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.

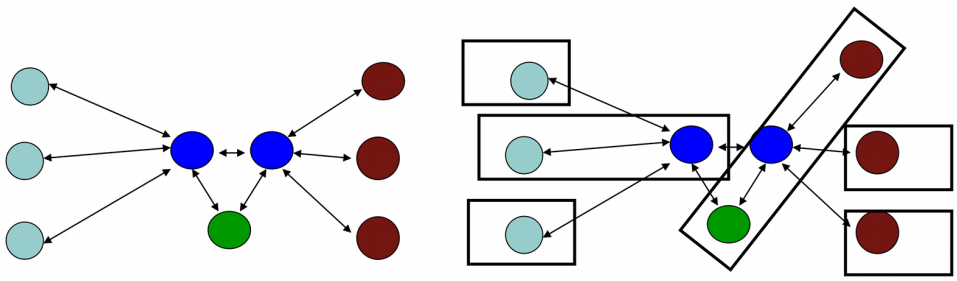


Figure 7.1: Abstract (left) and concrete (right) view of a configuration. Boxes represent nodes or virtual machines, circles represent components.

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”

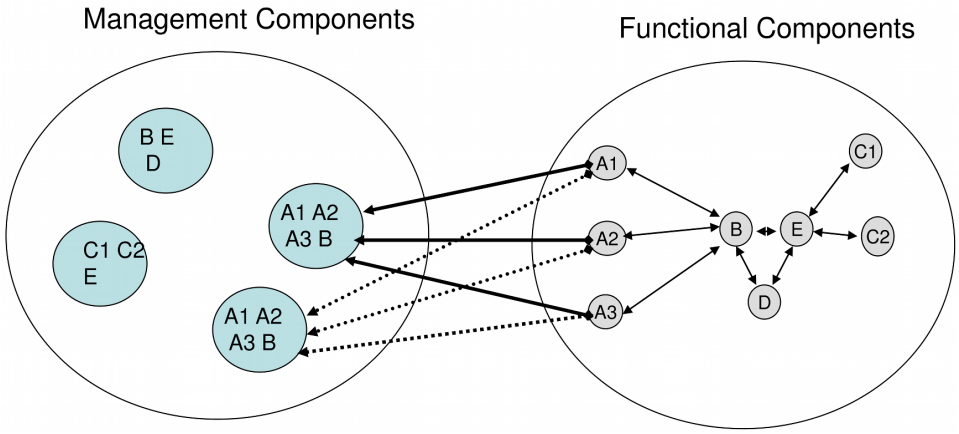


Figure 7.2: Abstract configuration of a self-managing application

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.

