent latency requirement even under high load.

Dynamo is primarily designed for the applications that require high write availability. It provides *eventual consistency* so that it sacrifices data consistency under certain failure scenarios or high write concurrency in order to achieve higher availability, better operational performance, and scalability. Conflicting versions are tolerated in the system during writes. However, the divergent versions must be detected and eventually reconciled. This is done during reads.

Nodes in the system form a ring structured overlay network and use consistent hashing for data partitioning. The system exposes two operations - get(key) and put(key, object, context) where context represents metadata about the object, e.g.,

version implemented using vector clocks [129]. The context is kept in the store in order to help the system to maintain its consistency guarantee. Dynamo uses successor replication.

PNUTS

Yahoo!'s PNUTS [15] is a geographically distributed and replicated large-scale data storage system currently being used by a number of Yahoo! applications. The system offers relaxed data consistency guarantees in order to decrease latency of operations, improve access concurrency and scalability to be able to cope with ever-increasing load.

Although the eventual consistency model adopted by Dynamo is a good fit for many web services, the model is vulnerable to exposing inconsistent data to applications because, even though it guarantees all updates to reach all replicas eventually, it does not guarantee the same order of updates at different replicas. Therefore, for many web applications, this model is a weak and inadequate option for data consistency.

In contrast to Dynamo, PNUTS offers a stronger consistency model, called timeline consistency, to applications that can live with slightly stale but valid data. It has been observed that unlike traditional database applications many web applications typically tend to manipulate only one data record at a time. PNUTS focuses on maintaining consistency for single records and provides a novel *per-record timeline consistency model*, which guarantees that all replicas of a given record apply updates in the exact same order. Developers can control the level of consistency through the following operations: read-any, read-critical, read-latest, write, and test-and-set-write.

The per-record timeline consistency model is implemented by designating one replica of a record as the master replica to which all updates are directed to be serialized as described below. The mastership is assigned on a per-record basis, therefore different records of the same table can have masters at different regions. The mastership can migrate between regions depending on the intensity of updates within a region.

PNUTS uses Yahoo! Message Broker (YMB), which is a topic-based publish/-subscribe system, to implement its asynchronous replication with timeline consistency. When an update reaches the record's master, the master publishes it to the YMB in the region. Once published, the update is considered committed. YMB guarantees that updates published in a particular YMB cluster will be asynchronously propagated to all subscribers (replicas) and delivered in the publish order. Master replica leverages these reliable publish properties of YMB to implement timeline consistency. When the system detects a change in mastership of a particular record, it also publishes identity of the new master to YMB.

10.3 P2P Majority-Based Object Store

In this paper, we propose a distributed key-value object store supporting multiple consistency levels. Our store exposes an API of read and write operations similar to the API of PNUTS [15]. In our store, data are replicated at various nodes in order to achieve fault-tolerance and improve availability, performance, and scalability. Replication causes different versions of an object to co-exist at the same time. In contrast to PNUTS, which uses masters to provide timeline consistency, our system uses a majority-based mechanism to provide multiple consistency guarantees. Our approach to maintaining per-object consistency using a quorum, rather than a master, eliminates a potential performance bottleneck and a single point of failure exposed by the master replica, and allows using our store in a highly dynamic environment such as P2P networks.

Our system is based on a scalable structured P2P overlay network. We use consistent hashing scheme [130] to partition the key-value store and distribute partitions among peers. Each peer is responsible for a range of keys and stores corresponding key-value pairs. The hashing scheme has good scalability in a sense that when a peer leaves or joins the system, only immediate neighbors of the peer are affected as they need to redistribute their partitions.

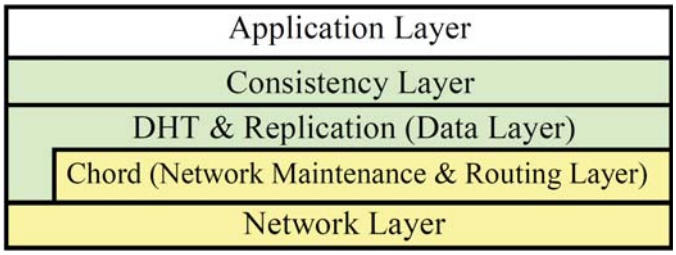

Figure 10.1: Architecture of a peer shown as layers

System Architecture

The architecture of a peer is depicted in Figure 10.1. It consists of the following layers.

- **Application Layer** is formed of applications that invoke read/write operations of the API exposed by the Consistency Layer to access the underlying key-value store in the Data Layer with various consistency levels.

- **Consistency Layer** implements the following read and write operations of the key-value store API. The operations follow timeline consistency and have semantics similar to semantics of operations provided by PNUTS [15].

  - *Read Any (key)* can return an older version of the data even after a successful write. This operation has lowest latency and can be used by applications that prefer fast data access over data consistency.
  - *Read Critical (key, version)* returns data with a version, which is the same or newer than the requested version. Using this operation, an application can enforce read-your-writes consistency, i.e., the application can read the version that reflects previous updates made by the application.
  - *Read Latest (key)* returns the latest version of the data associated with the given key, and is expected to have the highest latency compared to other reads. This operation is useful for applications to which consistency matters more than performance.
  - *Write (key, data)* stores the given data associated with the given key. The operation overwrites existing data, if any, associated with the given key.
  - *Test and Set Write (key, data, version)* writes the data associated with the key only if the given version is the same as the current version in the store, otherwise the update is aborted. This operation can be used to implement transactions.

- **DHT Layer** of a peer hosts a part of the DHT (the key-value object store) for which the peer is responsible according to consistent hashing. It also stores replicas of data of other peers. The read and write operations described above are translated into Get (key, attachment) and Put (key, value, attachment) operations implemented in the DHT Layer. The attachment argument contains metadata for the operations, e.g., a version number. A Get/Put operation looks up the node responsible for the given key using the underlying Chord Layer. After the key is resolved, the responsible node is contacted directly through the Network Layer to perform the requested operation. The DHT Layer also manages data handover and recovery caused by churn.

- **Chord Layer** performs lookup operations requested by the upper DHT Layer efficiently using the Chord lookup algorithm [83] and returns the address the node responsible for the given key. The layer enables nodes to join or leave the ring, also carries out periodic stabilization to keep network pointers correct in the presence of churn.

- **Network Layer** provides simple interfaces for sending/receiving messages to/from peers.

Algorithms

In this section, we describe the read/write operations introduced in Section 10.3 and present corresponding algorithms. Algorithm 10.1 includes common procedures

Algorithm 10.1 Replica Location and Data Access

```
 1: procedure GetNodes(key)                          ▷ Locates nodes responsible for replicas
 2:     for x ← 1, f do                                          ▷ f is the replication degree
 3:         id ← r(key, x)                          ▷ Calculate replica id using equation 10.1
 4:         nodes[x] ← Lookup(ids)                  ▷ Lookup node responsible for replica id
 5:     end for
 6:     return nodes[ ]                  ▷ Returns references to all nodes responsible for replicas
 7: end procedure

 8: receipt of ReadReq(key, rank) from m at n
 9:     (val, ver) ← LocalStore,Read(key, rank)
10:     sendto m : ReadResp(key, val, ver)

11: receipt of VerReq(key, rank) from m at n
12:     (val, ver) ← LocalStore.Read(key, rank)
13:     sendto m : VerResp(key, ver)

14: receipt of WriteReq(key, rank, value, ver) from m at n
15:     LocalStore.Write(key, rank, value, ver)                    ▷ Fails if data is locked
16:     sendto m : WriteAck(key)

17: receipt of LockReq(key, rank) from m at n
18:     ver ← LocalStore.Lock(key, rank)                           ▷ Fails if data is locked
19:     sendto m : LockAck(key, ver)

20: receipt of WriteUnlockReq(key, rank, value, ver) from m at n
21:     if LocalStore.IsLocked then                              ▷ Fails if data is unlocked
22:         LocalStore.Write(key, rank, value, ver)
23:         LocalStore,Unlock(key, rank)
24:         sendto m : WriteUnlockAck(key)
25:     end if
```

used by other algorithms. The algorithms are simplified and some practical issues such as timeouts and error handling are not presented.

Read-Any

The *Read-Any* operation (Algorithm 10.2) sends the read request to all nodes hosting replicas (*replicas* thereafter) and, as soon as it receives the first successful response, it returns the received data. If the data is found locally, Read-Any returns immediately. If no successful response is received within a timeout, the operation fails, e.g., raises an exception. Read-Any also fails if it receives failure responses from all replicas. A failure response is issued when data is not found at the expected node or lookup fails at the Chord layer because of churn. Although the default is to send requests to all replicas, an alternative design choice is to send requests to two random replicas [131] with a view to reducing the number of messages.

Read-Critical

The Read-Critical operation (Algorithm 10.3) sends read requests to all nodes hosting replicas (*replicas* thereafter) and, as soon as it receives data with a version not less than the required version, it returns the data it has received. Read-Critical fails in the case of timeout. It also fails if it receives failure responses or old versions

Algorithm 10.2 ReadAny

```
1: boolean firstResponse ← true
2: procedure ReadAny(key)                                   ▷ Called by application
3:     nodes[] ← GetNodes(key)                              ▷ Nodes hosting replicas
4:     for i ← 1, f do
5:         sendto nodes[i] : ReadReq(key, i)                ▷ Request replica i of key key
6:     end for
7: end procedure

8: receipt of ReadResp(key, val, ver) from m at n
9:     if firstResponse then                                ▷ Note that version ver is not used
10:        firstResponse ← false
11:        return (key, val)                                ▷ Return (key, val) pair to application
12:    else DoNothing( )
13:    end if
```

Algorithm 10.3 ReadCritical

```
1: integer version ← 0, boolean done ← false

2: procedure ReadCritical(key, ver)                         ▷ Called by application
3:     version ← ver
4:     nodes[] ← GetNodes(key)                              ▷ Nodes hosting replicas
5:     for i ← 1, f do
6:         sendto nodes[i] : ReadReq(key, i)
7:     end for
8: end procedure

9: receipt of ReadResp(key, val, ver) from m at n
10:    if not done and ver ≥ version then
11:        done ← true
12:        return (key, val, ver)                           ▷ Return (key, val, ver) to application
13:    else DoNothing( )
14:    end if
```

from all replicas. An alternative design choice is to send requests to a majority of replicas to reduce the number of messages. If all nodes in the majority are alive during the operation, it is guaranteed that the requested version will be found (if it exists) because write operations also use majorities.

Read-Latest

The Read-Latest operation (Algorithm 10.4) sends requests to all replicas and, as soon as it receives successful responses from a majority of replicas, it returns the latest version of the received data. Reading from a majority of replicas R guarantees to return the latest version because R always overlaps with the Write majority W, as $|R| + |W| > n$ (n is the number of replicas). Read-Latest fails in the case of timeout or when it receives failures from a majority of replicas.

Write

First, the Write operation (Algorithm 10.5) sends version requests to all nodes hosting replicas (*replicas* thereafter) and waits for responses from a majority of replicas. Requesting from a majority ensures that the latest version number is

Algorithm 10.4 ReadLatest

```
 1: integer version ← -1, count ← 0
 2: object value ← null, boolean done ← false

 3: procedure ReadLatest(key)                          ▷ Called by application
 4:     nodes[] ← GetNodes(key)                        ▷ Nodes hosting replicas
 5:     for i ← 1, f do
 6:         sendto nodes[i] : ReadReq(key, i)
 7:     end for
 8: end procedure

 9: receipt of ReadResp(key, val, ver) from m at n
10:     if not done then
11:         count ← count + 1
12:         if ver > version then                      ▷ Find the latest version and value
13:             version ← ver
14:             value ← val
15:         end if
16:         if count = f/2 + 1 then                    ▷ Reached majority?
17:             done ← true
18:             return (key, value, version)           ▷ Return to application
19:         end if
20:     else DoNothing( )
21:     end if
```

obtained. Note that for new inserts, the version number 0 is returned as nodes responsible for replicas do not have data. Next, the operation increments the latest version number and sends a write request with the new version of data to all replicas. When a majority of replicas has acknowledged the write requests, the Write operation successfully returns. If two or more distinct nodes try to write data with the same version number, the node with the highest identifier wins. The Write operation can fail for a number of reasons such as timeout, lookup failure, a replica is being locked by a Test-and-Set-Write operation, or collision with another write.

Test-and-Set-Write

The Test-and-Set-Write operation (Algorithm 10.6) starts with sending a lock request to all nodes hosting replicas (*replicas* thereafter). Each replica, if the data is unlocked, locks the data and sends a successful lock response together with the current version number to the requesting node. After receiving lock responses from a majority of replicas, the operation tests if the latest version number obtained from the majority, matches the required version number. If they do not match, the operation aborts (sends unlock requests to all replicas and returns). If they do match, the given data is sent to all replicas to be written with a new version number. Each of the replicas, which have locked data, writes the received new version, unlocks the data, and sends a write acknowledgement to the requesting node. As soon as acknowledgements are received from a majority of replicas, the operation successfully completes. Note that in order to ensure high read availability, read operations are allowed to read the locked data.

Algorithm 10.5 Write

```
1: integer maxVer ← -1, count ← 0, object value ← null
2: boolean done1 ← false, done2 ← false

3: procedure WRITE(key, val)                                  ▷ Called by application
4:     nodes[] ← GETNODES(key)                                ▷ Nodes hosting replicas
5:     value ← val
6:     for i ← 1, f do
7:         sendto nodes[i] : VERREQ(key, i)
8:     end for
9: end procedure

10: receipt of VERRESP(key, ver) from m at n
11:     if not done1 then
12:         count ← count + 1
13:         if ver > maxVer then                               ▷ Find the latest version
14:             maxVer ← ver
15:         end if
16:         if count = f/2 + 1 then                            ▷ Reached majority?
17:             done1 ← true
18:             maxVer ← maxVer + 1
19:             WRITEVER(key, maxVer)
20:         end if
21:     else DONOTHING( )
22:     end if

23: procedure WRITEVER(key,ver)
24:     nodes[] ← GETNODES(key)
25:     for i ← 1, f do
26:         sendto nodes[i] : WRITEREQ(key, i, value, ver)
27:     end for
28: end procedure

29: receipt of WRITEACK(key) from m at n
30:     if not done2 then
31:         count ← count + 1
32:         if count = f/2 + 1 then                            ▷ Majority
33:             done2 ← true
34:             return (key, SUCCESS)                          ▷ Return to application
35:         end if
36:     else DONOTHING( )
37:     end if
```

A Test-and-Set-Write operation can fail for a number of reasons such as timeout, lookup failure, a replica is being locked by another Test-and-Set-Write operation, or collision with another write. When the operation fails, the replicas locked by it have to be unlocked, and this is requested by the node which has initiated the operation. After the operation has completed, a late lock request issued by that operation might arrive at a replica which was not a part of the majority. In this case, according to the algorithm, the replica locks the data and sends a lock response to the requesting node, which, upon receiving the response, requests to unlock the data because the operation has been already completed.

Algorithm 10.6 Test-and-Set-Write

```
1: integer version ← 0, maxVer ← -1, count ← 0
2: object value ← null, boolean done1 ← false, done2 ← false

3: procedure TSWrite(key, val, ver)                          ▷ Called by application
4:     nodes[] ← GetNodes(key)                               ▷ Nodes hosting replicas
5:     value ← val, version ← ver
6:     for i ← 1, f do
7:         sendto nodes[i] : LockReq(key, i)
8:     end for
9: end procedure

10: receipt of LockAck(key, ver) from m at n
11:     if not done1 then
12:         count ← count + 1
13:         if ver > maxVer then                             ▷ Find the latest version
14:             maxVer ← ver
15:         end if
16:         if count = f/2 + 1 then                          ▷ Reached majority?
17:             done1 ← true
18:             if maxVer equals version then                ▷ Test version then set value
19:                 maxVer ← maxVer + 1
20:                 WriteUnlockVer(key, version)
21:             else
22:                 Abort( )                                 ▷ Unlocks the replicas
23:                 return (key, ABORTED)                    ▷ Return to application
24:             end if
25:         end if
26:     else DoNothing( )
27:     end if

28: procedure WriteUnlockVer(key,ver)
29:     nodes[] ← GetNodes(key)
30:     for i ← 1, f do
31:         sendto nodes[i] : WriteUnlockReq(key, i, value, ver)
32:     end for
33: end procedure

34: receipt of WriteUnlockAck(key) from m at n
35:     if not done2 then
36:         count ← count + 1
37:         if count = f/2 + 1 then                          ▷ Majority
38:             done2 ← true
39:             return (key, SUCCESS)                        ▷ Return to application
40:         end if
41:     else DoNothing( )
42:     end if
```

10.4 Discussion

Majority Versus Master

Even though the API of PNUTS is preserved in our key-value store and semantics of read/write operations are kept largely unchanged; our majority-based approach to implement the operations is different from the master-based approach adopted in PNUTS. In PNUTS, all writes to a given record are forwarded to the master replica, which ensures that updates are applied to all replicas in the same order. Serialization of updates through a single master works efficiently for PNUTS due

to the high write locality (the master is placed in the geographical region with the highest intensity of updates). The master-based consistency mechanism can generally hurt the scalability of a system. It leads to uneven load distribution and makes the master replica a potential performance bottleneck. Furthermore, the master represents a single point of failure and may cause a performance penalty by delaying read/write operations until the failed master is restored.

In order to eliminate the aforementioned potential drawbacks of using the master-based consistency mechanism in a distributed key-value store deployed in a dynamic environment, we propose to use a majority-based quorum technique to maintain consistency of replicas when performing read/write operations. Using a majority rather than a single master removes a single point of failure and allows the system to withstand a high level of churn in a dynamic environment that was not considered for the stable and reliable environment (data centers) of PNUTS. Our mechanism is decentralized so that any node in the system receiving a client request can coordinate read/write operations among replicas. This improves load balancing. However, delegating the coordination role to any node in the system while maintaining the required data consistency, incurs additional complexity due to distribution and concurrency.

Performance Model

The number of network hops and the corresponding number of messages for each API operation of both PNUTS and our system, are compared in Table 10.1. Worst case scenarios are considered for both systems and the replication degree is assumed to be 3. For simplicity, we abstract low-level details of communication protocols and count only the number of communication steps (as hops) and a corresponding number of messages. For example, in the case of PNUTS, we assume 2 hops and 2 messages for a roundtrip (request/response) interaction of a requesting node with a master in a local region; Interaction with a master in a remote region adds one more hop and entails one extra message. The asynchronous message propagation to replicas in YMB is not included (as, to our best knowledge, details of YMB were not published at the time this paper was written). For our store, we assume that it takes 2 hops to send a request from a requesting node to all replicas and receive responses from a majority, whereas the number of messages depends on the replication degree. It is worth noting that, taking into account the asynchronous messages propagated by YMB, both systems use about the same amount of messages, which is proportional to the number of replicas.

Other Approaches

Other approaches could have been used to implement our P2P-based object store. For example, a P2P based self-healing replicated state machine [90] could have been used to implement the read/write operations. However, we believe that Paxos based replicated state machine is too powerful for implementing timeline consistency and

Operation	P2P Object Store		PNUTS	
	Hops	messages	Hops	messages
Read-Any	2	4	2	2
Read-Critical	2	5	3	3
Read-Latest	2	5	3	3
Write	4	10	5	5
Test-and-Set-Write	4	10	5	5

Table 10.1: Analytical comparison of the cost of each operation

should only be used if stronger guarantees and/or more powerful operations (e.g., transactions) are required.

