# Niche Programming Guide

(SICS, KTH, INRIA)

July 8, 2009

**Abstract**

Deploying and managing distributed applications in dynamic Grid environments requires a high degree of autonomous management. Programming autonomous management in turn requires programming environment support and higher level abstractions to become feasible. We present Niche, which is a Distributed Component Management System (DCMS) that provides an API for programming self-managing component-based distributed applications. Application's functional and self-management code are programmed separately. The framework extends the Fractal component model by the component group abstraction and one-to-any and one-to-all communication patterns. Niche can automatically move application components responsible for self-management when necessary due to resource churn. The framework supports a network-transparent view of system architecture simplifying designing application self-* code. The framework provides a concise and expressive API for self-* code. Programming application self-* behaviours with Niche requires just a few dozens lines per application component. The implementation of the framework relies on scalability and robustness of the Niche structured p2p overlay network. We have also developed a distributed file storage service to illustrate and evaluate our framework. We discuss the current status of the Niche implementation and outline future work.

# Contents

# List of Figures

# Chapter 1

# Introduction

Niche is a distributed component management system (DCMS), which used to develop, deploy and execute a self-managing distributed application or service on a dynamic overlay network of the Democratic Grid.

Niche includes a component-based programming model with a set of APIs for development of self-managing applications and services, and a run-time execution environment for deployment and execution of the services. The run-time environment provides a set of overlay services, which mostly deal with connectivity of Niche resources and elements, e.g. nodes, components, component groups.

Niche programming model and API separate functional and non-functional (management) parts of the application. The management part includes self-* code that monitors and manages the functional part. Application components, component groups and bindings, are first-class entities in Niche that can be monitored and manipulated by the self-* code using an extensible actuation API provided by Niche. The actuation API defines a number of operations that can be performed to change architecture, configuration and operation of an application, e.g. deploy and (re)bind components, start/stop threads, move and replicate components, change component attributes.

The Niche programming model is partly based on the Fractal component model [7], in which components are bound and interact 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 membranes, introspection and intercession capabilities. This enables developing of manageable components. Niche supports location-transparent bindings and extends the original Fractal model with component groups that support one-to-any and one-to-many bindings.

Both, functional and management parts of an application can be pro-

grammed using the Niche component-based programming model and the corresponding API. However, elements of the management part (watchers, aggregators, managers) normally interact with each other using the event mechanism rather than the bindings used to interconnect functional components.

The management part of an application is constructed as a set of control loops each of which monitors some part of the application and reacts on predefined events (e.g. node failures, leaves or joins) and application-specific events (e.g. high load, low available storage capacity). The predefined events are fired by the run-time environment. When a control loop finds a symptom that requires some changes in the application, it plans and actuates necessary management actions using the Niche actuation API.

The control loops of the management part are built of distributed Management Elements, MEs, interacting through events delivered by the pub/-sub overlay service. MEs can be of three types: Watchers (Wi on Figure X), Aggregators (Aggr) and Managers (Mgr). A watcher monitors a part of the application, and can fire events when it finds some symptoms of the management concern. Aggregates are used to collect, filter and analyse events from watchers. An aggregator can be programmed to analyse symptoms and to issue change requests to managers. Managers do planning and execution of change requests. Niche programming environment provides basic MEs, publish/subscribe and actuation APIs. The Niche run-time environment includes a set of component containers that reside on a structured overlay network of VO computers; and a set of the overlay services (resource discovery, deployment, publish/subscribe, DHT).

The Niche run-time system allows initial deployment of a service or an application on the Democratic Grid. Niche relies on the underlying overlay services to discover and to allocate resources needed for deployment, and to deploy the application. The initial deployment code can be either manually written by the developer, or generated by Niche from the ADL description of the application architecture.

After/upon deployment, the management part of the application can monitor and react on changes in availability of resources by subscribing to resource events fired by Niche containers. In order to achieve robustness of the management part, MEs are transparently replicated in different Niche containers.

All elements of a Niche application - components, bindings, groups, management elements - are identified by unique identifiers (names) that enable location transparency. Niche uses the DHT functionality of the underlying structured overlay network for its lookup service. 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.

### Self-Managing Services Using Niche

In order to demonstrate Niche, we have developed two self-managing services:

**YASS** : Yet Another Storage Service;

**YACS** : Yet Another Computing Service.

The services can be deployed and provided on computers donated by users of the service or by a service provider. The services can operate even if computers join, leave or fail at any time. Each of the services has self-healing and self-configuration capabilities and can execute on a dynamic overlay network. Self-managing capabilities of services allows the users to minimize the human resources required for the service management. Each of services implements relatively simple self-management algorithms, which can be changed to be more sophisticated, while reusing existing monitoring and actuation code of the services.

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

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

# Chapter 2

# Component Programming Primer

This chapter aims to introduce the Fractal component model, to provide a basic primer on component programming, and to outline the main elements of the architecture description language (ADL).

## 2.1 The Fractal Component Model

Fractal is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications from operating systems and middleware to graphical user interfaces. The Fractal component model has the following main features:

- `hierarchical containment`: components can be nested at arbitrary levels (hence the "Fractal" name).
- `reflection`: components can be endowed with introspection and intercession capabilities.
- `sharing`: a given component instance can be included (or shared) by more than one component.
- `openness`: Fractal does not impose predefined semantics for reflective behaviour, component containment, and component binding.

Fractal components are runtime entities that communicate exclusively through well-defined access points, called *interfaces*. Interfaces can be divided into two kinds: *client interfaces* that emit operation invocations and *server interfaces* that receive them. Interfaces are connected through communication paths, called *bindings*. Fractal distinguishes primitive components from composite components, formed by hierarchically assembling other

components. Each Fractal component is made of two parts: the *membrane*, which embodies control behaviour, and the *content*, which consists of a finite set of sub-components. The membrane is composed of several *controllers*, which can take arbitrary (including user-defined) forms. Nevertheless, Fractal does define a useful set of four basic controllers. The *attribute controller* supports configuring component properties. The *binding controller* supports binding and unbinding client interfaces to server interfaces. The *content controller* supports listing, adding, and removing sub-components. The *life-cycle controller* supports starting and stopping the execution of a component. Finally, Fractal provides an *architecture description language* (ADL) for specifying configurations comprising components, their composition relationships, and their bindings.

Fractal is an ObjectWeb project, and further details are available at: `http://fractal.objectweb.org`

## 2.2 Implementing Fractal Components in Java

This section explains how to program Fractal components in Java. The example used is the HelloWorld example, found (more or less adapted) in all Fractal distributions. This example is a very simple application made of two primitive components inside a composite component (see the figure below).

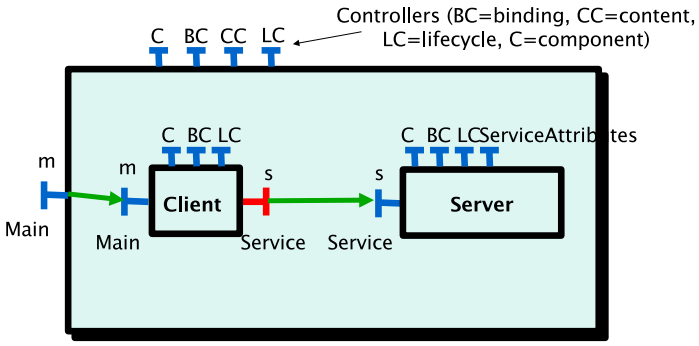


Figure 2.1: HelloWorld Fractal application.

The first primitive component is a "server" component that provides an interface to print messages on the console. It can be parameterized with two attributes: a "header" attribute to configure the header printed in front of each message, and a "count" attribute to configure the number of times each message should be printed. The other primitive component is a "client" component that uses the previous component to print some messages.

The server component provides a server interface named "s" of type Service, which contains a print method. It also has an AttributeController interface of type ServiceAttributes, which contains four methods to get and set the two attributes of the server component.

The client component provides a server interface named "m" of type Main, which contains a main method, called when the application is launched. It also has a client interface named "s" of type Service.

The composite component provides a server interface "m" that exports the corresponding interface of the client component.

The application can be programmed in two steps, by creating the component interfaces first, and then implementing these interfaces in the component classes.

### 2.2.1 Creating the component interfaces

Three interfaces must be implemented: the two "functional" interfaces Service and Main, and the attribute controller interface ServiceAttributes.

The functional interfaces are programmed "normally", i.e., the Fractal model does not impose any constraints on the Fractal functional component interfaces, except the fact that they must be public. The Service and Main interfaces are therefore very simple to implement:

```
public interface Service {
  void print (String msg);
}

public interface Main {
  void main (String[] args);
}
```

On the other hand, the attribute controller interfaces must extend the AttributeController interface, and must only have getter and setter method pairs (and they must be public too). The ServiceAttributes interface is therefore the following:

```
public interface ServiceAttributes extends AttributeController {
  String getHeader ();
  void setHeader (String header);
  int getCount ();
  void setCount (int count);
}
```

### 2.2.2 Implementing the component classes

The component classes must implement the previous interfaces, as well as some Fractal control interfaces.

The server component class, called ServerImpl, must implement the Service and ServiceAttributes interfaces. It may also implement the LifeCycleController interface, in order to be notified when it is started and stopped, but this is not mandatory. Since the server component does not have client interfaces, the ServerImpl class does not need to implement the BindingController interface, whose role is to manage the bindings involving the client interfaces of a given component. The ServerImpl class is therefore the following:

```
public class ServerImpl implements Service, ServiceAttributes {

  private String header = "";

  private int count = 0;

  public void print (final String msg) {
    for (int i = 0; i < count; ++i) {
      System.err.println(header + msg);
    }
  }

  public String getHeader () {
    return header;
  }

  public void setHeader (final String header) {
    this.header = header;
  }

  public int getCount () {
    return count;
  }

  public void setCount (final int count) {
    this.count = count;
  }
}
```

The client component class, called ClientImpl, must implement the Main interface. Since the client component has client interfaces, the class must also implement the BindingController interface, a basic Fractal control interface, presented next:

```
public interface BindingController {

  // Returns the names of the client interfaces of the component
```

```
  String[] listFc ();

  // Returns the interface to which the given client interface is bound
  Object lookupFc (String clientItfName) throws NoSuchInterfaceException;

  // Binds the client interface to the server interface
  void bindFc (String clientItfName, Object serverItf)

  // Unbinds the client interface
  void unbindFc (String clientItfName);
}
```

The ClientImpl class is therefore the following:

```
public class ClientImpl implements Main, BindingController {

  private Service service;

  public void main (final String[] args) {
    service.print("hello world");
  }

  public String[] listFc () {
    return new String[] { "s" };
  }

  public Object lookupFc (final String cItf) {
    if (cItf.equals("s")) {
      return service;
    }
    return null;
  }

  public void bindFc (final String cItf, final Object sItf) {
    if (cItf.equals("s")) {
      service = (Service)sItf;
    }
  }

  public void unbindFc (final String cItf) {
    if (cItf.equals("s")) {
      service = null;
    }
  }
}
```

## 2.3 Deploying Fractal applications

The simplest method to deploy an application is to describe the architecture of the application in the Architecture Description Language (ADL), and to pass this description to the deployment service. The HelloWorld application can be described as follows:

```
<definition name="helloworld.HelloWorld">

  <!-- Interface of the composite component-->
  <!-- NB. myhelloworld.* are the fully-qualified names of interfaces/classes
  <interface name="m" role="server" signature="myhelloworld.Main"/>

  <!-- The client sub-component -->
  <component name="client">
    <interface name="m" role="server" signature="myhelloworld.Main"/>
    <interface name="s" role="client" signature="myhelloworld.Service"/>

    <!-- Implementation of the client component -->
    <content class="myhelloworld.ClientImpl"/>
  </component>

  <!-- The server sub-component -->
  <component name="server">
    <interface name="s" role="server" signature="myhelloworld.Service"/>

    <!-- Implementation of the server component -->
    <content class="myhelloworld.ServerImpl"/>

    <!-- Attributes of the server component -->
    <attributes signature="myhelloworld.ServiceAttributes">
      <attribute name="header" value="-> "/>
      <attribute name="count" value="1"/>
    </attributes>

  </component>

  <!-- The binding between the composite and client -->
  <binding client="this.m" server="client.m" />

  <!-- The binding between client and server -->
  <binding client="client.s" server="server.s" />

</definition>
```

The HelloWorld example demonstrates the main concepts of the standard Fractal ADL; namely, component definitions, components, interfaces,

bindings, and attributes. Full details on the language can be found at `http://fractal.objectweb.org/fractaladl/`.

## 2.4 Extensions of standard Fractal ADL

Fractal ADL (Architecture Description Language) is an open and extensible language to define component architectures. More precisely, the language is made of a set of modules, each module defining an abstract syntax for a given architectural concern (e.g., component containment). The Fractal implementation includes a standard set of modules that allow describing component types, component implementations, component hierarchies and component bindings. The implementation then allows adding new modules for new architectural concerns and provides a modular toolset for processing the language extensions.

In the context of DCMS, we have extended the standard ADL in the following three areas: deployment, binding, and self-management.

### 2.4.1 Deployment extensions

The following ADL extract demonstrates the deployment extensions. It refines the client description in the HellWorld example with two new elements: packages and virtual nodes.

```
 ...
<component name="client">
    <interface name="m" role="server" signature="myhelloworld.Main"/>
    <interface name="s" role="client" signature="myhelloworld.Service"/>
    <content class="myhelloworld.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>
 ...
```

The *packages* element provides information about the software packages necessary for creating run-time components. Packaging currently relies on the OSGI standard, and packages are identified with a unique name in the OSGI bundle respository. The previous ADL extract states that the implementation of the client component (i.e., the class ClientImpl) is part of the OSGI bundle named "ClientPackage v1.3".

The *virtual node* element provides information about the placement of a component. Virtual nodes are logical component containers, automatically mapped to concrete nodes at deployment time. The mapping is based on the

associated resource requirements (*ResourceReqs* attribute) and the actual deployment environment. For example, the previous ADL extract states that the client should be deployed on a node with memory larger than 1 GB and CPU speed larger than 1Ghz. Resource requirements are expressed in the LDAP filter syntax.

### 2.4.2 Binding extensions

These extensions enable the ADL to represent one-to-any and one-to-all bindings. In the following example, a client component is connected to a group of two server components (server1, server2) using one-to-any invocation semantics (see figure).