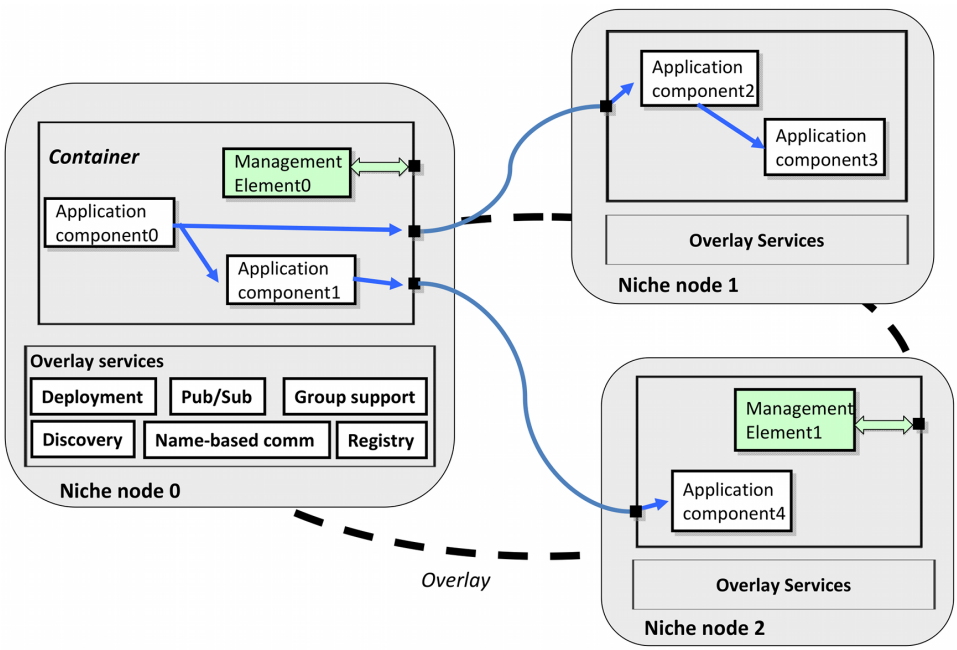


Figure 7.3: Niche architecture

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.

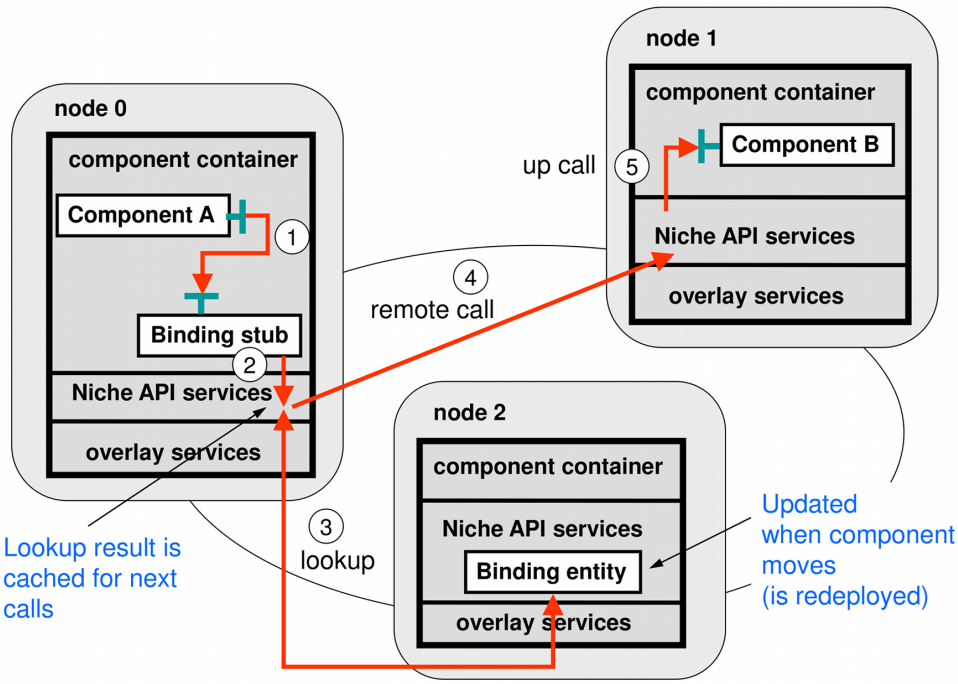


Figure 7.4: Steps of method invocation in Niche

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.

