Self Management of Large-Scale Distributed Systems by Combining Peer-to-Peer Networks and Components

Peter Van Roy (`pvr@info.ucl.ac.be`)
Université catholique de Louvain, Louvain-la-Neuve (UCL)

Ali Ghodsi, Seif Haridi ({`aligh,seif`}`@kth.se`)
Royal Institute of Technology, Stockholm (KTH)

Jean-Bernard Stefani (`Jean-Bernard.Stefani@inria.fr`)
Institut National de Recherche en Informatique et Automatique, Grenoble (INRIA)

Thierry Coupaye (`thierry.coupaye@francetelecom.com`)
France Telecom R&D

Alexander Reinefeld (`ar@zib.de`)
Zuse Institut Berlin (ZIB)

Ehrhard Winter (`Ehrhard.Winter@eplus.de`)
E-Plus Mobilfunk

Roland Yap (`ryap@comp.nus.edu.sg`)
National University of Singapore (NUS)

CoreGRID Technical Report
Number TR-0018
December 14, 2005

Institute on System Architecture

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

Self Management of Large-Scale Distributed Systems by Combining Peer-to-Peer Networks and Components

Peter Van Roy (pvr@info.ucl.ac.be)
Université catholique de Louvain, Louvain-la-Neuve (UCL)

Ali Ghodsi, Seif Haridi ({aligh,seif}@kth.se)
Royal Institute of Technology, Stockholm (KTH)

Jean-Bernard Stefani (Jean-Bernard.Stefani@inria.fr)
Institut National de Recherche en Informatique et Automatique, Grenoble (INRIA)

Thierry Coupaye (thierry.coupaye@francetelecom.com)
France Telecom R&D

Alexander Reinefeld (ar@zib.de)
Zuse Institut Berlin (ZIB)

Ehrhard Winter (Ehrhard.Winter@eplus.de)
E-Plus Mobilfunk

Roland Yap (ryap@comp.nus.edu.sg)
National University of Singapore (NUS)

CoreGRID TR-0018
December 14, 2005

Abstract

This report envisions making large-scale distributed applications self managing by combining *component models* and *structured overlay networks*. A key obstacle to deploying large-scale applications running on Internet is the amount of management they require. Often these applications demand specialized personnel for their maintenance. Making applications self managing will help removing this obstacle. Basing the system on a structured overlay network will allow extending the abilities of existing component models to large-scale distributed systems. A structured overlay network is a form of peer-to-peer network that provides strong guarantees on its behavior. The guarantees are provided for efficient communication, efficient load balancing, and self management in case of node joins, leaves, and failures. Component models, on the other hand, support dynamic configuration, the ability of part of the system to reconfigure other parts at run-time. By combining overlay networks with component models we achieve both low-level as well as high-level self management. We are working on a practical implementation of this vision. To show its usefulness, we will target multi-tier applications, and in particular we will consider three-tier applications using a self-managing storage service.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

1 Introduction

An important objective of CoreGRID is to investigate the scalability of distributed systems in the areas of Grid and Peer-to-Peer technologies. In our view, it is not possible to achieve scalability without first achieving self management, since a large-scale distributed system will otherwise be completely unmanageable. This report presents a vision for self management that combines two areas that were previously considered independently, namely peer-to-peer systems and component models. We observe that for large-scale distributed applications, each of these areas is missing an essential part, and that the essential part is to be found in the other area! Peer-to-peer research does low-level self management based on algorithms and ignores how applications are to be deployed and maintained during their lifecycles. Component research provides mechanisms for high-level self management but barely touches the world of distributed computing; distribution in the component area is limited mainly to client/server computing and cluster computing. Peer-to-peer systems, if rethought in terms of components, will be able to handle the application lifecycle. Component models, if built on top of a peer-to-peer infrastructure, will be able to handle decentralized distributed applications. In this report, we motivate this vision and compare it with existing work in both areas and in the area of autonomic computing. We then elaborate on this vision and explain how we are working toward it.

To illustrate the general usefulness of our vision, let us see how it applies to a popular application architecture, the multi-tier application. Multi-tier applications are the mainstay of industrial applications. A typical example is a three-tier architecture, consisting of a client talking to a server, which itself interfaces with a database (see Figure 1). The business logic is executed at the server and the application data and meta data are stored on the database. But multi-tier architectures are brittle: they break when exposed to stresses such as failures, heavy loading (the "slash-dot effect"), network congestion, and changes in their computing environment. This becomes especially cumbersome for large-scale systems. Therefore, cluster-based solutions are employed where the three-tier architecture is duplicated within a cluster with high speed interconnectivity between tightly coupled servers. In practice, these applications require intensive care by human managers to provide acceptable levels of service, and make assumptions which are *only* valid within a cluster environment, such as perfect failure detection.

