DESIGN OF A SELF-* APPLICATION USING P2P-BASED MANAGEMENT INFRASTRUCTURE

Konstantin Popov, Joel Höglund and Per Brand
Swedish Institute of Computer Science (SICS)
Stockholm, Sweden
{kost,joel,perbrand}@sics.se

Ahmad Al-Shishtawy
and Vladimir Vlassov
Royal Institute of Technology (KTH)
Stockholm, Sweden
{ahmadas,vladv}@kth.se

Nikos Parlavantzas
INRIA
Grenoble, France
nikolaos.parlavantzas@inria.fr

Abstract

We use a functionally simple distributed file storage service to demonstrate a framework for developing and managing self-* component-based applications for highly volatile Grid environments. The service replicates data for reliability, and provides for self-configuration, self-healing and self-tuning. The framework allows to develop application self-* behaviours as a distributed event-driven management application that is independent from application's functional code yet can interact with it when necessary. The framework provides a small set of abstractions that facilitate fault-tolerant application management, and fault-tolerance of the management itself. The framework uses and extends the self-* properties of the structured overlay network which it is built upon.

Keywords: Grid, component-based applications, self-management, P2P

1. Introduction

Deployment and run-time management of applications constitute a significant part of software's total cost of ownership. These costs increase dramatically for distributed applications that are deployed in volatile environments such as peer-to-peer overlays which aggregate heterogeneous, poorly managed resources on top of relatively unreliable networks. The autonomic computing initiative [9] advocates self-configuring, self-healing, self-optimizing and self-protecting (self-* thereafter) systems as a way to deal with the management complexity. Architecture-based self-* management [8] is performed at the level of applications' components and bindings. It has been shown useful for self-repair of component-based applications [1]. Component-based frameworks such as Fractal [3] provide for component reflection and management facilities that allow to separate applications' functional and management aspects.

We present a design of a self-* Fractal component-based application to be deployed in community-based Grids. Such Grids are envisioned to fill the gap between high-quality Grid environments and existing peer-to-peer systems that are limited to one single application and provide no coordinated resource management. Our application, a storage service, is intentionally simple from the functional point of view: it implements the "store a file" and "read a file" operations. Non-functionally, it provides for reliable storage using replication, and can self-heal, self-configure and self-optimize itself.

We use our application as a vehicle to demonstrate our management framework supporting self-* component-based applications, first introduced in [2]. The framework extends the Fractal-based Jade management system for cluster computing [1]. It allows to deploy Internet-based applications, and supports their self-* behaviors by providing an execution platform and a set of abstractions. The management framework relies on an external resource management service that for the purposes of our prototype is provided by the Grid's Virtual Organization (VO) management.

The management framework allows to program applications' self-* behaviours (self-* code thereafter) independently from application's functional code. The self-* code can manipulate the application in terms of the elements of its architecture: it can add and remove components and bindings between them using operations provided by the component management subsystem. The self-* code *senses* changes in the environment, such as failed resources, actions by VO management, and information communicated by the functional code.

Our management framework implements additional types of bindings not specified in the Fractal model, "one-to-all" and "one-to-any", which support scalable multicast communication essential for building fault-tolerant self-healing applications [2]. These bindings allow an application to treat a group of components as a single entity. Membership management of such a group is provided

by the self-* code and is transparent to the functional code. These types of bindings serve similar purpose to the multicast interfaces proposed in the Grid Component Model (GCM) [7].

We see management self-* code to be organized into a set of *management elements*, *watchers*, *aggregators* and *managers*, implemented by the application developer. Watchers monitor status of elements of the architecture. Aggregators maintain status information of an application by collecting information from different watchers. Managers monitor application status by listening to aggregators, and decide on and execute changes in the architecture. Management elements access and manipulate an architecture through *handles* that represent and provide access to architecture's elements. Handles are network-transparent, thus management elements can be executed on any computing node in the system. The framework allows programmer to control the location of management elements which can improve the performance of self-management and simplify handling of failures of nodes hosting management elements.