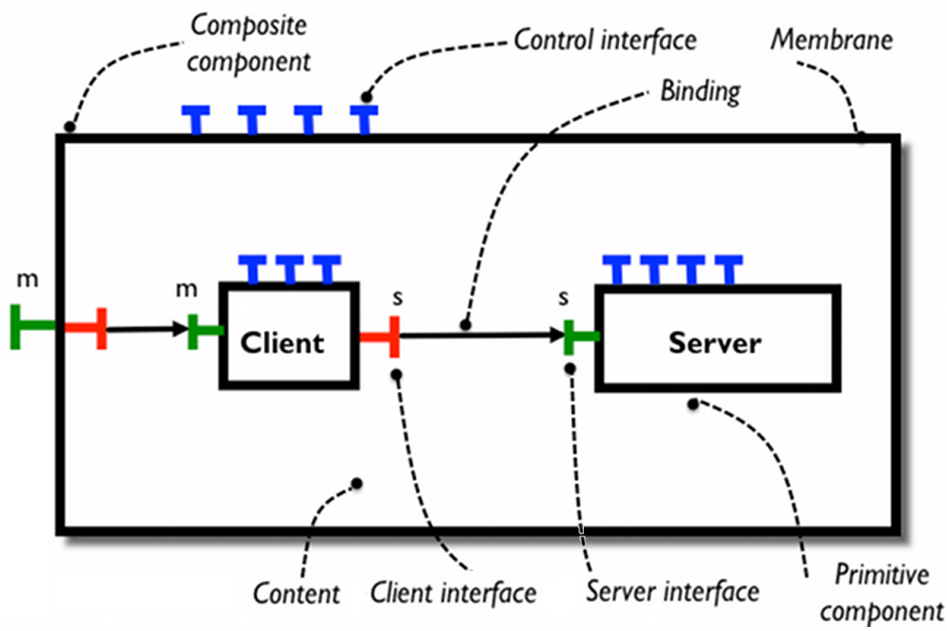


Figure 7.5: A composite Fractal component HelloWorld with two sub-components client and server