10.5 Evaluation

We present a simulation-based performance evaluation of our majority based key-value store in various scenarios. We are mainly interested in measuring the *operation latency* and the *operation success ratio* under different conditions by varying churn rate, request rate, network size, and replication degree. To evaluate the performance and to show the practicality of our algorithms, we built a prototype implementation of our object store using the *Kompics* [8] which is a framework for building and evaluating distributed systems in simulation, local execution, and distributed deployments. In order to make network simulation more realistic, we used the King latency dataset, available at [117], that contains measurements of the latencies between DNS servers obtained using the King [118] technique. To evaluate the performance of our algorithms in various churn scenarios, we have used the *lifetime-based node failure* model [114, 119] with the shifted Pareto lifetime Distribution. Note that the smaller the mean life time, the higher the level of churn. In our experiments, the mean lifetime of 10 and 30 minutes considered to be very low and used to stress test the system in order to find the breaking point.

When evaluating our approach, we did not make a quantitative comparison of our approach with PNUTS [15] mainly because we did not have access to details of PNUTS and YMB algorithms, which, to our best knowledge, were not published at the time this paper was written. Therefore we could not accurately implement PNUTS algorithms in our simulator in order to compare PNUTS with our system. Furthermore, PNUTS was designed to run on geographically distributed but fairly stable datacenters, whereas our system targets an Internet-scale dynamic P2P environment with less reliable peers and high churn. The reason for us to choose such unreliable environment over datacenters is mainly to reduce costs and improve data privacy. We expect that master-based read/write algorithms used in PNUTS will perform better in a stable environment whereas our quorum-based algorithms will win in a highly dynamic environment as discussed in Section 10.4.

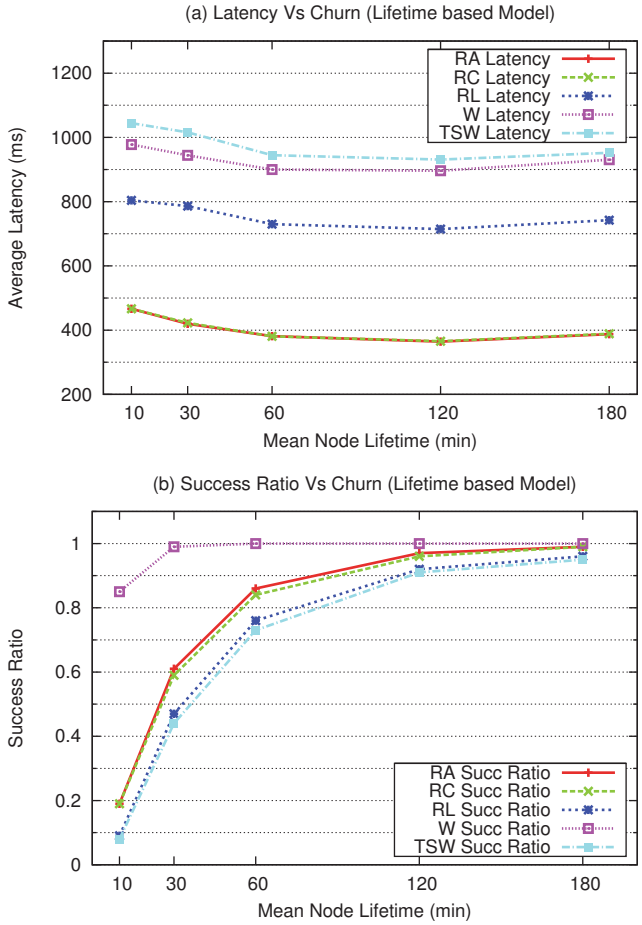

Figure 10.2: The effect of churn on operations (lower mean lifetime = higher level of churn)

Varying Churn Rate

In this experiment, the system is run with various churn rate represented as the mean node lifetime. Figure 10.2(a) shows the impact of churn on latency. The latency for each operation does not vary too much for different levels of churn.

As expected, Read-Any and Read-Critical perform much faster than Read-Latest. Read-Latest shows higher latency because it requires responses from a majority of replicas, whereas other reads do not require a majority in order to complete. Although the Read-Critical latency is expected to be higher than the Read-Any latency (as in the case for PNUTS) because the former requests a spe-

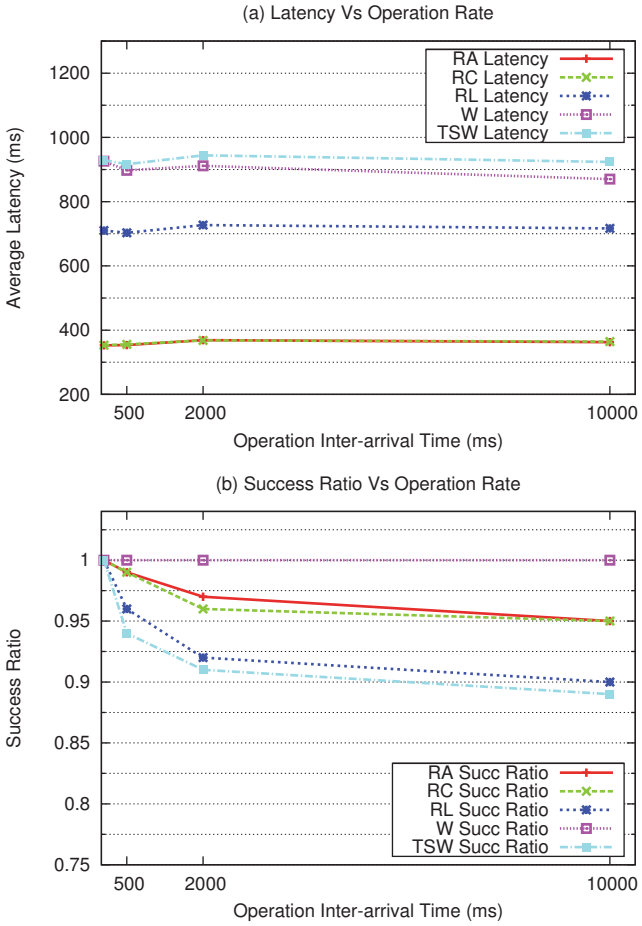

Figure 10.3: The effect of operation rate operations (lower inter-arrival time = higher op rate)

cific version, latencies of both operations in our case are almost identical. This is because a write operation sends updates to all replicas and completes as soon as it receives acknowledgements from a majority of replicas, and thus, the first (fastest) reply to a Read-Critical request with a high probability will come from a node which has the required version. Compare to reads, both Write and Test-and-Set-Write have higher latency because they involve more communication steps than reads. Furthermore, Test-and-Set-Write has slightly higher latency than Write due to possible contention because of locking.

Figure 10.2(b) shows operation success ratio versus churn rate (represented as

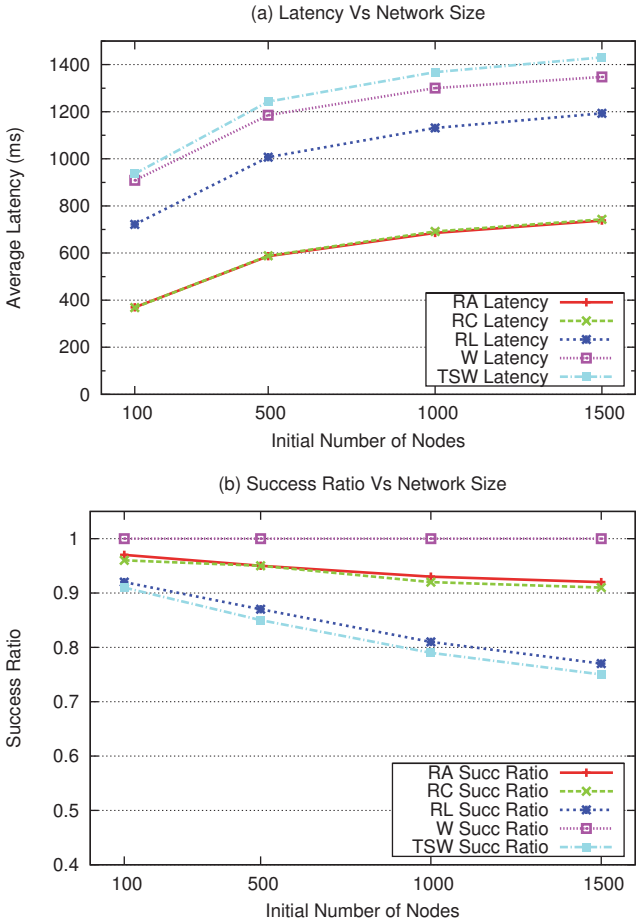

Figure 10.4: The effect of network size on operations

the mean node lifetime). As expected, the success ratio degrades for increasing churn rates. As mentioned in Section 10.3, there are several reasons for failures of operations due to churn. After analyzing the logs, the primary cause of failures has been identified as the unavailability of data at the responsible node. Another major reason is lookup failure.

Varying Operation Rate

Several experiments have been conducted to observe how the system performs under different load scenarios in a dynamic environment. Read/write operations are

generated using an exponential distribution of inter-arrival time of operations. The operation rate (load) is higher for lower mean inter-arrival time. Figure 10.3(a) shows the impact of the operation rate on latency. The latency for each operation does not vary too much for different operation rates. This is due to the simulation that assumes unlimited computing resources (no hardware bottlenecks). Nevertheless, these experiments show that our system can serve the increasing number of requests despite of churn (Figure 10.3(b)), as it quickly heals after failures. One interesting observation is that under the highest load in our experiments, operations have the highest success ratio. This is due to the fact that, when the intensity of writes in a dynamic environment increases, the effect of churn on the success ratio diminishes as the data are updated more often and, as a consequence, the success ration of operations improves.

Varying P2P Network Size

In this experiment, we evaluate the scalability by running the system with various network sizes (the number of nodes). Figure 10.4(a) shows the impact of the network size on latency. For all operations, the latency grows when the network size increases. However the increase is logarithmic because of the logarithmic latency in the Chord Layer.

Varying Replication Degree

In this experiment, the system is run with various replication degrees. Figure 10.5(a) shows the impact of the replication degree on the operation latency. The latency of Read-Any and Read-Critical is highest when there is no replication, but it noticeably decreases as more replicas are added. This is because both operations complete after receiving the first successful response, and having more replicas increase the probability to get the response from a fast (close) node and hence reduce the latency. For Read-Latest, Write, and Test-and-Set-Write operations the latency gets slower with increasing replication degree. This is because in these operations, a requesting node has to wait for a majority of responses, and as the number of replicas grows, the majority increases causing longer waiting time.

10.6 Conclusions and Future Work

We have presented a majority-based key-value store (architecture, algorithms, and evaluation) intended to be deployed in a large-scale dynamic P2P environment. The reason for us to choose such unreliable environment over datacenters is mainly to reduce costs and improve data privacy. Our store provides a number of read/write operations with multiple consistency levels and with semantics similar to PNUTS.

The store uses the majority-based quorum technique to maintain consistency of replicated data. Our majority-based store provides stronger consistency guarantees than guarantees provided in a classical DHT but less expensive than guarantees of

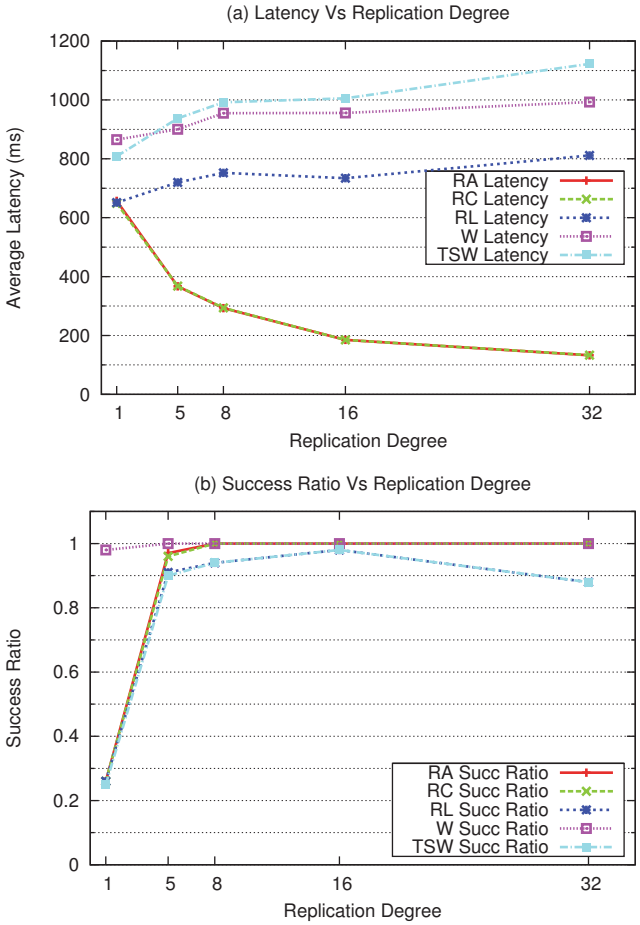

Figure 10.5: The effect of replication degree on operations

Paxos-based replication. Using majority allows avoiding potential drawbacks of a master-based consistency control, namely, a single-point of failure and a potential performance bottleneck. Furthermore, using a majority rather than a single master allows the system to achieve robustness and withstand churn in a dynamic environment. Our mechanism is decentralized and thus allows improving load balancing and scalability.

Evaluation by simulation has shown that the system performs rather well in terms of latency and operation success ratio in the presence of churn.

In our future work, we intend to evaluate our approach on larger scales and extreme values of load and churn rate, and to optimize the algorithms in order to

reduce the amount of messages and improve performance. As the proposed key-value store is to be used in a P2P environment, there is a need to ensure security and protect to personal information by using cryptographic means. This is also to be considered in our future work.

Acknowledgments

This research is supported by the E2E Clouds project funded by the Swedish Foundation for Strategic Research (SSF), and the Complex Service Systems (CS2) focus project, a part of the ICT-The Next Generation (TNG) Strategic Research Area (SRA) initiative at the KTH Royal Institute of Technology.

Part IV

Self-Management for Cloud-Based Storage Systems: Automation of Elasticity

Chapter 11

State-Space Feedback Control for Elastic Distributed Storage in a Cloud Environment

M. Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov

In The Eighth International Conference on Autonomic and Autonomous Systems ICAS 2012, pp. 18–27, St. Maarten, Netherlands Antilles, March, 2012.

Best Paper Award
http://www.iaria.org/conferences2012/awardsICAS12/icas2012_a1.pdf

State-Space Feedback Control for Elastic Distributed Storage in a Cloud Environment

M. Amir Moulavi[1], Ahmad Al-Shishtawy[1,2], and Vladimir Vlassov[1]

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

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

Abstract

Elasticity in Cloud computing is an ability of a system to scale up and down (request and release resources) in response to changes in its environment and workload. Elasticity can be achieved manually or automatically. Efforts are being made to automate elasticity in order to improve system performance under dynamic workloads. In this paper, we report our experience in designing an elasticity controller for a key-value storage service deployed in a Cloud environment. To design our controller, we have adopted a control theoretic approach. Automation of elasticity is achieved by providing a feedback controller that automatically increases and decreases the number of nodes in order to meet service level objectives under high load and to reduce costs under low load. Every step in the building of a controller for elastic storage, including system identification and controller design, is discussed. We have evaluated our approach by using simulation. We have developed a simulation framework EStoreSim in order to simulate an elastic key-value store in a Cloud environment and be able to experiment with different controllers. We have examined the implemented controller against specific service level objectives and evaluated the controller behavior in different scenarios. Our simulation experiments have shown the feasibility of our approach to automate elasticity of storage services using state-space feedback control.

11.1 Introduction

Web-based services frequently experience high workloads during their lifetime. A service can become popular in just an hour, and the occurrence of such high workloads has been observed more and more recently. Cloud computing has brought a great solution to the problem by requesting and releasing VM (Virtual Machine) instances that provide the service on-the-fly. This helps to distribute the loads among more instances. However, the high level load typically does not last for long and keeping resources in the Cloud costs money. This solution has led to Elastic Computing where a system running in the Cloud can scale up and down based on a dynamic property that is changing from time to time.

In 2001, P. Horn from IBM [5] marked the new era of computing as Autonomic Computing. He pointed out that the software complexity would be the next challenge of Information Technology. Growing complexity of IT infrastructures can undermine the benefits IT aims to provide. One traditional approach to manage the complexity is to rely on human intervention. However, considering the expansion rate of software, there would not be enough skilled IT staff to tackle the complexity of its management. Moreover, most of the real-time applications require immediate administrative decision-making and actions. Another drawback of the growing complexity is that it forces us to focus on management issues rather than improving the system itself.

Elastic Computing requires automatic management that can be provided using results achieved in the field of Autonomic Computing. Systems that exploit Autonomic Computing methods to enable automated management are called self-managing systems. In particular, such systems can adjust themselves according to the changes of the environment and workload. One common and proven way to apply automation to computing systems is to use elements of control theory. In this way a complex system, such as a Cloud service, can be automated and can operate without the need of human supervision.

In this paper, we report our experience in designing an elasticity controller for a key-value storage service deployed in a Cloud environment. To design our controller, we have adopted a control theoretic approach. Automation of elasticity is achieved by providing a feedback controller that continuously monitors the system and automatically changes (increases or decreases) the number of nodes in order to meet Service Level Objectives (SLOs) under high load and to reduce costs under low load. We believe that this approach to automate elasticity has a considerable potential for practical use in many Cloud-based services and Web 2.0 applications including services for social networks, data stores, online storage, live streaming services.

Our second contribution presented in this paper is an open-source simulation framework called EStoreSim (Elastic key-value Store Simulator) that allows developers to simulate an elastic key-value store in a Cloud environment and be able to experiment with different controllers.

The rest of the paper is organized as follows. In Section 11.2, we define the problem of automated elasticity and describe the architecture of an elastic Cloud-based key-value store with feedback control. Section 11.3 presents different approaches to system identification. In Section 11.4, we show how we construct a state-space model of our elastic key-value store. We continue in Section 11.5 by presenting the controller designing for our storage. Section 11.6 summarises steps of controller design including system identification. In Section 11.7, we describe the implementation of our simulation framework EStoreSim. Experimental results are presented in Section 11.8 followed by a discussion of related work in Section 11.9. Finally, our conclusion and our future work are given in Section 11.10.

11.2 Problem Definition and System Description

Our research reported here aims at automation of elasticity of a key-value store deployed in a Cloud environment. We want to automate the management of elastic storage instances depending on workload. a Cloud environment allows the system that is running in the Cloud to scale up and down in few minutes in response to load changes. In-time and proper decisions regarding the size of the system in response to the changes in the workload is very critical when it comes to enterprise and scalable applications.

In order to achieve elasticity of a key-value store in the Cloud, we adopt a control theoretic approach to designing a feedback controller that automatically increases and decreases the number of storage instances in response to changes in workload in order to meet SLOs under high load and to reduce costs under low load. The overall architecture of the key-value store with the feedback controller is depicted in Fig. 11.1.

