Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)
Thematic Priority 2: Information Society Technologies

D1.5: Full Specification and Final Prototypes of Overlay Services

Due date of deliverable: June 2009.

Actual submission date: 20 July 2009.

Start date of project: 1 June 2006 Duration: 30 months

Organisation name of lead contractor for this deliverable: SICS

Revision: 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level

PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contents

Abbreviations and acronyms used in this document

Abbrev./acronym	Def.	Description
VO		Virtual Organisation
DCMS		Distributed Component Management System
ADL		Architecture Description Language

Grid4All list of participants

Role	Part. #	Participant name	Part. short name	Country
CO	1	France Telecom	FT	FR
CR	2	Institut National de Recherche en Informatique en Automatique	INRIA	FR
CR	3	The Royal Institute of technology	KTH	SWE
CR	4	Swedish Institute of Computer Science	SICS	SWE
CR	5	Institute of Communication and Computer Systems	ICCS	GR
CR	6	University of Piraeus Research Center	UPRC	GR
CR	7	Universitat Politècnica de Catalunya	UPC	ES
CR	8	ANTARES Produccion & Distribution S.L.	ANTARES	ES

1 Introduction

Deployment and run-time management of applications constitute a large part of software's total cost of ownership. These costs increase dramatically for distributed applications that are deployed in dynamic environments such as unreliable networks aggregating heterogeneous, poorly managed computing resources. A notable example of such environments are community-based Grids where individuals and small organizations create ad-hoc Grid virtual organizations (VOs) that utilize otherwise unused computing resources to mutual benefits. Community-based Grids are meant to provide "best-effort" services to their participants, but because of the nature of their resources cannot provide more strict quality-of-service (QoS) guarantees. Community-based Grids fill the gap between high-quality Grid environments deployed for large-scale scientific and business applications, and existing peer-to-peer systems which are limited to a single application.

Application and Grid system management by humans renders many small and simple applications impossible or economically infeasible to run in such dynamic environments. Grid VO members join and leave sporadically, and their resources go frequently on- and off-line. Applications have to deal with resource and software failures, availability of new resources, and changes in load on applications and/or their performance requirements. Adjusting applications according to the aforementioned factors can either consume too much time of human administrators, or be physically impossible to perform. This issue is largely non-existing in conventional Grid systems where resources are stable of high-quality, and QoS requirements are fixed for a long period of time.

The autonomic computing initiative [18, 21] advocates autonomic applications and systems as a way to reduce the management costs of such applications. Autonomic systems can manage themselves independently (autonomically) without outside control, in particular by humans. Key capabilities of autonomic systems are self-configuration, self-healing, self-optimization and self-protection, commonly referred to as self-*. With the architectural approach to self-* management [17], autonomic systems have a hierarchical architecture where each element is autonomic on its own. Autonomic elements also need to provide certain introspection interfaces like status inquiry that enable self-* behaviours of higher-level elements in the application architecture. Self-* behaviours in an autonomic element can be implemented by feedback control loops that collect and aggregate information about elements of the architecture and the environment, and uses this information to update the element's architecture according to its algorithm of autonomic management. The architectural approach to self-* management of component-based applications has also been shown useful for retrofitting self-repair capabilities into legacy applications [8].

This document presents Niche – a distributed component management service (DCMS) and its application programming interface (API) designed to faciliate programming distributed self-* applications for dynamic environments introduced above. Niche intends to reduce the cost of deployment and run-time management of applications by allowing developers to program application self-* behaviours that do not require intervention by a human operator.

1.1 Context

A high-level description of the specific challenges in Grid4All dynamic and volatile environments and how Niche meets them is presented in deliverable 6.7A, the white paper on Niche. The focus in this deliverable is on describing what Niche is and how the application developer can make best use of Niche.

1.2 Introduction to Niche

Our framework separates functional and self-* parts of the application architecture. The functional code implements the application behaviours according to the functional requirements of the application. The self-* part implements the self-* behaviours according to the non-functional requirements such as quality of service and scalability, failures, dynamically changing application load and adapting to changes in the operation environment. The framework facilitates programming fault-tolerant scalable applications since both functional and non-functional parts are distributed.

The functional code of applications is developed in an extended Fractal component model [10]. In Fractal, application architectures are composed from *components* that encapsulate behaviours and/or nested sub-components, and exposes *interfaces* with well-defined types. Component interfaces are interconnected by *bindings*. Niche implements location-independent communication between distributed components, facilitating separation of concerns between application development and deployment on particular sets of networked resources. In addition to that, Niche introduces component groups and bindings to groups. Groups enable the "one-to-all" and "one-to-any" communication patterns that are useful for building scalable, fault-tolerant and self-healing applications [9, 1]. For example, component failures in a group can be handled independently of the functional code by a ME responsible for self-healing.

Self-management is expected to be implemented as control loops that receive events from sensors, analyze events, generate actuation plans and finally perform the actuation. Acutation plans reconfigure the application architecture. Self-management is written by application developers, as it is application dependent and Niche supplies the application developer with the necessary support to achieve this. The architecture of the application's self-* part, or just *self-* architecture* thereafter, is organized as a distributed network of application-specific *management elements* (MEs). MEs react to input events from sensors or other MEs, and generate output events and/or manage application architecture. MEs are implemented by application developer as Fractal components. Niche provides the environment for ME execution and inter-ME communication. MEs in the self-* architecture *sense* changes in the environment by means of events generated by Niche or by application-specific *sensors*, e.g. component state changes. MEs can *actuate* changes in the architecture – add and remove components and bindings between them.

Applications using our framework rely on external resource management providing resource discovery and allocation services. Niche does not manage resources on its own. Clearly, resource management is a VO-wide service and needs to arbitrate when different applications compete for scarce resource. In the Grid4All environment resource management is provided by Virtual Organization (VO) resource management services. Niche API functions related to component life-cycle management operate using *resource* entities which encapsulate the physical resources necessary for component deployment and execution. Separation of concerns between distributed components and MEs managed by Niche and resource management allows system designers to use Niche in different Grid environments.

Niche implements a distributed infrastructure that interconnects computers providing resources for execution of self-* component-based applications. Computers in Niche infrastructure execute Niche processes that act as containers for elements of application achitecture. Since Niche infrastructure aggregates the same unreliable computing resources, Niche is self-organizing and self-healing upon resource churn (volatility). Specifically, Niche can dynamically remove and add new computers to the infrastructure, and tolerate unexpected failures. Niche is implemented on the DKS overlay network [9] providing for scalable fault-tolerant address lookup operation, and for network sensing functionality that Niche provides to application self-* architecture. Thus, the Niche framework allows application developers to exploit the self-* properties of structured overlay networks using a simple programming model that specifically targets designing application self-* architecture, and hides unnecessary details of management of a P2P infrastructure.

The first contribution of our work is a simple yet expressive self-* management framework. The framework provides *(a)* network-transparent distribution of architectural elements, yet it *(b)* allows applications to be network-aware. The framework provides also *(c)* high-level abstractions for designing application self-* architecture.

Network-transparency is a property of communication systems and middleware when communicating parties are not aware of the fact that they interact over a network. Network-transparency enables the programming of both functional and self-* parts of applications independently of particular application deployment configurations. The configurations change due to availability of computing and networking resources. Execution of MEs is location-independent, i.e. inter-MEs communication is network-transparent and effects of actuation commands do not depend on which node they are issued. In particular, Niche implements application-transparent migration of MEs when the nodes they are running on are about to leave, and when new resources join to achieve better load balance.