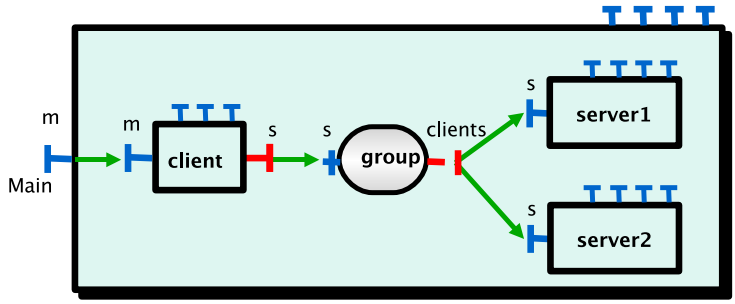


Figure 2.2: Application using group communication

This example can be described in ADL as follows:

```
<definition name="MyHelloGroup">

  <interface name="r" role="server" signature="myhelloworld.Main"/>

  <component name="client">
    <interface name="r" role="server" signature="myhelloworld.Main"/>
    <interface name="s" role="client" signature="myhelloworld.Service"/>
    <content class="myhelloworld.ClientImpl"/>
  </component>

   <component name="group">
    <interface name="s" role="server" signature="myhelloworld.Service"/>
    <interface name="clients" role="client" signature="myhelloworld.Service"
              cardinality="collection"/>
    <content class="GROUP"/>
  </component>
```

```
  <component name="server1">
    <interface name="s" role="server" signature="myhelloworld.Service"/>
    <content class="myhelloworld.ServerImpl"/>
  </component>

  <component name="server2">
    <interface name="s" role="server" signature="myhelloworld.Service"/>
    <content class="myhelloworld.ServerImpl"/>
  </component>

  <binding client="this.r" server="client.r" />
  <binding client="client.s" server="group1.s" bindingType="groupAny"/>
  <binding client="group1.clients1" server="server1.s"/>
  <binding client="group1.clients2" server="server2.s"/>

</definition>
```

As seen in this description, representing one-to-any bindings involves using 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 "collection" value indicates that the group has an arbitrary number of client interfaces). Invoking the group involves creating a binding to its server interface. One-to-any and one-to-all invocation semantics are represented by setting the *bindingType* attribute to "groupAny" or "groupAll".

### 2.4.3 Self-management extensions

The ADL extract below demonstrates the current self-management extensions. It refines the previous group communication example to add a management component.

```
<definition name="MyHelloGroup">

  <interface name="r" role="server" signature="myhelloworld.Main"/>

  <component name="mgr" definition="org.objectweb.jasmine.jade.ManagementType">
    <content class="org.objectweb.jasmine.jade.examples.managerimpl.ManagerImpl"/>
  </component>

  <!-- As above ... -->
  ...

</definition>
```

Management components have a predefined definition "Management-Type". This definition contains a set of client interfaces that correspond

to the DCMS API (see Chapter 4) that management components require for their operation. These interfaces are implicitly bound by the system after management components are instantiated. In the current prototype, the management components capture only the initial deployment and configuration of the self-management behaviour. In other words, the component implementation (e.g., the ManagerImpl class) contains the code for creating, configuring, and activating the set of DCMS management elements (MEs). More specifically, this code is located in the implementation of the LifeCycleController interface (start operation) of the manager. A future version will enable describing in ADL the deployment and configuration of individual MEs.

# Chapter 3

# Self-Managing Applications with Niche

The purpose of this chapter is to gradually introduce Niche – the Distributed Component Management System (DCMS) and its concepts without the burden of full details of its API and current limitations of the prototype. Information presented in this chapter should suffice to understand the formal API description in Chapter 4, the YASS example in Chapter 5, and the discussion of features, limitations and future extensions of the current prototype in Chapters 6 and 7. The presentation in this Chapter is informal, and particular syntax of examples can stray from the existing Niche and YASS code for the sake of presentation clarity.

## 3.1 Functional Code and Self-Management

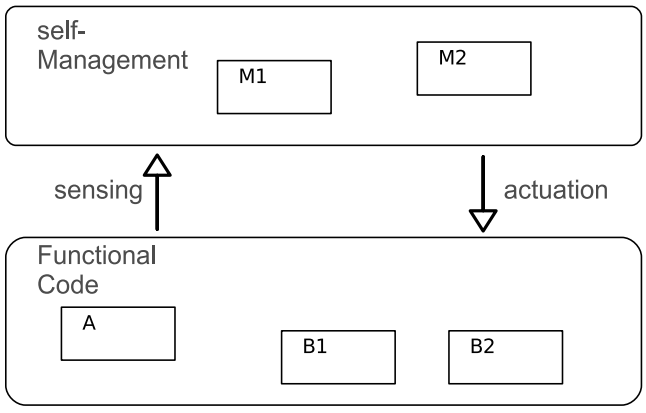


Figure 3.1: Application Architecture with Niche.

Niche is designed to support the development of self-managing behaviors of distributed applications. An application in the Niche framework contains a component-based implementation of the application's functional specification (the lower part of figure 3.1, with components `A`, `B1` and `B2`). During the development of this part of the application – we refer to it as the *functional code* thereafter – designers focus on algorithms, data structures and architectural patterns that fit the application's functional specification. The functional code can fulfill its purpose under a certain range of conditions in the environment such as availability of resources, user load and stability of computers hosting application components.

When the environment changes beyond the assumptions of the functional code, for instance when the user load becomes too high or an important application component fails, the application should either self-heal, self-configure, self-optimize or self-protect. These behaviors are commonly known as *self-* behaviors*, or *self-management*. The component-based implementation of application's self-* behaviors *senses* changes in the environment and adjusts the application architecture accordingly (the upper part of figure 3.1, with components `M1` and `M2`).

Niche provides APIs that allow the application developer to program deployment and management of application components, their interconnection, and also sensing state changes in environment and application components. The APIs provide *network-transparent* services, which means that the effects of an API method invocation are the same regardless where on the network the invocation took place. Niche implements a run-time infrastructure that aggregates computing resources on the network used to host and execute application components, which we discuss in more detail in Section 3.15.

## 3.2 Component-Based Functional Code

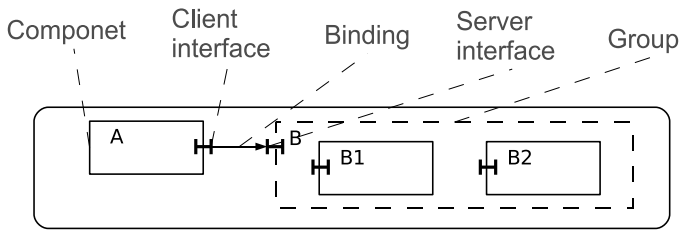


Figure 3.2: Functional Code in a Niche-based Application.

The functional part of application architectures is composed using *components* and *bindings*, see Figure 3.2, which are introduced in the Fractal component model and discussed in detail in Section 2.1. A Fractal com-

ponent contains code or sub-components, and its functionality is accessed through *server interfaces* which separate the component's implementation from other components using the component's functionality. A component's *client interface* manifests external services the component rely upon. A binding interconnects a client interfaces of one component with a server interface of another component. The Niche programming model introduces also *groups*. A Niche group represents a set of similar components and allows the designer to use them as it were one singe component. Niche groups can be exploited to improve application scalability and robustness, as discussed in detail in Section 3.12.

## 3.3 Application Self-Management by Control Loops

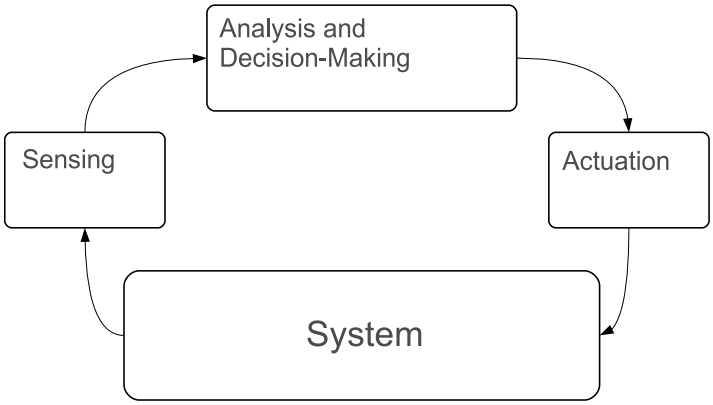


Figure 3.3: Management Control Loops.

*Self-** behaviours in Niche-based applications are implemented by *management control loops* we call just *control loops* when it is clear from the context. Control loops go through the *sensing*, *analysis and decision making*, and finally the *actuation* stages (see figure 3.3). The sensing stage obtains information about changes in the environment and components' state. The analysis and decision-making stage processes this information and decides on necessary actions, if any needed, to adjust the application architecture to the new conditions. During the actuation stage the application architecture is updated according to the decisions of the analysis and decision-making stage.

In a particular application there can be multiple control loops each controlling different kinds of application's self-* behaviors. These loops need in general to coordinate with each other in order to maintain the application's architecture consistently, which we address in Section 3.5.

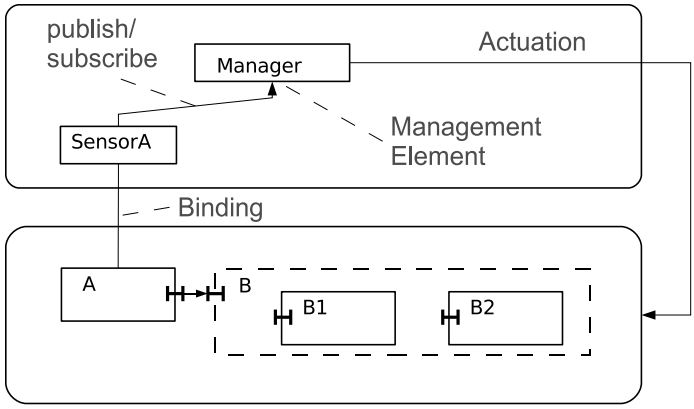


Figure 3.4: Self-Management in a Niche-based Application.

The sensing stages of Niche-based application's control loops are implmented by *sensors*. There are two types of sensors – application-specific sensors that provide information about status of individual components in the application, and sensors provided by the Niche run-time system that deliver information about the environment, such as notifications about component failures. Application-specific sensors are specific to individual component types, and implemented together with those components. At run-time, instances of application-specific sensor types can be dynamically deployed and attached to individual components. On Figure 3.4, `SensorA` is an application-specific sensor that provides necessary status information of component `A`.

The implementation of the analysis and decision-making stages of Niche-based application's control loops consists of *management elements* (MEs). MEs are stateful entities that process input *management events*, or just *events* thereafter (Section 3.8), according to their internal state, and can emit output events and/or manipulate the architecture using the Niche management *actuation* API implemented by Niche and introduced in this document. In general, in a Niche-based application there are multiple MEs that can be organized in different architectural patterns that are discussed in Section 3.5. On Figure 3.4, `Manager` is a management element.

Management events serve for communication between individual MEs and sensors that form application's management control loops. Management events are delivered asynchronously between MEs. MEs are *subscribed* to and receive input events from sensors and other MEs. Subscriptions can be thought of as bindings for unidirectional asynchronous communication for specific event types. Subscriptions are first-class entities that are explicitly manipulated using the Niche API, that is, the programmer designing the architecture of application's self-management explicitly creates subscriptions

between management elements to their sources of input events.

The Niche API provides functions for the actuation stage of application's management control loops. In particular, it provides for deployment of components of functional and self-management parts, and to interconnect those components.

## 3.4 Watchers, Aggregators and Managers

In a Niche-based application there can be multiple management loops implementing different kinds of self-* behaviors. Different loops should coordinate in order to ensure the coherency of architecture management, for which the coordination schemes discussed in Section 3.5 can be exploited. Individual management elements can participate in several control loops.

We distinguish the following roles of management elements that constitute the body of the analysis and decision-making stages of management control loops: *watchers*, *aggregators* and *managers*. In the simplest case, as on Figure 3.4, all roles can be performed by one single management element, but in a typical Niche application different roles are implemented by different MEs.

Watchers monitor the status of individual components and groups. Watchers are connected to and receive events from sensors that are either implemented by the programmer or provided by the management framework itself. Watchers provide also certain functionality that simplify programming of watching component groups, as discussed in Section 3.12. Watchers are intended to watch components of the same type, or components that are similar in some respect.

An aggregator is subscribed to several watchers and maintains partial information about the application status at a more coarse-grained level. There can be several different aggregators dealing with different types of information within the same control loop. Within an application as a whole there can be different aggregators acting in different management control loops.

Managers use the information received from different watchers and aggregators to decide on and actuate (execute) the changes in the architecture. Managers are meant to possess enough information about the status of the application's architecture as a whole in order to be able to maintain it. In this sense managers are different from watchers and aggregators where the information is though more detailed but limited to some parts, properties and/or aspects of the architecture. For example, in a data storage application a manager needs to know the current capacity, the design capacity and the number of online users in order to meet a decision whether additional storage elements should be allocated, while a storage capacity aggregator knows only the current capacity of the service, and different storage capac-

ity watchers monitor status and capacity of corresponding groups of storage elements.

## 3.5 Orchestration of Multiple Control Loops

In the Niche framework, application self-management can contain multiple management loops. The decisions made by different loops to change the architecture according to new conditions should be coordinated in order to maintain consistency of the architecture and to avoid its oscillation over time.

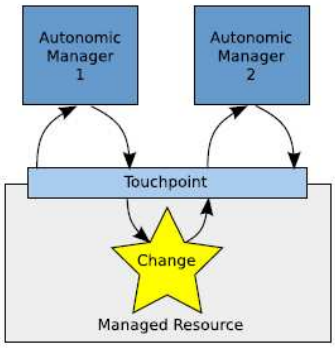


**Fig. 1.** The stigmergy effect.

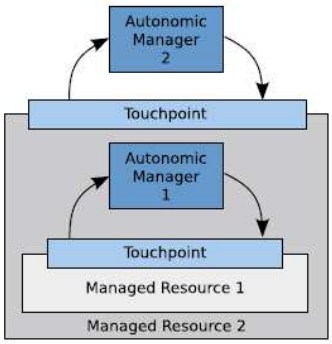


**Fig. 2.** Hierarchical management.

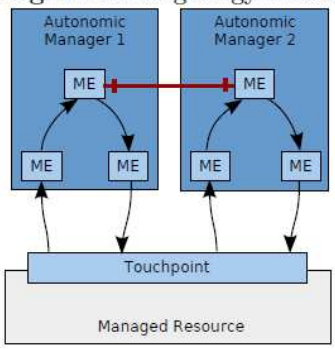


**Fig. 3.** Direct interaction.

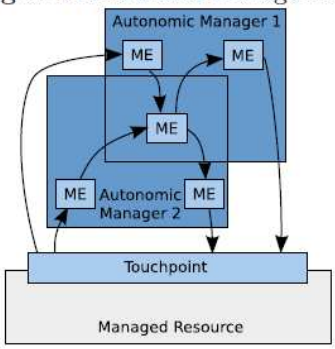


**Fig. 4.** Shared Management Elements.

Figure 3.5: Orchestration of Multiple Control Loops.

The following four methods can be used to coordinate the operation of several management control loops:

**Stigmergy** is a way of indirect communication and coordination between agents. 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 control loops and the environment is the managed application. See Fig. 1 on Figure 3.5.