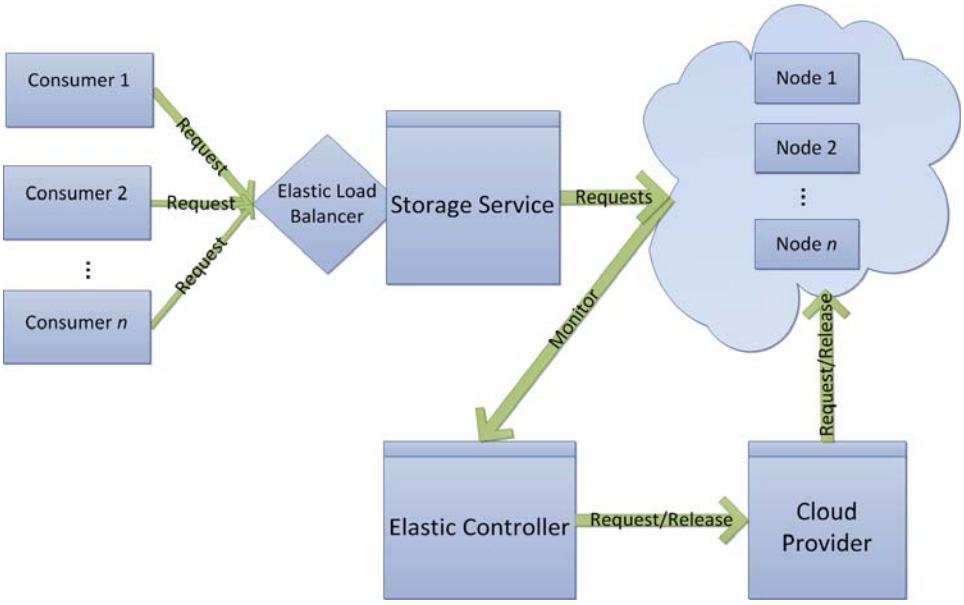

Figure 11.1: Architecture of the Elastic Storage with feedback control of elasticity

End-users request files that are located in the storage Cloud nodes (instances). All the requests arrive at the Elastic Load Balancer (ELB) that sits in front of all storage instances. The Elastic Load Balancer decides to which instance the request should be dispatched. In order to do this, the Elastic Load Balancer tracks the CPU load and the number of requests sent previously to each instance and based on that it determines the next node that can serve the incoming request. In addition to the

performance metrics that it tracks, ELB has the file tables with information about file replica locations since more than one instance can have a replica of the same file in order to satisfy the replication degree.

The Cloud Provider (Fig. 11.1) is an entity that is responsible for launching a new storage instance or terminating the existing one on requests of the Elasticity Controller.

Our system contains the Elasticity Controller, which is responsible for controlling the number of storage instances in the Cloud in order to achieve the desired SLO (e.g., download time). The Controller monitors the performance of the storage instances (and indirectly the quality of service) and issues requests to scale the number of instances up and down in response to changes in the measured quality of service (compared to the desired SLO). These changes are caused by changes in the workload, which is not controllable and is considered to be a disturbance in terms of control theory.

In the following two sections, we provide the relevant background and present steps of the *design of the controller* including *system identification* [132].

11.3 Approaches to System Identification

In this section, we present methods of system identification, which is the most important step in the design of a controller. It deals with how to construct a model to identify a system. System identification allows us to build a mathematical model of a dynamic system based on measured data. The constructed model contains a number of transfer functions, which define how the output depends on past/present inputs and outputs. Based on the transfer functions and desired properties and objectives, a control law is chosen. System identification can be performed using one of the following approaches.

First principle approach is one of the de facto approaches to identification of computer systems [21]. It can be considered as a consequence of the queue relationship. The first principle approach is developed based on knowledge of how a system operates. For example, this approach has been used in some studies and systems like [133–143]. However, there are some shortcomings with this approach that have been stated in [132]. It is very difficult to construct a first principle model for a complex system. Since this approach considers detailed information about the target systems, it requires an ongoing maintenance by experts. Furthermore, this approach does not address model validation.

Empirical approach starts by identifying the input and output parameters like the first principle approach. But rather than using a transfer function, an *autoregressive moving average* (ARMA) model is built and common statistical techniques are employed to estimate the ARMA parameters [132]. This approach is also known as *Black Box* [21]; and it requires minimal knowledge of the system. Most of the systems in our studies have employed a black-box approach rather than a first-principle approach for system identification, e.g., [21, 132, 144–147]. This is mainly because

the relationship between inputs and outputs of the system is complex enough so that the first-principle system identification cannot be done easily. One of the empirical approaches is to build a State-Space Model, which requires more knowledge of the internals of the system. We use the state-space model approach for system identification as described in the next section.

11.4 State-Space Model of the Elastic Key-Value Store

A state-space model provides a scalable approach to model systems with a large number of inputs and outputs [21]. The state-space model allows dealing with higher order target systems without a first-order approximation. Since the studied system executes in a Cloud environment, which is complex and dynamic in a sense of dynamic set of VMs and applications, we have chosen state-space modeling as the system identification approach. Another benefit of using the state-space model is that it can be extended easily. Suppose that after the model is built, we find more parameters to control the system. This can be accommodated by the state-space model without affecting the characteristic equations as shown later in Section 11.6 where we summarize a generic approach for system identification and controller design

The main idea of the state-space approach is to characterize how the system operates in terms of one or more variables. These variables may not be directly measurable. However, they can be sufficient in expressing the dynamics of the system. These variables are called *state variables*.

State Variables and the State-Space Model

In order to define the state variables for our system, first we need to define the inputs and measured outputs since the state variables are related to them. In particular, state variables can be used to obtain the measured output. It is possible for a state variable to be a measured output like it is in our case.

In our case, the system input is the number of nodes (instances) denoted by $\texttt{NN}(k)$ at time k. The measured system outputs (and hence state variables) are the following:

- *average CPU load* $\texttt{CPU}(k)$: the average CPU load of all instances currently running in the Cloud during the time interval $[k-1, k]$;
- *interval total cost* $\texttt{TC}(k)$: the total cost of all instances during the time interval $[k-1, k]$;
- *average response time* $\texttt{RT}(k)$: the average time required to start a download during the time interval $[k-1, k]$.

The value of each state variable at time k is denoted by $x_1(k)$, $x_2(k)$ and $x_3(k)$. The offset value for input is $\bar{u}_1(k) = \texttt{NN}(k) - \widehat{\texttt{NN}}$, where $\widehat{\texttt{NN}}$ is the operating point

for the input. The offset values for outputs are

$$\bar{y}_1(k) = \texttt{CPU}(k) - \widehat{\texttt{CPU}} \tag{11.1}$$
$$\bar{y}_2(k) = \texttt{TC}(k) - \widehat{\texttt{TC}} \tag{11.2}$$
$$\bar{y}_3(k) = \texttt{RT}(k) - \widehat{\texttt{RT}} \tag{11.3}$$

where $\widehat{\texttt{CPU}}$, $\widehat{\texttt{TC}}$ and $\widehat{\texttt{RT}}$ are operating points for corresponding outputs.

The state-space model uses state variables in two ways [21]. First, it uses state variables to describe the dynamics of the system and how $\mathbf{x}(k+1)$ can be obtained from $\mathbf{x}(k)$. Second, it obtains the measured output $\mathbf{y}(k)$ from state $\mathbf{x}(k)$.

State-space dynamics for a system with n states is described as follows

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \tag{11.4}$$
$$\mathbf{y}(k) = \mathbf{C}\mathbf{x}(k) \tag{11.5}$$

where $\mathbf{x}(k)$ is a $n \times 1$ vector of state variables, $\mathbf{A}$ is a $n \times n$ matrix, $\mathbf{B}$ is a $n \times m_I$ matrix, $\mathbf{u}(k)$ is a $m_I \times 1$ vector of inputs, $\mathbf{y}$ is a $m_O \times 1$ vector of outputs and $\mathbf{C}$ is a $m_O \times n$ matrix.

According to equations 11.4 and 11.5, we can describe dynamics of our system as follows:

- Average CPU Load (CPU) is dependant on the number of nodes in the system and previous CPU load, thus it becomes

$$\begin{aligned} x_1(k+1) = \texttt{CPU}(k+1) = \\ a_{11}\texttt{CPU}(k) + \\ b_{11}\texttt{NN}(k) + \\ 0 \times \texttt{TC}(k) + 0 \times \texttt{RT}(k) \end{aligned} \tag{11.6}$$

- Total Cost (TC) is dependant on the number of nodes in the system (the more nodes we have, the more money we should pay) and the previous TC, hence it becomes

$$\begin{aligned} x_2(k+1) = \texttt{TC}(k+1) = \\ a_{21}\texttt{TC}(k) + \\ b_{21}\texttt{NN}(k) + \\ 0 \times \texttt{RT}(k) + 0 \times \texttt{CPU}(k) \end{aligned} \tag{11.7}$$

- Average Response Time (RT) is dependant on the number of nodes in the system and the CPU load, so it is

$$\begin{aligned} x_3(k+1) = \texttt{RT}(k+1) = \\ a_{31}\texttt{CPU}(k) + a_{33}\texttt{RT}(k) + \\ b_{31}\texttt{NN}(k) + \\ 0 \times \texttt{TC}(k) \end{aligned} \tag{11.8}$$

In each equation (11.6, 11.7, 11.8) terms with zero factor include those state variables that do not affect the corresponding state variable definition. Thus their coefficient is zero. This is to ensure that there is no relation between those state variables or the relation is negligible and can be ignored. Their presence in the equations is for the sake of clarity and completeness. In order to prove that there is no relation or that it is negligible one should do a sensitivity analysis to investigate this, but it is out of the scope of this paper.

The output for the system at each time point k is equivalent to the corresponding state variable:

$$\mathbf{y}(k) = I_3\mathbf{x}(k) \tag{11.9}$$

The outputs are the same as the internal state of the systems at each time. That is why the matrix $\mathbf{C}$ is an identity matrix, i.e., a diagonal matrix of 1's. The matrices of coefficients are:

$$\mathbf{A} = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ a_{31} & 0 & a_{33} \end{bmatrix} \tag{11.10}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} \tag{11.11}$$

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{11.12}$$

Parameter Estimation

In Section 11.4, we have derived the State-Space model (Equations 11.6-11.12) that describes the dynamics of an elastic key-value store. There are two matrices $\mathbf{A}$ and $\mathbf{B}$ that contain the unknown coefficients for the equations 11.6-11.8. In order to use the model to design the controller we need to estimate the coefficient matrices $\mathbf{A}$ and $\mathbf{B}$.

Parameter estimation is done using experimental data. In this research, we use data obtained from the simulation framework EStoreSim that we have built, rather than from a real system, because the major focus is on controller design and the simulation framework allows us to experiment with different controllers. We have implemented a simulation framework EStoreSim (described in Section 11.7) of a Cloud system. Using the framework we can obtain experimental data for system identification.

To get the data, we have designed and run an experiment, in which we feed the system with an input signal and observe the output and internal state variable periodically. We change the input (which is the number of nodes in the system) by

increasing it from a small number of nodes a to a large number of nodes b and then back from b to a in a fixed period of time, and measure outputs (CPU load, cost, and response time). In this way, we ensure the complete coverage of the output signals in their operating regions by the input signal (the number of nodes). Load should be generated according to an arbitrary periodic function to issue a number of downloads per seconds. The period of the function should be chosen such that at least one period is observed during the time of changing the input between $[a, b]$.

For example, using the modeler component of our framework EStoreSim (Section 11.7), we scale up the number of nodes from 2 to 10 and then scale down from 10 to 2. Every 225 seconds a new node is either added or removed (depending on whether we scale up or down); sampling of training data (measuring outputs) is performed every 10 seconds.

When identifying the system, the workload is modeled as a stream of requests issued by the request generator component where the time interval between two consecutive requests forms a triangle signal in the range $[1, 10]$ seconds as follows: the first request is issued after 10 seconds, the second after 9 seconds, etc. The requests are received by the load balancer component in the Cloud provider component. After each scaling up/down the system will experience 2 triangle loads of requests between 1 to 10 seconds. The time needed to experience 2 triangles is $4\sum_{i=1}^{10} i$, which is 220 seconds. That is why we have selected 225 seconds as the action time.

Once training data are collected, they can be used to compute the matrices $\mathbf{A}$ and $\mathbf{B}$ using the *multiple linear regression* method. We use the `regress(y,X)` function of Matlab to calculate matrices:

$$\mathbf{A} = \begin{bmatrix} 0.9 & 0 & 0 \\ 0 & 0.724 & 0 \\ 5.927 & 0 & 0.295 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 2.3003 \\ 0.0147 \\ 77.8759 \end{bmatrix}$$

11.5 Controller Design

In this section, we describe how the feedback controller for the elastic storage deployed in a Cloud environment is designed. The controller design starts by choosing an appropriate controller architecture according to system properties. There are three common architectures for state-space feedback control, namely, *Static State Feedback*, *Precompensated Static Control* and *Dynamic State Feedback*. A good comparison between these architectures can be found in [21]. A close investigation in this comparison reveals that dynamic state feedback control is more suitable for

a Cloud system since it has disturbance rejection that the other two architectures lack. Disturbance (in terms of control theory) is observed in a Cloud in the form of changes in the set of virtual machines and workload of Cloud applications. Thus we choose dynamic state feedback control as our controller architecture for autonomic management of elasticity.

Dynamic State Feedback

Dynamic state feedback can be viewed as a State-Space analogous to PI (Proportional Integral) control that has good disturbance rejection properties. It both tracks the reference input and rejects disturbances. We need to augment the state vector with the control error $e(k) = r - y(k)$ where r is the reference input. We use integrated control error, which describes the accumulated control error. The integrated control error is denoted by $x_I(k)$ and computed as

$$x_I(k+1) = x_I(k) + e(k)$$

The augmented state vector is $\begin{bmatrix} \mathtt{x}(k) \\ x_I(k) \end{bmatrix}$. The control law is

$$u(k) = -\begin{bmatrix} \mathtt{K}_p & K_I \end{bmatrix} \begin{bmatrix} \mathtt{x}(k) \\ x_I(k) \end{bmatrix} \tag{11.13}$$

where $\mathtt{K}_p$ is the feedback gain for $\mathtt{x}(k)$ and K_I is the gain associated with $x_I(k)$.

LQR Controller Design

An approach to controller design is to focus on the trade-off between control effort and control errors. The *control error* is determined by the squared values of state variables, which are normally the difference from their operating points. The *control effort* is quantified by the square of $u(k)$, which is the offset of the control input from the operating point. By minimizing control errors we improve accuracy and reduce both settling times and overshoot and by minimizing control effort, system sensitivity to noise is reduced.

Least Quadratic Regulation (LQR) design problem is parametrized in terms of relative cost of control effort (defined by matrix $\mathbf{R}$) and control errors (defined by matrix $\mathbf{Q}$). The quadratic cost function to minimize is the following [21]:

$$J = \frac{1}{2} \sum_{k=0}^{\infty} \left[\mathbf{x}^\top(k) \mathbf{Q} \mathbf{x}(k) + \mathbf{u}^\top(k) \mathbf{R} \mathbf{u}(k) \right] \tag{11.14}$$

where $\mathbf{Q}$ must be positive semidefinite (eigenvalues of $\mathbf{Q}$ must be nonnegative) and $\mathbf{R}$ must be positive definite (eigenvalues of $\mathbf{R}$ must be positive) in order for J to be nonnegative.

After selecting the weighting matrices $\mathbf{Q}$ and $\mathbf{R}$, the controller gains $\mathbf{K}$ can be computed using the Matlab `dlqr` function that takes as parameters the matrices

A, **B**, **Q**, and **R**. The performance of the system with the designed controller can be evaluated by simulation. If the performance is not appropriate, the designer can select new **Q** and **R** and recompute the vector gain **K**.

In our example, the matrices **Q** and **R** are defined as follows:

$$\mathbf{Q} = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} 1 \end{bmatrix}$$

We have given 100 to the element that corresponds to CPU Load to emphasize that this state variable is more important compared to the others. One can give a high weight to total cost TC to trade off cost for performance. Using the Matlab `dlqr` function we compute the controller gains `K = dlqr(A, B, Q, R)`. For example, using the results of system identification in the example in Section 11.4, the controller gains (corresponding to the measured outputs of the elastic storage, CPU, TC, and RT) are:

$$\mathbf{K} = \begin{bmatrix} 0.134 & 1.470162e-06 & 0.00318 \end{bmatrix}$$

Fuzzy Controller

The main purpose in using an additional fuzzy controller is to optimize the control input produced by the Dynamic State Feedback Controller that we have designed in Section 11.5. A fuzzy controller uses heuristic rules that define when and what actions the controller should take. The output of the Dynamic State Feedback Controller (control input) is redirected together with measured outputs to the fuzzy controller, which decides if the control input should affect the system or not. The overall architecture for controllers is demonstrated in Fig. 11.2.

There is one important case that the dynamic state feedback controller cannot act accordingly. Let us assume that there are some instances with high CPU load. Since the average is high, the controller will issue a control request to add a number of new instances. The new instances will be launched and will start to serve requests. But at the beginning of their life cycle they have low CPU load, thus the average CPU load that is reported back to the controller can be low. The controller then assumes that the CPU load has dropped, and it requests to remove some nodes.

A closer look at the CPU loads reveals that we can not judge the system state by only the average CPU load. Hence the fuzzy controller also takes into account the standard deviation of CPU load. In this way, if the feedback controller gives an order to reduce the number of nodes when there is high standard deviation for CPU loads, the fuzzy controller will not allow this control input to affect the system, thus reducing the risk of unexpected results and confusions for the controller that may

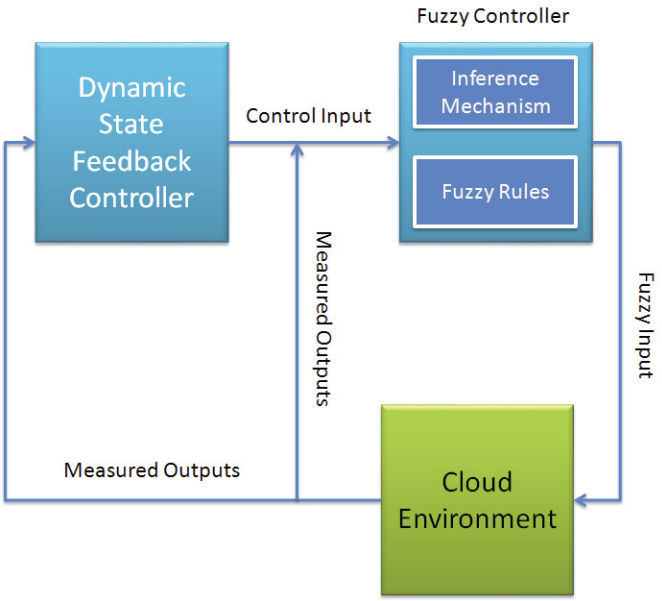

Figure 11.2: Controllers Architecture

cause oscillations. This will lead to a more stable environment without so many unnecessary fluctuations.