Network-awareness is another property of communication systems and middleware when communicating parties can be notified about events in the network that are of importance for them, and can control the ways network is used for their interaction. Usually network-awareness complements network-transparency. Niche gives the programmer a useful degree of network awareness by allowing it to control the co-location of architecture elements, which can improve the performance of self-management and simplify handling of failures of nodes hosting management elements.

Niche relieves application programmers and system administrators from low-level details of implementing and managing the functional and self-* parts of applications. In particular, it provides high-level deployment and component sensing abstractions hiding low-level details of dealing with computing and networking resources. Furthermore, Niche provides for flexible and application-transparent replication of management elements, effectively giving the programmer a conceptually simple platform for programming robust self-management architecture.

Our second contribution is the implementation model for our churn-tolerant management platform that leverages the self-* properties of a structured overlay network.

A general model for ensuring coherency and convergence of distributed self-* management is out of scope of this work. We believe, however, that our framework is general enough for arbitrary self-management control loops, and can be used for implementing higher-level abstractions for component-based self-managing applications, such as behavioural skeletons [2] in the GCM component model [15]. Our example application demonstrates usability of our approach in practice.

We proceed to gradually introduce the role of Niche in the context of Grid4All, the related work, and then the concepts of Niche without the burden of full details of its API and current limitations. Information presented in this deliverable should suffice to understand the formal API description in Niche API document, the YASS demo application example that is bundled with the Niche distribution, and the discussion of features, limitations and future Niche extensions in the Niche documentation. The presentation here is informal, and particular syntax of examples can stray from the existing Niche and YASS code for the sake of presentation clarity.

1.3 Outline of Deliverable

In section 2 we show Niche is positioned in the Grid4All architecture, and how it interacts with other VO-services. In section 3 we describe the Niche programming model. This section is written as a series of executive summaries of various programming aspects. The full details are given in the appendices. Finally in section 4 we discuss related work.

2 Niche in Grid4All

The Grid4All project aims to prototype Grid software building blocks that can be used by non-expert users in dynamic Grid environments such as community-based Grids. In these environments, computing resources are volatile, and possibly of low-quality and poorly managed. Because of the dynamic nature and ad-hoc, peer-to-peer styles of creating virtual organizations in community-based Grids, availability and consumption of computing resources cannot be coordinated. Networking can be slow and unreliable, and some computers can reside behind firewalls restricting inbound and outbound traffic to some but not all other computers in the Grid. Finally, typical users of community-based Grids are believed to have neither time nor qualification for managing their applications and the Grid system itself.

Both the applications and the Grid (i.e. Niche and other platform services) must be self-managing to achieve this vision. In particular, they must automatically adjust themselves to available resources and load demands that change over time, self-repair itself after hardware and software failures, protect themselves from security threats, and at the same time be reasonably efficient with resource consumption.

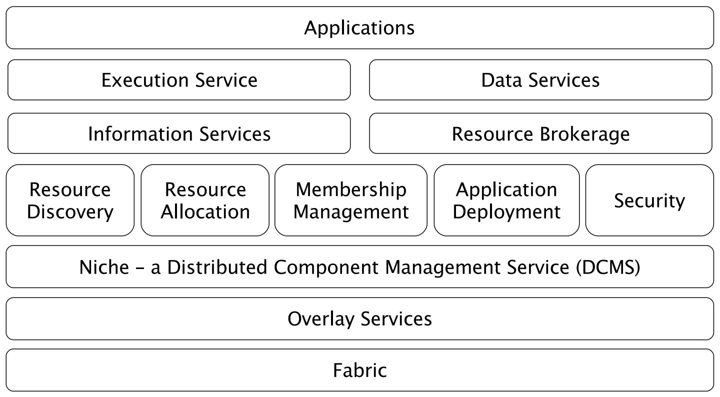


Figure 1: Niche in the Grid4All Architecture Stack.

Niche and Overlay Services are two of the Grid4All building blocks, depicted in figure 1. Niche allows developers to design and implement self-* component-based services that can be executed in the Grid4All environment using the Grid4All's Resource Management services. There can be multiple instances of the Niche run-time system in the same Grid4All Virtual Organization, and each instance can host multiple applications. Resource Management Services – Resource Discovery and Allocation – are part of Grid4All Virtual Organization (VO) Management services, along with Membership Management, Application Deployment and Security.

Niche uses the Overlay Services to implement the distributed Niche run-time infrastructure. Computers participating in the VO (Grid fabric) execute Niche processes that form together the self-* Niche infrastructure. Each Niche process can act as a container for elements of different applications. Overlay Services provide for scalable fault-tolerant address lookup operation, distributed hash tables and network sensing behaviours. Niche processes host, execute and provide services for elements of the application's architecture. Niche-based applications are in general distributed on a number of computers in the VO.

Resource Discovery&Allocation Services also utilize the Overlay Services. A basic implementation of these services is bundled with the Niche run-time system that provides a simple first-in-first-serve resource management policy. The main purpose of this basic implementation is to allow developers to test their applications. Other VO services can as well be (re)written using Niche.

Niche-based applications contain a part that implements the application's functional specification, and another part – self-* architecture – that provides the self-* behaviours, as outlined in figure 2. The functional part contains a number of interconnected components, and in general it can continue to operate

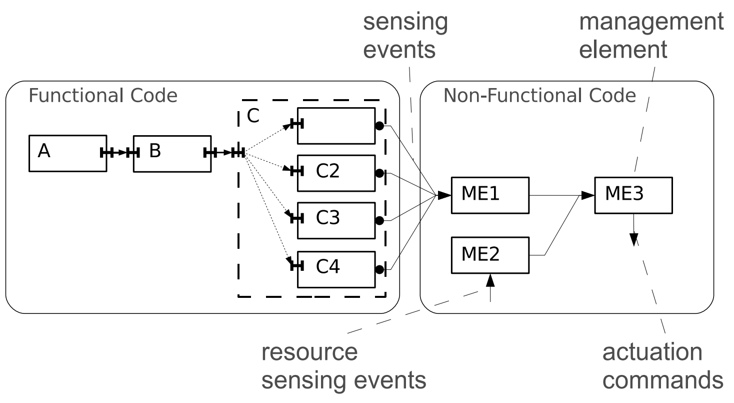


Figure 2: Two Parts of Niche-based Applications.

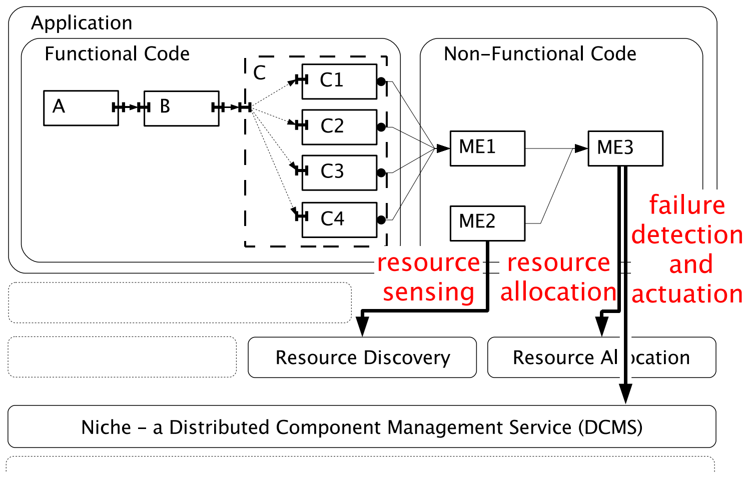


Figure 3: Management Elements Utilizing Grid4All Services.

without any changes as long as its constituent components are operational, the properties of the environment such as QoS of networking do not change, and requirements on the service such as service capacity are fixed. The non-functional part can *adapt* the functional part to such changes by means of reconfiguration, i.e. creating, removing and re-connecting components. The extent of capabilities of self-* architecture is determined by the application developer when he programs it. The non-functional part contains a number of interconnected *management elements* (MEs) that receive *sensing* information about the functional part and computer failures in the environment, and can upon need *actuate* changes in the application's architecture Sensing and actuation services are implemented by Niche using the Overlay Services that link together and monitor computers in the Niche infrastructure, see figure 3. Both functional and non-functional parts of Niche-based applications are distributed and therefore potentially scalable and fault-tolerant.