**Hierarchical Management** imply that some control loops can monitor and control other autonomic control loops (Fig. 2). The lower level control loops are considered as a managed resource for the higher level control loops. Higher level control loops can sense and affect the lower level ones.

**Direct Interaction** can technically be achieved by subscribing or binding appropriate management elements (typically managers) from different control loops to one another (Fig. 3). Cross control loop bindings can be used to coordinate them and avoid undesired behaviors such as race conditions and oscillations.

**Shared Management Elements** is another way of communication and coordination of different control loops (Fig. 4). Shared MEs can be used to share state (knowledge) and to synchronize actions by different control loops.

## 3.6 Scalability and Fault-Tolerance of Control Loops

Management elements can form hierarchical structures that improve scalability of self-management. Hierarchical structures can also facilitate hiding unnecessary details from higher-level management elements, thus simplifying design and maintainability of self-management code. In particular, lower-level watchers and aggregators can hide fine-grained details about individual components present in the system from higher-level management elements, in particular managers.

Application designers can also improve scalability and fault-tolerance of application self-management by distributing the responsibility of managing the application architecture over a group of sibling managers. The virtual synchrony approach for building scalable and fault-tolerant distributed systems [3, 4] can be used to achieve this: managers in the group can be programmed to receive all input events from all aggregators and watchers and thus be aware of the state of all other sibling managers, but each individual manager would maintain only the part of the architecture that is assigned to it.

It might be possible to improve fault-tolerance of self-management by deploying identical control loops, with some necessary coordination using e.g. the models outlined in Section 3.5. Fault-tolerance of self-management can be also improved by replicating individual management elements, as discussed in Section 3.14.

## 3.7 Management Elements and Sensors with Niche

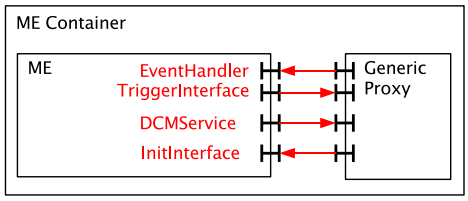


Figure 3.6: MEs in Niche.

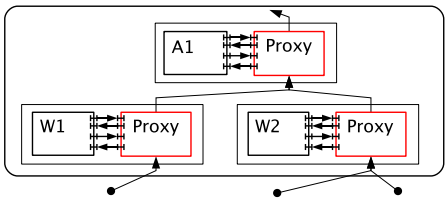


Figure 3.7: Composition of MEs.

Management Elements are programmed by the application developer as regular (centralized) Fractal components (Figure 3.6). MEs need to possess certain client and server interfaces, as explained below. Niche infrastructure provides for ME deployment and inter-ME communication; application developers do not need to explicitly program either of it. Note that MEs can be watched by watchers exactly as components implementing the application's functional specification.

The Niche run-time system hosts each ME in an *ME container* (Figure 3.6). ME containers are elements of the Niche infrastructure that enable operation of application-specific MEs. Specifically, an ME container provides an instance of the application-independent component called *ME Generic Proxy* with interfaces matching the interfaces of the ME.

The concepts of ME container and generic proxies is a part of the Niche implementation model; we present it here solely in order to provide some intuition behind Niche operation concerning MEs. ME containers are in particular important for reliable self-* behaviors achieved by means of replication of management elements, as discussed in Section 3.14.

In our Java-based Niche prototype, MEs are Fractal components implemented as Java classes according to the Java implementation of the Fractal framework. MEs can manipulate the application architecture using the Niche Java-based API provided through server interfaces of ME generic proxies. Niche infrastructure and application-specific ME components interact using certain data structures that identify elements of the application architecture, as discussed in Section 3.9. ME generic proxies provide for communication between MEs, see Figure 3.7. When a ME is deployed, Niche finds a suitable computer among those interconnected by Niche, creates the ME generic proxy and connects application-specific ME component to the proxy. Hosting and executing MEs is accounted to Niche processes executed on individual physical nodes. For example, in a Grid environment the members of a Virtual Organization (VO) would execute Niche processes on their computers. Note that components implementing the application's functional specification are hosted on first-class resources managed by dedicated resource management services, as discussed in Section 3.11. Niche

attempts to evenly balance the load of ME hosting.

Application-specific ME components can have the following client interfaces (see also figure 3.6):

- `TriggerInterface` interface with the `trigger` method used to emit events generated by the management element
- `DCMService` interface that provides Niche API for controlling functional and non-functional application components

Application-specific ME components need to provide the following server interfaces:

- `EventHandler` interface with the `eventHandler` method used when a management event arrives to the ME
- `InitInterface` interface used to (re)configure the management elements

The ME components can have further client and server interfaces which can be bound to functional and management components in the application, under certain restrictions discussed in Niche documentation.

Watchers receive information about status of components by means of component-specific sensors. Sensors are Fractal components. Individual types of sensors are designed to be bound to and receive status information from specific types of components in the application. Application developer designs sensors together with components they can sense. Sensor functionality is not integrated directly into the components because different sensors can be attached to the same component in different situations and at different points in time. Instead, components that need to be sensed implement a minimal interface that can provide enough information to sensors whenever needed. This facility in a component should not draw computing resources when not in use. When an application-specific sensor is deployed, it resides on the same node and interacts with the component through primitive Fractal bindings. In figure 3.8, sensor `Sensor A` is deployed for component `A`, and two instances of `Sensor B` are deployed for each of the components `B1` and `B2` from a group.

Sensors are deployed as a two-part structure similar to MEs, see figure 3.9. Sensors and `Sensor Generic Proxy` components provided by Niche interact through the following two interfaces. Application-specific sensor components can use the following client interface:

- `TriggerInterface` interface with the `trigger` method used to emit new events

and need to provide the following server interfaces :

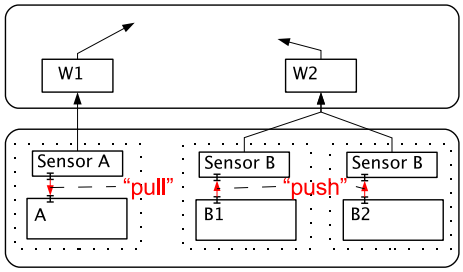


Figure 3.8: Structure of Application-Specific Sensors.

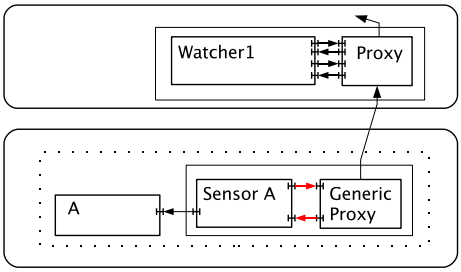


Figure 3.9: Composition of Application-Specific Sensors.

- `SensorInterface` interface used to control sensors

When a sensor is deployed, Niche locates the component for which the sensor is to be deployed, deploys both parts of the sensor appropriately and interconnects them (see figure 3.9). Using the facilities of the Fractal component model [7], application developer also specifies two lists of interfaces – for information "pull" and "push" between the sensor and the component being sensed (see figure 3.8), and Niche uses this information to connect the named application-specific sensor component interfaces to matching interfaces of the component being sensed.

## 3.8 Management Events

Sensors and management elements communicate asynchronously by means of *management events*, or just *events* thereafter. Events are objects of corresponding management event classes that are defined either by Niche itself, or are application-specific. When a subscription between a pair of sensors or MEs is being created, the subscription's management event type is identified by the event's class name.

Niche-specific events are generated by sensors implemented by Niche. These include in particular the `ComponentFailEvent` that identifies a failed component.

Applications define their own management event classes and generate corresponsing management events. In our current Java-based Niche prototype, events classes must be serializable. The Niche run-time system delivers events according to the established subscriptions.

## 3.9 Architecture Representation in Self-Management Code

Elements of the application architecture – components, bindings, groups and MEs – are identified by unique identifiers we call *Niche Id:s*, or just *Id:s* when it is clear from the context. Identifiers are unique in the scope of a Niche run-time infrastructure. For example, in a Grid environment the members of a Virtual Organization (VO) will usually run together a single instance of the Niche infrastructure, and different VOs will have separate infrastructures. MEs receive information about status of application architecture elements and manipulates them using the Id:s. Id:s of architecture elements are network-transparent, which allows application developers to design application architecture and its self-* behaviours independently of particular application deployment configurations. In our Java-based prototype of Niche, Id:s are represented in self-* code as Java objects of certain type. We discuss the implementation and performance characteristics of our Niche prototype in Section 3.15.

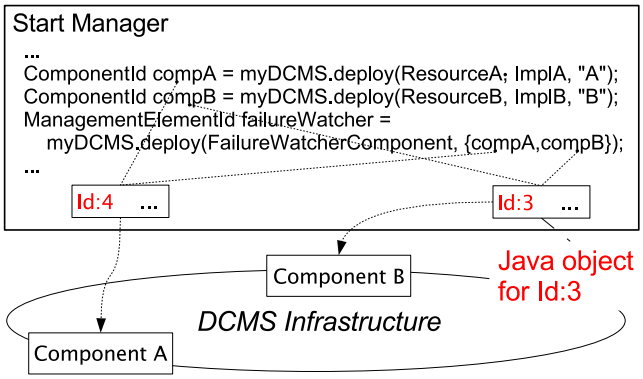


Figure 3.10: Application Architecture Representation in Self-* Code.

In figure 3.9 a snippet of self-* code from a ME called `StartManager` is presented. The code deploys two components and a "failure watcher" that oversees them. Note that this type of code fragments can be generated automatically by high-level tools, in particular by the Grid4All Application Deployment service. There are two components named `A` and `B` represented in self-* code by `compA` and `compB`, respectively. Id:s are introduced in self-* code by Niche API calls that deploy functional components and MEs. In figure 3.9, `compA` and `compB` are results of the `deploy` API calls that deploy components `A` and `B` implemented by Java classes `ImplA` and `ImplB` on resources `ResourceA` and `ResourceB`, respectively. Resource management is discussed in Section 3.11. Id:s are passed to Niche API invocations when

operations are to be performed on corresponding architecture elements, like deallocating a component. In the example, `compA` and `compB` are passed to the `deploy` Niche API call that deploys a watcher implemented by the Java class `FailureWatcherComponent`. `FailureWatcherComponent` watchers expects to receive Id:s of components to watch upon initialization.

Note that Id:s are *network-transparent*: multiple MEs on different nodes can access and manipulate the same architecture element by means of the element's Id. Niche API operations on Id:s have the same effect regardless of the location of the nodes with MEs issuing the operations, and the location of architecture elements identified by Id:s. In the example In figure 3.9, the `failureWatcher` ME will in general be deployed on a different physical node from the one where the `StartManager` is deployed itself, yet both MEs posses references to Niche Id Java objects representing `A` and `B`. Niche Id:s can also be included in application-specific management events passed between MEs, and thus used by the recipient MEs. Different physical nodes necessarily have different Java objects representing the same Niche Id, as discussed in Section 3.15.

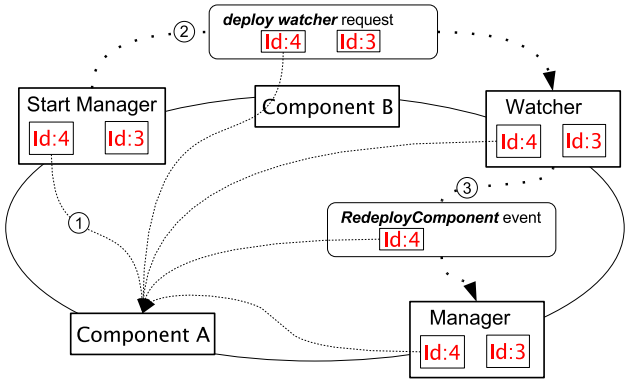


Figure 3.11: Sharing Niche Id:s between distributed MEs.

A possible sequence of actions and events is illustrated on Figure3.9. First, designated by (1), components `A` and `B` are deployed by the `StartManager`. Next (2), `StartManager` issues a request to deploy the `Watcher` ME that upon initialization obtains Id:s of `A` and `B`. Eventually (3), the watcher senses the failure of `A` (using Niche sensing - not illustrated on the Figure), and generates a `RedeployComponent` event that identifies `A` and is delivered to the `Manager` ME (the subscription is established beforehand by e.g. the same `StartManager`). At this point, `Manager` can update its architecture representation and/or perform management actions, like redeploying `A`.

We continue illustrating the manipulation of the application architecture in Section 3.10 using the code for initial deployment as an example.

## 3.10 Initial Deployment of Applications

```
DCMService myDCMS;
ComponentId compA = myDCMS.deploy(ResourceA, ImplA, "A");
ComponentId compB = myDCMS.deploy(ResourceB, ImplB, "B");
myDCMS.bind(compA, "clientInterfaceA",
            compB, "serverInterfaceB");
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA, compB});
ManagementElementId aggregator =
   myDCMS.deploy(AggregatorComponent, {compA, compB});
(void) myDCMS.subscribe(aggregator, manager, "status");
(void) myDCMS.subscribe(compA, aggregator, "componentFailure");
(void) myDCMS.subscribe(compB, aggregator, "componentFailure");
```

Figure 3.12: Example of Self-Management Code with Niche.

Initial deployment of applications is performed using the Niche API. In the case when the initial architecture of the application's functional part is specified using an Architecture Description Language (ADL) specification as discussed in Section 2.3, the application deployment service interprets the ADL specification and invokes corresponding Niche API functions. An example sequence of commands executed by Niche is shown in figure 3.12, In this example, `myDCMS` is an object that provides the Niche API. The first `deploy` method deploys a component `A` implemented by `ImplA` on a resource `ResourceA`. We discuss the resource management in Section 3.11. `deploy` invocations contain also symbolic names of components in the application architecture, `A` and `B` in our example – this information is necessary in order to connect the ADL specification of application's initial architecture with the application's self-* architecture, as discussed later in this Section.

Next, the client interface on component `A` named `clientInterfaceA` is bound to the server interface `serverInterfaceB` on component `B`:

```
myDCMS.bind(compA, "clientInterfaceA",
            compB, "serverInterfaceB");
```

Next, the management element `manager` is deployed using the following method:

```
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA, compB});
```

Here, the new manager `manager` will receive the list {`compA, compB`} as the argument for initialization. This argument is not interpreted by Niche itself. After initialization, the manager will possess the Id:s of `compA` and `compB` and therefore will be able to control these two components.

Finally, both MEs – `manager` and `aggregator` – are interconnected. The manager is subscribed to the aggregator for application-specific `status` events:

```
myDCMS.subscribe(aggregator, manager, "status");
```

The aggregator is subscribed for the predefined by Niche `componentFailure` environment sensing events that are generated by Niche when `compA` or `compB` fail:

```
myDCMS.subscribe(compA, aggregator, "componentFailure");
myDCMS.subscribe(compB, aggregator, "componentFailure");
```

The self-* architecture manipulates application components using their Id:s. If the initial deployment of the application is performed by the application deployment based on an ADL specification, the Id:s of the components are known to the deployment service and are not known initially to the self-* architecture. This problem is solved by means of a component registry that maps symbolic component names to their Id:s.