Stability Analysis of Controller

A system is called stable if all bounded inputs produce bounded outputs. The BIBO theorem [21] states that for a system to be stable, its poles must lie within the unit circle (have magnitude less than 1). In order to calculate the poles for the controller we need to get the eigenvalues of matrix `A` that are 0.2951, 0.9 and 0.7247. As it is obvious from the values, all of the poles reside within the unit circle thus the controller is stable.

11.6 Summary of Steps of Controller Design

This section summarizes the steps needed to design a controller for an elastic storage in a Cloud environment. The steps described below are general enough to be used to design a controller for an elastic service in a Cloud. The design process consists of two stages: system identification and controller design. The design steps are as follows: the system identification stage includes steps 1-9; and the remaining steps (10-12) belong to the stage of the controller design.

1. Study system behavior in order to identify the inputs and outputs of the system.

2. Place inputs and outputs in $\mathbf{u}(k)$ and $\mathbf{y}(k)$ vectors respectively.

3. Select n system outputs that you want to control and place them in state variable vector $\mathbf{x}$. The outputs should be related to SLOs and performance metrics.

4. Select m system inputs that you will use to control. These system inputs will be the outputs of your controller. The system outputs should depend on the system inputs These inputs should have the highest impact in your system. In some systems there might be only one input that has high impact whereas in other systems there might be several inputs that together have high impact. To assess the impact you might need to do sensitivity analysis.

5. Define state variables that describe the dynamics of the system. State variables can be equivalent to system outputs selected in step 3. Each state variable can depend on one or more other state variables and system inputs. Find the relation between the next value for a state variable to other state variables and system inputs and construct the characteristic equations as follows (see also Equation 11.4).

$$\begin{aligned} x_1(k+1) &= a_{11}x_1(k) + \ldots + a_{1n}x_n(k) \\ &\quad + b_{11}u_1(k) + \ldots + b_{1m}u_m(k) \\ x_2(k+1) &= a_{21}x_1(k) + \ldots + a_{2n}x_n(k) \\ &\quad + b_{21}u_1(k) + \ldots + b_{2m}u_m(k) \\ &\vdots \\ x_n(k+1) &= a_{n1}x_1(k) + \ldots + a_{nn}x_n(k) \\ &\quad + b_{n1}u_1(k) + \ldots + b_{nm}u_m(k) \end{aligned}$$

6. Place coefficients from the previous equations into two matrices $\mathbf{A}$ and $\mathbf{B}$. Some of the coefficients can be zero:

$$\mathbf{A}_{n\times n} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{bmatrix}$$

$$\mathbf{B}_{n\times m} = \begin{bmatrix} b_{11} & \ldots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \ldots & b_{nm} \end{bmatrix}$$

7. In order to simplify the design of controller, one can assume that outputs of the systems are equal to state variables, thus matrix **C** is an identity matrix:

$$\mathbf{C}_{n\times n} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

8. Design an experiment, in which the system is fed with its inputs. Inputs in the experiment should be changed in such a way that they cover their ranges at least one time. A range for an input is the interval that the values of the input will most likely belong to when the system operates. The selection of ranges can be based on industry's best practices. All inputs and outputs should be measured periodically with a fixed time interval T. Store collected data for each equation in a separate file called x_i.

9. In Matlab, for each file x_i, load the file and extract each column of data in a separate matrix. Use the function `regress` to calculate the coefficients. Repeat this for every file. At the end you will have all the coefficients that are required for matrices **A** and **B**.

10. Choose a controller architecture for feedback control: such as dynamic state feedback control, which is, in our opinion, more appropriate for a Cloud-based elastic service (as discussed in Section 11.5).

11. Construct matrices **Q** and **R** as described in Section 11.5. Remember to put high weights in matrix **Q** for those state variables that are of more importance.

12. Use the Matlab function `dlqr` with matrices **A**, **B**, **Q** and **R** as parameters to calculate the vector **K** of controller gains. Perform stability analysis of the controller checking whether its poles reside within the unit circle (Section 11.5).

11.7 EStoreSim: Elastic Key-Value Store Simulator

We have implemented a simulation framework, which we call EStoreSim, that allows developers to simulate an elastic key-value store in a Cloud environment and to experiment with different controllers. We have selected Kompics as the implementation tool. Kompics [148] is a message-passing component model for building distributed systems using event-driven programming. Kompics components are reactive state machines that execute concurrently and communicate by passing data-carrying typed events through typed bidirectional ports connected by channels. For further information please refer to the Kompics programming manual and the tutorial on its web site [148].

Implementation is done in Java and Scala languages [149] and the source is publicly available at [150]. The overall architecture of EStoreSim is shown in Fig. 11.3. The simulator includes the following components.

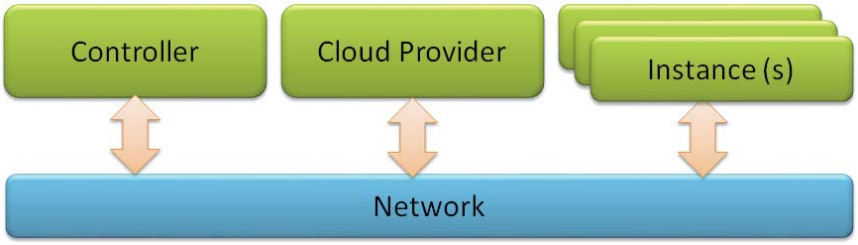

Figure 11.3: Overall Architecture of the EStoreSim Simulation Framework

Cloud Instance Component represents an entire storage instance within a Cloud. The component architecture for instance is shown in Fig. 11.4.

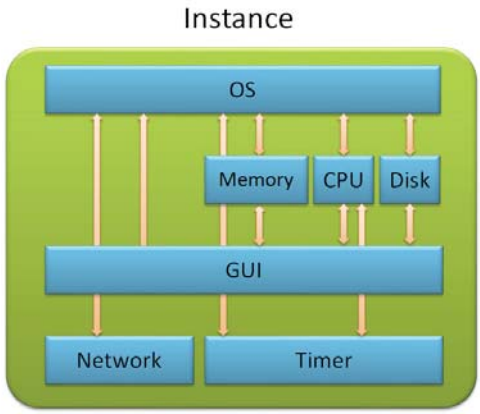

Figure 11.4: Cloud Instance Component Architecture

Cloud Provider Component represents an important unit in the implementation. It is the heart of a simulated Cloud computing infrastructure and provides vital services to manage and administer the nodes (VM instances) within the Cloud. The Cloud provider component architecture is shown in Fig. 11.5.

Elasticity Controller represents the controller that can connect to the Cloud provider and retrieve information about the current nodes in the system. The main responsibility of the controller component is to manage the number of nodes currently running in the Cloud. In other words, it attempts to optimize the cost and satisfy some SLO parameters. The overall component architecture is shown in Fig. 11.6.

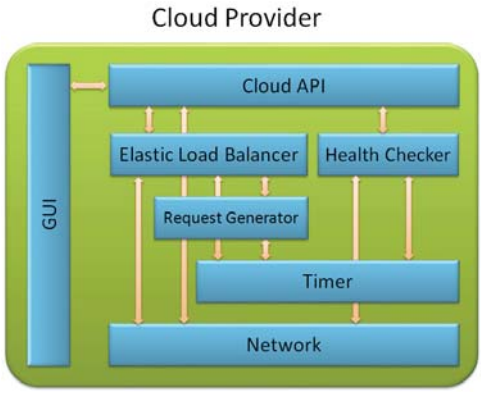

Figure 11.5: Cloud Provider Component Architecture

For further information on EStoreSim please refer to [150].

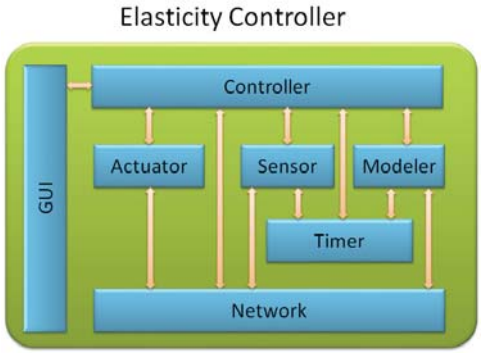

Figure 11.6: Elasticity Controller Component Architecture

11.8 Experiments

We have conducted a number of simulation experiments using EStoreSim in order to evaluate how the use of an elasticity controller in a Cloud-based key-value store improves the operation of the store by reducing the cost of Cloud resources and the number of SLO violations. The baseline in our experiments is a non-elastic key-value store, i.e., a key-value store without the elasticity controller.

For evaluation experiments, we have implemented a dynamic state feedback controller with the parameters (controller gains) calculated according to the controller design steps (Section 11.5). The controller is given reference values of the system

outputs that correspond to SLO requirements. Values of system outputs (average CPU load `CPU`, Total Cost `TC`, and average Response Time `RT`) are fed back into the controller periodically. When the controller gets the values, it calculates and places the next value of the number of nodes `NN` on its output. The controller output is a real number that should be rounded to a natural integer. We round it down in order to save the total cost the Cloud generates. One can assume two boundaries, which are defined as follows:

- L (Lower boundary): the minimum number of instances that should exist in the Cloud at all times;
- U (Upper boundary): the maximum number of instances that is allowed to exist in the Cloud at all times.

Hence if the value of controller output is smaller than L or greater than U, then the value should be discarded. If the calculated output of the controller is Θ, the number of nodes is defined as follows:

$$\texttt{NN} = \begin{cases} L & \text{if} \quad \Theta \leqslant L \\ \Theta & \text{if} \quad L < \Theta < U \\ U & \text{if} \quad U \leqslant \Theta \end{cases} \tag{11.15}$$

If the number of current nodes in the system is $\texttt{NN}'$ and the control input (output of the controller) is `NN`, then the next control action is determined as follows:

$$\text{Next action} = \begin{cases} \text{scale up with } \texttt{NN} - \texttt{NN}' \text{ nodes} & \text{if} \quad \texttt{NN}' < \texttt{NN} \\ \text{scale down with } \texttt{NN}' - \texttt{NN} \text{ nodes} & \text{if} \quad \texttt{NN} < \texttt{NN}' \\ \text{no action} & \text{otherwise} \end{cases} \tag{11.16}$$

We have conducted two series of experiments to prove our approach to elasticity control. By these experiments we check whether the elasticity feedback controller operates as expected. In the first series (which we call SLO Experiment), the load is increased to a higher level. This increase is expected to cause SLO violation that is detected by the feedback controller, which adds nodes in order to meet SLO under high load. In the second series (which we call Cost Experiment), the load decreases to a lower level. This causes the controller to release nodes in order to save cost under low load. The instance configuration for these experiments are as follows:

- CPU frequency: 2 GHz;
- Memory: 8 GB;
- Bandwidth: 2 MB/s;
- Number of simultaneous downloads: 70.

There are 10 data blocks in the experiments with sizes between 1 to 5 MB. Note that the same configuration is used in the system identification experiments.

SLO Experiment: Increasing Load

In this series we conducted two experiments: one with controller and another without controller. In the results and figures presented below, they are denoted by `w/controller` and `w/o controller`, respectively. Each experiment starts with three warmed up instances. By a warmed up instance we mean that in this instance each data block is requested at least once thus it resides in the memory of this instance.

Workload that is used for this experiment is of two levels: normal and high. Under the normal load the time interval between consecutive requests is selected from a uniform random distribution in the range [10, 15] seconds that corresponds to an average request rate of 4.8 requests per minute. Under the high load the time interval between consecutive requests is selected from a uniform random distribution in the range [1, 5] seconds that corresponds to an average request rate of 20 requests per minute. The experiment starts with normal load and after 500 seconds the workload increases to the high level. This is shown in Fig. 11.7.

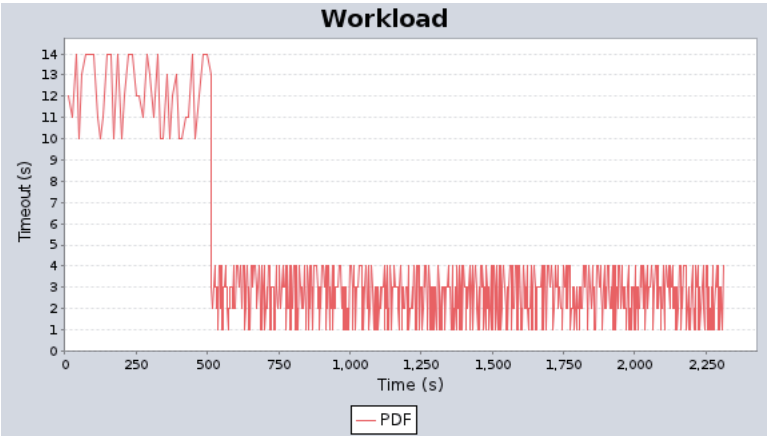

Figure 11.7: SLO Experiment Workload

Sensing of instance output is done every 25 seconds. In the case of controller, actuation is performed every 100 seconds. Thus there are 4 sets of measured data at each actuation time that the controller should consider. In order to calculate values of the system output, the controller computes averages of data sets. The duration of each experiment is 2000 seconds with warm up of 100 seconds. SLO requirements are as follows:

- Average CPU Load: $\leqslant 55\%$

Table 11.1: SLO Violations

SLO Parameter Violation (%)	w/ Controller	w/o Controller
CPU Load	17.94	72.28
Response Time	2.12	7.073
Bandwidth	35.89	74.69

- Response Time: $\leqslant 1,5$ seconds
- Average Bandwidth per download: > 200000 B/s

For each experiment the percentages of SLO violations are calculated for each aforementioned SLO requirement based on Equation 11.17. The result is shown in Table 11.1.

$$SLO\ Violations = 100\% \times \frac{Number\ of\ SLO\ Violations}{Total\ Number\ of\ SLO\ Checks} \tag{11.17}$$

Checking of SLO is done at each estimate (sensing) of the *Average CPU Load* and *Average Bandwidth per download* and each estimate of *Response Time*.

This experiment gives us interesting results that are discussed in this section. `NL` and `HL` in figures 11.8-11.12 indicate periods of *Normal Load* and *High Load* respectively.

Fig. 11.8 depicts the Average CPU Load for the aforementioned experiments. The Average CPU Load is the average of all nodes' CPU Loads at each time the sensing is performed. As one can see in Fig. 11.8, CPU loads for the experiment with the controller is generally lower than the same experiment without the controller. This is due to the controller that launches new instances under high workloads causing a huge drop in average CPU Load.

Fig. 11.9 depicts the Average Response Time for the experiments. By response time we mean the time that it takes for an instance to respond to a request that download is started and not the actual download time. As it is seen from the diagram, the average response time for the experiment with the controller is generally lower than the experiment without controller. This is because in case of having a fixed number of instances (3 in this experiment), there would be congestion by the number of requests an instance can process. This increases the responsivity of an instance. However, in the case that the controller launches new instances, no instance will actually go under high number of requests.

Fig 11.10 shows the total cost for the experiments. Interval total cost means that total cost is calculated for each interval in which the senses are done. As can be observed from the diagram, the interval total cost for the experiment with the controller is much higher than the experiment without the controller. This is because launching new instances will cost more money than having a fixed number

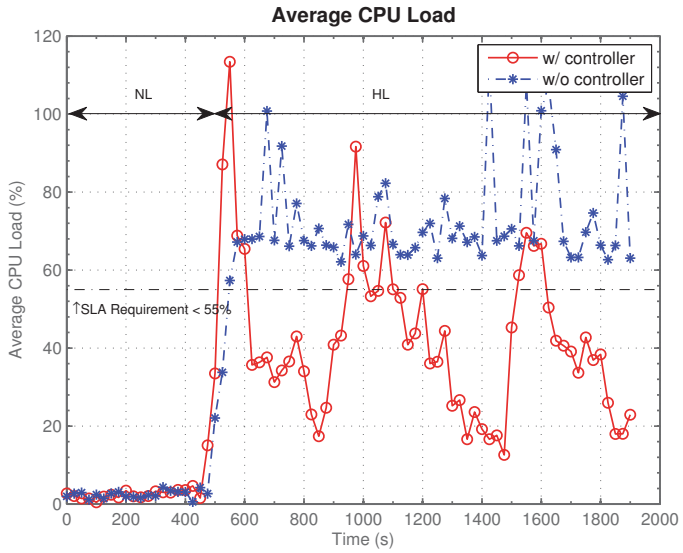

Figure 11.8: SLO Experiment - Average CPU Load

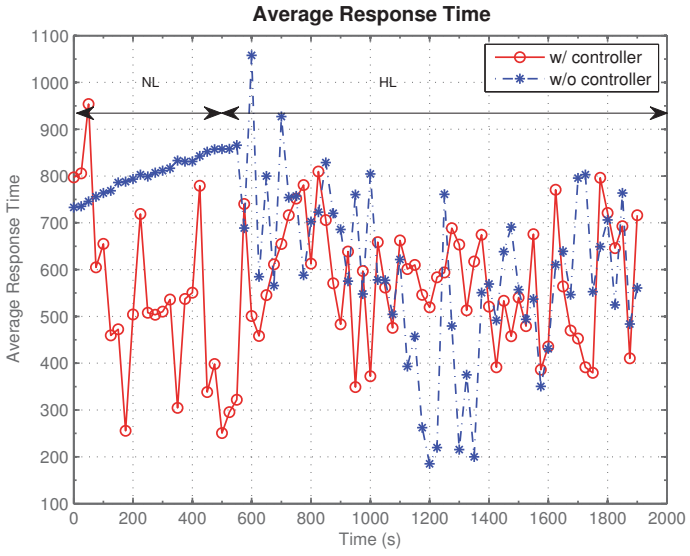

Figure 11.9: SLO Experiment - Average Response Time

	w/ controller	w/o controller
Total Cost ($)	14.4528	8.6779

Table 11.2: Total Cost for each SLO experiment

of instances available in the Cloud. This experiment has high load of requests for the system in which the controller is more likely to scale up and resides in that mood than to scale down. It should be noted that costs are computed according to Amazon EC2 price list.

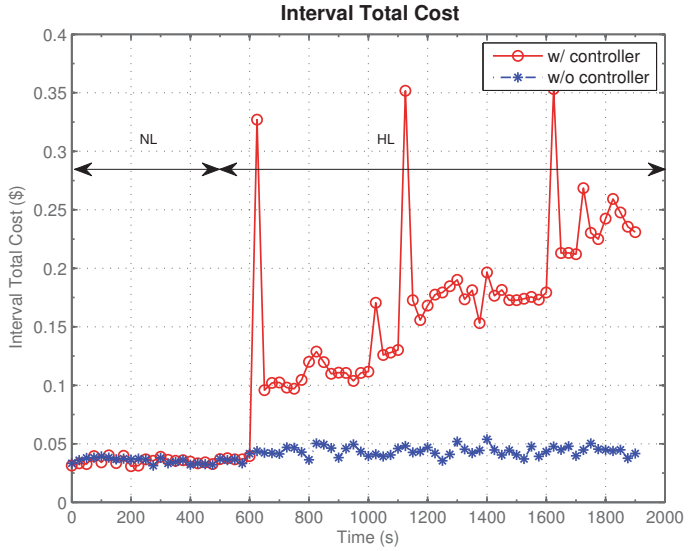

Figure 11.10: SLO Experiment - Interval Total Cost

Calculated total cost for each experiment is given in Table 11.2.