Execution of Niche-based services in the Grid4All environment, see figures 3 and 4, relies on Resource Management services which, in turn, can depend on other Grid4All VO Management Services. Using the Resource Management services, the application's management elements (MEs) can discover and allocate *resources* necessary for component deployment and execution. Once resources are obtained, the MEs use the Niche to deploy application components on them. The relationship between Niche and external resource management services is discussed further in Section E. In turn, the operation of VO management services is supported by Overlay Services that provide for fault-tolerant, scalable and

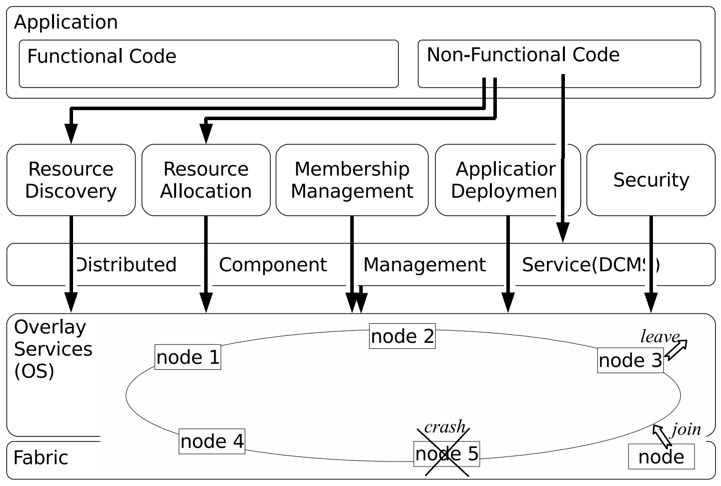


Figure 4: Niche and VO Services Executed on Overlay.

efficient communication between computers in the VO.

VO Management services provide also the Application Deployment Service that performs initial deployment of applications given their description in the high-level Architecture Description Language (ADL). An ADL description specifies application's components, their resource requirements and their interconnections. The Deployment service utilizes the VO Resource Management services and Niche to allocate resources, deploy and interconnect components according to the ADL specification.

3 Niche Outlined

The first thing the application developer needs to understand to develop self-managing distributed applications is how to model management, how to separate management code from functional code, and how they interact. This interaction requires that the functional code is properly encapsulated and structured, in an application architecture. Structured encapsulation (component-based) has been found to be useful in developing software for centralized applications. For distributed applications it is not only useful but necessary as the interaction between components on different machines need to be made visible. Components need hooks in order that management can control them and thereby the entire application architecture. Niche is based on the Fractal component model and currently the application developer must develop his functional code using it. Potentially Niche could be adapted to other component models that provide similar hooks and triggers. The programmer needs to understand the control loop paradigm and that his task in programming the management part of an application is to react to sensing events and, if needed, to react by issuing appropriate actuation commands. The programmer also needs to understand the basics of sensing (monitoring) and actuation. These basics are described in appendix A.

The application developer needs to understand how to structure the management part of an application. An important property of Niche is that management need not be centralized. Indeed, decentralization of management is crucial for achieving efficient and robust management in Grid4All dynamic environments (as described in deliverable 6.7A). The management of a distributed application may, for instance, be programmed hierarchically. Management may also be divided by aspects. For instance, one manager deals with self-healing (recovering from failure) and another deals with self-tuning (e.g. load-balance). This is described in appendix B.

Management code is composed of management elements (or components) as described in appendix B. The details of how these elements can be connected (and how they communicate through events) to one another as well as to functional components is described in appendix C.

In appendix D we describe how architectural elements are identified, pointing out that Niche provides the VO-wide identifiers when these elements are created. An important aspect of Niche is that these identifiers are location-transparent, and Niche ensures that messages (events) are delivered (at least once) even when management elements move. In this appendix we also describe one way to achieve initial deployment of applications.

In appendix E we describe resource management, and the relationship between Niche and the VO resource management services.

In appendix F we describe component groups. The reason that Niche directly supports component groups as primitives rather than to provide a subsystem library is for efficiency, in particular, greatly reduced messaging overhead. By providing groups as primitives, we can provide efficient group sensing and actuation. Groups are very common in applications, where the size of the group varies according to load or resource applicability. The notion of groups is integrated in the component model, with two different kinds of bindings, one-to-any and one-to-all.

In appendix G we describe how the application developer can control the location of management elements. The general model is that during the deployment of an application the functional elements are deployed on suitable discovered and allocated resources. Management elements are also deployed by Niche, but this is done transparently with no programmer directed resource discovery and allocation necessary. In general (and in all our case studies) management consumes very few computing resources, as they wait passively until some event triggers. When triggered they will performs some small computation and perform some actuation, and then become passive again. A small slice of each machine in a VO is reserved for application management needs. However, particularly in a VO with heterogeneous network latencies, location of management elements may be important, e.g. by keeping management elements close to those components that it is monitoring. This is an interesting research topic but one that we did not tackle. However, we do provide for one special case and elements can be co-located.

In appendix H we describe how management elements can be replicated for fault-tolerance.

Finally in appendix I we describe how Niche was implemented and how it is makes use of the underlying structured overlay network.

On the Niche web site (http//niche.sics.se) there is more material. First there is a quick start guide, with a hello world application, and information as how to install Niche in a VO. The programming guide includes a primer on the Fractal programming model and a detailed specification of the Niche API. In addition the application YASS (Yet Another Storage Service) is presented in detail illustrating how Niche was used and the reasoning behind how it was structured. In addition one chapter is dedicated to features and limitation. Finally, the web site contains user guides on some applications that have been developed using Niche, including information as how to install and run them.

4 Related Work

As our work on Niche advocates a particular programming model for distributed self-* component-based applications, we relate our work to research on autonomic computing in general and in research about component and programming models for distributed systems in particular.

Autonomic Management. The vision of autonomic management as presented in [18, 21] inspired a considerable industrial and academic interest in self-* systems[4]. In many solutions the support for self-management is added through a centralized manager. One of the approaches that tries to add some support for decentralized fault-tolerant self-management is Unity [11]. In the Unity framework, application self-healing and self-configuration are enabled by designing system components as autonomic elements responsible for their own self-management. Autonomic elements self-assemble into complete self-managing systems based on their high-level roles using the "goal-driven self-assembly" technique. Policies governing the autonomic behaviours of individual elements are stored in *policy repositories*. The Unity framework proposes to organize autonomic elements into *clusters* that provide for self-healing: when one of the elements in a cluster fails, it is restored using the information provided by remaining elements. The Unity project applied the concept of element clustering to self-healing policy repositories. Compared to Unity, Niche provides a simple and concrete programming model for programming arbitrary distribution patterns of application self-* architecture, which is not limited to policy-based self-management using policies from replicated policy repositories.