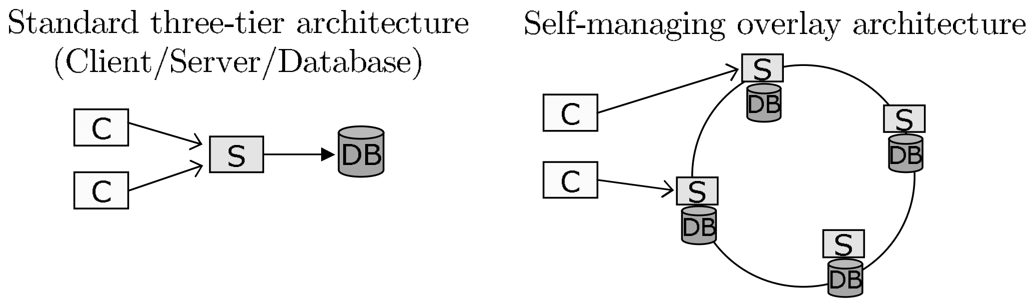
Standard three-tier architecture
(Client/Server/Database)
C
C
S
DB
Self-managing overlay architecture
C
C
S
DB
S
DB
S
DB
S
DB

Figure 1: A traditional and a self-managing three-tier architecture.

Lack of self management is not only pervasive in multi-tier architectures, but a problem in most distributed systems. For example, deploying a distributed file system across several organizations requires much manual configuration, as does adding another file server to the existing infrastructure. If a file server crashes, most file systems will stop functioning or fail to provide full service. Instead, the system should reconfigure itself to use another file server. This desirable behavior is an example of self management.

In this vision, we intend to make large-scale distributed applications such as these self managing. In this report we outline our vision of a general architecture which combines research on structured overlay networks together with research on component models. These two areas each provide what the other lacks: structured overlay networks provide a robust communications infrastructure and low-level self-management properties for Internet-scale distributed systems, and component models provide the primitives needed to support dynamic configuration and enable high-level self-management properties. We present our general architecture and outline the work that is needed to realize it. We will use this outline as a guide to help us orient our own research.

Structure of the report

This report is an extended version of a paper presented at the CoreGRID Integration Workshop held in Pisa, Italy, in November 2005. The report is divided into five main sections:

- Section 2 gives a definition of self management as we use it in this report.
- Section 3 compares our vision with related work in the areas of structured overlay networks (peer-to-peer systems), component models, and autonomic computing.
- Section 4 gives detailed motivation to explain why combining structured overlay networks with component models is a good approach for realizing self management. This section also presents in more detail our motivating example, a three-tier e-commerce application.
- Section 5 summarizes our recent work toward realizing the vision of self management.
- Section 6 gives a brief conclusion.

2 Definition of Self Management

Self management and self organization are overloaded terms widely used in many fields. We define self management along the same lines as done in [1], which can be summarized in that the system should be able to reconfigure itself to handle changes in its environment or requirements without human intervention but according to high-level management policies. It is important to give a precise definition of self management that makes it clear what parts can be handled automatically and what parts need application programmer or user (system administrator) intervention. The user then defines a self management policy and the system implements this policy. Self management exists on all levels of the system. At the lowest level, self management means that the system should be able to automatically handle frequent addition or removal of nodes, frequent failure of nodes, load balancing between nodes, and threats from adversaries. For large-scale systems, environmental changes that require some recovery by the system become normal and even frequent events. For example, failure becomes a normal situation: the probability that at a given time instant some part of the system is failed approaches 1 as the number of nodes increases. At higher levels, self management embraces many system properties. For our approach, we consider that these properties are classified in four axes of self management: *self configuration*, *self healing*, *self tuning*, and *self protection*.

To be effective, self management must be designed as part of the system from its inception. It is difficult or impossible to add self management a posteriori. This is because self management needs to be done at many levels of the system. Each level of the system needs to provide self management primitives ("hooks") to the next level.