Fig. 11.11 depicts the Average bandwidth per download. If an instance has a bandwidth of 4 Mb/s and has two current downloads running, the bandwidth per download is 2 Mb/s. As can be seen from the diagram, the experiment with controller shows significantly higher bandwidth per download. This is mainly because the instances receive less number of requests and bandwidth is divided among less requests also. This will end up having higher bandwidth available on each instance.

Fig 11.12 shows the number of nodes for each experiment. As we discussed earlier the number of nodes is constant for experiment without controller. However, for the experiment with the controller the number of nodes is changed over time hence the SLO requirements can be met.

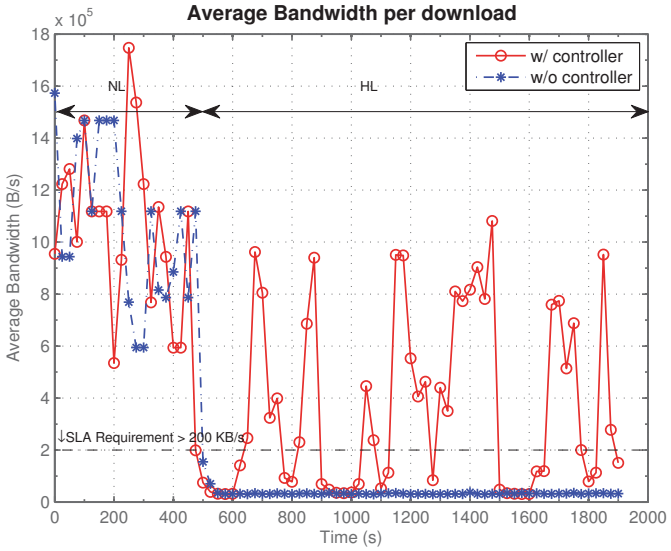

Figure 11.11: SLO Experiment - Average Bandwidth per download (B/s)

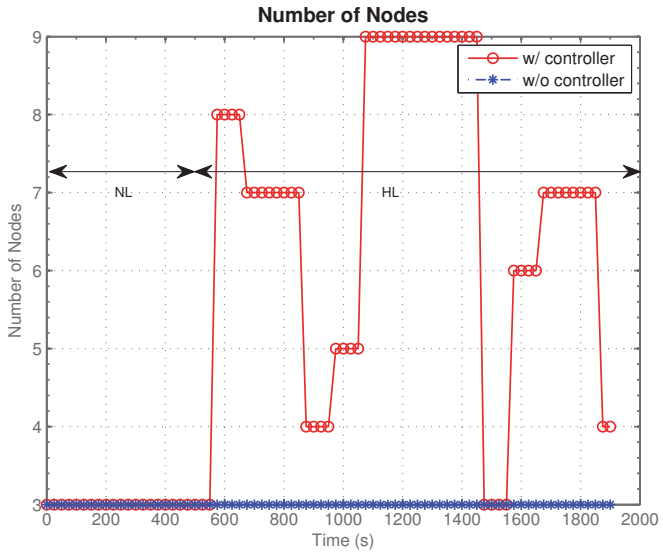

Figure 11.12: SLO Experiment - Number of Nodes

	w/ controller	w/o controller
Total Cost ($)	10.509	16.5001

Table 11.3: Total Cost for Cost experiment

Cost Experiment: Decreasing Load

The purpose of this series of experiments is to show that the controller can save the total cost by releasing instances when the load is low. Each experiment in this series starts with 7 instances. The duration of the experiment is 2000 seconds.

In this series we use different workloads of two levels: high and low. In the high load the time interval between consecutive requests is selected from a uniform random distribution in the range [1, 3] seconds that corresponds to a request rate of 30 requests per minute. In the low load the time interval between consecutive requests is selected from a uniform random distribution in the range [15, 20] seconds that corresponds to a request rate of about 3.4 requests per minute. Unlike the SLO experiment, the cost experiment starts with a high load, which changes to a low load after 500 seconds as shown in Fig. 11.13.

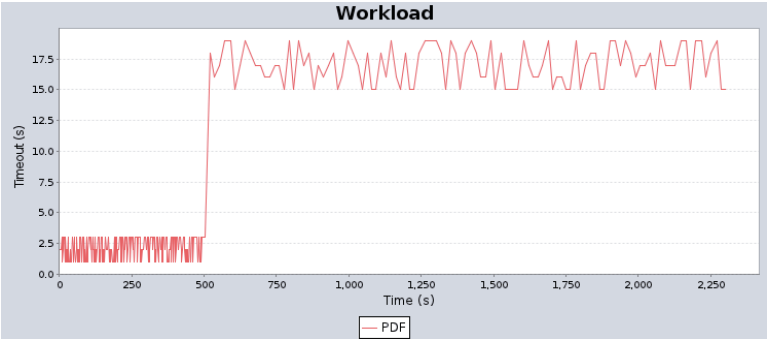

Figure 11.13: Cost Experiment workload

The result of the cost experiment shown in Table 11.3 is interesting. It is observed that the total cost in the experiment with the controller is actually lower than the total cost in the experiment without the controller unlike in the SLO experiment. This is because the controller removes instances under low load and that results in cost savings. The reason that this experiment has lower cost than the previous one is that L (lower bound on number of nodes) is not equal to the initial number of nodes and it is smaller. Hence controller can scale down the number of nodes to L.

11.9 Related Work

There are many projects that use elements of control theory for providing automated control of computing systems including Cloud-based services [14, 21, 132, 136–138, 141, 144–147]. Here we consider two related pieces of work [14, 145], which are the closest to our research aiming at automation of elasticity of storage services.

The SCADS Director proposed in [14] is a control framework that reconfigures a storage system at run time in response to workload fluctuations. Reconfiguration includes adding/removing servers, redistributing and replicating data between servers. The SCADS Director employs the Model-Predictive Control technique to predict system performance for the given workload using a performance model of the system and make control decisions based on prediction. Performance modeling is performed by statistical machine learning.

Lim et al. [145] have proposed a feedback controller for elastic storage in Cloud environment. The controller consists of three components: Horizontal Scale Controller responsible for scaling the storage; Data Rebalancer Controller that controls data transfer for rebalancing after scaling up/down; and the State Machine that coordinates the actions of the controllers in order to avoid wrong control decisions caused by interference of rebalancing with applications and sensor measurements.

To our knowledge both aforementioned projects do not explicitly use cost as a controller input (state variable, system output) in the controller design. In contrast, we use state-space feedback control and explicitly include the total cost of Cloud instances as a state (system output) variable in the state-space model (when identifying the system) and as a controller input in the controller design (when determining controller gains). This allows us to use a desired value of cost in addition to the SLO requirements to automatically control the scale of the storage by trading off performance for cost.

11.10 Conclusion and Future Work

Elasticity in Cloud computing is an ability of a system to scale up and down (request and release resources) in response to changes in its environment and workload. Elasticity provides an opportunity to scale up under high workload and to scale down under low workload to reduce the total cost for the system while meeting SLOs. We have presented our experience in designing an elasticity controller for a key-value store in a Cloud environment and described the steps in designing it including system identification and controller design. The controller allows the system to automatically scale the amount of resources while meeting performance SLO, in order to reduce SLO violations and the total cost for the provided service. We also introduced our open source simulation framework (EStoreSim) for Cloud systems that allows to experiment with different controllers and workloads. We have conducted two series of experiments using EStoreSim. Experiments have shown the feasibility of our approach to automate elasticity control of a key-value store in a

Cloud using state-space feedback control. We believe that this approach can be used to automate elasticity of other Cloud-based services.

In our future work, we will study other controller architectures such as model predictive control, and conduct experiments using real-world traces. We will also research on using feedback control for other elastic Cloud-based services.

Acknowledgments

This research is supported by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research, the Complex Service Systems focus project, a part of the ICT-TNG Strategic Research Area initiative at the KTH Royal Institute of Technology, and by the Testbed for E2E Clouds RCLD-project funded by EIT ICT Labs.

Chapter 12

ElastMan: Autonomic Elasticity Manager for Cloud-Based Key-Value Stores

Ahmad Al-Shishtawy and Vladimir Vlassov

Technical Report TRITA-ICT/ECS R 12:01, ISSN 1653-7238, ISRN KTH/ICT/ECS/R-12-01-SE, KTH Royal Institute of Technology, Stockholm, Sweden, August 2012.

ElastMan: Autonomic Elasticity Manager for Cloud-Based Key-Value Stores

Ahmad Al-Shishtawy[1,2], and Vladimir Vlassov[1]

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

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

Abstract

The increasing spread of elastic Cloud services, together with the pay-as-you-go pricing model of Cloud computing, has led to the need of an elasticity controller. The controller automatically resizes an elastic service, in response to changes in workload, in order to meet Service Level Objectives (SLOs) at a reduced cost. However, variable performance of Cloud virtual machines and nonlinearities in Cloud services, such as the diminishing reward of adding a service instance with increasing the scale, complicates the controller design. We present the design and evaluation of ElastMan, an elasticity controller for Cloud-based elastic key-value stores. ElastMan combines feedforward and feedback control. Feedforward control is used to respond to spikes in the workload by quickly resizing the service to meet SLOs at a minimal cost. Feedback control is used to correct modeling errors and to handle diurnal workload. To address nonlinearities, our design of ElastMan leverages the near-linear scalability of elastic Cloud services in order to build a scale-independent model of the service. Our design based on combining feedforward and feedback control allows to efficiently handle both diurnal and rapid changes in workload in order to meet SLOs at a minimal cost. Our evaluation shows the feasibility of our approach to automation of Cloud service elasticity.

12.1 Introduction

The growing popularity of Web 2.0 applications, such as wikis, social networks, and blogs, has posed new challenges on the underlying provisioning infrastructure. Many large-scale Web 2.0 applications leverage elastic services, such as elastic key-value stores, that can scale horizontally by adding/removing servers. Voldemort [9], Cassandra [10], and Dynamo [11] are few examples of elastic storage services.

Cloud computing [3], with its pay-as-you-go pricing model, provides an attractive environment to provision elastic services as the running cost of such services becomes proportional to the amount of resources needed to handle the current workload. The independence of peak loads for different applications enables Cloud

providers to efficiently share the resources among the applications. However, sharing the physical resources among Virtual Machines (VMs) running different applications makes it challenging to model and predict the performance of the VMs [39,40].

Managing the resources for Web 2.0 applications, in order to guarantee acceptable performance, is challenging because of the highly dynamic workload that is composed of both gradual (diurnal) and sudden (spikes) variations [41]. It is difficult to predict the workload particularly for new applications that can become popular within few days [12,13]. Furthermore, the performance requirement is usually expressed in terms of upper percentiles which is more difficult to maintain than the average performance [11, 14].

The pay-as-you-go pricing model, elasticity, and dynamic workload of Web 2.0 applications altogether call for the need for an elasticity controller that automates the provisioning of Cloud resources. The elasticity controller leverages the horizontal scalability of elastic services by provisioning more resources under high workloads in order to meet required service level objectives (SLOs). The pay-as-you-go pricing model provides an incentive for the elasticity controller to release extra resources when they are not needed once the workload decreases.

In this paper, we present the design and evaluation of ElastMan, an *Elast*icity *Man*ager for elastic key-value stores running in Cloud VMs. ElastMan addresses the challenges of the variable performance of Cloud VMs, dynamic workload, and stringent performance requirements expressed in terms of upper percentiles by combining feedforward control and feedback control. The feedforward controller monitors the current workload and uses a logistic regression model of the service to predict whether the current workload will cause the service to violate the SLOs or not, and acts accordingly. The feedforward controller is used to quickly respond to sudden large changes (spikes) in the workload. The feedback controller directly monitors the performance of the service (e.g., response time) and reacts based on the amount of deviation from the desired performance specified in the SLO. The feedback controller is used to correct errors in the model used by the feedforward controller and to handle gradual (e.g., diurnal) changes in workload.

Due to the nonlinearities in elastic Cloud services, resulting from the diminishing reward of adding a service instance (VM) with increasing the scale, we propose a scale-independent model used to design the feedback controller. This enables the feedback controller to operate at various scales of the service without the need to use techniques such as gain scheduling. To achieve this, our design leverages the near-linear scalability of elastic service. The feedback controller controls the number of nodes indirectly by controlling the average workload per server. Thus, the controller decisions become independent of the current number of instances that compose the service.

The major contributions of the paper are as follows.

- We leverage the advantages of both feedforward and feedback control to build an elasticity controller for elastic key-value stores running in Cloud environments.

- We propose a scale-independent feedback controller suitable for horizontally scaling services running at various scales.
- We describe the complete design of ElastMan including various techniques required to automate elasticity of Cloud-based services.
- We evaluate effectiveness of the core components of ElastMan using the Voldemort [9] key-value store running in a Cloud environment against both diurnal and sudden variations in workload.
- We provide an open source implementation of ElastMan with detailed instructions on how to repeat our experiments.

The rest of this paper is organized as following. Section 12.2 summarizes key concepts necessary for the paper. In Section 12.3 we describe the basic architecture of the target system we are trying to control. We continue by describing the design of ElastMan in Section 12.4. This is followed by the evaluation in Section 12.5. Related work is discussed in Section 12.6. We discus future work in Section 12.7 followed by conclusions in Section 12.8.

12.2 Background

In this section we lay out the necessary background for the paper. This include Web 2.0 applications, Cloud computing, elastic services, feedback control, and feed-forward control.

Web 2.0 Applications

Web 2.0 applications, such as Social Networks, Wikis, and Blogs, are data-centric with frequent data access [37]. This poses new challenges on the data-layer of multi-tier application servers because the performance of the data-layer is typically governed by strict Service Level Objectives (SLOs) [14] in order to satisfy costumer expectations.

With the rapid increase of Web 2.0 users, the poor scalability of a typical data-layer with ACID [38] properties limited the scalability of Web 2.0 applications. This has led to the development of new data-stores with relaxed consistency guarantees and simpler operations such as Voldemort [9], Cassandra [10], and Dynamo [11]. These storage systems typically provide simple key-value storage with eventual consistency guarantees. The simplified data and consistency models of key-value stores enable them to efficiently scale horizontally by adding more servers and thus serve more clients.

Another problem facing Web 2.0 applications is that a certain service, feature, or topic might suddenly become popular resulting in a spike in the workload [12, 13]. The fact that storage is a stateful service complicates the problem since only a particular subset of servers host the data related to the popular item.

These challenges have led to the need for an automated approach, to manage the data-tier, that is capable of quickly and efficiently responding to changes in the workload in order to meet the required SLOs of the storage service.

Cloud Computing and Elastic Services

Cloud computing [3], with its pay-as-you-go pricing model, provides an attractive solution to host the ever-growing number of Web 2.0 applications. This is mainly because it is difficult, specially for startups, to predict the future load that is going to be imposed on the application and thus the amount of resources (e.g., servers) needed to serve that load. Another reason is the initial investment, in the form of buying the servers, that is avoided in the Cloud pay-as-you-go pricing model.

To leverage the Cloud pricing model and to efficiently handle the dynamic Web 2.0 workload, Cloud services (such as key-value stores in the data-tier of a Cloud-based multi-tier application) are designed to be elastic. An Elastic service is designed to be able to scale horizontally at runtime without disrupting the running service. An elastic service can be scaled up (e.g., by the system administrator) in the case of increasing workload by adding more resources in order to meet SLOs. In the case of decreasing load, an elastic service can be scaled down by removing extra resource and thus reducing the cost without violating the service SLOs. For stateful services, scaling is usually combined with a rebalancing step necessary to redistribute the data among the new set of servers.

Feedback versus Feedforward Control

In computing systems, a controller [21] or an autonomic manager [5] is a software component that regulates the nonfunctional properties (performance metrics) of a target system. Nonfunctional properties are properties of the system such as the response time or CPU utilization. From the controller perspective these performance metrics are the *system output*. The regulation is achieved by monitoring the target system through a monitoring interface and adapting the system's configurations, such as the number of servers, accordingly through a control interface (*control input*). Controllers can be classified into feedback or feedforward controllers depending on what is being monitored.

In feedback control, the system's output (e.g., response time) is being monitored. The controller calculates the control error by comparing the current system's output to a desired value set by the system administrators. Depending on the amount and sign of the control error, the controller changes the control input (e.g., number of servers to add or remove) in order to reduce the control error. The main advantage of feedback control is that the controller can adapt to disturbance such as changes in the behaviour of the system or its operating environment. Disadvantages include oscillation, overshoot, and possible instability if the controller is not properly designed. Due to the nonlinearity of most systems, feedback controllers are ap-

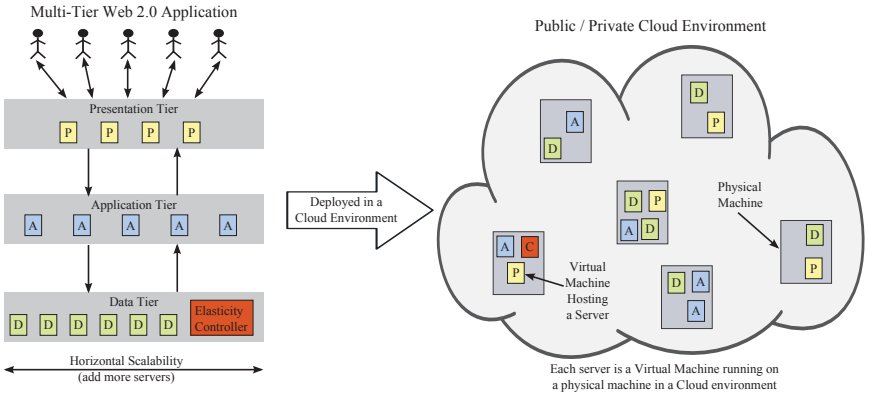

Figure 12.1: Multi-Tier Web 2.0 Application with Elasticity Controller Deployed in a Cloud Environment

proximated around linear regions called the operating region. Feedback controllers work properly only around the operating region they where designed for.

On the other hand, in feedforward control, the system's output is not being monitored. Instead the feedforward controller relies on a model of the system that is used to calculate the systems output based on the current system state. For example, given the current request rate and the number of servers, the system model is used to calculate the corresponding response time and act accordingly to meet the desired response time. The major disadvantage of feedforward control is that it is very sensitive to unexpected disturbances that are not accounted for in the system model. This usually results in a relatively complex system model compared to feedback control. The main advantages of feedforward control include being faster than feedback control and avoiding oscillations and overshoot.

12.3 Target System

We are targeting multi-tier Web 2.0 applications as depicted in the left side of Figure 12.1. We are focusing on managing the data-tier because of its major effect on the performance of Web 2.0 applications, which are mostly data centric [37]. Furthermore, the fact that storage is a stateful service makes it harder to manage as each request can be handled only by a subset of the servers that store replicas of the particular data item in the request.

For the data-tier, we assume horizontally scalable key-value stores due to their popularity in many large scale Web 2.0 applications such as Facebook and LinkedIn. A typical key-value store provides a simple put/get interface. This simplicity enables key-value stores to efficiently partition the data among multiple servers and thus to scale well to a large number of servers.