Biological systems inspired several design patterns for autonomous systems [3], in particular for load balancing (e.g. [29]), managing large-scale networks ([19]) and robust aggregation of information in volatile distributed environments ([20]). Similar patterns are also deployed in wireless sensor/actuator networks that also require self-management for successful operation in volatile environments.

Some approaches, notably ICENI [28], Automate [32] and GRIDKIT [16] rely on P2P to support some of the self-* aspects in Grid systems such as self-healing. Our work also aims at a high-level yet simple programming model for developing self-* applications that benefits from P2P technologies used in the implementation of its run-time system. In our programming model, application developers deal with high-level concepts such as "management elements" and "sensors" as opposed to P2P's "nodes" and "messages".

For example, the AutoMate project [32] proposes a programming framework called Accord to deal with challenges of dynamism, scale, heterogeneity and uncertainty in Grid environments [25]. Accord is based on the concept of an autonomic element that represents any (self-)manageable entity, e.g. resource, component, or object. Self-* behaviours are separated from application's functional behaviours. The autonomic management in the Accord framework is guided by rules of two types: (i) behaviour rules, which control functional behaviours of autonomic elements and applications; and (ii) interaction rules, which control the interactions between elements and their environments and coordinate an autonomic application. The Accord framework provides support for run-time composition of elements and run-time injection of rules. Rules are triggered by sensing data and executed by the Accord's run-time system. Accord suggests also an elaborated scheme for conflict resolution in rule execution. In the AutoMate software stack, dynamic composition of autonomic elements is performed using a coordination framework called Rudder [24] that contains a core coordination substrate providing messaging services, and a multi-agent infrastructure of the peer element managers and a composition manager. In [25], authors illustrate how the Accord framework can be used to realize each of the four aspects of self-management: self-configuration, self-optimization, self-healing and self-protection.

Our approach complements the AutoMate's architectural stack: AutoMate's self-* rules can be implemented using our framework. Our work proposes a more generic and conceptually far simpler programming model, as opposed to a more specialized and restricting, rule-based one in Accord. Our model focuses on ability to program fault-tolerant and scalable management architecture by means of explicit distribution by the programmer and implicit replication provided by Niche, and can also be used to provide self-* management support for legacy applications.

Autonomic management of distributed architectures is enabled by ability to manage its networked elements. In particular, the Web Services Distributed Management (WSDM) framework [34] provides such a standardized specification for Web Services. If the Niche model would be adapted for retrofitting self-* capabilities to existing web-based applications, the Niche actuation services could be implemented according to the WSDM specification.

Self-* behaviours and in particular self-healing of a software architecture usually involves checkpointing – saving the state of architecture's elements on regular intervals, and re-using the saved state when the element needs to be restored after a crash. Recent work on distributed checkpointing in Cliques [14] demonstrates how to store checkpoints in a distributed fashion in order to reduce load on managers in a self-* architecture which then do not have to store checkpointing information and therefore posess more capacity to deal with management issues. Such methods can be introduced in our framework to support stateful applications.

Component Models. Among the proposed component models which target building distributed systems, the traditional ones, such as the Corba Component Model or the standard Enterprise JavaBeans were designed for client-server relationships assuming highly available resources. They provide very limited support for dynamic reconfiguration. Other component models, such as OpenCOM [12], allow dynamic flexibility, but their associated infrastructure lacks support for operation in dynamic environments.

The Grid Component Model, GCM [15], is a recent component model that specifically targets grid programming. GCM is defined as an extension of Fractal and its features include many-to-many communications with several different semantics, and autonomic components. GCM defines simple "autonomic managers" that embody autonomic behaviours and expose generic operations to execute autonomic operations, accept QoS contracts, and to signal QoS violations. However, GCM does not seem to offer mechanisms to ensure the efficient operation of self-* architecture in large-scale environments. We are also not aware of GCM publications that describe GCM implementation models. Thus, the work on GCM can be seen as largely complementary to our work and thanks to the common ancestor, we believe that our results can be exploited within a future GCM implementation. *Behavioural skeletons* [2] aim to model recurring patterns of component assemblies equipped with correct and effective self-management schemes. Behavioural skeletons are being implemented using GCM, but the concept of reusable, domain-specific, self-management structures can be equally applied using our component framework.

GCM also defines collective communications by introducing new kinds of cardinalities for component interfaces: multicast, and gathercast [5]. This enables *one-to-N* and *N-to-one* communication. However GCM does not define groups as a first class entities, but only implicitly through bindings. Therefore GCM groups can not explicitly addressed and managed by management elements. GCM also does not mention how to handle failures and dynamism (churn) and who is responsible to maintain the group. Our one-to-all binding can utilise the multicast service, provided by the underlying P2P overlay, to provide more scalable and efficient implementation in case of large groups. Also our model supports mobility so members of the group can change their location without affecting the group.

A component model designed specifically for structured overlay networks and wide scale deployment is p2pCM [31], which extends the DERMI [30] object middleware platform. The model provides replication of component instances, component lifecycle management and group communication, including *anycall* functionality to communicate with the closest instance of a component. The model does not offer higher-level programming abstractions such as sensors, group watchers and application-transparent replication of management elements, and the support for self-healing and issues of consistency are only partially addressed.

Our work builds on the technical work on the Jade component-management system [8]. Jade implements the Fractal component model specification [10]. Jade utilizes the Java RMI, and is limited to cluster environments as it relies on small and bounded communication latencies between nodes and centralized management. Our work on Niche focuses not only a high-level yet simple programming model, but also on the implementation model of the P2P-based Niche run-time system and its performance.

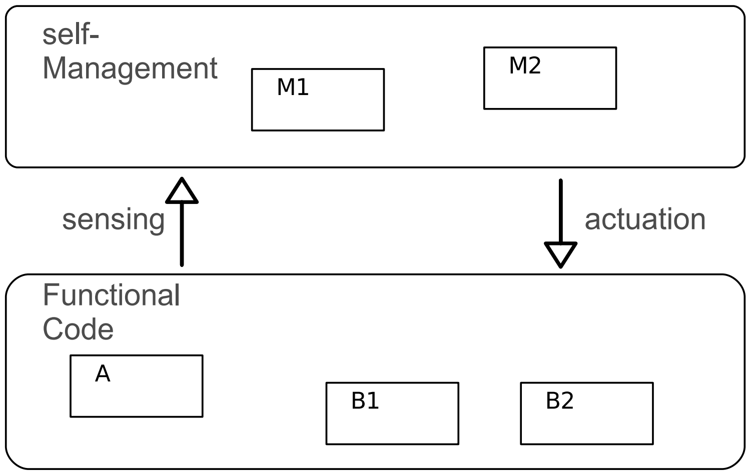


Figure 5: Application Architecture with Niche.

A Basics

A.1 Functional Code and Self-Management

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 5, 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 5, 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 I.

A.2 Component-Based Functional Code

The functional part of application architectures is composed using *components* and *bindings*, see Figure 6, which are introduced in the Fractal component model and discussed in detail in Niche programming guide. A Fractal component 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.

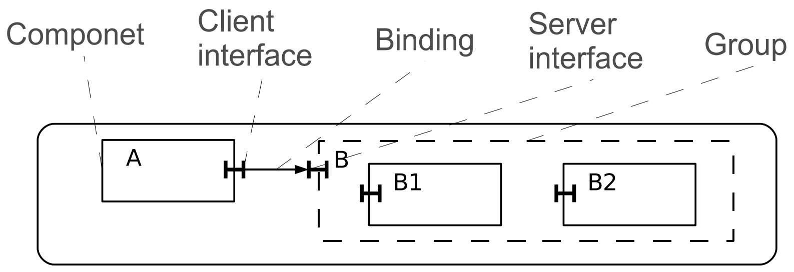


Figure 6: Functional Code in a Niche-based Application.