The key to supporting self management is a service architecture that is a framework for building large-scale self-managing distributed applications. The heart of the service architecture is a component model built in synergy with a structured overlay network, to provide the following self-management properties:

1. *Self configuration*: Specifically, the infrastructure provides primitives so that the service architecture will continue to work when nodes are added or removed during execution. We will provide primitives so that parts of the application can be upgraded from one version to another without interrupting execution (online upgrade) . We will also provide a component trading infrastructure that can be used for automating distributed configuration processes.
2. *Self healing*: The service architecture will provide the primitives for continued execution when nodes fail or when the network communication between nodes fails, and will provide primitives to support the repair of node configurations. Specifically, the service architecture will continue to provide its basic services, namely communication and replicated storage, and will provide resource trading facilities to support repair mechanisms. Other services are application-dependent; the service architecture will provide the primitives to make it easy to write applications that are fault-tolerant and are capable of repairing themselves to continue respecting service level agreements.
3. *Self tuning*: The service architecture will provide the primitives for implementing load balancing and overload management. We expect that both load balancing and online upgrade will be supported by the component model, in the form of introspective operations (including the ability to freeze and restart a component and to get/set a component's state).

4. *Self protection*: Security is an essential concern that has to be considered globally. In a first approximation, we will consider a simple threat model, in which the nodes of the service architecture are considered trustworthy. We can extend this threat model with little effort for some parts, such as the structured overlay network, for which we already know how to protect against more aggressive threat models, such as Sybil attacks.

2.1 Feedback loops

An essential feature of self management is that it adds feedback loops throughout the system. A feedback loop consists of (1) the detection of an anomaly, (2) the calculation of a correction, and (3) the application of the correction. Whereas in non-self-managing systems, the feedback loops often go through a human being, in a self-managing system this is avoided. The human being is still needed, but to manage the loop from the outside and not to be part of the loop. Feedback loops can exist within one level or cross levels. An example of a loop within a level is failure handling in a structured overlay network: it contains a feedback loop to detect a node failure and to reorganize its routing information. In some cases, it can be important for failure handling to cross levels. For example, the low level detects a network problem, a higher level is notified and decides to try another communication path, and the low level then implements that decision (see Figure 2).

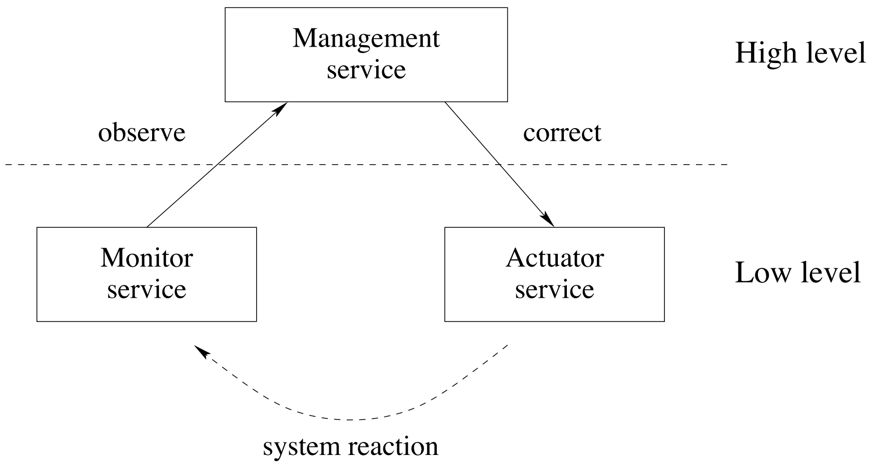

Figure 2: An example of a feedback loop that traverses system levels.

Because of the feedback loops, it is important that the system behavior converges (no oscillatory, chaotic, or divergent behavior). In the future, we intend to model formally the feedback loops, to confirm convergent behavior (possibly changing the design), and to validate the model with the system. This research was pioneered in the classic work of Norbert Wiener on Cybernetics [2]. We will apply it to the area of large-scale distributed systems. We will use techniques from several domains. One important source of inspiration is the physics of complex systems. For example, Krishnamurthy et al [3] have used the master equation approach to formalize the algorithms of the Chord structured overlay network and to derive its behavior under various rates of node leaves and joins. Another source of inspiration is the domain of electronic circuit design. Electronic systems frequently use feedback loops, in particular negative feedback, to keep the system near its desired behavior. For example, phase-locked loops (PLLs) are used in demodulators to track a signal that can have strong random fluctuations in frequency. If the incoming signal deviates in frequency from the PLL's internal oscillator then a correction is applied to the oscillator. This kind of negative feedback works for systems that can be approximated by linear or monotonic behavior. Unfortunately, the formal model of a computer system is generally highly nonlinear or nonmonotonic, so that naive approximations may not work. It is one of our goals to explore this aspect of feedback loops. We will use our experience in exploring how to program systems with monotonic behaviors (which are usually called *declarative*; see chapters 2 to 4 in [4]). In addition, it may be possible to exploit oscillatory or chaotic behavior to enhance certain characteristics of the system. We will explore this aspect as well.