The management framework is self-* on its own: it can accomodate new nodes joining a system, tolerate failures, and self-optimize following system load. The management framework is implemented on the Niche overlay network [2]. Niche provides for reliable communication, including multicast, and distributed hash table functionality with symmetric replication. Niche's replication and communication are used to implement failure-tolerant management elements and reliable delivery and processing of events.

Our simple storage service, called YASS, contains user front-ends that are connected to a group of storage elements. A front-end has no knowledge about actual configuration of the service. Individual file replicas are grouped, so the front ends access replicas independently of self-* code that maintains the replication factor.

Our first contribution is a simple self-* management model and matching framework that despite its simplicity is sufficient to support the demonstrated self-* application behaviours. Our second contribution is the implementation model for the management framework, which leverages the self-* properties of the Niche overlay network to provide failure-tolerant management elements.

In this paper we present our current prototype of the management framework that is planned to be extended into a number of directions. In particular, the current prototype is limited in the following senses: (a) self-* code including code for initial deployment is written in a low-level, imperative style, while using a declarative "architecture description language" could reduce the complexity imposed to application developers; (b) we do not provide a general model for ensuring coherency and convergence of distributed self-* management. Our example application, however, demonstrates that these properties are attainable in practice.

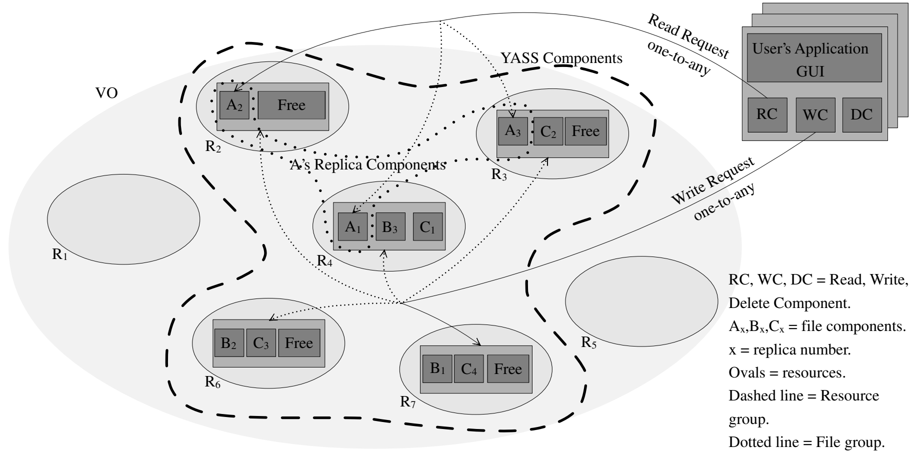

Figure 1. The YASS Architecture

2. Application

Our application – "yet another storage service" (YASS) – allows to store, read and delete files. The service replicates files for the sake of robustness and scalability. We target the service for highly volatile Grid environments, where resources can join, gracefully leave or fail at any time. YASS automatically maintains the file replication factor upon resource churn, and scales itself based on the load on the service and resource availability.

A YASS consists out of *front-end components* and *storage components* Fig. 1. The front-end components and deployed on user machines and provide the read, write and delete components implementing the user interface. Storage components are composed of *file components* that keep files. The ovals R_i in Fig. 1 represent resources contributed to a Virtual Organization (VO). Some of the resources are used to deploy storage components, shown as rectangles.

A user store request can be sent to any storage component that will try to find some $r - 1$ more different storage components, where r is the file's replication degree, with enough free space to store a single file copy. The user will send the file in parallel to the r storage components, resulting in the dynamic creation of r new file components, which together form a *file group*. Read and delete requests can be sent to any of the r file components in the group.