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 F.

A.3 Application Self-Management by Control Loops

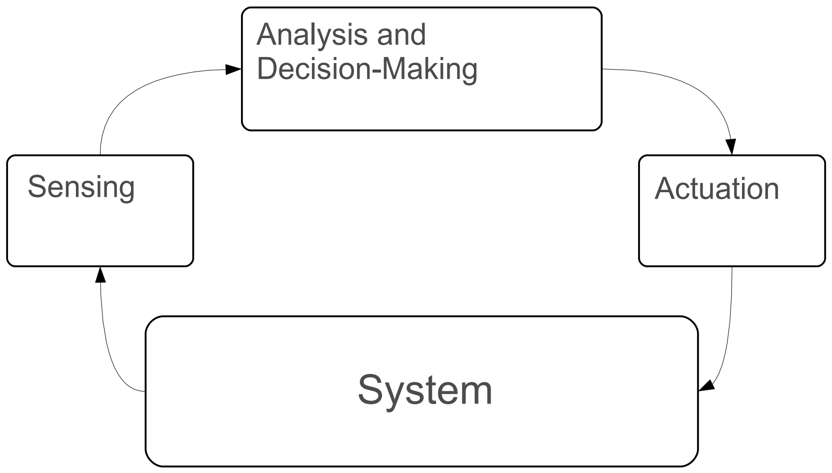


Figure 7: Management Control Loops.

*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 7). 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 B.2.

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

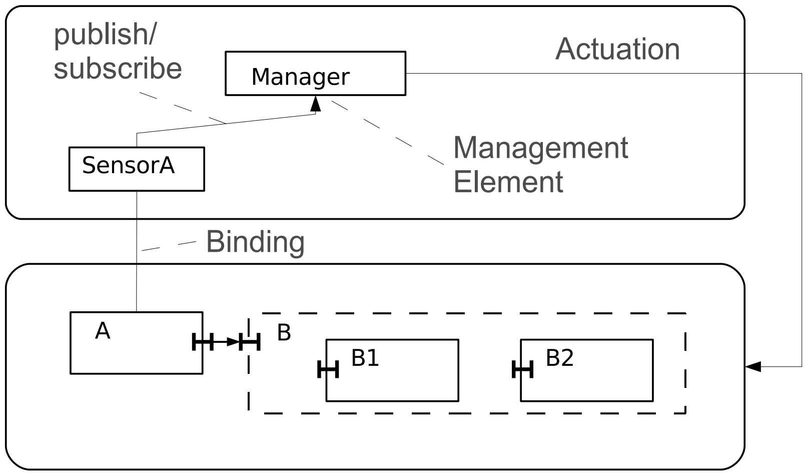


Figure 8: Self-Management in a Niche-based Application.

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 8, `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 C.2), 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 B.2. On Figure 8, `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.

B Structuring Management

B.1 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 B.2 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 8, 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 F. 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 capacity watchers monitor status and capacity of corresponding groups of storage elements.

B.2 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.

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 9.

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.

B.3 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.

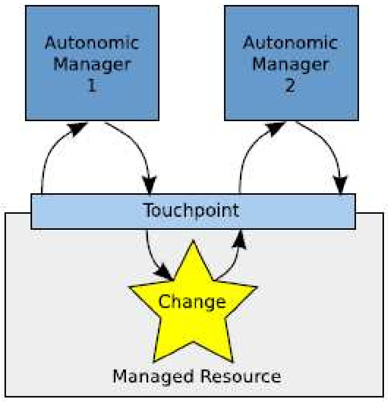


Fig. 1. The stigmergy effect.

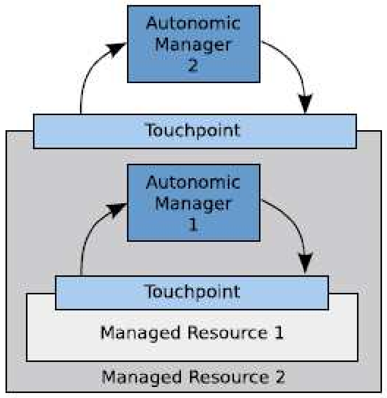


Fig. 2. Hierarchical management.

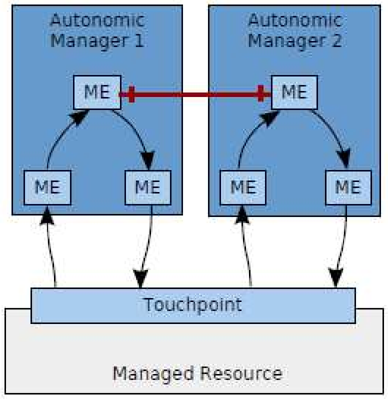


Fig. 3. Direct interaction.

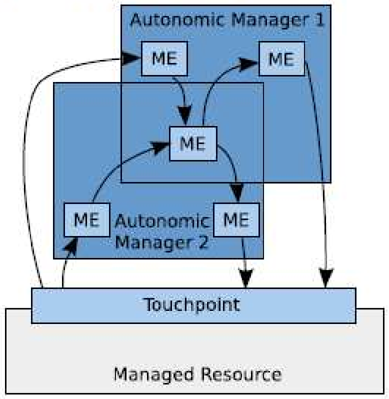


Fig. 4. Shared Management Elements.

Figure 9: Orchestration of Multiple Control Loops.

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 [6, 7] 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 B.2. Fault-tolerance of self-management can be also improved by replicating individual management elements, as discussed in Section H.

C Sensing and Events

C.1 Management Elements and Sensors with Niche

Management Elements are programmed by the application developer as regular (centralized) Fractal components (Figure 10). 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

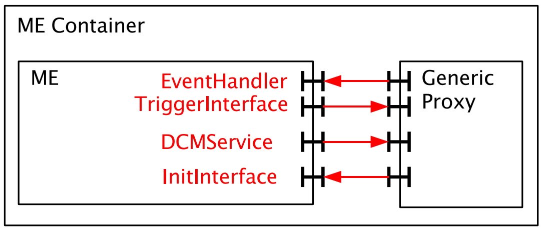


Figure 10: MEs in Niche.

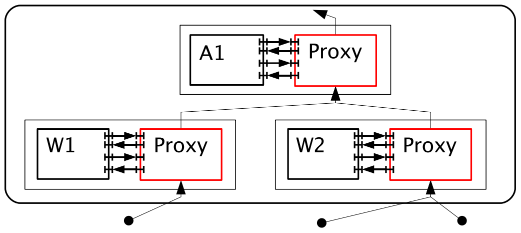


Figure 11: Composition of MEs.

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 10). 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 H.

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 D.1. ME generic proxies provide for communication between MEs, see Figure 11. 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 E. Niche attempts to evenly balance the load of ME hosting.

Application-specific ME components can have the following client interfaces (see also figure 10):

- `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 12, 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.

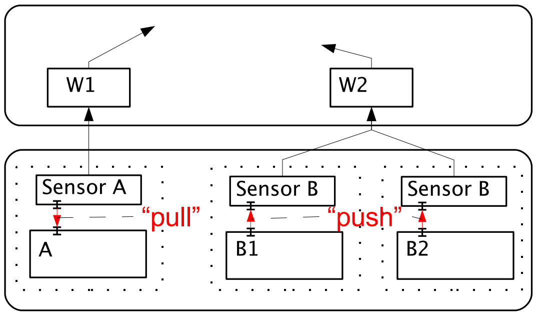


Figure 12: Structure of Application-Specific Sensors.