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-

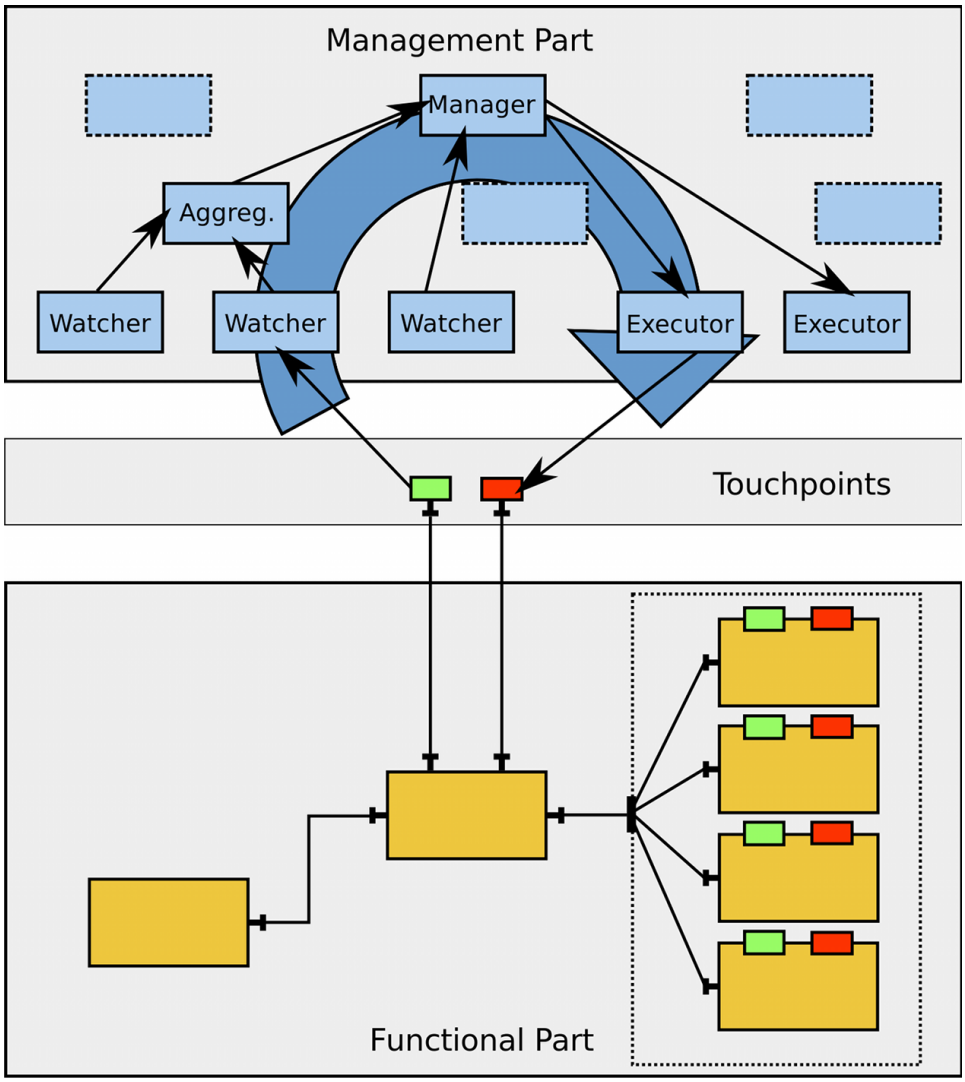


Figure 7.6: Hierarchy of management elements in a Niche application

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"/>
```

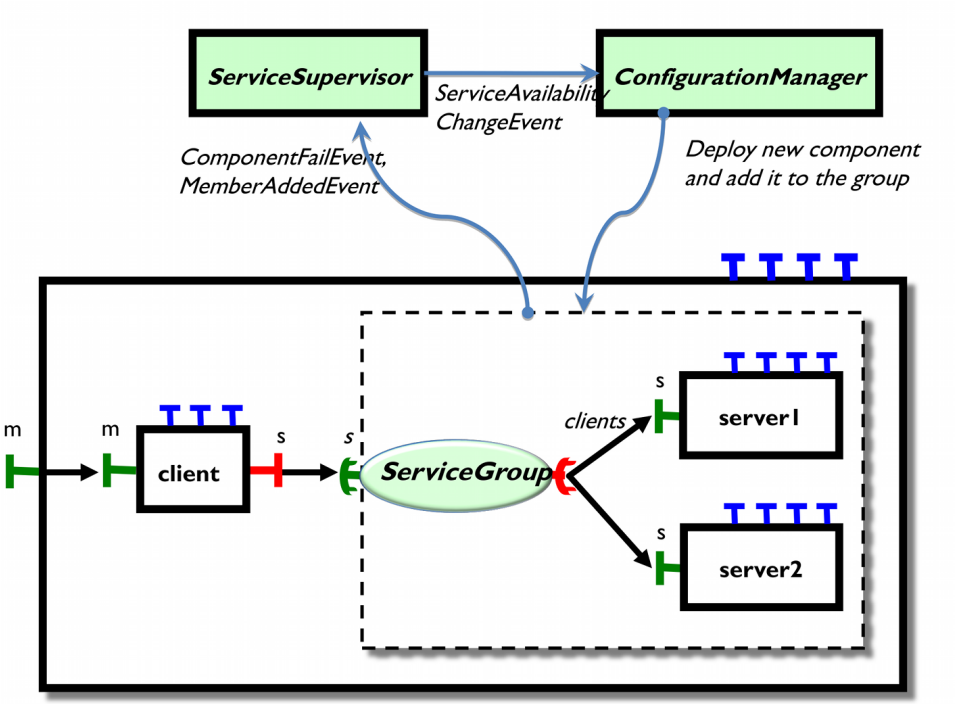


Figure 7.7: HelloGroup application

```
  <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];
```

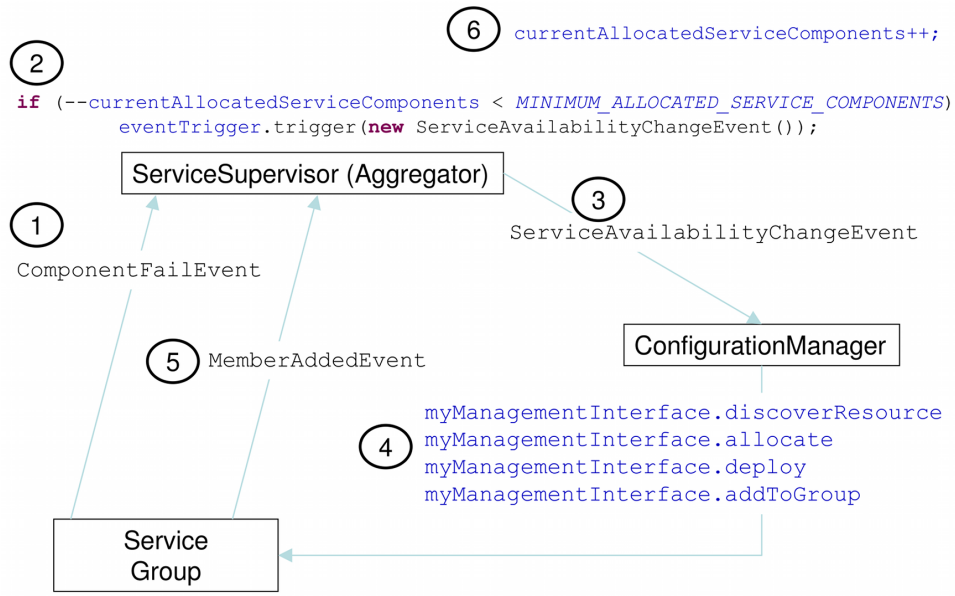


Figure 7.8: Events and actions in the self-healing loop of the HelloGroup application