When the application is deployed by the deployment service, the symbolic names of components are taken from the ADL specification, and used as arguments to `deploy` methods as discussed in Section 3.10. As the side-effect of the deployment, Niche records the mapping, such that later on the management elements can obtain the component Id:s by the component symbolic names:

```
ComponentId compA = myDCMS.lookup("A");
```

## 3.11 Resource Management and Niche

Resource discovery and management for components implementing the application's functional specification is out of the scope of Niche. Niche is designed to be used together with one or several external resource management services. Deployment and management of resources for MEs is transparent to the application developer and is performed by Niche, as discussed in Section 3.7.

Applications, Niche and resource management services share the `NodeRef` and `ResourceRef` abstract data types. Objects of these types represent physical resources such as memory and CPU cycles.

Applications use the "discovery" request served by resource management services to discover free resources in the Niche infrastructure, see figure 3.13. Resource management services respond with `NodeRef` objects representing free resource(s) on a computing node available to the particular application. Resources discovered this way are not allocated to any application, in particular, a free resource discovered by an application can concurrently

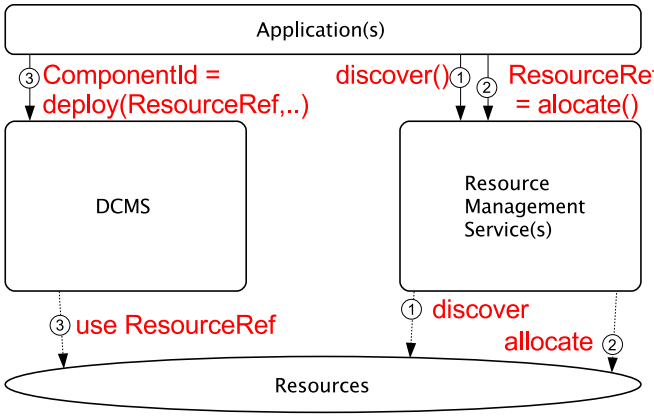


Figure 3.13: Resource Management with Niche.

be discovered and become used by another application. Applications can reserve a part or whole `NodeRef` resource for own usage by means of the "allocate" request to resource management services. If allocation fails due to activity of other applications, the application can try to allocate another previously discovered resource, or restart the discovery from scratch. The result of the "allocate" request is a `ResourceRef` object that represents a resource that is reserved for use by the calling application.

One of the arguments of the "deploy" Niche API function that provides for component deployment (see Section 3.10) is a `ResourceRef` object representing a resource to be used for deploying a particular component. Resources are "consumed" when applications deploy components, i.e. the same resource cannot be used for more than one "deploy" invocation, and for every component Niche verifies its resource usage with respect to properties of `ResourceRef`"s used for deployment of that component.

## 3.12 Groups and Group Sensing

Niche allows the programmer to group components together forming first-class entities called *Niche groups*, or just *groups* thereafter. Components can be bound to groups through *one-to-any* and *one-to-all* bindings, which is an extension of the Fractal model [7]. For functional code, a group of components acts as a single entity. Group membership management is provided by the self-* architecture and is transparent to the functional code. With a one-to-any binding, a component can communicate with a component randomly chosen at run-time from the given group. With a one-to-all binding, it will communicate with all elements of the group. Irrespective of type of bindings, the content of the group can be changed dynamically affecting

neither the component with binding's client interface (binding source), nor components in the group providing the binding server interfaces.

Niche are created by means of the `createGroup` Niche API call:

```
GroupId groupBId =
   myDCMS.createGroup({compB1, compB2}, {"groupServerInterface"});
```

The first argument is the initial list of components in the group, and the second argument is the list of server interfaces the group will provide. Server interfaces provided by a group can be used when making a binding to the group.

Once a group is created, the `GroupId` object can be used as a destination in binding construction call.

```
myDCMS.bind(compA, "clientInterfaceA",
            groupBId, "groupServerInterface",
            ONE_TO_ANY);
```

Here, the client interface `clientInterfaceA` of the component `compA` is bound to the `groupServerInterface` service interface provided by the group `groupBId`, using the one-to-any communication pattern.

Figure 3.14: Watching Groups in Niche-based Applications.

Groups can be watched by a single watcher. When a watcher for a group is being deployed, application programmer specifies the application-specific part of sensors to be used to generate sensing events for the watcher. Niche automatically deploys and removes sensors as the group membership changes. Subscriptions between watchers and dynamically deployed sensors are implicit and managed automatically by the Niche run-time system. In this sense group watchers are different from other management elements where the subscriptions for input management events are first-class and explicitly maintained by the application. The automatic management of sensors for group members is implemented using the SNR abstraction as described in Section 3.15. In figure 3.14, if the component `B2` is removed from the group `B`, then the corresponding sensor will be automatically removed from `B2`. Conversely, if a new component `B3` is added to `B`, then the sensor specified by the programmer for the watcher `W2` will be deployed on `B3`. Note that watchers can also watch other management elements, thus the self-* architecture can be designed to be self-* on its own.

To deploy a watcher and associate it with a group one needs to specify ManagementDeployParameters as follows:

```
params = new ManagementDeployParameters();
params.describeWatcher(watcherImpl, "watcherW",
```

```
                        initialArguments, groupId)
ManagementElementId watcher =
   myDCMS.deploy(ManagementDeployParameters params);
```

Here, the first line constructs a "parameter container" that is filled by the second line, specifying the Java class implementing the watcher (`watcherImpl`), the symbolic name of the ME for the component registry (`"watcherW"`), watcher's initial arguments, and the Id of the group to be watched (`groupId`). Finally, the watcher is deployed with the `deploy` Niche method.

The watcher, when initialized, must specify the sensor type that Niche will automatically deploy on components in the group:

```
myDeploySensorsInterface.deploySensor(sensorImpl,
          "sensorEvent", sensorParameters,
          clientInterfaces, serverInterfaces);
```

Here, `sensorImpl` is the Java class implementing the sensor, `"sensorEvent"` is the event generated by sensors and processed by the watcher, `sensorParameters` are parameters needed for sensor initialization, and the last two arguments are the lists of client and server interfaces used to by Niche to connect sensors to their components using "push" and/or "pull" sensing methods, as discussed in Section 3.7.

## 3.13 Controlling Location of Management Elements

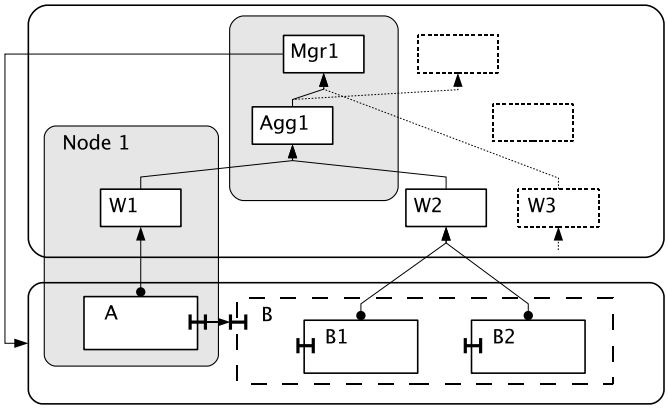


Figure 3.15: Co-location of MEs.

By default, for the sake of load balancing the Niche run-time system attempts to evenly distribute MEs on available computers. In order to reduce communication latency, application developers can control co-location

of MEs, see figure 3.15. ME deployment API calls allow the programmer to specify another architecture element so that the new element will always reside on the same node with the specified one (see also API section for "deploy"). Co-location of MEs can improve the performance of self-management and simplify handling of failures of nodes hosting management elements. However, this facility should be used only when necessary as excessive co-location of MEs can reduce fault-tolerance of application's self-* architecture.

## 3.14 Reliable Self-* Behaviours By Replication

The developer of a self-* architecture can request Niche to host some MEs such that to a certain degree the failures of nodes forming the Niche infrastructure is transparent to the execution of the selected MEs. This is achieved by means of replication of MEs: Niche creates several ME containers (introduced in Section 3.7) each hosting a replica of the ME, as discussed in more detail in Section 3.15.

In the simplest form, the programmer indicates that an ME should be replicated:

```
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA}, REPLICATED);
```

After the deployment, the programmer can manage the set of replicas as one single ME. In particular, it can remove the whole set at once, and subscribe and unsubscribe it to sources and sinks of input and output events. Niche restores failed replicas automatically and transparently using the state of one of the alive ones.

In this simplest form, every replica receives input from all alive sources of events it is subscribed to. Individual replicas can learn their index in the replica set, and e.g. trigger output events only if the index is zero. Individial replicas in the set can fail, and it takes some time to restore them. Input events can be delivered in different order to different replicas, so the programmer must *not* assume that all replicas have the same internal state and produce the same output events in the same order. Instead, replicas should be programmed such that their internal states "self-converge" after a while in a fault-free run, by means of e.g. redundancy in input events and/or auxiliary (replicated) MEs acting as shared storage.

The programmer can also request replication of MEs with consistency guarantees using the `SYNCHRONIZED` flag of ME deployment method:

```
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA}, SYNCHRONIZED);
```

In this case, Niche guarantees the same order of delivery of input events to such synchronized replicas, and exactly one output event is delivered from the set. This property is guaranteed also when a synchronized ME is restored after a node failure.

If the synchronized ME is programmed to be deterministic, i.e. its behaviour is completely determined by the ME's initial state and the sequence of input events, then individual elements in the ME set will pass the same sequence of internal state and produce the same sequence of output events. This approach of providing fault-tolerant services is known as state machine replication [14, 17].

The mechanism of synchronized MEs is a conceptually simple model for programming robust self-management architectures: it gives the programmer the illusion of hosting MEs on failure-free computers.

## 3.15 The Implementation Model of Niche

Niche implementation relies on structured overlay networking (SON), overlay thereafter. The overlay provides to Niche scalable and self-* address lookup and message delivery services. The overlay is used by Niche to implement bindings between components and message-passing between MEs, storage of architecture representation and also failure sensing.

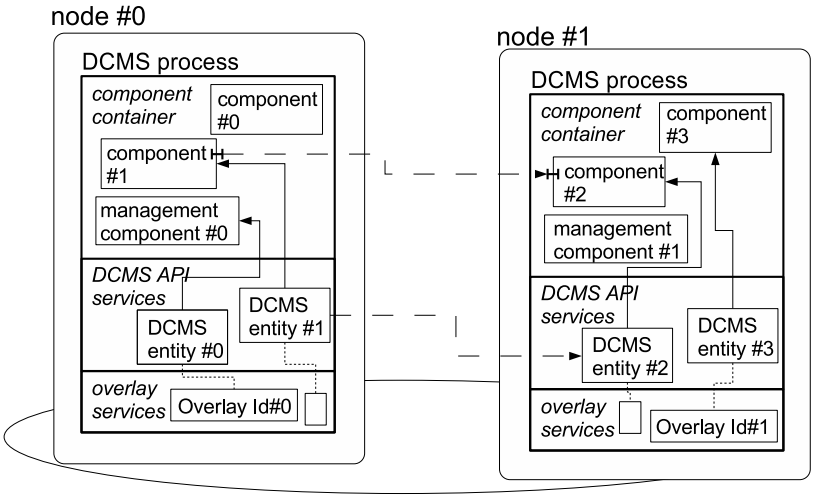


Figure 3.16: Niche infrastructure.

At runtime, Niche infrastructure integrates Niche container processes on several physical nodes using an overlay middleware, DKS [6, 8] in our current prototype. On each physical node there is a local Niche process that provides the Niche API to applications, see figure 3.16. The overlay allows to locate entities stored on nodes of the overlay. On the overlay, entities are

assigned unique overlay identifiers, and for each overlay identifier there is a physical node hosting the identified element. Such a node is usually called a "responsible" node for the identifier. Note that responsible nodes for overlay entities change upon churn. Every physical node on the overlay and thus in Niche also has an overlay identifier, and can be located and contacted using that identifier.

Niche maintains several types of entities we refer to as *Niche* or *DCMS entities*, in particular components of the application architecture and internal DCMS entities maintaining representation of the application's architecture. Niche entities are distributed on the overlay. For example, in figure 3.16 "DCMS entity #0" represents the management component #0. DCMS entities can have references between each other, direct or indirect. For example, in the figure the "DCMS entity #1" can refer to "DCMS entity #2" representing component #2, which is necessary to implement the binding between components #1 and #2. Functional components are situated on specified physical nodes, while MEs and entities representing the architecture might be moved upon churn between physical nodes.

DCMS entities are identified by *DCMS Id:s*. A DCMS Id contains an overlay identifier, "Overlay Id" in the figure 3.16, and a further local identifier. The local identifier allows Niche to distinguish multiple entities assigned to the same overlay identifier. Note that DCMS Id:s act as both unique identifiers and addresses of entities in Niche. In particular, DCMS entities representing components contain the overlay Id of the physical node that the component has been deployed on, and an identifier local to the node. The latter one is mapped by the Niche process to the physical address of the component (in our prototype, a Java object implementing the component). If no co-location constraints are specified for MEs and other DCMS entities, Niche tries to place them on different computers from the Niche infrastructure for the sake of load balancing, by means of assigning random overlay Id:s.

Figure 3.17 illustrates the important types of DCMS entities. There is a management element that deploys another management element with the identifier `Id:3` that we refer to as `ME/Id:3` thereafter. Using the identifier `Id:3`, the Niche process that deployed `ME/Id:3` on a remote node can access it for subsequent operations. `ME/Id:3` deals with a binding `Binding/Id:1`. The `Binding/Id:1` connects the component `Component/Id:6` to a group `Group/Id:5` that contains, among other, component `Component/Id:4`. Components are accessed in general through *references* (`ComponentRef/Id:2` and `ComponentRef/Id:8` in figure), which allows Niche to change the location of the component without affecting other elements of the application self-management.