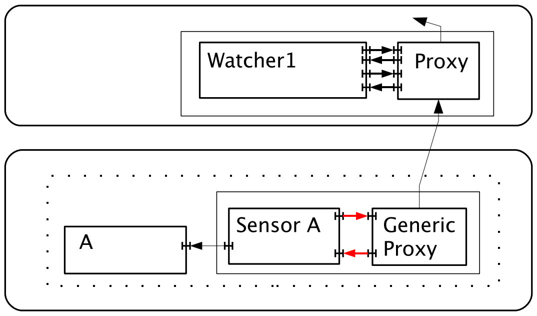


Figure 13: Composition of Application-Specific Sensors.

Sensors are deployed as a two-part structure similar to MEs, see figure 13. 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 :

- `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 13). Using the facilities of the Fractal component model [10], application developer also specifies two lists of interfaces – for information "pull" and "push" between the sensor and the component being sensed (see figure 12), and Niche uses this information to connect the named application-specific sensor component interfaces to matching interfaces of the component being sensed.

C.2 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.

D Architectural Representation

D.1 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 I.

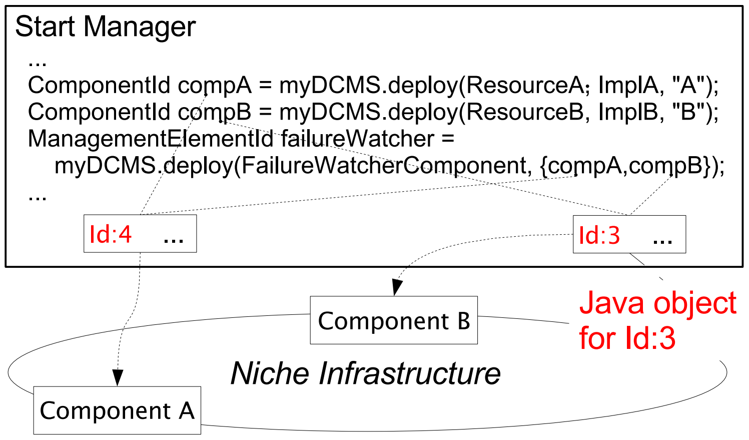

Start Manager
...
ComponentId compA = myDCMS.deploy(ResourceA, ImplA, "A");
ComponentId compB = myDCMS.deploy(ResourceB, ImplB, "B");
ManagementElementId failureWatcher =
myDCMS.deploy(FailureWatcherComponent, {compA,compB});
...
Id:4 ...
Id:3 ...
Java object for Id:3
Component B
Niche Infrastructure
Component A

Figure 14: Application Architecture Representation in Self-* Code.

In figure D.1 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 D.1, `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 E. 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 D.1, 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 I.

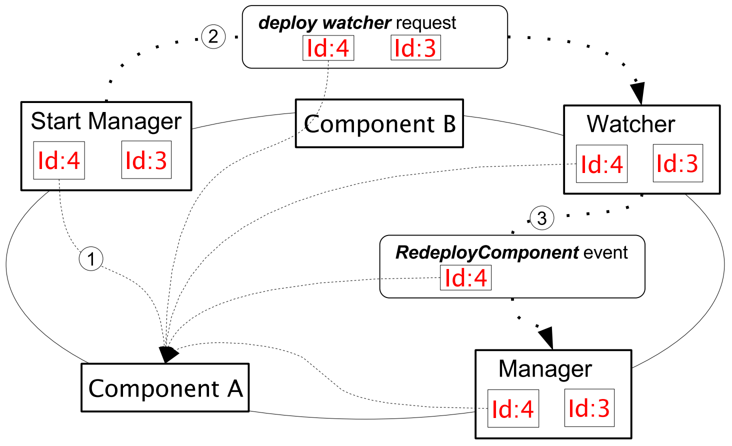


Figure 15: Sharing Niche Id:s between distributed MEs.

A possible sequence of actions and events is illustrated on FigureD.1. 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 D.2 using the code for initial deployment as an example.