```
        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).

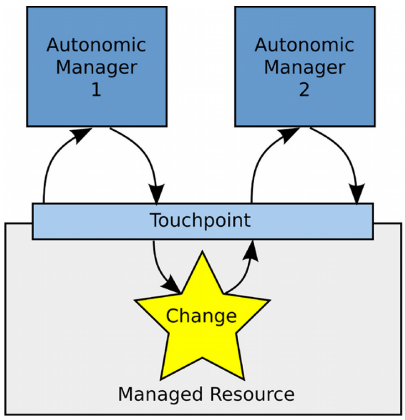
- **Orchestration:** Although autonomic managers can be designed independently, multiple autonomic managers, in the general case, are not independent since they manage the same system and there exist dependencies between management tasks. Therefore they need to interact and coordinate their actions in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

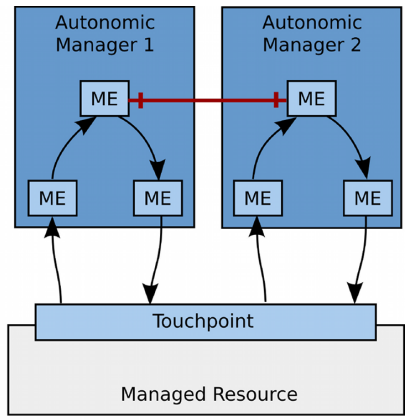
In this section, our major focus is on the orchestration of autonomic managers as the most challenging and less studied problem. The actions and objectives of the other stages are more related to classical issues in distributed systems such as partitioning and separation of concerns, and optimal placement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

Orchestrating Autonomic Managers

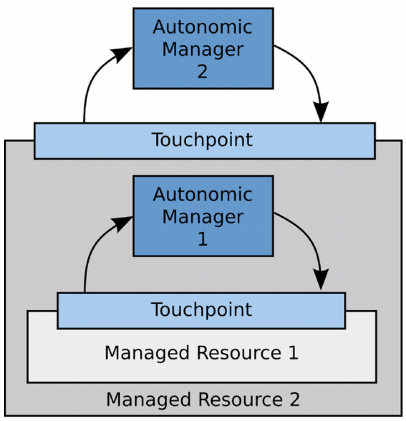
Autonomic managers can interact and coordinate their operation in the following four ways as discussed below and illustrated in Figure 7.9: indirect interactions via the managed system (stigmergy); hierarchical interaction (through touch points); direct interaction (via direct bindings); sharing of management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



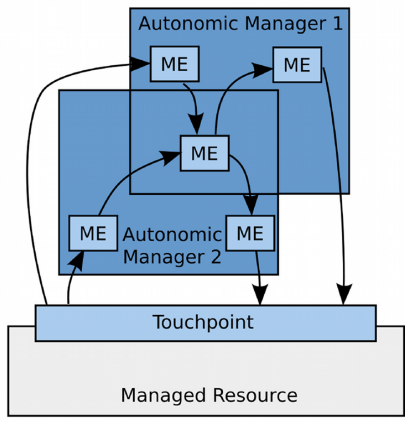
a. The stigmergy effect.



b. Direct interaction.



c. Hierarchical management.



d. Shared 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.