3 Related Work

Our approach to self management can be considered a computer systems approach. That is, we give a precise definition of self management in terms of computer system properties, namely configuration, fault tolerance, performance, and security. To make these properties self managing, we propose to design a system architecture and the protocols it needs. But in the research community self management is sometimes defined in a broader way, to touch on various parts of artificial intelligence: learning systems, swarm intelligence (a.k.a. collective intelligence), biologically-inspired systems, and learning from the immune system [1]. We consider that these artificial intelligence approaches are worth investigating in their own right. However, we consider that the computer systems approach is a fundamental one that has to be solved, regardless of these other approaches. In what follows, we focus on the computer systems approach.

Let us characterize the advantages of our proposed architecture with respect to the state of the art in computer systems. There are three areas to which we can compare our approach:

1. *Structured overlay networks and peer-to-peer systems.* Current research on overlay networks focuses on algorithms for basic services such as communication and storage. The reorganizing abilities of structured overlay networks can be considered as low-level self management. We extend this to address high-level self management such as configuration, deployment, online updating, and evolution, which have been largely ignored so far in structured overlay network research.

2. *Component-based programming.* Current research on components focuses on architecture design issues and not on distributed programming. We extend this to study component-based abstractions and architectural frameworks for large-scale distributed systems, by using overlay networks as an enabler.

3. *Autonomic systems.* Most autonomic systems focus on individual autonomic properties, specific self-managed systems, or focus on specific elements of autonomic behavior. Little research has considered the overall architectural implications of building self-managed distributed systems. Our position is unique in this respect, combining as it does component-based system construction with overlay network technology into a service architecture for large-scale distributed system self management.

We now present these areas in more detail and explain where the contribution of our approach fits. In Sections 4 and 5 we then go deeper into the specifics of our approach.

3.1 Structured overlay networks and peer-to-peer systems

Research on peer-to-peer networks has evolved into research on structured overlay networks, in particular on Distributed Hash Tables (DHTs). The main differences between popular peer-to-peer systems and structured overlay networks are that the latter provide strong guarantees on routing and message delivery, and are implemented with more efficient algorithms. The research on structured overlay networks has matured considerably in the last few years [5, 6, 7]. Hardware infrastructures such as PlanetLab have enabled DHTs to be tested in realistically harsh environments. This has led to structured peer-to-peer communication and storage infrastructures in which failures and system changes are handled gracefully.

At their heart, structured overlay networks enable the nodes in a distributed system to organize themselves to provide a shared directory service. Any application built on top of an overlay can add information to this directory locally, which immediately results in the overlay system distributing the data onto the nodes in the system, ensuring that the data is replicated in case some of the nodes become unavailable due to failure.

The overlay guarantees that any node in the distributed system can access data inserted to the directory efficiently. The efficiency, calculated as the number of reroutes, is typically $log_k(N)$, where N is the number of nodes in the system, and k is a configurable parameter. The overlay makes sure that the nodes are interconnected such that data in the directory always can be found. The number of connections needed vary in different system, but are typically in the range $O(1)$ to $O(\log N)$, where N is the number of nodes in the overlay.

Though most overlays provide a simple directory, other abstractions are possible too. More recently, a relational view of the directory can be provided [8], and the application can use SQL to query the relational database for information. Most ordinary operations, such as selection, projection, and equi-joins are supported.

All structured overlays provide self management in presence of node joins and node departures. This means that a running system will adapt itself if new nodes arrive or if some nodes depart. Self management is done at two distinct layers: the communication layer and the storage management layer.

When nodes join or leave the system, the communication layer of the structured peer-to-peer system will ensure that the routing information present in the system is updated to adapt to these changes. Hence, routing can efficiently be done in presence of dynamism. Similarly, the storage management layer maintains availability of data by transferring data which is stored on a departing node to an existing node in the system. Conversely, if a new node arrives, the storage management layer moves part of the existing data to the new node to ensure that data is evenly distributed among the nodes in the system. Hence, data is self configured in presence of node joins and leaves.