The minimum requirements to manage a key-value store using our approach (described in Section 12.4) is as follows. The store must provide a monitoring interface that enables the monitoring of both the workload and the latency of put/get operations. The store must also provide an actuation interface that enables the horizontal scalability by adding or removing service instances.

Because storage is a stateful service, actuation (adding or removing service instances) must be combined with a rebalance operation. The rebalance operation redistributes the data among the new set of servers in order to balance the load among them. Many key-value stores, such as Voldemort [9] and Cassandra [10], provide tools to rebalance the data among the service instances. In this paper, we focus on the control problem and rely on the built-in capabilities of the storage service to rebalance the load. If the storage does not provide such service, techniques such as rebalancing using fine grained workload statistics proposed by Trushkowsky et al. [14], the Aqueduct online data migration proposed by Lu et al. [42], or the data rebalance controller proposed by Lim et al. [43] can be used.

In this work we target Web 2.0 applications running in Cloud environments such as Amazon's EC2 [44] or private Clouds. The target environment is depicted on the right side of Figure 12.1. We assume that each service instance runs on its own VM; Each Physical server hosts multiple VMs. The Cloud environment hosts multiple such applications (not shown in the figure). Such environment complicates the control problem. This is mainly due to the fact that VMs compete for the shared resources. This high environmental noise makes it difficult to model and predict the performance of VMs [39, 40].

12.4 Elasticity Controller

The pay-as-you-go pricing model, elasticity, and dynamic workload of Web 2.0 applications altogether call for the need for an elasticity controller that automates the provisioning of Cloud resources depending on load. The elasticity controller leverages the horizontal scalability of elastic Cloud services by provisioning more resources under high workloads in order to meet the required SLOs. The pay-as-you-go pricing model provides an incentive for the elasticity controller to release extra resources when they are not needed once the workload starts decreasing.

In this section we describe the design of ElastMan, an elasticity controller designed to control the elasticity of key-value stores running in a Cloud environment. The objective of ElastMan is to regulate the performance of key-value stores according to a predefined SLO expressed as the 99th percentile of read operations latency over a fixed period of time.

Controlling a noisy signal, such as the 99th percentile, is challenging [14]. The high level of noise can mislead the controller into taking incorrect decisions. On the other hand, applying a smoothing filter in order to filter out noise, may also filter out a spike or, at least, delay its detection and handling. One approach to control noisy signals is to build a performance model of the system, thus avoiding

the need for measuring the noisy signal. The performance model used to predict the performance of the system given its current state (e.g., current workload). However, due to the variable performance of Cloud VMs (compared to dedicated physical servers), it is difficult to accurately model the performance of the services running in the Cloud.

To address the challenges of controlling a noisy signal and variable performance of Cloud VMs, ElastMan consists of two main components, a feedforward controller and a feedback controller. ElastMan relies on the feedforward controller to handle rapid large changes in the workload (e.g., spikes). This enables ElastMan to smooth the noisy 99th percentile signal and use feedback controller to correct errors in the feedforward system model in order to accurately bring the 99th percentile of read operations to the desired SLO value. In other words, the feedforward control is used to quickly bring the performance of the system near the desired value and then the feedback control is used to fine tune the performance.

The Feedback Controller

The first step in designing a feedback controller is to create a model of the target system (the key-value store in our case) that relates the control input to the system output (i.e., how a change in the control input affects the system output). For computing systems, a black-box approach is usually used [21], which is a statistical technique used to find the relation between the input and the output. The process of constructing a system model using the black-box approach is called *system identification*.

System identification is one of the most challenging steps in controller design. This is because a system can be modelled in many different ways and the choice of the model can dramatically affect the performance and complexity of the controller. The model of the system is usually a linear approximation of the behaviour of the system around an *operating point* (within an operating region). This makes the model valid only around the predefined point.

In order to identify a key-value store, we need to define what is the *control input* and the *system output*. In feedback control we typically monitor (measure) the system output that we want to regulate, which is, in our case, the 99th percentile of read operations latency over a fixed period of time (called R99p thereafter). The feedback controller calculates the error, which is the difference between the *setpoint*, which in our case is the required SLO value of R99p, and the measured system output as shown in equation 12.1.

$$e(t) = \text{Setpoint}_{\text{SLO\_R99p}} - \text{Measured}_{\text{R99p}}(t) \tag{12.1}$$

For the control input, an intuitive choice would be to use the number of storage servers. In other words, to try to find how changing the number of nodes affects the R99p of the key-value store. However, there are two drawback for this choice of a model. First, the model does not account for the current load on the sys-

tem. By load we mean the number of operations processed by the store per second (i.e., *throughput*). The latency is much shorter in an underloaded store than in an overloaded store. In this case, the load is treated as disturbance in the model. Controllers can be designed to reject disturbances but it might reduce the performance of the controller. Using the number of servers (which we can control) as a control input seems to be a natural choice since we can not control the load on the system as it depends on the number of clients interacting with our web application.

The second drawback is that using the number of servers as input to our model makes it nonlinear. For example, adding one server to a store having one server,doubles the capacity of the store. On the other hand, adding one server to a store with 100 servers increases the capacity by only one percent. This nonlinearity makes it difficult to design a controller because the model behaves differently depending on the size of the system. This might require having multiple controllers responsible for different operating regions corresponding to different sizes of the system. In control theory, this approach is known as gain scheduling.

In the design of the feedback controller for ElastMan, we propose to model the target store using the average throughput per server ans the control input. Although we can not control the total throughput on the system, we can indirectly control the average throughput of a server by adding/removing servers. Adding servers to the system reduces the average throughput per server, whereas removing servers reduces the average throughput per server. Our choice is motivated by the near linear scalability of elastic key-value stores (as discussed in Section 12.5). For example, when we double both the load and the number of servers, the R99p of the store remains roughly the same as well as the average throughput per server.

The major advantage of our proposed approach to model the store is that the model remains valid as we scale the store, and it does not dependent on the number of nodes. The noise in our model is the slight nonlinearity of the horizontal scalability of the elastic key-value store and the variable behaviour of the VMs in the Cloud environment. Note that this noise also exist in the previous model using the number of servers as control input.

In our proposed model, the operating point is defined as the value of the average throughput per server (input) and corresponding desired R99p (output); Whereas in the previous model, using the number of servers as a control input, the operating point is the number of servers (input) and corresponding R99p (output). Using our proposed model, the controller remains in the operating region (around operating point) as it scales the storage service. The operating region is defined around the value of the average throughput per server that produces the desired R99p regardless of the current size of the store. This eliminates the need for gain scheduling and simplifies the system identification and the controller design.

Given the current value of R99p, the controller uses the error, defined in Equation 12.1, to calculate how much the current throughput per server (called u) should be increased or decreased in order to meet the desired R99p defined in the SLO of the store. We build our controller as a classical PI controller described by Equation 12.2. The block diagram of our controller is depicted in Figure 12.2. The

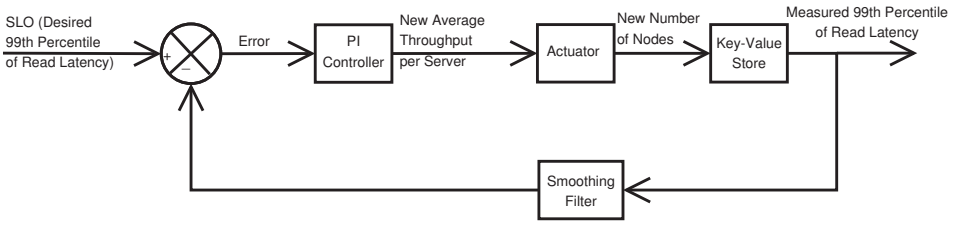

Figure 12.2: Block Diagram of the Feedback controller used in ElastMan

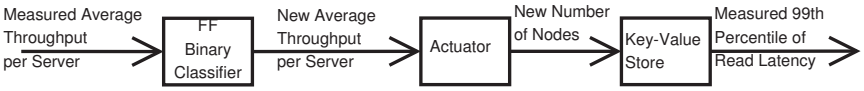

Figure 12.3: Block Diagram of the Feedforward controller used in ElastMan

controller design step involves using the system model to tune the controller gains, $\mathbf{K_p}$ and $\mathbf{K_i}$, which is out of the scope of the paper.

$$u(t+1) = u(t) + \mathbf{K_p}e(t) + \mathbf{K_i}\sum_{x=0}^{t} e(x) \tag{12.2}$$

The actuator uses the control output $u(t+1)$, which is the new average throughput per server, to calculate the new number of servers according to Equation 12.3.

$$\text{New Number of Servers} = \frac{\text{Current Total Throughput}}{\text{New Average Throughput per Server}} \tag{12.3}$$

The actuator uses the Cloud API to request/release resources and uses the elasticity API to add/remove new servers and also uses the rebalance API of the store to redistribute the data among servers.

The Feedforward Controller

ElastMan uses a feedforward model predictive controller to detect and quickly respond to spikes (rapid changes in workload). A model predictive controller uses a model of the system in order to reason about the current status of the system and make decisions. The block diagram of the feedforward controller is depicted in Figure 12.3. For our system we use a binary classifier created using logistic regression as proposed by Trushkowsky et al. [14]. The model is trained offline by varying the average intensity and the ratio of read/write operations per server as shown

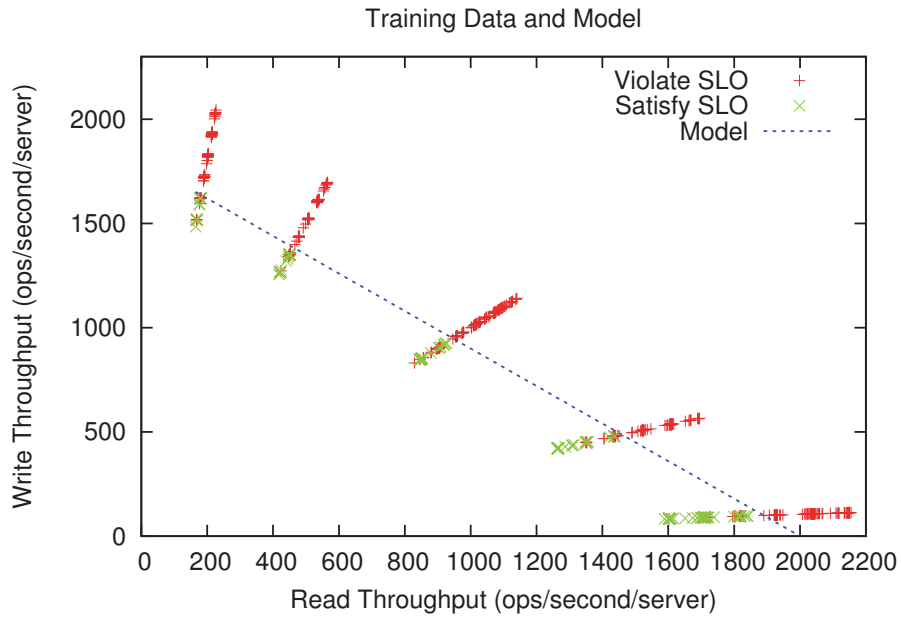

Figure 12.4: Binary Classifier for One Server

in Figure 12.4. The final model is a line that splits the plane into two regions. The region on and below the line is the region where the SLO is met whereas the region above the line is the region where the SLO is violated. Ideally, the average measured throughput should be on the line, which means that the SLO is met with the minimal number of servers.

In a very large system, averaging of throughput of servers may hide a spike that can occur on a single server or a small number of servers. In order to detect such spikes, the large system can be partitioned and each partition can be monitored separately.

The controller uses the model to reason about the current status of the system and make control decisions. If the measured throughput is far below the line, this indicates that the system is underloaded and servers (VMs) could be removed and vice versa. When a spike is detected, the feedforward controller uses the model to calculate the new average throughput per server. This is done by calculating the intersection point between the model line and the line connecting the origin with the point that corresponds to the measured throughput. The slope of the latter line is equal to the ratio of the write/read throughput of the current workload mix.

Then the calculated throughput is given to the actuator, which computes the new number of servers (using Equation 12.3) that brings the storage service close

to the desired operating point where the SLO is met with the minimal number of storage servers.

Note that the feedforward controller does not measure the R99p nor does make decisions based on error but relies on the accuracy of the model to check if the current load will cause an SLO violation. This makes the feedforward controller sensitive to noise such as changes in the behavior of the VM provided by the Cloud.

Elasticity Controller Algorithm of ElastMan

Our elasticity controller ElastMan combines the feedforward controller and feedback controller. The feedback and feedforward controllers complement each other. The feedforward controller relies on the feedback controller to correct errors in the feedforward model; whereas the feedback controller relies on the feedforward controller to quickly respond to spikes so that the noisy R99p signal that drives the feedback controller is smoothed. The flowchart of the algorithm of ElastMan is depicted in Figure 12.5.

The ElastMan elasticity controller starts by measuring the current 99th percentile of read operations latency (*R99p*) and the current average throughput (*tp*) per server. The *R99p* signal is smoothed using a smoothing filter resulting in a smoothed signal (*fR99p*). The controller then calculates the error e as in Equation 12.1. If the error is in the deadzone defined by a threshold around the desired *R99p* value, the controller takes no action. Otherwise, the controller compares the current *tp* with the value in the previous round. A significant change in the throughput (workload) indicate a spike. The elasticity controller then uses the feedforward controller to calculate the new average throughput per server needed to handle the current load. On the other hand, if the change in the workload is relatively small, the elasticity controller uses the feedback controller which calculates the new average throughput per server based on the current error. In both cases the actuator uses the current total throughput and the new average throughput per server to calculate the new number fo servers (Equation 12.3).

During the rebalance operation, which is needed in order to add or remove servers, both controllers are disabled as proposed by Lim et al. [43]. The feedback controller is disabled because the rebalance operation adds a significant amount of load on the system that causes increase in R99p. This can mislead the feedback controller causing it to wrongly add more resources. However if the storage system supports multiple rebalance instances or modifying the currently running rebalance instance, the feedforward controller can still be used. This is because the feedforward controller relies on the measured throughput of read/write operations (and it does not count rebalance operations) thus it will not be affected by the extra load added by the rebalancing operation.

Because the actuator can only add complete servers in discreet units, it will not be able to fully satisfy the controller actuation requests which are continuous values. For example, to satisfy the new average throughput per server, requested by the elasticity controller, the actuator might calculate that 1.5 servers are needed to be

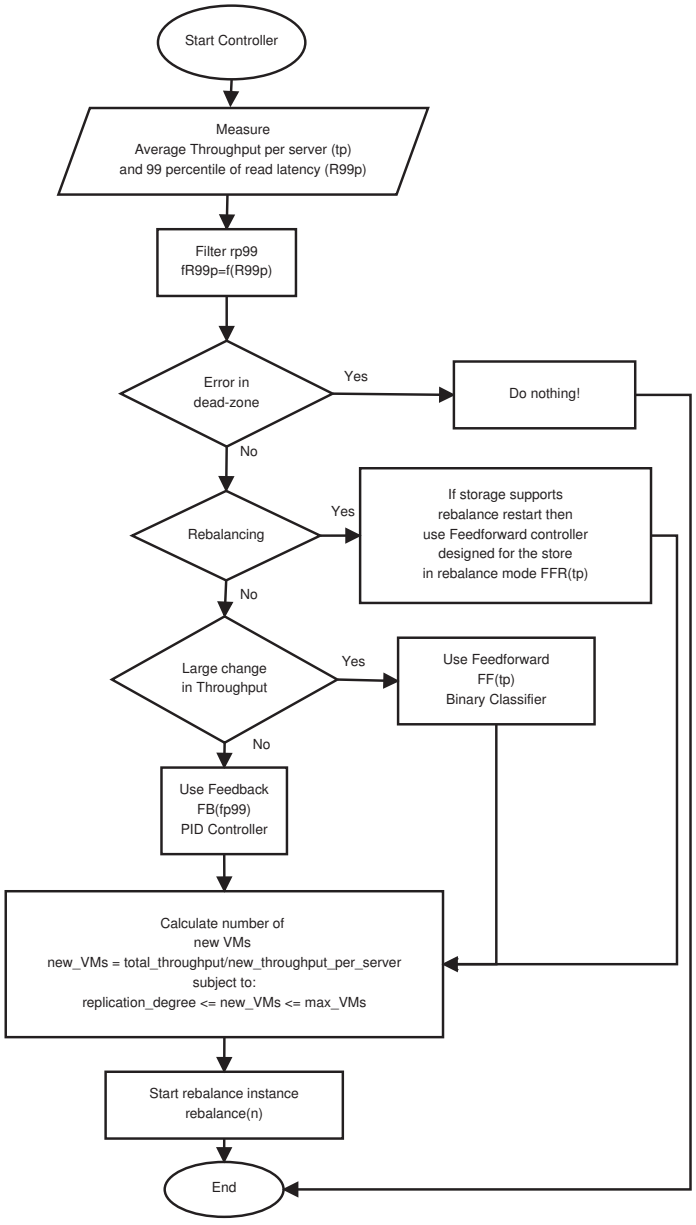

Figure 12.5: Combined Feedback and Feedforward Flow Chart

added (or removed). The actuator solves this situation by rounding the calculated value up or down to get a discrete value. This might result in oscillation, where the controller continuously adds and removes one node. Oscillations typically happen when the size of the storage cluster is small, as adding or removing a server have bigger effect on the total capacity of the storage service. Oscillations can be avoided by using the *proportional thresholding* technique as proposed by Lim et al. [43]. The basic idea is to adjust the lower threshold of the dead zone, depending on the storage cluster size, to avoid removing a server that will result in SLO violation and thus will request the server to be added back again causing oscillation.

12.5 Evaluation

We implemented ElastMan, our elasticity controller, in order to validate and evaluate the performance of our proposed solution to control an elastic key-value stores running in Cloud environments. The source code of ElastMan is publicly available[1].

Experimental Setup

In order to evaluate our ElastMan implementation, we used the Voldemort (version 0.91) Key-Value Store [9] which is used in production at many top Web 2.0 applications such as LinkedIn. We kept the core unmodified. We only extended the provided Voldemort client that is part of the Voldemort performance tool. The Voldemort performance tool is based on the YCSB benchmark [151]. Clients in our case represent the Application Tier shown in Figure 12.1. The clients connect to the ElastMan controller. Clients continuously measure the throughput and the 99th percentile of read operations latency. The controller periodically (every minute in our experiments) pulls the monitoring information from clients and then executes the control algorithm described in Section 12.4. ElastMan actuator uses the rebalance API of Voldemort to redistribute the data after adding/removing Voldemort servers.

Each client runs in its own VM in the Cloud and produces a constant workload of 3000 operations per second. The workload consists of 90% read operations and 10% read-write transactions unless otherwise stated. The total workload is increased by adding more client VMs and vice versa. This mimics the horizontal scalability of the Application Tier shown in Figure 12.1.

We run our experiments on a local cluster consisting of 11 Dell PowerEdge servers. Each server is equipped with two Intel Xeon X5660 processor (12 cores, 24 HW threads in total), and 44 GB of memory. The cluster runs Ubuntu 11.10. We setup a private Cloud using OpenStack Diablo release [92].