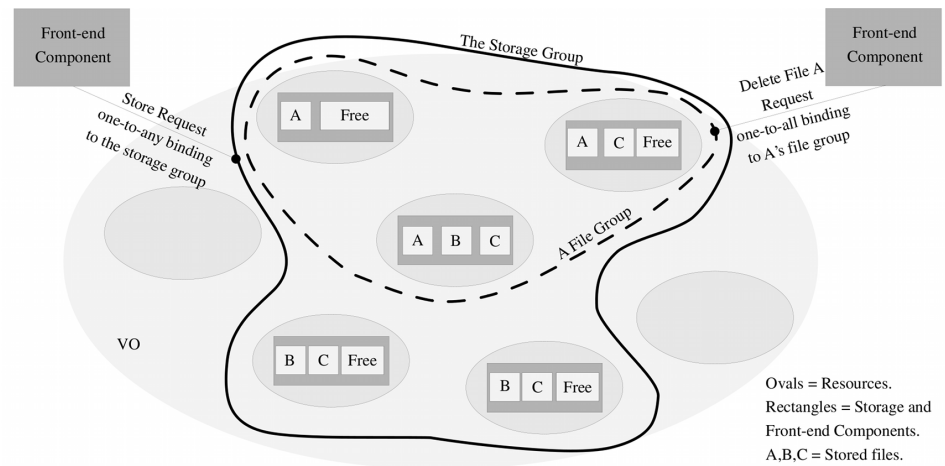


Figure 7.10: YASS functional design

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).

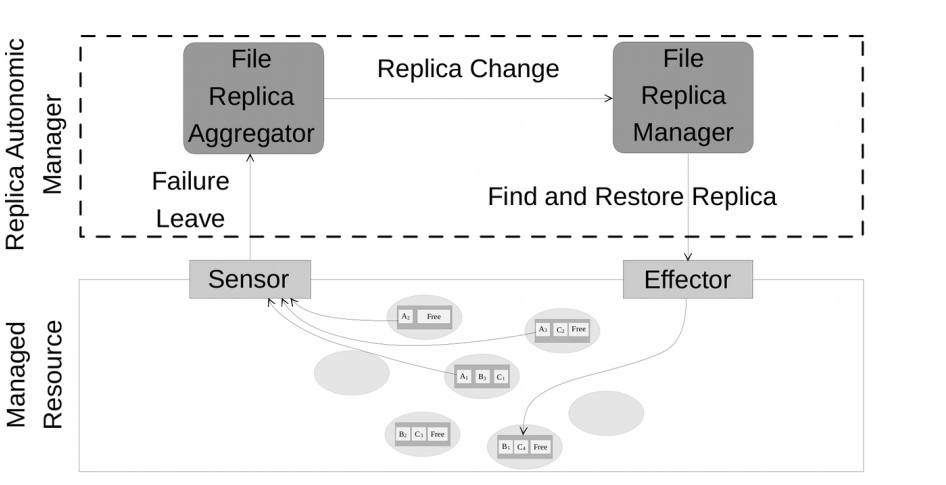


Figure 7.11: Self-healing control loop for restoring file replicas.

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.

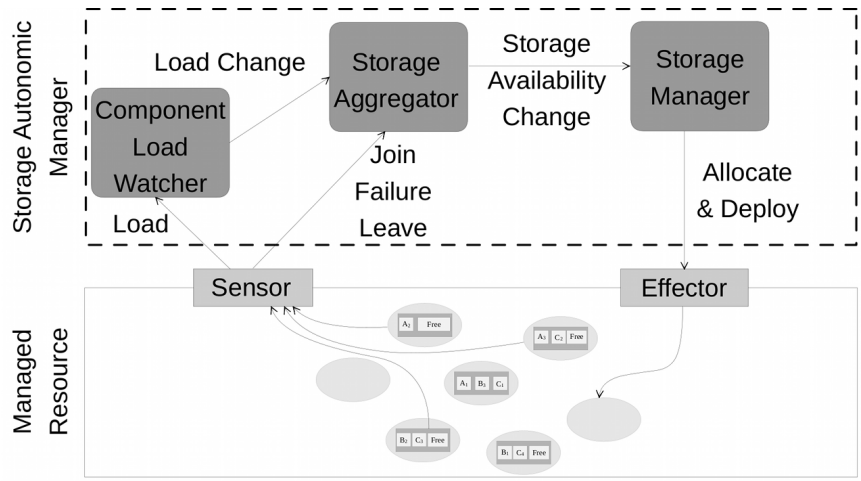


Figure 7.12: Self-configuration control loop for adding storage

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.