In addition to the handling of node joins and leaves, the peer-to-peer system self heals in presence of link failures. This requires that the communication layer can accurately detect failures and correct routing tables accordingly. Moreover, the communication layer informs the storage management layer such that data is fetched from replicas to restore the replication degree when failures occur.

Much research has also been conducted in making peer-to-peer systems self tuning. There are many techniques employed to ensure that the heterogeneous nodes that make up the peer-to-peer system are not overloaded [9]. Self tuning is considered with respect to amount of data stored, amount of routing traffic served, and amount of routing information maintained. Self tuning is also applied to achieve proximity awareness, which means that routing done on the peer-to-peer network reflects the latencies in the underlying network.

Lately, research has been conducted in modeling trust to achieve security in large-scale systems [10]. In essence, a node's future behavior can be predicted by judging its previous behavior. The latter information can be acquired by regularly asking other nodes about their opinion about other nodes.

3.2 Component-based programming

The main current de-facto standards in distributed software infrastructures, Sun's J2EE, Microsoft .Net, and OMG CORBA, provide a form of component-based distributed programming. Apart from the inclusion of publish-subscribe facilities (e.g. the JMS publish-subscribe services in J2EE), support for the construction of large-scale services is limited. Management functions are made available using the traditional manager agent framework [11] but typically do not support online reconfiguration or autonomous behavior (which are left unspecified). Some implementations (e.g. JBoss) have adopted a component-based approach for the construction of the middleware itself, but they remain limited in their reconfiguration capabilities (coarse-grained, mostly deployment time, no support for unplanned software evolution).

Component models supported by standard platforms such as J2EE (the EJB model) or CORBA (the CCM model) are non-hierarchical (an assemblage of several components is not a component), and provide limited support for component introspection and dynamic adaptation. These limitations have been addressed in work on adaptive middleware (e.g. OpenORB, Dynamic TAO, Hadas, that have demonstrated the benefits of a reflective component-based approach to the construction of adaptive middleware). In parallel, a large body of work on architecture description languages (e.g. ArchJava, C2, Darwin, Wright, Rapide, Piccola, Acme or CommUnity) has shown the benefits of explicit software architecture for software maintenance and evolution. The component models proposed in these experimental prototypes, however, suffer from several limitations:

1. They do not allow the specification of component structures with sharing, a key feature required for the construction of software systems with resource multiplexing.

2. They remain limited in their adaptation capabilities, defining, for those that do provide such capabilities, a fixed meta-object protocol that disallows various optimizations and does not support different design trade-offs (e.g. performance vs. flexibility).

3. Finally, and most importantly, they lack abstractions for building large distributed structures.

Compared to the current industrial and academic state of the art in component-based distributed system construction, our approach intends to extend a reflective component-based model, the Fractal model [12], that subsumes the capabilities of the above models (it caters to points (1) and (2)) in order to address point (3).

3.3 Autonomic systems

The main goal of autonomic system research is to automate the traditional functions associated with systems management, namely configuration management, fault management, performance management, security management, and cost management [11]. This goal is becoming of utmost importance because of increasing system complexity. It is this very realization that prompted major computer and software vendors to launch major R&D initiatives on this theme, notably, IBM's Autonomic Computing initiative and Microsoft's Dynamic Systems initiative.

The motivation for autonomic systems research is that networked environments today have reached a level of complexity and heterogeneity that make their control and management by human administrators more and more difficult. The complexity of individual elements (a single software element can literally have thousands of configuration parameters), combined with the brittleness inherent of today's distributed applications, makes it more and more difficult to entertain the presence of a human administrator in the "management loop". Consider for instance the following rough figures: One-third to one-half of a company's total IT budget is spent preventing or recovering from crashes, for every dollar used to purchase information storage, 9 dollars are spent to manage it, 40% of computer system outages are caused by human operator errors, not because they are poorly trained or do not have the right capabilities, but because of the complexities of today's computer systems.