D.2 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 16: 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 D.2, 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 16, 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 E. 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 D.2. 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");
```

E 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 C.1.

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 17. 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 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.

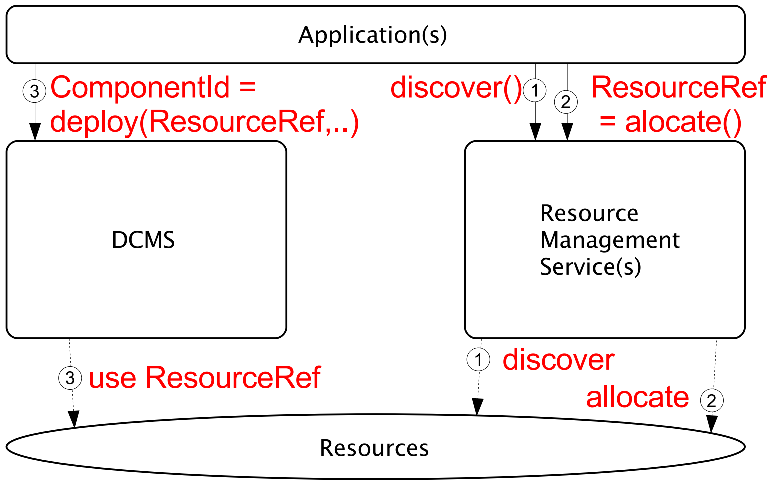

Application(s)
3 ComponentId = deploy(ResourceRef,..)
discover() 1
2 ResourceRef = alocate()
DCMS
Resource Management Service(s)
3 use ResourceRef
1 discover
allocate 2
Resources

Figure 17: Resource Management with Niche.

One of the arguments of the “deploy” Niche API function that provides for component deployment (see Section D.2) 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.

F 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 [10]. 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 18: 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 I. In figure 18, 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 C.1.

G Controlling Location of Management Elements

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 19. 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.

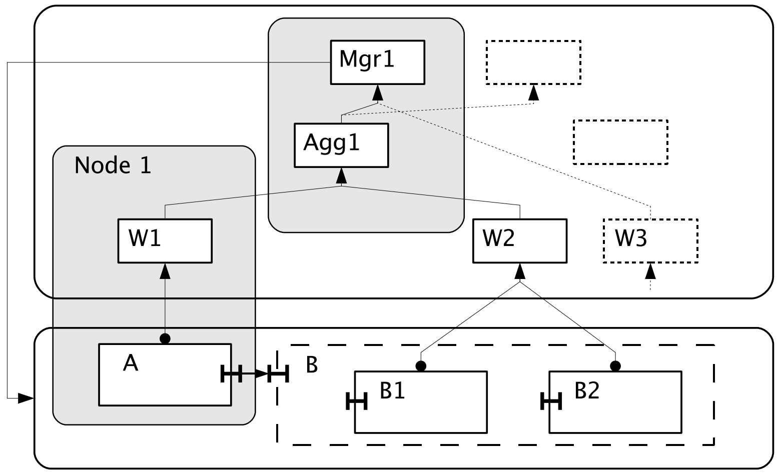


Figure 19: Co-location of MEs.

H Replication of Management Elements

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 C.1) each hosting a replica of the ME, as discussed in more detail in Section I.

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 [23, 33].

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.

I 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.

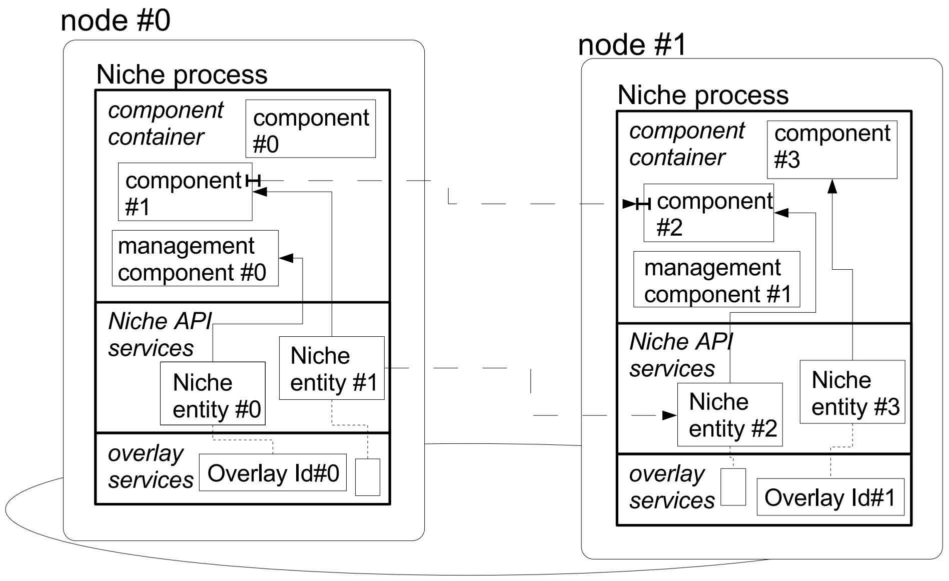
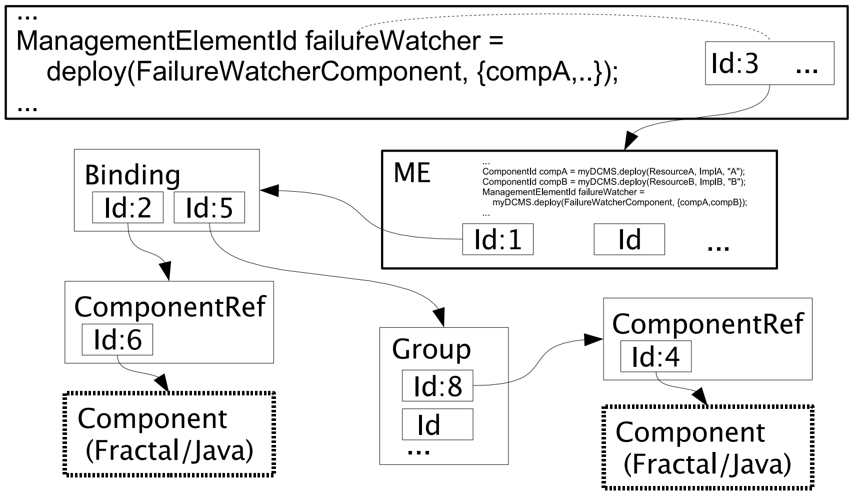


Figure 20: Niche infrastructure.

At runtime, Niche infrastructure integrates Niche container processes on several physical nodes using an overlay middleware, DKS [9, 13] in our current prototype. On each physical node there is a local Niche process that provides the Niche API to applications, see figure 20. 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 *Niche entities*, in particular components of the application architecture and internal Niche entities maintaining representation of the application's architecture. Niche entities are distributed on the overlay. For example, in figure 20 "Niche entity #0" represents the management component #0. Niche entities can have references between each other, direct or indirect. For example, in the figure the "Niche entity #1" can refer to "Niche 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.

Niche entities are identified by *Niche Id:s*. A Niche Id contains an overlay identifier, "Overlay Id" in the figure 20, and a further local identifier. The local identifier allows Niche to distinguish multiple entities assigned to the same overlay identifier. Note that Niche Id:s act as both unique identifiers and addresses of entities in Niche. In particular, Niche 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 Niche 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.



```
...
ManagementElementId failureWatcher =
   deploy(FailureWatcherComponent, {compA,..});
...
```

```
...
ComponentId compA = myDCMS.deploy(ResourceA, ImplA, "A");
ComponentId compB = myDCMS.deploy(ResourceB, ImplB, "B");
ManagementElementId failureWatcher =
   myDCMS.deploy(FailureWatcherComponent, {compA,compB});
...
```

Figure 21: Id:s and References in self-* Architecture.

Figure 21 illustrates the important types of Niche 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 binding 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 Niche 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 Niche entities and refreshes the cache contents automatically. Caching policy is determined by the Niche implementation and affects its performance.

Figure 22 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 Niche entities, their Niche Id:s are assigned as follows: the overlay Id is taken from the Niche 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) [9, 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 Niche

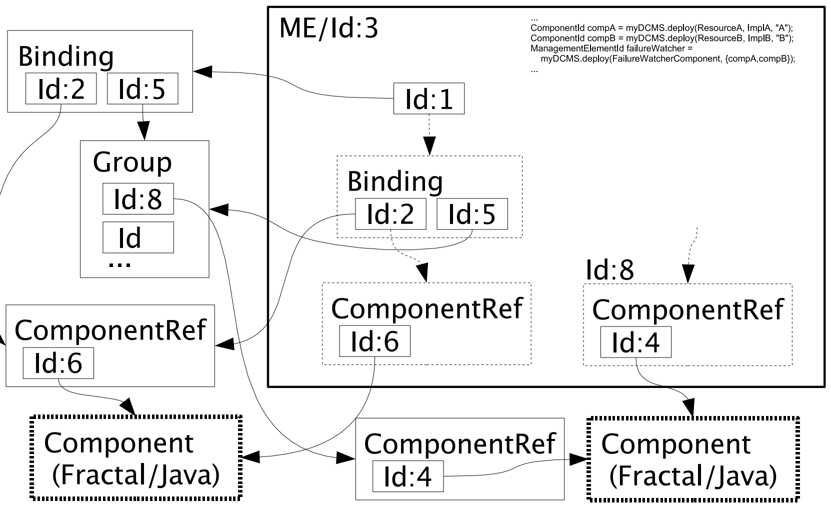