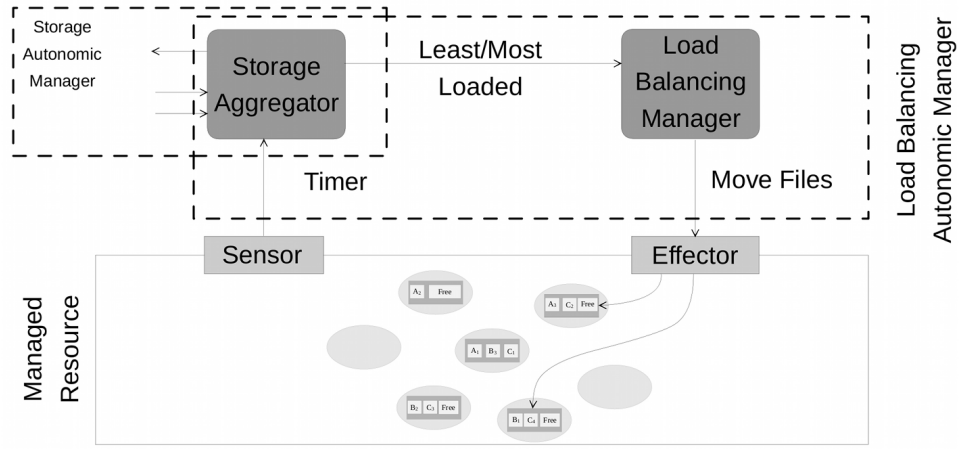


Figure 7.14: Sharing of management elements used to implement the self-optimization control loop for load balancing

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.