If, say, `ME/Id:3` wants to perform an operation on the binding `Binding/Id:1`, it can either delegate the execution of the operation to the node where `Binding/Id:1` resides, or obtain and maybe also cache a replica of the bind-

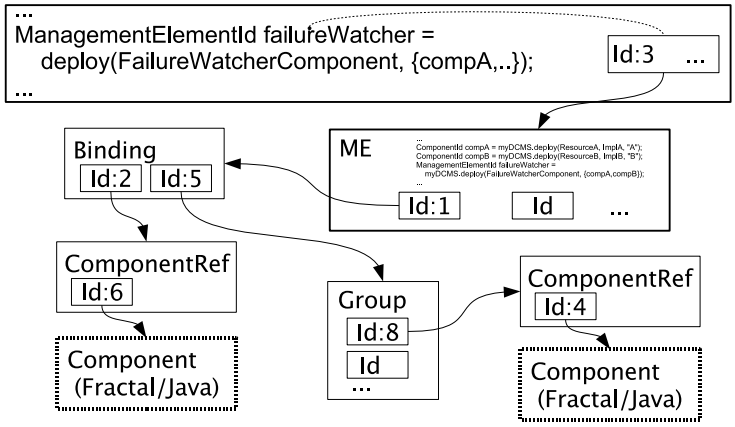


Figure 3.17: Id:s and References in self-* Architecture.

ing entity and execute the operation locally on the replica. For the same entity, some operations on it can be executed using a locally cached replica, while other operations can always require remote execution on the entity itself. Depending on the subset of entity operations that can be executed on a replica, Niche processes cache remote entities partially or entirely. Thus, by "replica" we mean an entity that represents a remote DCMS entity and contains enough information to support local execution of some operations on the entity. Specifically, a replica is not necessarily an verbatim copy of the entity. For instance, in our current Niche prototype a node invoking a binding can attempt to use the cached replica of the binding entity, while binding update is always executed on the binding entity itself. Niche detects invalid identifiers in cached DCMS entities and refreshes the cache contents automatically. Caching policy is determined by the Niche implementation and affects its performance.

Figure 3.18 depicts `ME/Id:3` caching the binding `Binding/Id:1`, the component reference `ComponentRef/Id:2` identifying the component with the binding's client interface, and the component reference `ComponentRef/Id:8` that identifies a member of the group `Group/Id:5`.
Dashed arrows depict the operation of the Niche caching mechanism: when Niche discovers that it possess a cached replica of an entity, then it will use it instead of accessing the remote entity itself.

If MEs or groups are co-located with other DCMS entities, their DCMS Id:s are assigned as follows: the overlay Id is taken from the DCMS Id of the entity to be co-located with, and a fresh local identifier is chosen.

Groups are implemented using *Set of Network References* (SNR) [6, 1] which is a primitive data abstraction that is used to associate a *name* with a set of *references*. SNRs can be thought of as DCMS component reference entities containing multiple references. A "one-to-any" or "one-to-all" bind-

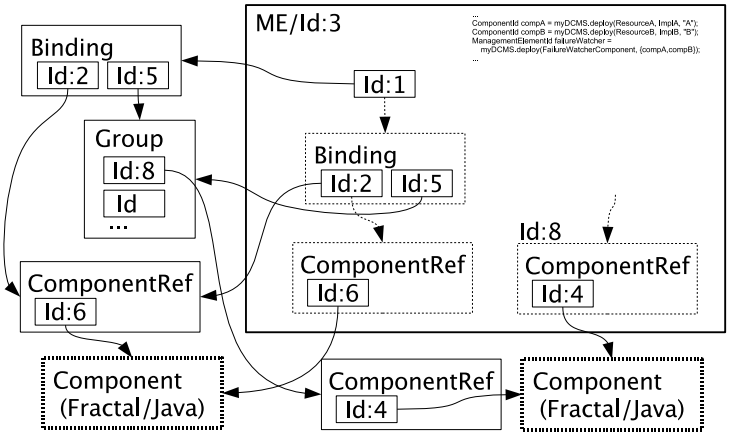


Figure 3.18: Caching of DCMS entities.

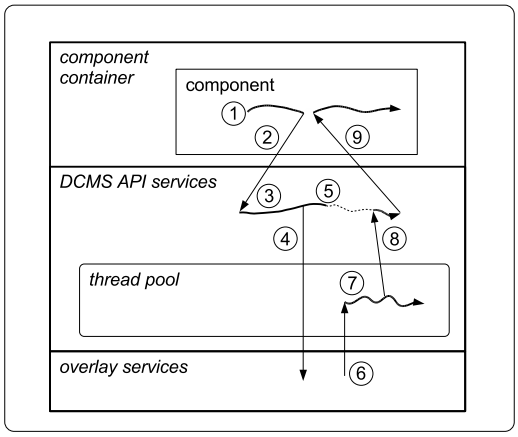


Figure 3.19: Threads of Control in Niche.

ing to a group means that when a message is sent through the binding, the group SNR's location is encapsulated in the group Id, and one or more of the group references from the SNR are used to send the message depending on the type of the binding. A group can grow or shrink transparently from group user's point of view. Finally SNRs support group sensing. Adding a watcher to a group causes deployment of sensors for each element of the group according to the group's SNR. Changing group membership transparently causes deployment/unemployment of sensors for the corresponding elements.

Niche API operations that involve DCMS entities located on remote nodes are implemented using Niche messages, which, in turn, use the underlying overlay network. Operations on a single remote entity without a

return value and without synchronization on completion requires a single Niche message which is delivered asynchronously with respect to the thread initiating the operation. Other types of operations involving a single remote entity require two consequent "request-response" Niche messages. If an operation on an entity involves sub-operations on some further entities, the total number of Niche messages for the operation increases correspondingly. Every Niche message requires an overlay address lookup operation that returns the address of an overlay node with the given overlay Id which is taken from the DCMS Id, and an overlay network message send operation to that address. Overlay address lookup operations usually require the time logarithmic to the size of the overlay, and results of lookup operations are cached by the overlay services layer in Niche processes.

Figure 3.19 illustrates operation of Niche processes. Thread (position 1 in figure) in an application component invokes a Niche operation (2) through the synchronous Niche interface. Niche handles the request (3) which can involve sending messages to remote Niche nodes (4) which, in turn, is handled by the overlay services. If a response message(s) is expected from a remote Niche node in order to complete the request, the thread (1,3) eventually blocks inside Niche (5). Response(s) from remote nodes (6) are handled by threads managed by the Niche thread pool (7). Response handler wakes up (8) the client thread (1,3) blocked inside Niche. Eventually the Niche operation finishes (9) and the thread in the application component (1) resumes the execution. Note that threads from the Niche pool never wait for incoming messages from remote nodes, and thus are never blocked except while waiting in the thread pool for a new request to handle.

The overlay services layer detects failures of other nodes in the Niche infrastructure when it fails to deliver to them pending messages. In this case, messages are handed back to the Niche layer that analyzes and handles the condition. For instance, when a one-to-any binding invocation fails because the destination component picked from the group has failed, Niche will attempt to pick another destination component from the group and repeat the operation. On the other hand, if a node with a resource to be used by a deployment operation has failed, Niche deployment operations fails and this condition is reported to the ME that invoked the operation.

The implementation model for synchronized MEs is presented in figure 3.20. ME generic proxies implement the inter-replica consensus algorithm that totally orders ME input events. One of the better-known algorithms solving this kind of consensus in unreliable environments – the Paxos protocol [13, 16] – has the latency of 3 messages, as opposed to 1 message latency for delivery of input events to non-synchronized and non-replicated MEs. Solutions that allow Paxos to deal with replicas that are restored after node failures [15] do not change the message complexity of Paxos in normal operation. A total-order broadcast algorithm achieving the latency of 2 messages is known [18], but it is unclear whether it can be adopted to

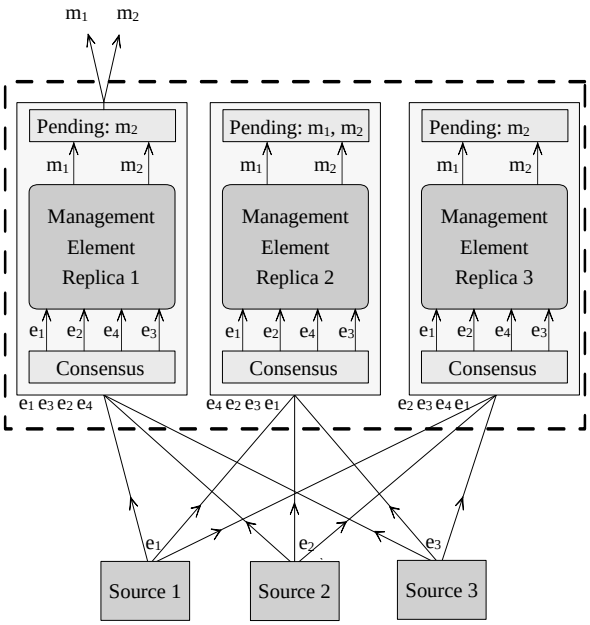


Figure 3.20: Replication of Synchronized MEs in Niche.

deal with replicas that are restored after node failures.

ME generic proxies contain also a queue of pending outgoing events and commands issued by MEs but not yet acknowledged by the recipients. Only the *primary* replica really sends out the events and commands. Acknowledgments are received by all replicas so that a secondary replica can resume exactly where the failure occurred.

In our prototype, the overlay services layer also maintains a pool of threads for managing incoming overlay messages. Threads in Niche API services and overlay services compete for common system resources.

Figure 3.21 illustrates Niche operation with the behaviour of a binding invocation. In figure, component 1 on node 0 is bound to component 2 on node 1. On node 0, Niche with the assistance of the component container created a binding stub – a special type of component with a matching server interface, so that component 1 is actually bound to the stub (position 1 in figure). When component 1 invokes its client interface, the implementation of the server interface in the binding stub calls Niche (2) to deliver the binding invocation to node 1. Niche retrieves the binding destination from the binding entity (3) or uses the cached binding replica, and sends the message to binding destination node (4). At the point, node 1 can associate the incoming request with the binding destination – server interface of component 2, and invokes it (5).

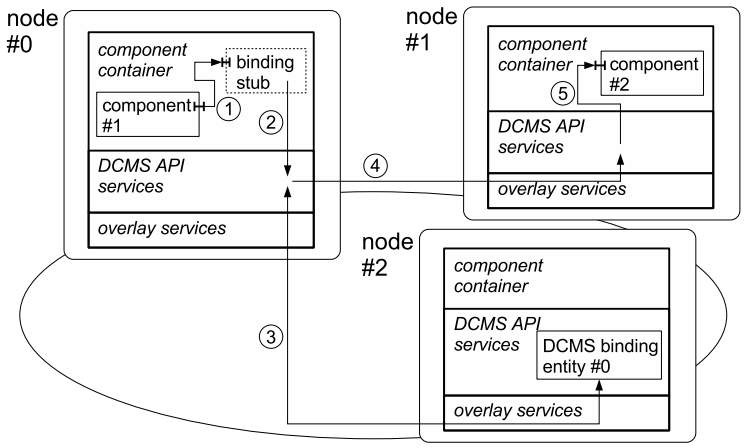


Figure 3.21: Bindings in Niche.

# Chapter 4

# DCMS API

## 4.1 Interfaces

### 4.1.1 Interface **NicheActuatorInterface**

The `NicheActuatorInterface` class

This class fills three purposes: - It acts as the interface class for Niche operations which are available only for management elements, while including the operations available to all components through extending NicheComponentSupportInterface - It acts as the interface class for proxies created by Jade - It gives access to primitive resource management services for systems and applications that do not provide those services themselves

Declaration

public interface NicheActuatorInterface
**implements** NicheComponentSupportInterface

Methods

- *allocate*
  `public ArrayList` **allocate**`( Serializable` **destinations**`, Object` **descriptions** `)`
  - **Usage**
    - Allocates a (part of a) discovered node, which is needed before deploying components
  - **Parameters**

    - `destinations` - Either a single ResourceId or an ArrayList<ResourceId>for bulk operation
    - `descriptions` - Either a single description or an ArrayList of descriptions for bulk operation. The format of allocate description will depend on the resource management being implemented
  - **Returns** - A list of the allocated resource identifiers, null if the operation could not be completed for the resource

---

- *cancelTimer*
  `public void` **cancelTimer**`( long` **timerId** `)`
  - **Usage**
    - Cancels a timer previously registered with registerTimer
  - **Parameters**
    - `timerId` -

---

- *deallocate*
  `public void` **deallocate**`( ResourceId` **resourceId** `)`
  - **Usage**
    - Frees a previously allocated resource
  - **Parameters**
    - `resourceId` - The reference to the resource which should be deallocated

---

- *deploy*
  `public ArrayList` **deploy**`( Serializable` **destinations**`, Serializable` **descriptions** `)`
  - **Usage**
    - Deploys one or more fractal components as specified by one or more component descriptions.
      As of now the code of the component to be deployed has to exist on the receiving computer.
  - **Parameters**
    - `destinations` - Either a single (allocated) ResourceId of an ArrayList<ResourceId>for bulk operation
    - `descriptions` - Either a single description or an ArrayList of descriptions for bulk operation. *Insert text from Nikos*
  - **Returns** - A list containing one or more global component ids

---

- *deployManagementElement*
  `public NicheId` **deployManagementElement**`( ManagementDeployParameters` **description**`, IdentifierInterface` **destination** `)`

- **Usage**
  - Deploys a management element as specified by a management element component description.
    As of now the code of the component to be deployed has to exist on the receiving computer.
- **Parameters**
  - `description` - A management element description *Insert text from Nikos*
  - `destination` - The reference to another management element, with which the new element should be collocated
- **Returns** - A management element id

---

- *discover*
  `public ArrayList` **discover**`( Serializable` **requirements** `)`
  - **Usage**
    - Method to ask the resource manager for currently free nodes matching the requirements
  - **Parameters**
    - `requirements` - The format of requirement description will depend on the resource management being implemented
  - **Returns** - A list of all nodes which can provide resources matching the requirements, null if none could be found

---

- *getComponentType*
  `public ComponentType` **getComponentType**`( String` **adlName** `)`
  - **Usage**
    - Returns the Jade component type corresponding to a given adl name. If the component type was not previously generated on the node where the method call is done, it will be generated on the first invocation. This therefore requires that the given adl name corresponds to an existing adl file.
  - **Parameters**
    - `adlName` -
  - **Returns** - A Jade component type object

---

- *getGroupTemplate*
  `public GroupId` **getGroupTemplate**`( )`
  - **Usage**

* Returns an 'empty' GroupId to be used as template when specifying which interfaces that should be automatically bound by the system when a component becomes member in a specific group.