The Voldemort server is run in a VM with 4 cores and 6GB of memory, The Voldemort Clients run in a VM with 2 cores and 4GB of memory.

[1]The source code together with detailed instructions on how to repeat our experiments will be made publicly available after acceptance notification

Voldemort Servers	Min Client VMs	Max Client VMs	Max Cluster Total Cores	Max Cluster Total Mem (GB)	Cluster Load
9	1	12	60	102	22.7%
18	2	24	120	204	45.5%
27	3	36	180	306	68.2%
36	4	48	240	408	90.9%
45	5	60	300	510	113.6%
54	6	72	360	612	136.3%

Table 12.1: Parameters for the workload used in the scalability test.

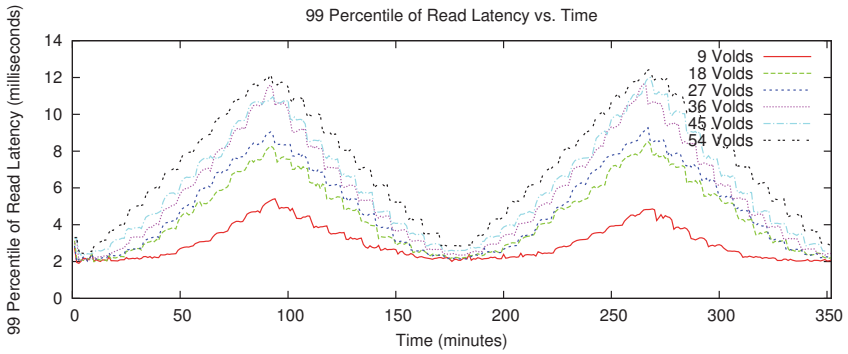

Figure 12.6: 99th percentile of read operations latency versus time under relatively similar workload

Horizontal Scalability of Key-Value stores

In this experiment we evaluated the scalability of the Voldemort key-value store in a Cloud environment. We gradually increased the cluster size and relatively scaled the workload as well. The amount of workload is described in Table 12.1.

The results, summarized in Figure 12.6 and Figure 12.7, shows the near linear scalability of Voldemort under normal load on our Cloud. That is between 45%-90% wich corresponds to 18 to 36 Voldemort servers. However, when the Cloud environment is underloaded (9 Voldemort servers at 22.7%) or when we Over-provision our Cloud environment (45 and 54 Voldemort servers at 113.6% and 136.3%), we notice a big change of the scalability of the Voldemort storage servers. This shows that the variable performance of the Cloud VMs can dramatically affect the performance and thus the ability to accurately model the system. This have motivated us to use the feedback controller in ElastMan to compensate such inaccuracies.

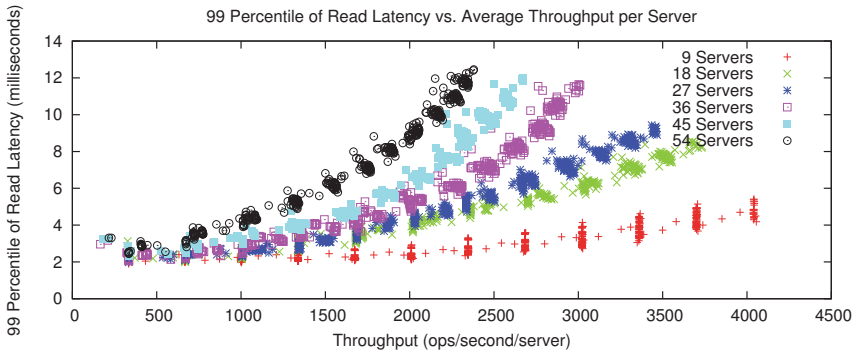

Figure 12.7: 99th percentile of read operations latency versus average throughput per server

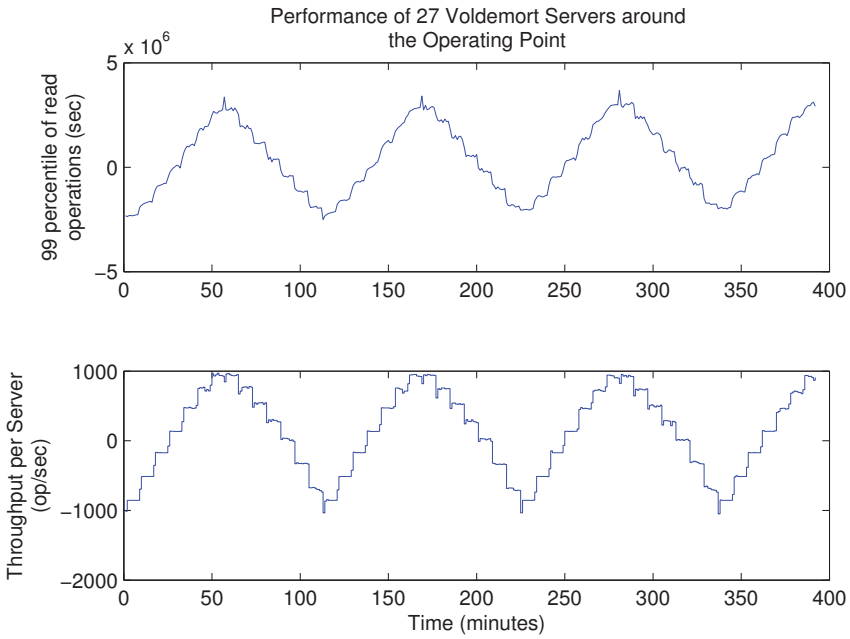

Figure 12.8: Training Data.

System Identification and Controller Design

We have used the average scenario of 27 Voldemort servers to create a model of the Voldemort storage service. The model is used to design and tune the feedback

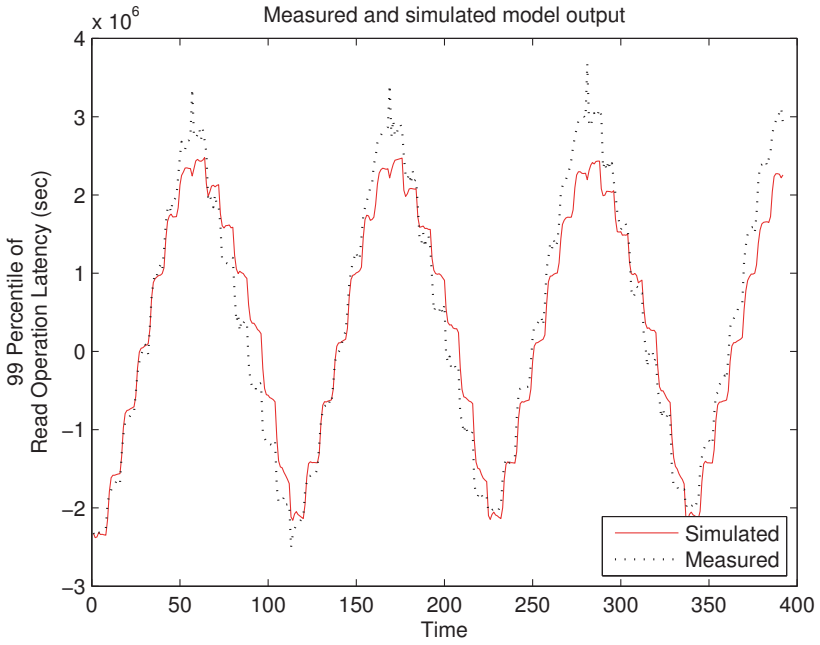

Figure 12.9: Model Performance.

controller. We used black-box system identification. The training data is shown in Figure 12.8. The performance of our model is shown in Figure 12.9 and Figure 12.10.

Varying Workload

We have tested ElastMan controller with both gradual diurnal workload and sudden increase/decrease (spikes) in workload. The goal of ElastMan controller is to keep the 99th percentile of read operation latency (R99p) at a predefined value (setpoint) as specified in the service SLO. In our experiments we choose the value to be 5 milliseconds in 1 minute period. Since it is not possible to achieve the exact specified value, we defined a 0.5 millisecond region around our setpoint with 1/3 above and 2/3 below. The controller does not react in this region which is known ad the deadzone for the controller. Note that we measure the latency from the application tier (see Figure 12.1). The overall latency observed by the clients depends on the request type that usually involves multiple read/write operations, and processing.

We start by applying gradual diurnal workload to the Voldemort cluster. The experiment starts with 9 Voldemort servers each running in its own VM. We set the maximum number of Voldemort VMs to 38. The total throughput applied on the

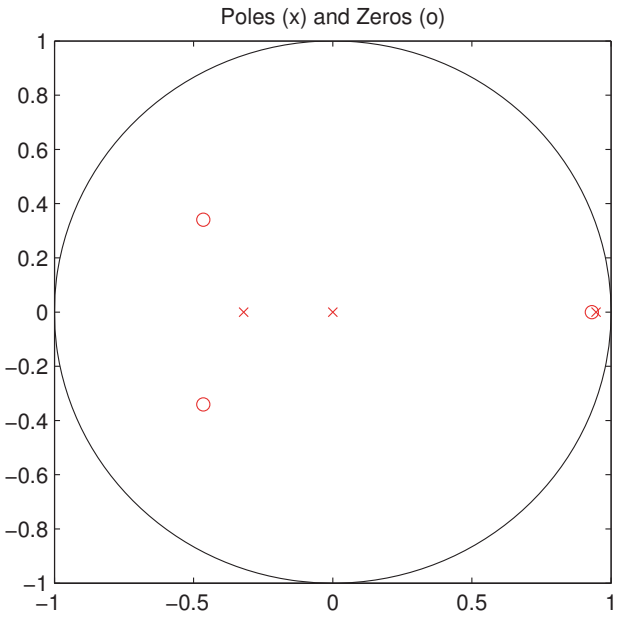

Figure 12.10: Zeros and Poles of our Model.

cluster starts with about 35000 requests per seconds and then increased to about 80000 requests per seconds. ElastMan controller is started after 30 min warm-up period. The results of our experiment is depicted in Figure 12.11. Up so far in this experiment, ElastMan relies mainly on the PI feedback controller since there are no sudden changes on the workload. ElastMan is able to keep the R99p within the desired region most of the time.

We continue the experiment by applying workload spikes with various magnitudes after 900 minutes. The results of the second part of the experiment is depicted in Figure 12.12. At the beginning of a spike, ElastMan (according to the algorithm) uses the feedforward controller since it detects large change in the workload. This is followed by using the feedback controller to fine tune the R99p at the desired value. For example, at time 924, the feedforward controller added 18 and at time 1024 added 14 nodes in order to quickly respond to the spike. Another example is at time 1336 after the spike where the feedforward controller removed 15 nodes.

Figure 12.13 depicts the performance of a Voldemort cluster with fixed number of servers (18 virtual servers).

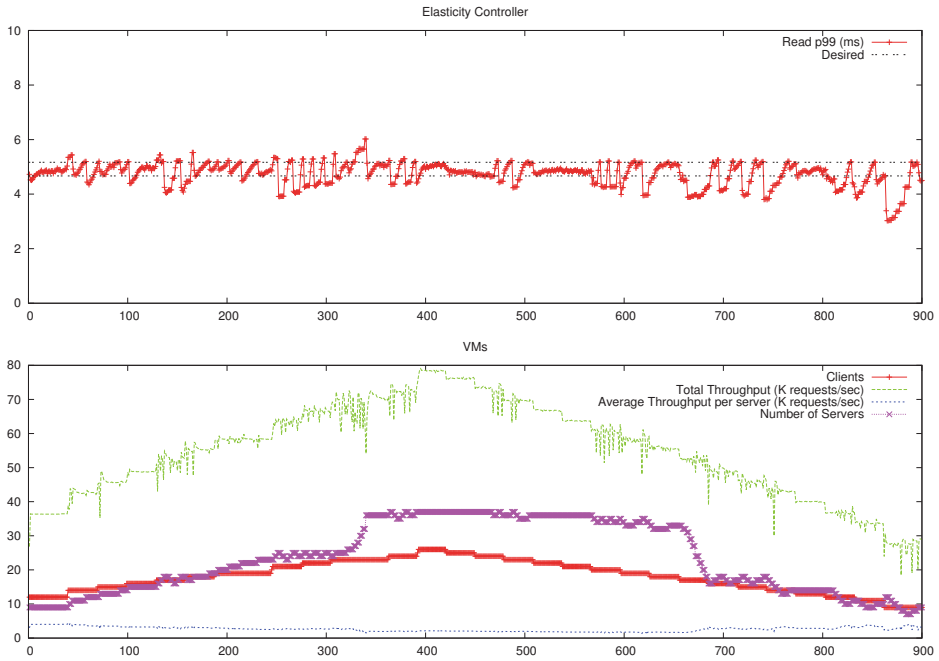

Figure 12.11: ElastMan controller performance under gradual (diurnal) workload

12.6 Related Work

There are many projects that use techniques such as control theory, machine learning, empirical modeling, or a combination of them to achieve SLOs at various levels of a multi-tier Web 2.0 application.

Lim et al. [43] proposed the use of two controllers. An integral feedback controller is used to keep the average response time at a desired level. A cost-based optimization is used to control the impact of the rebalancing operation, needed to resize the elastic storage, on the response time. The authors also propose the use of proportional thresholding, a technique necessary to avoid oscillations when dealing with discrete systems. The design of the feedback controller relies on the high correlation between CPU utilization and the average response time. Thus, the control problem is transformed into controlling the CPU utilization to indirectly control the average response time. Relying on such strong correlation might not be valid in Cloud environments with variable VM performance nor for controlling using 99th percentile instead of average. In our design, the controller uses a smoothed signal of the 99th percentile of read operations directly to avoid such problems. It is not clear how the controller proposed in [43] deals with the nonlinearity resulting

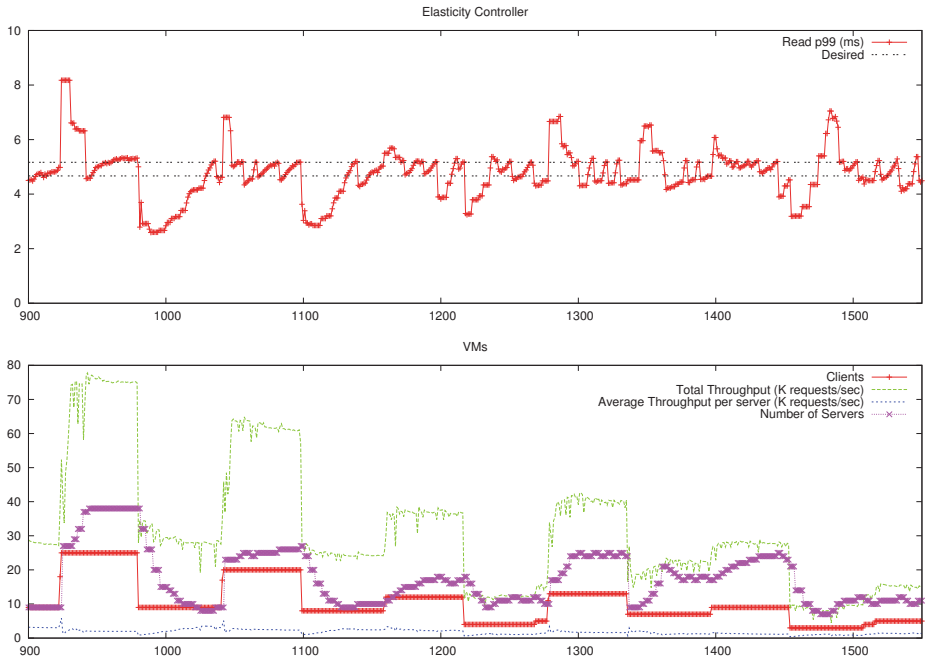

Figure 12.12: ElastMan controller performance with rapid changes (spikes) in workload

from the diminishing reward of adding a service instance with increasing the scale. Thus, it is not clear if the controller can work at different scales, a property that is needed to handle diurnal workload. In our approach we rely on the near-linear scalability of horizontally scalable stores to design a scale-independent controller that indirectly controls the number of nodes by controlling the average workload per server needed to handle the current workload. Another drawback in using only feedback controller is that it has to be switched off during rebalancing. This is because of the high disturbance resulting from the extra rebalancing overhead that can cause the feedback controller to incorrectly add more servers. We avoid switching off elasticity control during rebalancing, we use a feedforward controller tuned for rebalancing. The feedforward controller does not measure latency and thus will not be disturbed by rebalancing and can detect real increase/decrease in workload and act accordingly.

Trushkowsky et al. [14] were the first to propose a control framework for controlling upper percentiles of latency in a stateful distributed system. The authors propose the use of a feedforward model predictive controller to control the upper percentile of latency. The major motivation for using feedforward control is to

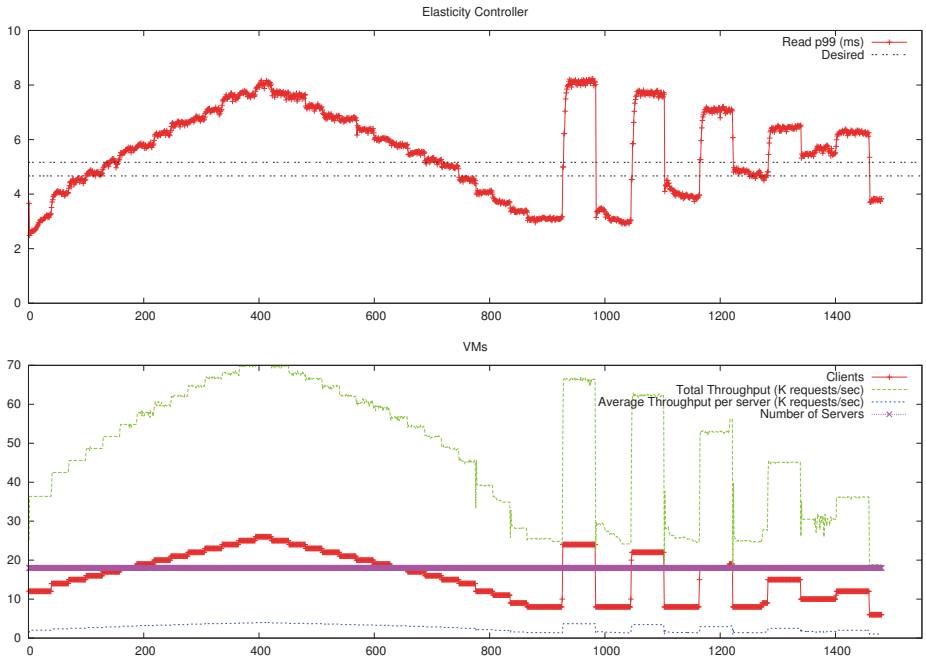

Figure 12.13: Voldemont performance with fixed number of servers (18 virtual servers)

avoid measuring the noisy upper percentile signal necessary for feedback control. Smoothing the upper percentile signal, in order to use feedback control, may filter out spikes or delay the response to them. The major drawback of using only feedforward is that it is very sensitive to noise such as the variable performance of VMs in the Cloud. The authors relies on replication to reduce the effect of variable VM performance, but in our opinion, this might not be guaranteed to work in all cases. Our approach combines both feedback and feedforward control, enabling us to leverage the advantages of both and avoid disadvantages. We rely of feedforward to quickly respond to spikes. This enables us to smooth the upper percentile signal and use feedback control to handle gradual workload and thus deal with modeling errors resulting from uncontrolled environmental noise. The authors [14] also propose the use of fine grained monitoring to reduce the amount of data transfer during rebalancing. This significantly reduces the disturbance resulting from the rebalance operation. Fine grain monitoring can be integrated with our approach to further improve the performance.