```
...
ComponentId compA = myDCMS.deploy(ResourceA, ImplA, "A");
ComponentId compB = myDCMS.deploy(ResourceB, ImplB, "B");
ManagementElementId failureWatcher =
   myDCMS.deploy(FailureWatcherComponent, {compA,compB});
...
```

Figure 22: Caching of Niche entities.

component reference entities containing multiple references. A “one-to-any” or “one-to-all” binding to a group means that when a message is sent through the binding, the group 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 Niche 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 Niche 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 23 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

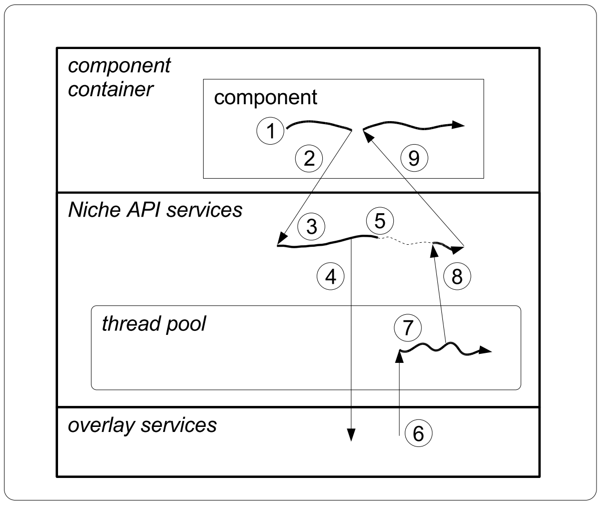


Figure 23: Threads of Control in Niche.

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 24. 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 [22, 27] – 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 [26] do not change the message complexity of Paxos in normal operation. A total-order broadcast algorithm achieving the latency of 2 messages is known [35], but it is unclear whether it can be adopted to 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 25 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).

References

[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.

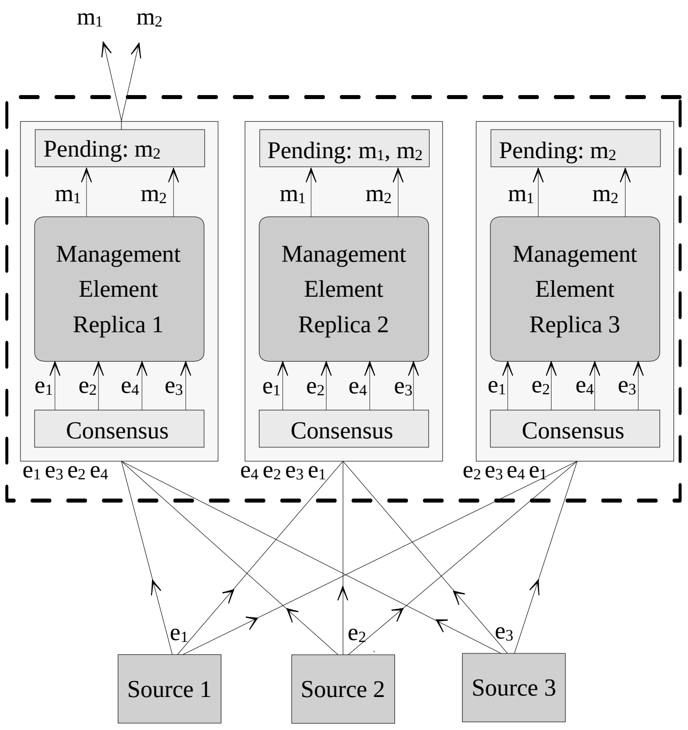


Figure 24: Replication of Synchronized MEs in Niche.

[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] Ö. Babaoglu, G. Canright, A. Deutsch, G. Di Caro, F. Ducatelle, L. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, and T. Urnes. Design patterns from biology for distributed computing. *ACM Transactions on Autonomous Adaptive Systems*, 1(1):26–66, September 2006.

[4] Özalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad P. A. Van Moorsel, and Maarten Van Steen, editors. *Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations*, volume 3460 of *LNCS*. Springer, 2005. The book is a result from a workshop at Bertinoro, Italy, Summer 2004.

[5] Francoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective interfaces for distributed components. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 599–610, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[8] 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.

[9] 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.

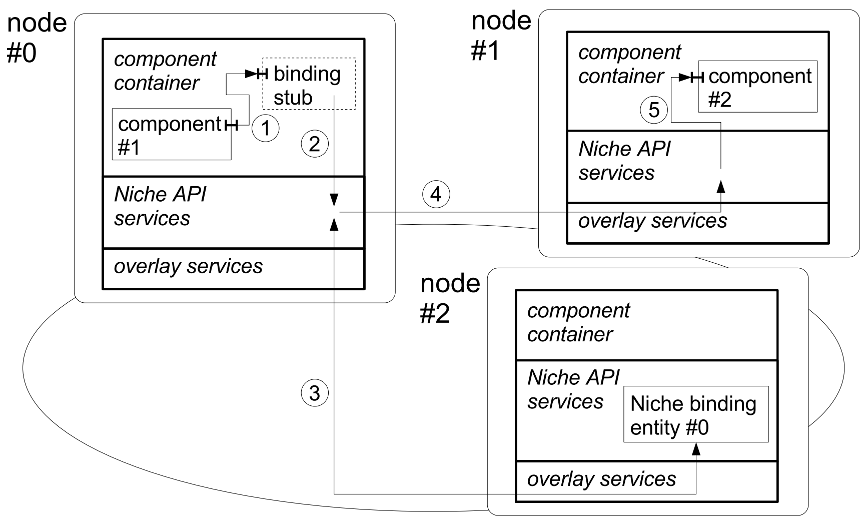


Figure 25: Bindings in Niche.

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

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

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

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

[14] D. Kondo F. Araujo, P. Domingues and L. Moura Silva. Using cliques of nodes to store desktop grid checkpoints. In *Proceedings of CoreGRID Integration Workshop*, pages 15–26, April 2008.

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

[16] P. Grace, G. Coulson, G.S. Blair, L. Mathy, W.K. Yeung, W. Cai, D.A. Duce, and C.S. Cooper. GRIDKIT: Pluggable overlay networks for grid computing. In R. Meersman and Z. Tari, editors, *Proc. Distributed Objects and Applications (DOA'04)*, volume 3291 of *LNCS*, pages 1463–1481, Cyprus, October 2004. Springer.

[17] 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.

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

[19] M. Jelasity and Ö. Babaoglu. T-Man: Gossip-based overlay topology management. In Sven Brueckner, Giovanna Di Marzo Serugendo, David Hales, and Franco Zambonelli, editors, *Engineering Self-Organising Systems, Third International Workshop (ESOA 2005). Revised Selected Papers*, volume 3910 of *LNCS*, pages 1–15, Utrecht, The Netherlands, July 25, 2005, 2006. Springer.

[20] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.

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

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

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

[24] Zhen Li and Manish Parashar. A decentralized agent framework for dynamic composition and coordination for autonomic applications. In *16th International Workshop on Database and Expert Systems Applications (DEXA 2005)*, pages 165–169, Copenhagen, Denmark, August 22–26 2005. IEEE Computer Society.

[25] H. Liu and M. Parashar. Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 36(3):341–352, 2006.

[26] 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.

[27] 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.

[28] A. Mayer, S. McGough, N. Furmento, J. Cohen, M. Gulamali, L. Young, A. Afzal, S. Newhouse, and J. Darlington. ICENI: An integrated Grid middleware to support e-Science. In Vladimir Getov and Thilo Kielmann, editors, *Proceedings of the Workshop on Component Models and Systems for Grid Applications*, volume 1 of *CoreGRID series*, pages 109–124, Saint Malo, France, June 26 2005. Springer.

[29] A. Montresor, H. Meling, and Ö. Babaoglu. Messor: Load-balancing through a swarm of autonomous agents. In Gianluca Moro and Manolis Koubarakis, editors, *Agents and Peer-to-Peer Computing, First International Workshop, Revised and Invited Papers*, volume 2530 of *LNCS*, pages 125–137, Bologna, Italy, July 2002, 2003. Springer.

[30] C. Pairot, P. García, and A. Gómez-Skarmeta. Dermi: A new distributed hash table-based middleware framework. *IEEE Internet Computing*, 08(3):74–84, 2004.

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

[32] M. Parashar, Z. Li, H. Liu, V. Matossian, and C. Schmidt. Enabling autonomic grid applications: Requirements, models and infrastructure. In *Self-Star Properties in Complex Information Systems*, volume 3460 of *LNCS*, pages 273–290. Springer, 2005.

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

[34] Web services distributed management. http://docs.oasis-open.org/wsdm/wsdm-muws1-1.1-spec-os-01.htm, http://docs.oasis-open.org/wsdm/wsdm-muws2-1.1-spec-os-01.htm and http://docs.oasis-open.org/wsdm/wsdm-mows-1.1-spec-os-01.htm.

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