- **Returns** - An empty GroupId

---

- *getId*
  `public NicheId` **getId**`( )`

---

- *getLogger*
  `public LoggerInterface` **getLogger**`( )`

  - **Usage**
    * Allows the applications/system developer to reuse the logging functionality already present in Niche
  - **Returns** - A reference to the Niche log4j-logger

---

- *oneShotDiscoverResource*
  `public NodeRef` **oneShotDiscoverResource**`( Serializable` **requirements** `)`

  - **Usage**
    * A shorthand to grab just one node matching the requirements
  - **Parameters**
    * `requirements` - The format of requirement description will depend on the resource management being implemented
  - **Returns** - The first found resource that matched the requirements, null if none could be found

---

- *redeployManagementElement*
  `public void` **redeployManagementElement**`( ManagementDeployParameters` **description**`, IdentifierInterface` **oldId** `)`

  - **Usage**
    * Redeploys a management element which has failed due to no, or insufficient replication
  - **Parameters**
    * `description` - A management element description
    * `destination` - The id of the failed ME to be recreated
  - **Returns** - A management element id

---

- *registerTimer*
  `public long` **registerTimer**`( EventHandlerInterface` **managementElement**`, Class` **eventClassName**`, int` **timerDelay** `)`

- **Usage**
  - Allows a management element to register a one-off timer. When the time delay has expired the eventHandler method of the management element will be called with a event of class eventClassName
- **Parameters**
  - `managementElement` - The management element which will be called when the timer goes off
  - `eventClassName` - The event which will be generated
  - `timerDelay` - The timer delay in milliseconds
- **Returns** - A timer id which is needed for cancellation

---

- *sendOnBinding*
  `public Object` **sendOnBinding**`( Object` **localBindId**`, Invocation` **invocation**`, ComponentId` **shortcut** `)`
  - **Usage**
    - Used by Jade-created interface proxies for one-way bindings. Semi-synchronous - propagates a method invocation and waits until the message is on the network.
  - **Parameters**
    - `localBindId` - The id of the proxy
    - `invocation` - The wrapped method invocation
    - `shortcut` - Gives the possibility to specify a specific receiver out of a group
  - **Returns** - Any object as specified by the interface description

---

- *sendWithReply*
  `public Object` **sendWithReply**`( Object` **localBindId**`, Serializable` **invocation** `)`
  - **Usage**
    - Used by Jade-created interface proxies for two-way bindings. Synchronous - propagates a method invocation and waits for a reply.
  - **Parameters**
    - `localBindId` - The id of the proxy
    - `invocation` - The wrapped method invocation
  - **Returns** - Any object as specified by the interface description

---

- *subscribe*
  `public Subscription` **subscribe**`( IdentifierInterface` **source**`, IdentifierInterface` **sink**`, String` **eventName** `)`
  - **Usage**

* Adds a new sink to the event generating management element source

– **Parameters**

* `source` - The id of the management element generating the events of intrest, or the id of the group defining the scope of interest in case of a subscription to an infrastructure event
* `sink` - The id of the management element interested in the event
* `eventName` - The full classname of the event

– **Returns** - A subscription which can be used to later change or stop the subscription

---

- *subscribe*
  `public Subscription` **subscribe**`( IdentifierInterface` **source**`, IdentifierInterface` **sink**`, String` **eventName**`, Serializable` **tag** `)`

  – **Usage**

  * Adds a new sink to the event generating management element source

  – **Parameters**

  * `source` - The id of the management element generating the events of intrest, or the id of the group defining the scope of interest in case of a subscription to an infrastructure event
  * `sink` - The id of the management element interested in the event
  * `eventName` - The full classname of the event
  * `tag` - Can be used to filter events based on the tag

  – **Returns** - A subscription which can be used to later change or stop the subscription

---

- *subscribe*
  `public Subscription` **subscribe**`( IdentifierInterface` **source**`, String` **sink**`, String` **eventName**`, IdentifierInterface` **sinkLocation** `)`

  – **Usage**

  * Adds a new sink to the event generating management element source, given that the sink is present

  – **Parameters**

  * `source` - The id of the management element generating the events of intrest
  * `sink` - The ADL name of the management element interested in the event

    * `eventName` - The full classname of the event
    * `sinkLocation` - The id of any element known to be collocated with the sink
  - **Returns** - A subscription which can be used to later change or stop the subscription, or null if the sink did not exist

---

- *testingOnly*
  `public NicheAsynchronousInterface` **testingOnly**`( )`

---

- *unsubscribe*
  `public boolean` **unsubscribe**`( Subscription` **subscription** `)`
  - **Usage**
    * Cancels a subscription
  - **Parameters**
    * `subscription` - The subscription specifying source, sink and event to stop listening to

---

- *update*
  `public void` **update**`( Object` **objectToBeUpdated**`, Object` **argument**`, int` **type** `)`
  - **Usage**
    * Generic method to update groups or management elements
  - **Parameters**
    * `objectToBeUpdated` - The id of the management element which should be updated
    * `argument` - The update message, or the item to add/remove, depending on the type
    * `type` - The type specifying the update operation, as given by the constants in *NicheComponentSupportInterface*

### 4.1.2 Interface **NicheComponentSupportInterface**

---

The `NicheComponentSupportInterface` class. Gives access to group management, (Niche wide) bind operations and primitive query support the query sytem state.

Declaration

---

```
public interface NicheComponentSupportInterface
```

FIELDS

- public static final int ADD_TO_GROUP
- public static final int ADD_TO_GROUP_AND_START
- public static final int REMOVE_FROM_GROUP
- public static final int REMOVE_GROUP
- public static final int GET_CURRENT_MEMBERS

METHODS

- *addToGroup*
  `public void` **addToGroup**`( Object` **newItem**`, Object` **groupId** `)`
  - **Usage**
    - Add a new component to an existing group
  - **Parameters**
    - `newItem` - A ComponentId representing the component to be added to the group. The component has to share the same interfaces as the previous group members, as it will be automatically become part of the existing bindings related to the group.
    - `groupId` - The id of the existing group

- *bind*
  `public BindId` **bind**`( Object` **sender**`, String` **senderInterface**`, Object` **receiver**`, String` **receiverInterface**`, int` **type** `)`
  - **Usage**
    - Binds the fractal client interface of 'client' to server interface of 'server'
  - **Parameters**
    - `client` - Normally a ComponentId. Can also be a GroupId, in which case bindings are created between all members of 'client' to the server
    - `clientInterface` - The ADL name of the client interface
    - `server` - Either a single ComponentId or a GroupId where all group members expose 'serverInterface'
    - `serverInterface` - The ADL name of the server interface

        * `type` - The type of the bindId: one-to-one, one-to-any, one-to-many, defined by constants in *currently* JadeBindInterface
    - **Returns** - A bindId id

---

- *bind*
  `public void` **bind**`( String` **senderInterface**`, Object` **receiver**`, String` **receiverInterface**`, int` **type** `)`
    - **Usage**
        * Binds the fractal client interface of the component calling the method to the server interface of 'server'
    - **Parameters**
        * `clientInterface` - The ADL name of the client interface
        * `server` - Either a single ComponentId or a GroupId where all group members expose 'serverInterface'
        * `serverInterface` - The ADL name of the server interface
        * `type` - The type of the bindId: one-to-one, one-to-any, one-to-many, defined by constants in *currently* JadeBindInterface
    - **Returns** - A bindId id

---

- *createGroup*
  `public GroupId` **createGroup**`( SNR` **template**`, ArrayList` **items** `)`
    - **Usage**
        * Creates a new group based on the given template and the components in the array list.
    - **Parameters**
        * `template` - A template which defines the interfaces which the group should manage upon membership changes
        * `items` - An array list of all components which should be part of the group. The components must have at least one interface in common.
    - **Returns** - A group id representing the new group

---

- *createGroup*
  `public GroupId` **createGroup**`( String` **templateName**`, ArrayList` **items** `)`
    - **Usage**
        * Creates a new group based on the template name and the components in the array list. This requires that the template has been previously created and registered with the template name

- **Parameters**
  - `templateName` - A template name which refers to a template which defines the interfaces which the group should manage upon membership changes
  - `items` - An array list of all components which should be part of the group. The components must have at least one interface in common.
- **Returns** - A group id representing the new group

---

- *getResourceManager*
  `public SimpleResourceManager` **getResourceManager**`( )`
  - **Usage**
    - Gives components access to the local resource manager, which can be used to get the component id based on the ADL name
  - **Returns** - The local resource manager

---

- *query*
  `public Object` **query**`( IdentifierInterface` **queryObject**`, int` **queryType** `)`
  - **Usage**
    - Generic query method to ask queries about elements in the system. The available query-types are as of now given as constants by this class
  - **Parameters**
    - `queryObject` - The id of the element which the query is concerning
    - `queryType` - The type specifying the query operation, as given by the constants in `NicheComponentSupportInterface`
  - **Returns** - The return type is dependent on the query - it can be a single object/identifierinterface or an arraylist of objects/identifierinterfaces

---

- *registerGroupTemplate*
  `public boolean` **registerGroupTemplate**`( String` **templateName**`, SNR` **template** `)`
  - **Usage**
    - Registers a group template to be used for subsequent group creation, where the user has specified which interfaces that the group should make available to any component bound to that group. Please note the created template name is only valid locally, for one node in the system
  - **Parameters**

    * `templateName` - A String representing the name of the template.
    * `template` - The group template with the interfaces of interest specified
  - **Returns** - True if the template was successfully registered, false if there already existed a template with that name

---

- *removeFromGroup*
  `public void` **removeFromGroup**`( Object` **item**`, Object` **groupId** `)`
  - **Usage**
    * Removes a component from an existing group
  - **Parameters**
    * `item` - A ComponentId representing the component to be removed from the group.
    * `groupId` - The id of the existing group

---

- *removeGroup*
  `public void` **removeGroup**`( GroupId` **gid** `)`
  - **Usage**
    * Removes an existing group. All watchers subscribed through that group will no longer be notified of changes to the previous group members, although the components themselves will remain unaltered
  - **Parameters**
    * `gid` - The id of the group to remove

---

- *unbind*
  `public void` **unbind**`( IdentifierInterface` **binding** `)`
  - **Usage**
    * Removes a previously established binding
  - **Parameters**
    * `binding` - The id of the binding to remove.

---

- *update*
  `public void` **update**`( Object` **objectToBeUpdated**`, Object` **argument**`, int` **type** `)`
  - **Usage**
    * Generic method to update groups or management elements. The available update-types are as of now given as constants by this class, but they might later be moved to the `DCMSInterface`

– **Parameters**
  * `objectToBeUpdated` - The id of the management element which should be updated
  * `argument` - The update message, or the item to add/remove, depending on the type
  * `type` - The type specifying the update operation, as given by the constants in `NicheComponentSupportInterface`

## 4.2 Interfaces

### 4.2.1 Interface **EventHandlerInterface**

The `EventHandlerInterface` class This interface must be implemented by all management elements that want to be able to subscribe to events, and have them delivered

#### Declaration

```
public interface EventHandlerInterface
```

#### Methods

- *eventHandler*
  `public void` **eventHandler**`( Serializable` **event**`, int` **flag** `)`

  - **Usage**
    - This method is invoked by the system when an event matching a previously done subscription is delivered.
  - **Parameters**
    - `event` - The event coming from one of the sources to which the ME is subscribed
    - `flag` - A flag which indicates whether the event has arrived normally (value zero) or during a period of churn, so that the ME has been moved or restored in between event creation and event delivery

### 4.2.2 Interface **InitInterface**

The `InitInterface` class
This interface must be implemented by all management elements to be properly initialized by the Niche framework. Please observe that the system gives no guarantees about the order which these methods are called upon ME creation

#### Declaration

```
public interface InitInterface
```

- *init*
  `public void` **init**`( dks.niche.interfaces.NicheActuatorInterface` **actuator** `)`
  - **Usage**
    - The system will invoke this method on a management element during its creation.
  - **Parameters**
    - `parameters` - An instance of the NicheActuatorInterface to be used by the ME

---

- *init*
  `public void` **init**`( Serializable [] parameters )`
  - **Usage**
    - The system will invoke this method on a management element during its creation, if it is being created for the first time as a result of managment deployment
  - **Parameters**
    - `parameters` - The initialArguments parameter of the ManagementDeployParameters instance used to deploy the ME
  - **See Also**
    - `ManagementDeployParameters`

---

- *initId*
  `public void` **initId**`( NicheId` **id** `)`
  - **Usage**
    - The system will invoke this method on a management element during its creation
  - **Parameters**
    - `parameters` - The NicheId of the ME being initialized

---

- *reinit*
  `public void` **reinit**`( Serializable [] parameters )`
  - **Usage**
    - The system will invoke this method on a management element during its creation, if it is being recreated after a churn event
  - **Parameters**

* `parameters` - The ME-parameters as given by the getAttributes method of the MovableInterface

– **See Also**

* `MovableInterface` ( in 4.2.3, page 57)

### 4.2.3 Interface **MovableInterface**

The `MovableInterface` class Any management interface which wants support from the system to be automatically moved and redeployed upon churn needs to implement this interface.

Declaration

```
public interface MovableInterface
```

Methods

- *getAttributes*
  `public Serializable` **`getAttributes( )`**

  – **Usage**

  * The system will call this method on a management element which is about to be moved or copied.

  – **Returns** - An array of any parameters the management element is dependent on to be properly re-initialized. It is the responsibility of the ME designer to record the state in the way expected by the re-init method of the same ME class.

  – **See Also**

  * `InitInterface` ( in 4.2.2, page 55)

### 4.2.4 Interface **TriggerInterface**

The `TriggerInterface` class This interface is used by management elements that want to be able to trigger events

Declaration

```
public interface TriggerInterface
```

- *removeSink*
  `public void` **removeSink**`( String` **sinkId** `)`

- *trigger*
  `public void` **trigger**`( Serializable` **event** `)`

  - **Usage**
    - Triggers an event
  - **Parameters**
    - `event` - The event, which will be matched against current subscriptions. All management elements which has subscribed to the triggering element for that type of event will get notified

- *trigger*
  `public void` **trigger**`( Serializable` **event**`, Serializable` **tag** `)`

  - **Usage**
    - Triggers an event with a special tag for filtering
  - **Parameters**
    - `event` - The event, which will be matched against current subscriptions. If there are subscriptions matching the event type, they will also be checked against the tag.
    - `tag` -

- *triggerAny*
  `public void` **triggerAny**`( Serializable` **event** `)`

  - **Usage**
    - Triggers an event
  - **Parameters**
    - `event` - The event, which will be matched against current subscriptions. Out of the matching subscriptions, one random subscriber will get notified

## 4.3 Classes

### 4.3.1 Class ManagementDeployParameters

The `ManagementDeployParameters` class. An instance of the class is needed as parameter for the call to `deploy` provided by `DCMS`.

To deploy management elements, the element specific methods `describe...` should be used.