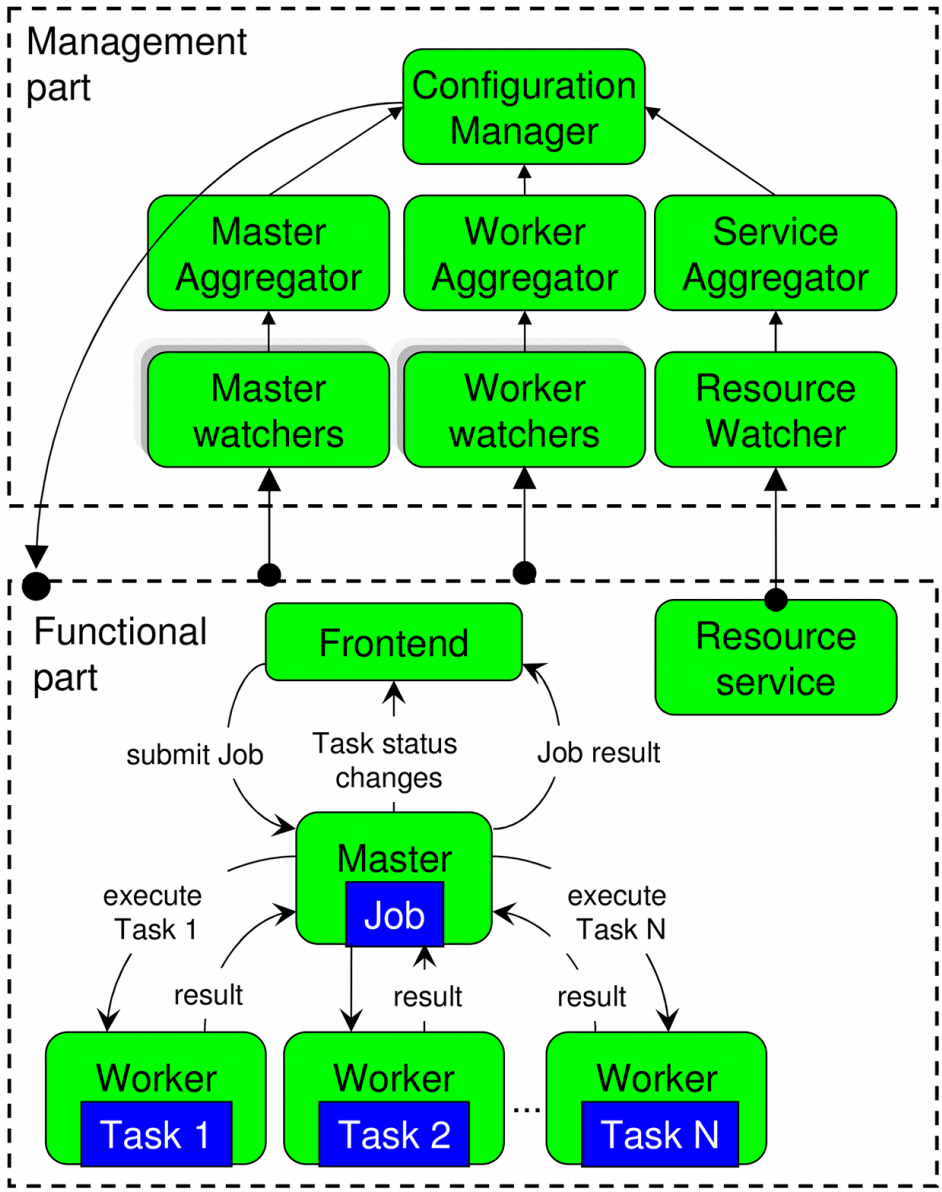


Figure 7.15: Architecture of YACS (yet another computing service)

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.

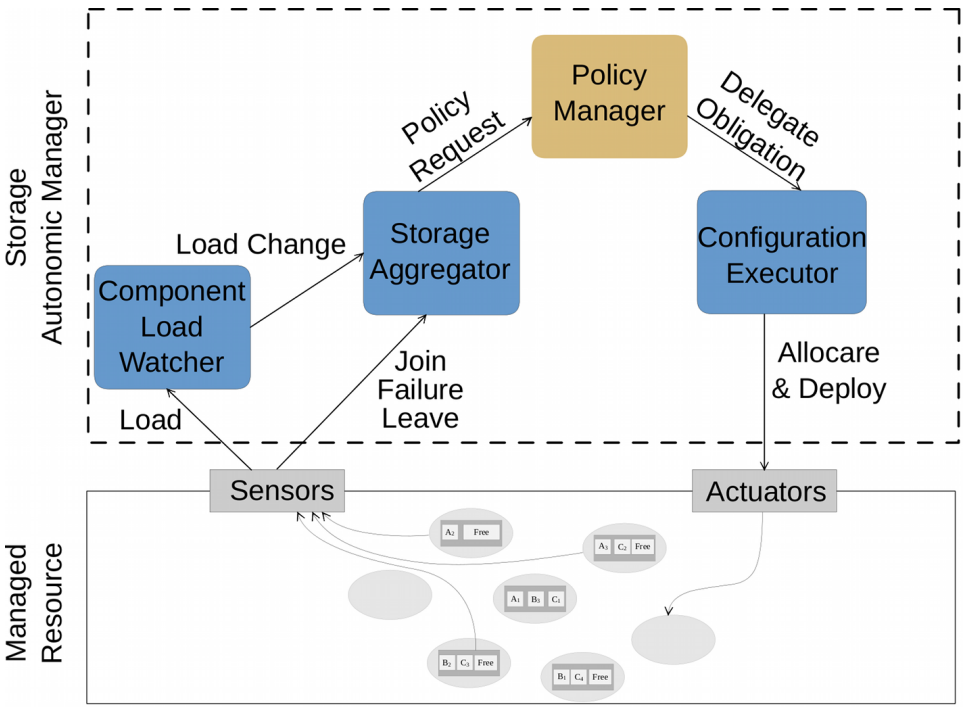


Figure 7.16: YASS self-configuration using policies

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.