IBM's autonomic computing initiative, for instance, was introduced in 2001 and presented as a "grand challenge" calling for a wide collaboration toward the development of computing systems that would have the following characteristics: self configuring, self healing, self tuning and self protecting, targeting the automation of the main management functional areas (self healing dealing with responses to failures, self protecting dealing with responses to attacks, self tuning dealing with continuous optimization of performance and operating costs). Since then, many R&D projects have been initiated to deal with autonomic computing aspects or support techniques. For example, we mention the following projects that are most relevant to our vision: the recovery-oriented computing project at UC Berkeley, the Smartfrog Project at HP Research Labs in Bristol, UK, and the Swan project at INRIA, Alcatel, France Telecom. Compared to these projects, the uniqueness of our approach is that it combines structured overlay networks with component models for the development of an integrated architecture for large-scale self-managing systems. Each complements the other: overlay networks support large-scale distribution, and component models support reconfiguration. None of the aforementioned projects provide such a combination, which gives a uniform architectural model for self-managing systems. Note also that many of the above-mentioned projects are based on cluster architectures, whereas our approach targets distributed systems that may be loosely coupled.

4 Synergy of Overlays and Components

The foundation of our approach is to combine a structured overlay network with a component model. Both areas have much matured in recent years, but they have been studied in isolation. It is a basic premise of our approach that their combination will enable achieving self management in large-scale distributed systems. This is first of all because structured overlay networks already have many *low-level* self-management properties. Structured overlay network research has achieved efficient routing and communication algorithms, fault tolerance, handling dynamism, proximity awareness, and distributed storage with replication. However, almost no research has been done on deployment, upgrading, continuous operation, and other *high-level* self-management properties.

We explain what we mean with lack of high level self management in overlay networks by the following concrete problems. An overlay network running on thousands of nodes will occasionally need a software upgrade. How can a thousand node peer-to-peer system, dispersed over the Internet, be upgraded on the fly without interrupting existing services, and how do we ensure that it is done securely? How can it be guaranteed that the new version of the overlay software will not break when deployed on a node which does not have all the required software? For example, the new version might be making calls to certain libraries which might not be available on every node.

To continue the example, nodes in the overlay might provide different services or may run different versions of the services. For instance, an overlay might provide a rudimentary routing service on every node. But it might be that high-level services, such as a directory service, do not exist on every node. We need to be able to introspect nodes to find out such information, and, if permitted, install the required services on the remote machine at runtime. Even if the nodes do provide a directory service, it might be of different incompatible versions. For example, a node might be running an old version which stores directory information in memory, while another node has support for secure and persistent data storage.

The above mentioned research issues have been ignored by the peer-to-peer community. By using components, we can add these high-level self-management properties, such as deployment, versioning, and upgrade services. Recent research on component models, such as the Fractal model [12], is adding exactly those abilities that are needed for doing self management (such as reification and reflection abilities).

4.1 A three-tier e-commerce application

We now give a motivational example which will show how the overlay and the component model is used to build a scalable fault-tolerant application. Imagine an e-commerce application that allows users to use their web browser to buy books. The user can browse through the library of books, and add/remove books to its shopping cart. When the user has finished shopping, it can decide to either make a purchase or cancel it.

Traditionally, the above application is realized by creating a three-tier architecture, where the client makes request to an application server, which uses a database to store session information, such as the contents of the shopping cart.

In our system (see Figure 1), there will be several application servers running on different servers, possibly geographically dispersed running on heterogeneous hardware. Each application server is a node in a structured overlay network and can thus access the storage layer, which is a distributed hash table provided by the overlay. The storage layer has a thin layer which provides a relational view of the directory, allowing SQL queries, and supports transactions on top of the distributed hash table. A user visiting the shopping site will be forwarded by a load balancer to an appropriate server which can run the e-commerce application. The component model will enable the load balancer to find the server which has the right contextual environment, e.g. with J2EE installed and with certain libraries, and which is not overloaded. Thereafter the request is forwarded to the appropriate server, which uses the overlay storage layer to store its session state.

To continue our above example, we would like the e-commerce application to be self healing and provide *failover*. This can be realized by providing a failover component which periodically checkpoints by invoking an interface in the e-commerce application forcing it to save its entire state and configuration to the overlay storage . Should the application server crash, the failure detectors in the crashed node's neighborhood will detect this. One such neighbor is chosen by the overlay and the application is executed on that node. The component model ensures that the last saved state will be loaded by making calls to a standard interface in e-commerce application which will load the session state.