#### Declaration

public class ManagementDeployParameters
**extends** Object
**implements** Serializable

#### Constructors

- *ManagementDeployParameters*
  `public` **ManagementDeployParameters**`( )`
  - **Usage**
    - Standard empty constructor

#### Methods

- *bind*
  `public void` **bind**`( String` **clientComponentName**`, String` **clientInterfaceName**`, String` **serverComponentName**`, String` **serverInterfaceName** `)`
  - **Usage**
    - Allows the user to specify local bindings which should be initiated at component deploy time
  - **Parameters**
    - `clientComponentName` - Local component ADL name
    - `clientInterfaceName` - Local component client interface name
    - `serverComponentName` - Local component ADL name
    - `serverInterfaceName` - Local component server interface name

- *deployComponent*
  `public void` **deployComponent**`( String` **ADL**`, String` **componentName**`, ComponentType` **componentType**`, .Map` **context** `)`
  - **Parameters**
    - `ADL` - The ADL file name containing the component description
    - `componentName` - The new component name. If null then the name in the ADL will be used
    - `context` - used for example to set attribute=value

---

- *deployComponent*
  `public void` **deployComponent**`( String` **ADL**`, String` **managementElementName**`, ComponentType` **componentType**`, .Map` **context**`, int` **type**`, Serializable []` **initialArguments**`, boolean` **reliable**`, boolean` **movable**`, boolean` **start**`, NicheId` **managedComponentId** `)`
  - **Parameters**
    - `ADL` - The ADL file name containing the component description
    - `componentName` - The new component name. If null then the name in the ADL will be used
    - `context` - used for example to set attribute=value
    - `type` - used for the framework to automatically bind the management element depending on type

---

- *describeAggregator*
  `public void` **describeAggregator**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable []` **initialArguments** `)`
  - **Usage**
    - Describes an aggregator to be deployed
  - **Parameters**
    - `className` - The class name of the java class file implementing the management element
    - `componentName` - The new component name. If null then the name from the ADL will be used
    - `initialArguments` - An array of initial arguments to be passed to the management element init method

---

- *describeExecutor*
  `public void` **describeExecutor**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable [] initialArguments, NicheId` **actuatedComponentId** `)`

---

- *describeManager*
  `public void` **describeManager**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable [] initialArguments` `)`
  - **Usage**
    - Describes a manager to be deployed
  - **Parameters**
    - `className` - The class name of the java class file implementing the management element
    - `componentName` - The new component name. If null then the name from the ADL will be used
    - `initialArguments` - An array of initial arguments to be passed to the management element init method

---

- *describeSensor*
  `public void` **describeSensor**`( String` **className**`, String` **componentName**`, Serializable [] initialArguments` `)`
  - **Usage**
    - Describes a sensor to be deployed. This deployment can only be done by the responsible watcher
  - **Parameters**
    - `className` - The class name of the java class file implementing the sensor
    - `componentName` - The new component name.
    - `initialArguments` - An array of initial arguments to be passed to the sensor init method

---

- *describeWatcher*
  `public void` **describeWatcher**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable [] initialArguments, NicheId` **watchedComponentId** `)`
  - **Usage**
    - Describes a watcher to be deployed
  - **Parameters**
    - `className` - The class name of the java class file implementing the management element

        * `componentName` - The new component name. If null then the name from the ADL will be used
        * `initialArguments` - An array of initial arguments to be passed to the management element init method
        * `watchedComponentId` - The new id of the component, or group, with which the watcher is associated

---

- *getReInitParameters*
  `public Serializable` **getReInitParameters**`( )`

---

- *getType*
  `public int` **getType**`( )`

---

- *keepAlive*
  `public boolean` **isReliable**`( )`
    - **Returns** - Tells whether the new component is declared to be reliable

---

- *lifeCycle*
  `public void` **lifeCycle**`( String` **componentName**`, boolean` **start** `)`
    - **Usage**
        * The method can be used together with the 'deploy' settings to specify whether the component should be started directly after deployment. It can also be used on its own to remotly start an already deployed component
    - **Parameters**
        * `componentName` - The component name you want to start or stop
        * `start` - true to start it & false to stop it

---

- *setAttributes*
  `public void` **setAttributes**`( ArrayList` **attributes** `)`

---

- *setAttributes*
  `public void` **setAttributes**`( String` **componentName**`, String` **controllerName**`, .Map` **attributes** `)`
    - **Usage**
        * Method used to specify initial attribute values of component attributes. Requires the component to implement corresponding attribute controller.
    - **Parameters**
        * `componentName` - Component ADL name

* `controllerName` - Classname of component attribute controller
* `attributes` - A map specifying pairs

---

- *setReInitParameters*
  `public void` **setReInitParameters**`( Serializable [] param )`

---

- *setReliable*
  `public void` **setReliable**`( boolean  reliable )`

  - **Usage**
    * Specifies whether the new component should be reliable, that is if the runtime system should keep the element replicated despite churn

Figure 5.1: YASS Functional Part

# Chapter 5

# DCMS Use Case: YASS

## 5.1 Architecture

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

### 5.1.1 Application functional design

A YASS instance consists out of *front-end components* which are deployed on user machines and *storage components* Figure 5.1. Storage components

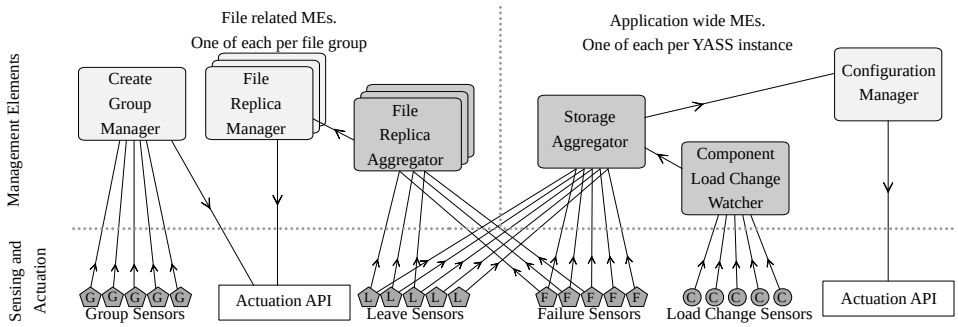


Figure 5.2: YASS Non-Functional Part

are composed of *file components* representing files. The ovals in Figure 5.1 represent resources contributed to a Virtual Organization (VO). Some of the resources are used to deploy storage components, shown as rectangles.

The service contain two types of groups, storage group and file group. The Storage group containing all storage components and file group containing all replicas of a specific file. Each instance of YASS have only one storage group and several file groups depending on the number of stored files (one file group per stored file).

A user store request is sent (using one-to-any binding between the front-end and the storage group) to an arbitrary storage component that will try to find some $r$ different storage components, where $r$ is the file's replication degree, with enough free space to store a file replica. These replicas together will form a *file group* containing the $r$ dynamically created new file components. The user will then use a one-to-all binding to the file group to send the file in parallel to the $r$ replicas in the file group. Read requests can be sent to any of the $r$ file components in the group using the one-to-any binding between the front-end and the file group. Similarly, a delete request is sent to all file components using one-to-all binding between the front-end and the file group.

### 5.1.2 Application non-functional design

**Configuration of application self-management.** The Figure 5.2 shows the architecture of the watchers, aggregators and managers used by the application.

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

When new file-groups are formed by the functional part of the appli-

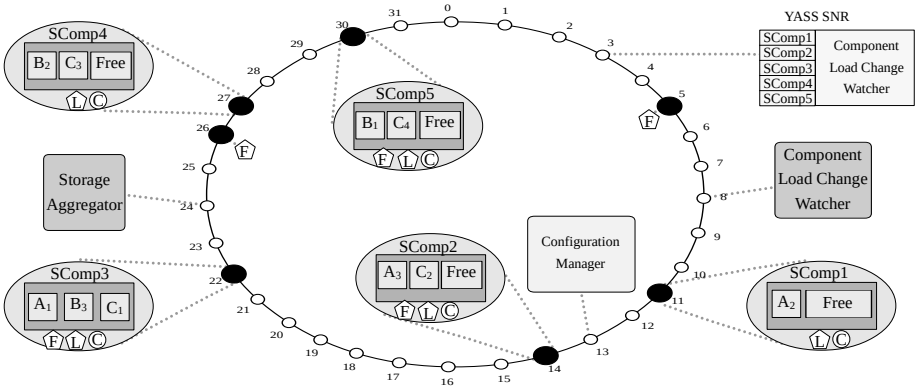


Figure 5.3: Parts of the YASS application deployed on the management infrastructure.

cation, the management infrastructure propagates group-creation events to the CreateGroup-manager which initiates a FileReplica-aggregator and a FileReplica-manager for the new group. The new FileReplica-aggregator is subscribed to resource leave- and resource fail-events of the resources associated with the new file group.

Before explaining these management elements in more details we'll present an example that will help in better understanding.

### 5.1.3 Test-cases and initial evaluation

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

Figure 5.3 shows the management elements associated with the group of storage components. The black circles represent physical nodes in the P2P overlay Id space. Architectural entities (e.g. SNR and MEs) are mapped to ids. Each physical node is responsible for Ids between its predecessor and itself including itself. As there is always a physical node responsible for an id, each entity will be mapped to one of the nodes in the system. For instance the *Configuration Manager* is mapped to id 13, which is the responsibility of the node with id 14 which means it will be executed there.

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

*Resource leave.* An infrastructure sensor signals that a resource is about to leave. For each file stored at the leaving resource, the associated FileReplica-aggregator is notified and issues a replicaChange-event which is forwarded

to the FileReplica-manager. The FileReplica-manager uses the one-to-any binding of the file-group to issue a FindNewReplica-event to any of the components in the group.

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

**Application Self-configuration.** With self-configuration we mean the ability to adapt the system in the face of dynamism, thereby maintaining its capability to meet functional requirements. This is achieved by monitoring the total amount of allocated storage. The Storage-aggregator is initialized with the amount of available resources at deployment time and updates the state in case of resource leaves or failures. If the total amount of allocated resources drops below given requirements, the Storage-aggregator issues a storageAvailabilityChange-event which is processed by the Configuration-manager. The Configuration-manager will try to find an unused resource (via the external resource management service) to deploy a new storage component, which is added to the group of components. Parts of the Storage-aggregator and Configuration-manager pseudocode is shown in Listing 5.1, demonstrating how the stateful information is kept by the aggregator and updated through sensing events, while the actuation commands are initiated by the manager.

**Application Self-optimization.** In addition to the two above described test-cases we have also designed but not fully tested application self-optimization. With self-optimization we mean the ability to adapt the system so that it, besides meeting functional requirements, also meets additional non-functional requirements such as efficiency. This is achieved by using the ComponentLoad-watcher to gather information on the total system load, in terms of used storage. The storage components report their load changes, using application specific load sensors. These load-change events are delivered to the Storage-aggregator. The aggregator will be able to determine when the total utilization is critically high, in which case a StorageAvailabilityChange-event is generated and processed by the Configuration-manager in the same way as described in the self-configuration section. If utilization drops below a given threshold, and the amount of allocated resources is above initial requirements, a storageAvailabilityChange-event is generated. In this case the event indicates that the availability is higher than needed, which will cause the Configuration-manager to query the ComponentLoad-watcher for the least loaded storage component, and instruct it to deallocate itself, thereby freeing the resource. Parts of the Configuration-manager pseudocode is shown in Listing 5.2, demonstrating how the number of storage components can be adjusted upon need.

Listing 5.1: Pseudocode for parts of the Storage-aggregator

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

Listing 5.2: Pseudocode for parts of the Configuration-manager

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

## 5.2 Implementation

### 5.2.1 The Component Load Sensor

The component load sensor is implemented in `yass.sensors.LoadSensor`. It is responsible for monitoring the load of a storage component. Sensors can be *push* and/or *pull* style. The load sensor is a push sensor. When a file is added to or removed from a storage component, the new load is *pushed* by the storage component to the associated load sensor. This is done using `yass.sensors.LoadChangeInterface`. The `LoadSensor` triggers a `ComponentStateChangeEvent` when the storage component load changes with a predefined *delta*.

### 5.2.2 The Component Load Watcher

The component load watcher is implemented in `yass.watchers.LoadWatcher`. It is subscribed to all `yass.sensors.LoadSensor` sensors. Currently the load watcher only forwards the `ComponentStateChangeEvent` triggered by the load sensors.

The watcher must specify the sensor that it requires to be able to watch component(s). For example the component load watcher requires the component load sensor to be deployed to be able to watch storage components. It is the responsibility of the infrastructure to deploy the sensor and bind it with the watched component and do the subscriptions for the events generated by the sensor.

The sensor specification is done when initializing the watcher in `init()`. It includes the following:

- Sensor class name.
- Event class name that is generated by the sensor.
- Any parameters needed to initialise the sensor (delta for load sensor).
- Name of fractal client interface(s) used by pull sensors. A server interface with the same name must exist on the watched component.
- Name of fractal server interface(s) used by push sensors. A client interface with the same name must exist on the watched component.

Here is how the load watcher specifies the load sensor:

```
deploySensor.deploySensor("yass.sensors.LoadSensor",
   "yass.sensors.LoadSensorEvent", sensorParameters,
   null, new String[] { "pushLoadChange" });
```

### 5.2.3 The Storage Aggregator

The storage aggregator is implemented in `yass.aggregators.StorageAggregator`. It is responsible for keeping the state of the system. When a resource fail or leave it remove the amount of storage that was allocated on the leaving resource and if the remaining storage is less than a predefined threshold it informs the ConfigurationManager through a `StorageAvailabilityChangeEvent`. When a new storage component joins the storage group the storage aggregator adds its storage capacity to the total storage. Currently the Component State Change event is not handled.

### 5.2.4 The Configuration Manager

The configuration manager is implemented in `yass.managers.ConfigurationManager`. It is registered to `StorageAvailabilityChangeEvent` from the storage aggregator. Upon receiving the event it checks if the load is higher than a threshold or the total capacity is less than the minimum threshold and if so it tries to find new resource to deploy a storage component on it.

### 5.2.5 The File Replica Aggregator

The file replica aggregator is implemented in `yass.aggregators.FileReplicaAggregator`. It monitors the members of the associated file group for resource leave and failure by subscribing to corresponding resource fail and leave sensors.

When a watched resource fails/leaves, it triggers a `ReplicaChangeEvent` indicating that action must be taken to restore the replication degree.

### 5.2.6 The File Replica Manager

The file replica manager is implemented in `yass.managers.FileReplicaManager`. When it receive a `ReplicaChangeEvent` it starts to find a new replica by sending a `ReplicaRestoreRequest` to any of the remaining members of the file group.

### 5.2.7 The Create Group Manager

The create group manager is implemented in `yass.managers.CreateFileGroupManager`. It is subscribed to the `CreateGroupEvent` which is an infrastructure event. The purpose of this manager is to detect when the functional part (storage components) creates a new file group. When a new file group is detected, the create group manager will create a new file replica aggregator and manager for the new file group.

### 5.2.8 The Start Manager

The start manager is implemented in `yass.managers.StartManager`. This is called at the deployment time. The purpose of this manager is to handle operations that are not yet implemented in the ADL.

## 5.3 Installation

YASS requires Jade and Niche to be installed and configured properly. In this section we will guide you, step by step, to prepair your environment to run and/or develop applications based on the DCMS.

### Download

You will need Jade, Niche, and YASS:

- If you are interested only in running YASS then it is enough for you to get Jade because we already added the JAR files for Niche and YASS in Jade. You can get Jade from the SVN repository at

  `https://gforge.inria.fr/projects/grid4all/` you will find it under `wp1/Jade`.

- You can also checkout the source code of YASS from the same repository under `wp1/YASS`.

- The Niche source code can be found at

  `svn://korsakov.sics.se/dks/`

### The Web Cache

The DCMS uses the Niche/DKS [6, 8], an overlay middleware built on the DKS structured overlay network. Any new resource/node that wants to join an existing overlay must first contact a node already inside the overlay to be able to join. The only exception is the first node which created the overlay.