Malkowski et al. [72] focus on controlling all tiers on a multi-tier application due to the dependencies between the tiers. The authors propose the use of an empiri-

cal model of the application constructed using detailed measurements of a running application. The controller uses the model to find the best known configuration of the multi-tier application to handle the current load. If no such configuration exists, the controller falls back to another technique such as a feedback controller. Our work is different in a way that we integrate and leverage the advantages of both feedforward and feedback control. Although the empirical model will generally generate better results, it is more difficult to construct. The binary classifier proposed by Trushkowsky et al. [14] which we use together with feedback control to compensate for modeling errors is simpler to construct and might be more suitable for Cloud environments with variable VM performance. However, if needed, the empirical model can be used in our approach instead of the binary classifier. The extension of our work to control all tiers is our future work.

12.7 Future Work

A Web 2.0 application is a complex system consisting of multiple components. Controlling the entire system typically involves multiple controllers, with different management objectives, that interact directly or indirectly [47]. In our future work, we plan to investigate the controllers needed to control all tiers of a Web 2.0 application and the orchestration of the controllers in order to correctly achieve their goals.

We also plan to extend our implementation and evaluation of ElastMan to include the proportional thresholding technique as proposed by Lim et al. [43] in order to avoid possible oscillations in the feedback control. We also plan to provide the feedforward controller for the store in the rebalance mode (when performing rebalance operation). This will enable us to adapt to changes in workload that might happen during the rebalance operation.

Since ElastMan runs in the Cloud, it is necessary in real implementation to use replication in order to guarantee fault tolerance. One possible way is to use Robust Management Elements [90], that is based on replicated state machines, to replicate ElastMan and guarantee fault tolerance.

12.8 Conclusions

The strict performance requirements posed on the data-tier in a multi-tier Web 2.0 application together with the variable performance of Cloud virtual machines makes it challenging to automate the elasticity control. We presented the design and evaluation of ElastMan, an Elasticity Manager for Cloud-based key-value stores that address these challenges.

ElastMan combines and leverages the advantages of both feedback and feedforward control. The feedforward control is used to quickly respond to rapid changes in workload. This enables us to smooth the noisy signal of the 99th percentile of read operation latency and thus use feedback control. The feedback controller is

used to handle gradual (diurnal) workload and to correct errors in the feedforward control due to the noise that is caused mainly by the variable performance of Cloud VMs. The feedback controller uses a scale-independent model by indirectly controlling the number of servers (VMs) by controlling the average workload per server. This enables the controller, given the near-linear scalability of key-value stores, to operate at various scales of the store.

We have implemented and evaluated ElastMan using the Voldemort key-value store running in a Cloud environment based on OpenStack. The evaluation results show that ElastMan can handle both gradual (diurnal) workload and quickly respond to rapid changes in the workload (spikes).

Part V

Bibliography

Bibliography

[1] 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.

[2] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *Int. J. High Perform. Comput. Appl.*, vol. 15, pp. 200–222, Aug. 2001.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

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

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

[6] 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.

[7] 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.

[8] C. Arad, J. Dowling, and S. Haridi, "Building and evaluating P2P systems using the Kompics component framework," in *Peer-to-Peer Computing (P2P'09)*, pp. 93–94, IEEE, Sept. 2009.

[9] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *The 10th USENIX Conference on File and Storage Technologies (FAST'12)*, February 2012.

[10] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[12] "Animoto's Facebook scale-up (visited June 2012)."

[13] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, (New York, NY, USA), pp. 241–252, ACM, 2010.

[14] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The scads director: scaling a distributed storage system under stringent performance requirements," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2011.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, pp. 1277–1288, August 2008.

[16] F. Dabek, *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, November 2005.

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

[18] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart, "An architectural approach to autonomic computing," in *Autonomic Computing, 2004. Proceedings. International Conference on*, ICAC2004, pp. 2–9, may 2004.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] M. Ohara, P. Nagpurkar, Y. Ueda, and K. Ishizaki, "The data-centricity of web 2.0 workloads and its impact on server performance," in *ISPASS*, pp. 133–142, IEEE, 2009.

[38] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. Berkeley, CA, USA: Osborne/McGraw-Hill, 2nd ed., 2000.

[39] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling virtual machine performance: challenges and approaches," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, pp. 55–60, Jan. 2010.

[40] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell, "VM3: Measuring, modeling and managing VM shared resources," *Computer Networks*, vol. 53, pp. 2873–2887, December 2009.

[41] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *Network, IEEE*, vol. 14, pp. 30 –37, may/jun 2000.

[42] C. Lu, G. A. Alvarez, and J. Wilkes, "Aqueduct: Online data migration with performance guarantees," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, (Berkeley, CA, USA), USENIX Association, 2002.

[43] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, (New York, NY, USA), pp. 1–10, ACM, 2010.

[44] "Amazon elastic compute cloud (visited June 2012)."

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

[54] R. Yanggratoke, F. Wuhib, and R. Stadler, "Gossip-based resource allocation for green computing in large clouds," in *Network and Service Management (CNSM), 2011 7th International Conference on*, pp. 1 –9, oct. 2011.

[55] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *The 13th IEEE/IFIP Network Operations and Management Symposium*, NOMS 2012, (Hawaii, USA), April 2012.

[56] E. Elmroth, J. Tordsson, F. Hernández, A. Ali-Eldin, P. Svärd, M. Sedaghat, and W. Li, "Self-management challenges for multi-cloud architectures," in *Proceedings of the 4th European conference on Towards a service-based internet*, ServiceWave'11, (Berlin, Heidelberg), pp. 38–49, Springer-Verlag, 2011.

[57] 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.

[58] 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.

[59] 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.

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

[61] 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.

[62] L. Bao, A. Al-Shishtawy, and V. Vlassov, "Policy based self-management in distributed environments," in *Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on*, pp. 256 –260, September 2010.

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

[64] 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.

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

[66] "The vision cloud project."

[67] 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.

[68] 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.

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

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

[71] 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, 2008.

[72] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann, "Automated control for elastic n-tier workloads based on empirical modeling," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, (New York, NY, USA), pp. 131–140, ACM, 2011.

[73] V. Vlassov, A. Al-Shishtawy, P. Brand, and N. Parlavantzas, "Niche: A platform for self-managing distributed applications," in *Formal and Practical Aspects of Autonomic Computing and Networking: Specification, Development, and Verification* (P. Cong-Vinh, ed.), ch. 10, pp. 241–293, IGI Global, 2012. ISBN13: 9781609608453.

[74] 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.

[75] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," in *EuroSys'06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pp. 103–115, ACM, 2006.

[76] M. A. Fayyaz, "Achieving self management in dynamic distributed environments," Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology (ICT), Stockholm, Sweden, May 2010.

[77] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007.

[78] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[79] W. Vogels, "Eventually consistent," *Queue*, vol. 6, pp. 14–19, October 2008.

[80] A. Al-Shishtawy, T. J. Khan, and V. Vlassov, "Robust fault-tolerant majority-based key-value store supporting multiple consistency levels," in *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, (Tainan, Taiwan), pp. 589 –596, December 2011.

[81] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, (New York, NY, USA), pp. 150–162, ACM, 1979.

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

[83] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM'01*, pp. 149–160, Aug. 2001.

[84] T. M. Shafaat, M. Moser, A. Ghodsi, T. Schütt, S. Haridi, and A. Reinefeld, "On consistency of data in structured overlay networks," in *Proceedings of the 3rd CoreGRID Integration Workshop*, April 2008.

[85] M. A. Moulavi, A. Al-Shishtawy, and V. Vlassov, "State-space feedback control for elastic distributed storage in a cloud environment," in *The Eighth International Conference on Autonomic and Autonomous Systems ICAS 2012*, (St. Maarten, Netherlands Antilles), pp. 18–27, March 2012.

[86] 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.

[87] 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.

[88] 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.

[89] 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, October 2008.

[90] A. Al-Shishtawy, M. A. Fayyaz, K. Popov, and V. Vlassov, "Achieving robust self-management for large-scale distributed applications," in *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, pp. 31 –40, October 2010.

[91] A. Ghodsi, *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, KTH Royal Institute of Technology, School of Information and Communication Technology (ICT), 2006.

[92] "Openstack: Open source software for building private and public clouds.."

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

[94] 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.

[95] 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.

[96] 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.

[97] 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.

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

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] "Niche homepage."

[104] 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.

[105] 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.

[106] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi, "Efficient broadcast in structured p2p networks," in *IPTPS* (M. F. Kaashoek and I. Stoica, eds.), vol. 2735 of *Lecture Notes in Computer Science*, pp. 304–314, Springer, 2003.

[107] "OSGi service platform release (retrieved june 2010)."

[108] S. Sicard, F. Boyer, and N. De Palma, "Using components for architecture-based management," in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pp. 101–110, may 2008.

[109] A. T. Hannesson, "Yacs: Yet another computing service using niche," Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology (ICT), 2009.

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

[111] 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.

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

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

[114] 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.

[115] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[116] 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.

[117] "Meridian: A lighweight approach to network positioning." `http://www.cs.cornell.edu/People/egs/meridian`.

[118] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: estimating latency between arbitrary internet end hosts," in *IMW'02: 2nd ACM SIGCOMM Workshop on Internet measurment*, pp. 5–18, ACM, 2002.

[119] D. Leonard, Z. Yao, V. Rai, and D. Loguinov, "On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks," *IEEE/ACM Trans. Networking*, vol. 15, pp. 644–656, June 2007.

[120] B. M. Oki and B. H. Liskov, "Viewstamped replication: a general primary copy," in *PODC'88:* 7^{th} *Ann. ACM Symp. on Principles of Distributed Computing*, pp. 8–17, ACM, 1988.

[121] L. Lamport and M. Massa, "Cheap Paxos," in *DSN'04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, p. 307, IEEE Computer Society, June 28–July 1 2004.

[122] B. Charron-Bost and A. Schiper, "Improving fast Paxos: Being optimistic with no overhead," in *PRDC'06:* 12^{th} *Pacific Rim International Symp. on Dependable Computing*, pp. 287–295, IEEE, 2006.

[123] L. Lamport, "Generalized consensus and Paxos," Tech. Rep. MSR-TR-2005-33, MSR, Apr. 28 2005.

[124] F. Pedone and A. Schiper, "Generic broadcast," in *Distributed Computing, 13^{th} International Symposium* (P. Jayanti, ed.), vol. 1693 of *LNCS*, pp. 94–108, Springer, Sept. 27–29 1999.

[125] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI'99: 3^{rd} Symp. on Operating Systems Design and Implementation*, pp. 173–186, USENIX Association, 1999.

[126] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010.

[127] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 21, no. 5, pp. 123–138, 1987.

[128] A. Ghodsi, L. O. Alima, and S. Haridi, "Symmetric replication for structured peer-to-peer systems," in *Proceedings of The 3^{rd} Int. Workshop on Databases, Information Systems and P2P Computing*, (Trondheim, Norway), 2005.

[129] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[130] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, (New York, NY, USA), pp. 654–663, ACM, 1997.

[131] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 1094–1104, October 2001.

[132] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using control theory to achieve service level objectives in performance management," *Real-Time Syst.*, vol. 23, pp. 127–141, July 2002.

[133] D. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN*, vol. 17, no. 1, pp. 1–14, 1989.

[134] S. Keshav, "A control-theoretic approach to flow control," *SIGCOMM Comput. Commun. Rev.*, vol. 21, pp. 3–15, August 1991.

[135] B. Li and K. Nahrstedt, "A control-based middleware framework for quality-of-service adaptations," *Selected Areas in Communications, IEEE Journal on*, vol. 17, pp. 1632 –1650, sep 1999.

[136] A. Kamra, V. Misra, and E. M. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," *In International Workshop on Quality of Service (IWQoS*, pp. 47–56, 2004.

[137] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: a control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 80–96, August 2002.

[138] A. Robertson, B. Wittenmark, and M. Kihl, "Analysis and design of admission control in web-server systems," in *American Control Conference. Proceedings of the 2003*, vol. 1, pp. 254–259, june 2003.

[139] T. Abdelzaher and N. Bhatti, "Web content adaptation to improve server overload behavior," in *WWW8 / Computer Networks*, pp. 1563–1577, 1999.

[140] B. Li and K. Nahrstedt, "A control theoretical model for quality of service adaptations," in *In Proceedings of Sixth International Workshop on Quality of Service*, pp. 145–153, 1998.

[141] H. D. Lee, Y. J. Nam, and C. Park, "Regulating i/o performance of shared storage with a control theoretical approach," *NASA/IEEE conference on Mass Storage Systems and Technologies (MSST)*, April 2004.

[142] S. Mascolo, "Classical control theory for congestion avoidance in high-speed internet," in *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, vol. 3, pp. 2709 –2714 vol.3, 1999.

[143] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real-rate scheduling," in *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, (Berkeley, CA, USA), pp. 145–158, USENIX Association, 1999.

[144] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance isolation and differentiation for storage systems," in *In International Workshop on Quality of Service (IWQoS)*, pp. 67–74, 2004.

[145] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," *International Conf. on Autonomic Computing*, pp. 1–10, 2010.

[146] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *4th ACM European conf. on Computer systems*, pp. 13–26, 2009.

[147] C. Lu, T. Abdelzaber, J. Stankovic, and S. Son, "A feedback control approach for guaranteeing relative delays in web servers," in *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, pp. 51–62, 2001.

[148] "Kompics." `http://kompics.sics.se/`, accessed Oct 2011.

[149] "Scala language." `http://www.scala-lang.org/`, accessed Oct 2011.

[150] "EStoreSim: Elastic storage simulation framework." `https://github.com/amir343/ElasticStorage`, accessed Oct 2011.

[151] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.

Swedish Institute of Computer Science
SICS Dissertation Series

1. Bogumil Hausman, Pruning and Speculative Work in OR-Parallel PROLOG, 1990.
2. Mats Carlsson, Design and Implementation of an OR-Parallel Prolog Engine, 1990.
3. Nabiel A. Elshiewy, Robust Coordinated Reactive Computing in SANDRA, 1990.
4. Dan Sahlin, An Automatic Partial Evaluator for Full Prolog, 1991.
5. Hans A. Hansson, Time and Probability in Formal Design of Distributed Systems, 1991.
6. Peter Sjödin, From LOTOS Specifications to Distributed Implementations, 1991.
7. Roland Karlsson, A High Performance OR-parallel Prolog System, 1992.
8. Erik Hagersten, Toward Scalable Cache Only Memory Architectures, 1992.
9. Lars-Henrik Eriksson, Finitary Partial Inductive Definitions and General Logic, 1993.
10. Mats Björkman, Architectures for High Performance Communication, 1993.
11. Stephen Pink, Measurement, Implementation, and Optimization of Internet Protocols, 1993.
12. Martin Aronsson, GCLA. The Design, Use, and Implementation of a Program Development System, 1993.
13. Christer Samuelsson, Fast Natural-Language Parsing Using Explanation-Based Learning, 1994.
14. Sverker Jansson, AKL - - A Multiparadigm Programming Language, 1994.

15. Fredrik Orava, On the Formal Analysis of Telecommunication Protocols, 1994.
16. Torbjörn Keisu, Tree Constraints, 1994.
17. Olof Hagsand, Computer and Communication Support for Interactive Distributed Applications, 1995.
18. Björn Carlsson, Compiling and Executing Finite Domain Constraints, 1995.
19. Per Kreuger, Computational Issues in Calculi of Partial Inductive Definitions, 1995.
20. Annika Waern, Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction, 1996.
21. Björn Gambäck, Processing Swedish Sentences: A Unification-Based Grammar and Some Applications, 1997.
22. Klas Orsvärn, Knowledge Modelling with Libraries of Task Decomposition Methods, 1996.
23. Kia Höök, A Glass Box Approach to Adaptive Hypermedia, 1996.
24. Bengt Ahlgren, Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption, 1997.
25. Johan Montelius, Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, 1997.
26. Jussi Karlgren, Stylistic experiments in information retrieval, 2000.
27. Ashley Saulsbury, Attacking Latency Bottlenecks in Distributed Shared Memory Systems, 1999.
28. Kristian Simsarian, Toward Human Robot Collaboration, 2000.
29. Lars-åke Fredlund, A Framework for Reasoning about Erlang Code, 2001.
30. Thiemo Voigt, Architectures for Service Differentiation in Overloaded Internet Servers, 2002.
31. Fredrik Espinoza, Individual Service Provisioning, 2003.
32. Lars Rasmusson, Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design, 2002.
33. Martin Svensson, Defining, Designing and Evaluating Social Navigation, 2003.
34. Joe Armstrong, Making reliable distributed systems in the presence of software errors, 2003.

35. Emmanuel Frécon, DIVE on the Internet, 2004.

36. Rickard Cöster, Algorithms and Representations for Personalised Information Access, 2005.

37. Per Brand, The Design Philosophy of Distributed Programming Systems: the Mozart Experience, 2005.

38. Sameh El-Ansary, Designs and Analyses in Structured Peer-to-Peer Systems, 2005.

39. Erik Klintskog, Generic Distribution Support for Programming Systems, 2005.

40. Markus Bylund, A Design Rationale for Pervasive Computing - User Experience, Contextual Change, and Technical Requirements, 2005.

41. Åsa Rudström, Co-Construction of hybrid spaces, 2005.

42. Babak Sadighi Firozabadi, Decentralised Privilege Management for Access Control, 2005.

43. Marie Sjölinder, Age-related Cognitive Decline and Navigation in Electronic Environments, 2006.

44. Magnus Sahlgren, The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces, 2006.

45. Ali Ghodsi, Distributed k-ary System: Algorithms for Distributed Hash Tables, 2006.

46. Stina Nylander, Design and Implementation of Multi-Device Services, 2007

47. Adam Dunkels, Programming Memory-Constrained Networked Embedded Systems, 2007

48. Jarmo Laaksolahti, Plot, Spectacle, and Experience: Contributions to the Design and Evaluation of Interactive Storytelling, 2008

49. Daniel Gillblad, On Practical Machine Learning and Data Analysis, 2008

50. Fredrik Olsson, Bootstrapping Named Entity Annotation by Means of Active Machine Learning: a Method for Creating Corpora, 2008

51. Ian Marsh, Quality Aspects of Internet Telephony, 2009

52. Markus Bohlin, A Study of Combinatorial Optimization Problems in Industrial Computer Systems, 2009

53. Petra Sundström, Designing Affective Loop Experiences, 2010

54. Anders Gunnar, Aspects of Proactive Traffic Engineering in IP Networks, 2011

55. Preben Hansen, Task-based Information Seeking and Retrieval in the Patent Domain: Process and Relationships, 2011

56. Fredrik Österlind, Improving low-power wireless protocols with timing-accurate simulation, 2011