We might want our application to be self tuning, such that the e-commerce application running on an overloaded server is migrated to another application server . This could be solved using different approaches. One approach would be to have a component which saves the state of a session, and initiates another server to start the e-commerce application with the saved state. Notice that the level of granularity is high in this case as the component model would only define interfaces for methods which the e-commerce application would implement. These methods would then save the application specific state to the storage layer. Similarly, interfaces would be defined to tell the application to load its state from the storage layer. Another approach, with a low-level of granularity, would be to use a virtual machine such as Xen or VMWare. With these, the whole e-commerce application, its OS and state, would be moved to another machine. This would nevertheless require that the application is running on a common distributed file system, or is getting its data from a common database. The overlay could be used to either provide a self-managing distributed file system, or let the application use the overlay storage to fetch and store its data. The virtual machine approach has the additional advantage that it guarantees that applications running on the same machine are shielded securely from each other. At the same time, the virtual machine approach would not be able to run if the servers actual hardware differ.

5 Realizing the Synergy

To realize the synergy of overlay networks and components, we have started work in both areas. In the area of components, we are working on building the basic mechanisms of self management on top of the Fractal model [13, 12]. In the area of overlay networks, we are working on a decentralized service architecture, P2PKit, built on top of a structured overlay network [14]. We expect the work in both of these areas to interact and converge in the future. Current trends in component programming indicate that more and more programming will be done by means of components, and that the partition between programming language and component will be more and more weighted in favor of components.

5.1 Components for self management

The Fractal model is being used to construct self-managing systems. Quéma [13] describes several of these systems and the framework that is used to implement them. We briefly present the Fractal model and explain how it implements the mechanisms needed for self management.

5.1.1 The Fractal model

We have developed a component model, Fractal, that provides most of the primitives necessary for implementing self management. The Fractal model has the following abilities:

- *Compound components*: A component can be formed from other components.
- *Shared components*: A component can be a subcomponent of two different components. This is important for modeling resources, which often exist in a single component, while at the same time preserving encapsulation.
- *Introspection*: A component's execution can be observed by another component.
- *Configuration and reconfiguration*: Component instances can be installed, removed, and replaced dynamically. This work is done by other components in the system.

A Fractal component has one or more interfaces, where an *interface* is an access point for the component. Interfaces exist both for the functional and non-functional behavior of the component. A Fractal component usually has two parts: a membrane and a content. The membrane hosts the interfaces and the content consists of the subcomponents. A Fractal component can have several levels of control interface: attribute control (for setting the control state of the component), linking control (for managing linkage between interfaces), content control (for adding/removing subcomponents), lifecycle control (for stopping, starting, and other abilities related to the lifecycle).

5.1.2 Constructing self-managing systems with Fractal

Fractal has the support for implementing feedback loops similar to Figure 2 at the level of individual components. We explain how this support is implemented.

- *Actuators*: Using the control interfaces, a self-management service can control the system, for example to start and stop a component or to reconfigure part of the system (remove/add a subcomponent, create or remove a link, modify a component's code, etc.).
- *Monitors*: Fractal has defined a software framework called LeWYS that permits to define probes to observe parts of the system. The probes run independently of the rest of the system by means of an active communication mechanism called a *pump*, in which the probe is activated when needed and its results passed on to the higher level.
- *Management service*: The management service consists of several subcomponents, each dedicated to a different non-functional behavior such as deployment, fault tolerance, performance tuning, and so forth. Together, these subcomponents form a representation of the system being managed. Another entity (a human being, for example) can interact with this representation at the level of management policies. The subcomponents then perform the necessary actions so that the policies are implemented.

These mechanisms are currently defined in a centralized way, i.e., they are not being used for large-scale distributed systems. We intend to provide support for large-scale distribution by means of structured overlay networks.

5.2 Decentralized service architecture

We have developed a first version of a decentralized service architecture, called P2PKit, which uses a structured overlay network, called P2PS [14]. Both P2PKit and P2PS are written in Mozart, an advanced portable programming environment which is efficiently supported on most flavors of Unix and Windows and on Mac OS X [15]. Mozart provides a network-transparent distribution support that simplifies writing distributed applications. We are currently enhancing the distribution support so that it can use *reflective routing*: the basic routing of the system will then be defined by P2PS instead of being hard-wired into the system. We briefly present P2PS and P2PKit, and we explain how we intend to implement self management on top of them.

5.2.1 The P2PS library

P2PS is a communication library written in Mozart that provides a robust communications infrastructure with point-to-point and broadcast abilities. P2PS implements the Tango protocol [16], which is an extension of the DKS protocol implemented at SICS and KTH [17]. These protocols efficiently maintain their routing tables in the face of failures. Whenever a message is sent between nodes, the protocol takes advantage of this message to correct any errors in the routing tables. Furthermore, whenever a node fails, this is detected by neighboring nodes and they take action to correct routing tables. Finally, there are simple algorithms to handle prolonged network inactivity, to eject nodes that are no longer behaving correctly because of this. We are testing P2PS and DKS on top of the PlanetLab infrastructure and on top of the EVERGROW cluster infrastructure.