The problem now is how will a new node learn about the reference (ip/-port) of an existing node? The solution used is a simple web page that contains the references to the most recent nodes in the overlay. We assume that the URL of this page is known by the nodes that wants to join the overlay repsented by the page. We call this web page the web cache.

For each overlay you need one web cache. Two different overlays must use different web caches. This web cache is used to cache references to some nodes that are part of the overlay. We assume that all nodes know the URL of the web cache. This is configurable through the `Jade/etc/dks/dksParam-.prop` file as described in the next section. The first node that creates the overlay resets the content of the web cache and puts a reference to itself. Other nodes that join this overlay will use the web cache to get a reference to a node already in the overlay. The new joining node will then use this reference to join the overlay and will also add a reference to itself in the web cache.

To setup the web cache you will need a web server that can run PHP. You can use Apache web server for example. Apache comes with most Linux distributions. For Windows you can use for example www.easyphp.org. After installing your web server copy the `Jade/webcache/` folder to your `www` root folder.

### Configuration

The configuration files can be found in the `etc/` folder under the Jade source tree. The `Jade/etc/dks/dksParam.prop` file is used to set the "Arity" $K$ and the number of "Levels" $L$ for DKS routing. The ID space is $N = K^L$. The values for $K$ and $L$ must be the same for all nodes in the same overlay. This file is also used to specify the default IP and port number of each node in the overlay but can be changed in the code. The address of the web cache is also specified in this file. All nodes in a single overlay must share the same web cache. You might need to edit this file and set these values.

Search for `<Path to Jade>` at `Jade/etc` (including sub folders) and replace it with the correct path.

Go to `Jade/etc/oscar/` and edit `bundle-jadeboot.properties` and `bundle-jadenode.properties` to set appropriate values for `jadeboot.registry.host` and `jadeboot.discovery.host`. The JadeBoot is the node that starts Jade and the overlay and currently this node can not fail!

### If you are Disconnected from the Internet

In this case you'll have to copy `Jade/etc/www/repo.jasmine/` folder to your `www` root. Then search the `Jade/etc/oscar/` folder and the `Jade/etc/execute.properties` file for the text `repo.jasmine` and replace the remote url with your local one.

### For Developers Only

If you want to modify Niche/DKS you must place the new dks.jar file in the `Jade/external` folder. If you modify YASS you must place the new yass.jar file in the `Jade/external` folder and if you modified the fractal architecture then you must place the new `yass.fractal` file in the `jade/examples` folder.

If you want to run a new application you must add the JARs to `Jade/external` and the .fractal file to `Jade/examples`. Then search the Jade source tree for the text "yass.jar" and add similar lines for your application. Finally to run it (described in next section) the simple way is to add ant targets for your application similar to "testG4A-Yass-Deploy" and "testG4A-Yass-Start".

## 5.4 Running YASS

Compile Jade using the ant file `Jade/build-src.xml` default target. Then to start the Niche/Jade system you will need a boot node and several nodes (depending on the number of nodes you need). You'll need at least one boot and four nodes to run the demo. You can use the `Jade/build.xml` ant file. Use the `jadeboot` target to start a boot node then use the `jadenode` target several times to create as much nodes as you need. Alternatively you can use the `testG4A-init2N` target to start a boot and a node or the `testG4A-init4N` target to start a boot and three nodes.

For testing it is easier to first try to run the nodes on the same machine just to make sure that everything is working.

After starting the Niche/Jade system you can now deploy the YASS application to the overlay using the `testG4A-Yass-Deploy` target in the `Jade/build.xml` at the JadeBoot node. After deployment start the application using the `testG4A-Yass-Start` target.

You can now test the YASS application by storing/retrieving some files, killing some nodes, and joining some other nodes.

# Chapter 6

# Features and Limitations

## 6.1 Initial deployment

In the current prototype, only deployment of functional components is done through ADL. The rest, forming the group, establishing bindnings, and creating the self-management architecture is done in the YASS StartManager. The start manager is defined in ADL by adding a "definition= "org.objectweb.jasmine.jade.ManagementType"" tag after the component name.

Currently:

- The start-manager must be the first component to be declared in the ADL file.

- All bindings not established through ADL must declared with "contingency="optional"".

## 6.2 Demands on stability

In the first version of the prototype, the stable nodes must be present when the application is deployed, and remain present for the duration of the application lifetime.

## 6.3 Scope of registry

The component registry needed to locate deployed component based on their ADLName is currently only accessible from the boot-node from where the components where initially deployed.

## 6.4 Resource management

Currently there is no real resource management, parts of corresponding functionality is hardwired in Niche to suit the YASS demo. The following section will demonstrate how to set up the system in a valid way concerning the node id:s.

## 6.5 Id configuration

To make the prototype work, some settings files should be present and tuned for the specific scenario at hand to achieve desired behaviour, the *lines* file and the *stableNodes* file, two plain-text files in the main Jade folder which list desired node id:s.

In general, a joining node is assigned a random id from the ring id space, ranging from 0 to N. Especially for the prototype, it is desirable to be able to control the assignment of the id:s.

By specifying a *lines* file for each physical computer used, the nodes started on the computer will pick the id:s read form the lines file, in top down order. Since no two nodes should share the same id, the lines files should be non-intersecting. If more nodes are started on a computer than there are id:s in the lines file, the exceeding nodes will be assigned random id:s.

To achieve a very primitive "resource management" functionality, the same *lines* file is used to specify the amount of available storage each logical node offers the system, simply by typing "Id=AmountOfOfferedSpace".

The *stableNodes* file lists the id:s of the nodes which are assumed to be always present in the system for the duration of the test.

By setting these two files with care, nodes can join and leave the system without disrupting the management. The present requirement is that the id:s of any node joining or leaving has to closely follow an id of a stable node. See below for an example.

### 6.5.1 Id configuration example

The following shows a valid setup to run the YASS demo.
lines-file on:
computer 1: 5000=1, 420000=11
computer 2: 190050=950000
computer 3: 190000=1200000
computer 4: 830000=9500000
computer 5: 830050=9500000
stableNodes-file, on all computers: 190000, 420000, 830000

Listing 6.1: Code to show request-reply behaviour

```
private void fileWrite(String uniqueFileName,
                ComponentId initiator, ...) {

        //Local book-keeping
        ...
        fileWriteAck.fileWriteSucceeded(uniqueFileName, initiator);
}
```

With the above setup, two nodes should be started on the first computer and one on 2, 3 and 4. Thereafter the YASS application can be deployed. To ensure that management is not disrupted, it is the node present at computer 2 that should be stopped to demonstrate the self-* mechanisms. After the node is stopped, a new node can join with id 830050, which will be detected and put to use to replace the lost node.

## 6.6 Limitations of two-way bindings

Two-way bindings are now implemented for one-to-one and one-to-any bindings. Currently the system cannot aggregate multiple replies on a one-to-many binding. To receive multiple acknowledgements from a set of components, the following work-around can be used: Special attention is given interface method signatures; if the last argument of a method corresponds to a valid ComponentId, this will short-cut the ordinary binding, and be used to send directly to that designated component.

An example from YASS is given in 6.1. In the example the component which should reply is given access to the id of the requestor through the request invocation. The same id is then used to reply through the findReplicasAck interface.

## 6.7 Caching

In the current prototype functional components are assumed to be static in the sense of non-movable. Therefore the physical address of the components participating in a group or in a binding are cached by the group or the bind-object respectively. On the other hand there is no caching of group content by group users, which means each name based address is resolved on each use.

## 6.8 Lack of garbage collection

Currently if a group or a management element is removed, not all related bindings and sensors are removed.

## 6.9 YASS limitations

The current prototype does only store "virtual" files: space is reserved inside the storage component, and file metadata is stored, but the files are not actually transfered over the network.

# Chapter 7

# Future Extensions

## 7.1 Initial deployment

Eventually management elements that should be present from initial application deployment time will be deployed through ordinary ADL the same way as functional components. If needed, resource constraints for management elements could then be specified the same way as for functional components.

## 7.2 Resource Management

Useful improvements include the ability to reallocate resources.

<**TODO**> Nikos should contribute here. </**TODO**>

DCMS currently does not verify resource usage by application components. Moreover, DCMS does not yet define the way to specify required resources for use by specific application management elements, and therefore DCMS cannot verify resource usage by individual MEs even if there were a mechanism to do so. However, if resources that are consumed by MEs are reasonable and distributed fairly between different applications executed by DCMS, accounting of resource usage by individual MEs can be unnecessary.

## 7.3 Increased Tolerance to Churn: Joins, Leaves and Failures

The next released version of DCM will allow nodes with arbitrary id:s to join without disrupting existing management.

The following version will include ability to gracefully leave any node, which then might cause temporary delays in management response times, but no permanent disruption.

Before the end of the project we hope to implement transparent replication of management elements for increased robustness, which then will allow also nodes hosting management elements to fail. Replication of management elements will involve refining the meaning of bindings between MEs and functional components as such bindings will connect multiple replicas of the same ME to functional components. The set of ME replicas will act as a group, except that MEs and thus ME groups can provide also client interface(s) which will imply "many-to-one" or "many-to-many" communication patterns. We may find it necessary also to restrict the possible interaction types between MEs and functional components in order to make both the programming model and DMCS implementation reasonably simple and efficient.

## 7.4 Caching

We will implement and test different forms of caching, although the work towards increased efficiency will have lower priority than the work towards increased robustness.

## 7.5 Improved Garbage Collection

We will improve garbage collection when removing management elements from the system, although the work towards better garbage collection will have lower priority than the work towards increased robustness.

## 7.6 Replication of Architecture Element Handles

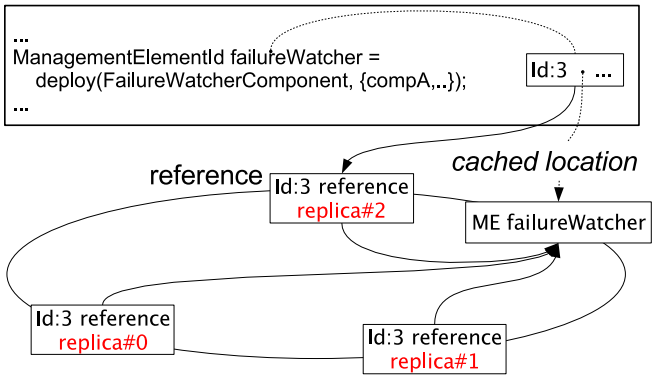


Figure 7.1: Replicated Handles.

The framework will support a network-transparent view of system architecture, which simplifies reasoning about and designing application self-* code. This will be facilitated by replication of internal DCMS entities representing architecture elements, such as references as shown on Figure 3.9. Different MEs access different entity replicas for read accesses. The SNR replication provides eventual consistency of SNR replicas, but transient inconsistencies are allowed. Similarly to handling of SNR caching, the framework recognizes out-of-date SNR references and repeats SNR access whenever necessary.

# Chapter 8

# Conclusions

In this document we introduce a programming model and API implemented by DCMS – distributed component management service. DCMS facilitates developing self-* applications for community-based Grids, as envisioned by Grid4All use cases. Our framework separates application functional and self-* code, and allows to design robust application self-* behaviours as a network of management elements. DCMS exploits a structured overlay network for naming and lookup, communication, and DHT overlay services. DCMS intends to reduce the cost of deployment and run-time management of applications by allowing to program application self-* behaviours that do not require intervention by a human operator, thus enabling many small and simple applications that in environments like Grid4All's community-based Grids are economically infeasible without self-management.

# Glossary

**API** – Application Programming Interface.

**Niche** is a Distributed Component Management Service (DCMS), a service supporting developing component-based self-* services. The Niche framework includes the programming model and the API specification.

**Niche groups** is an abstraction in the Niche programming model that allows the programmer to group together components. One-to-any and one-to-all bindings can be made to a group, and group membership management is independent of bindings to the group.

**DKS** – Distributed k-ary system [8], a structured overlay network.

**Fractal** component model [7] is a model for component-based applications. Components have interfaces that are connected by bindings. Fractal provides for nested components and hierarchical composition, and allows component sharing. The novel feature of Fractal is the provisioning for dynamic introspection, reconfiguration and life-cycle management of components.

**Groups** , see Niche groups.

**Management Events** are objects that are passed between sensors and MEs. There are event classes pre-defined by Niche, and application-specific event classes defined by the application developer.

**ME** – Management Elements. A distributed network of MEs constitute the implementation of self-* behaviours in an application. MEs communicate by means of events, and use the Niche API to manage the application architecture.

**ME containers** encapsulate application-specifc MEs and application-independent ME generic proxies.

**ME generic proxies** are components provided by Niche that are bound to MEs and implement inter-ME communication and management functions. ME generic proxies allow devlopers of Niche-based application to program MEs as regular Fractal components.

**Sensor** is a Niche entity that provides information to application self-* code, implemented as a network of MEs, about status of individual components and the environment. The former type of sensors is developed by application programmer together with application components, and the latter one is provided by Niche.

**Id,Identifier** is a concept in Niche. Id:s uniquely identify implementations and representations of elements of the application's architecture, like components, groups and bindings. Internally in Niche, Id:s are used to address all kinds of entities that are shared by multiple nodes and MEs, in particular – Niche nodes themselves and Niche reference entities.

**SNR** – Set of Network References, an abstraction provided by Niche that allows to maintain and monitor a set of references to entities on the overlay.

**Subscription** is an asynchronous communication channel between a pair of sensors or MEs for a specific type of management events.

**GCM** – Grid Component Model, a refinement of the Fractal component model. Provides for group communication through "collective interfaces", and facilates construction of autonomous component through "behavioural skeletons".

**P2P** – peer-to-peer networks and systems.

**Synchronized MEs** – replicated MEs that are maintained in a consistent (synchronized) state, such that such MEs can be seen as hosted on reliable (failure-free) computers.

# Index

# Bibliography


[1] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand. Enabling self-management of component based distributed applications. In *Proceedings of CoreGRID Symposium*, Las Palmas de Gran Canaria, Canary Island, Spain, August 25-26 2008. Springer. To appear.

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

[3] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Co., Greenwich, CT, USA, 1997.

[4] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications.* Springer, 2005.

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

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

[7] E. Bruneton, T. Coupaye, and J.-B. Stefani. The fractal component model. Technical report, France Telecom R&D and INRIA, February 5 2004.

[8] Distributed k-ary system (dks). http://dks.sics.se/.

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

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

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

[12] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[13] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[14] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[15] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. *ACM SIGOPS Operating Systems Review*, 40(4):103–115, 2006.

[16] D. Malkhi, F. Oprea, and L. Zhou. *Omega* meets Paxos: Leader election and stability without eventual timely links. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference (DISC' 05*, volume 3724 of *LNCS*, pages 199–213, Cracow, Poland, September 26–29 2005. Springer.

[17] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[18] P. Zielinski. Low-latency atomic broadcast in the presence of contention. *Distributed Computing*, 20(6):435–450, 2008.