5.2.2 Constructing self-managing systems with P2PKit

P2PKit allows to deploy *services*, which we define as a set of component instances that collaborate to provide a functionality. A service is typically decentralized, with one component per process. Services are constructed in layers. For example, a failure detection service can be input to an application service. We have implemented sample services for peer failure notification, replicating data, and network control and monitoring. These services are robust: they automatically adapt as peers join and leave the network. The P2PS library itself is being rewritten to have a component structure instead of being written in a monolithic fashion.

P2PKit uses a simple component model, Mozart functors, and can install components and provide components with the ability to communicate with each other. To enable self management, we will extend this component model with abilities similar to those in the Fractal model.

6 Conclusions

We have outlined an approach for building large-scale distributed applications that are self managing. The approach exploits the synergy between structured overlay networks and component models. Each of these areas has matured considerably in recent years, but in isolation. Each area lacks the abilities provided by the other. Structured overlay networks lack the deployment and configuration abilities of component models. Component models lack the decentralized distributed structure of structured overlay networks. By combining the two areas, we expect to eliminate both of these lacks and achieve a balanced approach to self management. We are working on both areas, notably on the Fractal component model and the P2PKit service architecture, and we expect to merge them and add self management mechanisms according to the approach of this report.

Additional acknowledgements

In addition to CoreGRID, this work is supported by the EVERGROW Integrated Project (contract number 001935), funded by the European Union in the sixth Framework Programme. This work is also based in part on experiments done using the PlanetLab network (`http://www.planet-lab.org`) for providing a worldwide network of around 600 machines hosted by 275 sites in 30 countries.

References

[1] Herrmann, K., Mühl, G., Geihs, K.: Self-management: The solution to complexity or just another problem? IEEE Distributed Systems Online (DSOnline) **6**(1) (2005)

[2] Wiener, N.: Cybernetics; or, Control and Communication in the Animal and the Machine. Wiley (1955)

[3] Krishnamurthy, S., El-Ansary, S., Aurell, E., Haridi, S.: A statistical theory of Chord under churn. In: The 4th Interational Workshop on Peer-to-Peer Systems (IPTPS'05). (2005)

[4] Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)

[5] Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: ACM SIGCOMM 2001, San Deigo, CA (2001) 149–160

[6] Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Lecture Notes in Computer Science **2218** (2001)

[7] Alima, L.O., Ghodsi, A., Haridi, S.: A Framework for Structured Peer-to-Peer Overlay Networks. In: LNCS post-proceedings of Global Computing, Springer Verlag (2004) 223–250

[8] Chun, B., Hellerstein, J.M., H., R., Jeffery, S.R., Loo, B.T., Mardanbeigi, S., Roscoe, T., Rhea, S., Shenker, S., Stoica, I.: Querying at internet scale. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (2004) 935–936

[9] Karger, D.R., Ruhl, M.: Simple efficient load balancing algorithms for peer-to-peer systems. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM Press (2004) 36–43

[10] Aberer, K., Despotovic, Z.: Managing trust in a peer-2-peer information system. In Paques, H., Liu, L., Grossman, D., eds.: Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM01), ACM Press (2001) 310–317

[11] Distributed Management Task Force: http://www.dmtf.org (2005)

[12] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: An Open Component Model and Its Support in Java. Lecture Notes in Computer Science **3054** (2004) International Symposium on Component-based Software Engineering (CBSE'2004).

[13] Quéma, V.: Vers l'exogiciel: Une approche de la construction d'infrastructures logicielles radicalement configurables. PhD thesis, Institut National Polytechnique de Grenoble (2005)

[14] Glynn, K.: P2PKit: A services based architecture for deploying robust peer-to-peer applications. http://p2pkit.info.ucl.ac.be. EVERGROW Project Deliverable (2006)

[15] Mozart Consortium: Mozart programming system. http://www.mozart-oz.org (2004)

[16] Carton, B., Mesaros, V.: Improving the scalability of logarithmic-degree DHT-based peer-to-peer networks. In: Euro-Par 2004. (2004)

[17] Alima, L.O., El-Ansary, S., Brand, P., Haridi, S.: DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In: The 3rd International workshop on Global and Peer-To-Peer Computing on large scale distributed systems - CCGRID2003, Tokyo, Japan (2003)