3. Component Model

Our management framework [2] supports an extended version of the Fractal component model [3]. The Fractal model allows to define software architectures based on composition and binding of components. The Fractal specification contains the notion of one-to-one bindings that we instantiate to *one-to-any* and *one-to-all* bindings. With a one-to-any binding, a component can communicate

with a component randomly chosen at run-time from a certain group. With a one-to-all binding, it will communicate with all elements of the group. In either case, the content of the group can change dynamically without affecting neither the source component nor other elements of the destination's group.

4. The Management Framework

An application in the framework consists of a component-based implementation of the application's functional specification and a separate part implementing the application's self-* behaviors. The framework implements component hosting and intra-component communication, and provides a distributed platform for reliable execution of self-* code.

Our work builds on the technical work on the Jade component-management system [1]. Jade's self-* behaviors are implemented in a "sensing – management decision making – actuation" loop. Jade utilizes the Java RMI, and is limited to cluster environments as it relies on small and bounded communication latencies between nodes.

Self-* code in our management framework consists of *management elements* (MEs thereafter). We subdivide MEs into *watchers*, *aggregators* and *managers*. Self-* code can access information from the *architecture registry* (AR thereafter). The framework's run-time system (RTS thereafter) provides for reliable hosting and communication between these entities.

Watchers monitor the status of individual architectural elements, or groups of similar elements. A watcher is a stateful entity that subscribes to and receives events from *sensors* that are either implemented by the element, or provided by the management framework itself. An aggregator is subscribed to several watchers and maintains partial information about the application status at a more coarse-grained level. A manager can be subscribed to several watchers and aggregators. The manager uses the information to decide on and execute the changes in the architecture. Managers manipulate the architecture using the management *actuation* API [1] implemented by the framework. The API provides in particular functions to obtain resources, deploy components, and manage and bind components. The AR provides network-transparent storage of *handles* to elements of application architecture. A handle is an entity that contains reference(s) to the corresponding element(s) of the architecture – components, bindings, and MEs. Functions of the aforementioned actuation API work on handles specified by their identifiers. Handles to architecture elements are implemented by *sets of network references* described below.

MEs are first-class entities that are dynamically created and destroyed as necessary, and recorded in the AR. A dedicated piece of application-specific management code performs initial deployment of the architecture and instantiation of self-* code.

The management framework allows the developer of self-* code to control location of MEs. For every management element the developer can specify a *container* where that element should reside. A container is a first-class entity which sole purpose is to ensure that entities in the container reside on the same physical node. This allows to eliminate network communication latencies between co-located MEs. The container's location can be explicitly defined by a location of a resource that is used to host elements of the architecture, thus eliminating the communication latency and overhead between architecture elements and managers handling them.

The management run-time system provides for reliable hosting of watchers and managers, and reliable delivery of events. The management framework provides for transparent replication of MEs for reliability, and reliable delivery of messages between replicated MEs. This is achieved by placing MEs inside *management element wrappers*, MEWs, that intercept incoming and outgoing events and invocations of actuation API, and coordinate with MEWs hosting other replicas of the same ME.

AR provides a weak consistency model: it is guaranteed that reads follow writes only within a single watcher or manager. The management framework assumes that the self-* code can recognize out-of-date information, and repeat the read operation until up-to-date data is read.

A **Set of Network References**, SNR [2], is a primitive abstraction that is used to associate a *name* with a set of *references*. References are used to access elements in the system and can be either direct or indirect. Direct references can be used without resolving, such as the location of a resource. Indirect references refer to other SNRs by names and needs to be resolved before use. In its simplest form, the SNR associates a name to one direct reference, such as a reference to a component deployed on a resource. We will refer to such simple SNRs by primitive SNRs. SNRs are also used to create homogeneous groups by associating a group name to a set of primitive SNRs.

One-to-any and one-to-all bindings are implemented by binding to SNRs with more than one reference. A binding to an SNR means that when a message is sent through the binding, the SNR name is resolved and one or more of the current references are used to send the message depending on the type of binding. SNRs also enable mobility of elements pointed to by the references. Management code can move components between resources, and by updating their references other elements can still find the components by name. A group can grow or shrink transparently from group user point of view. Finally SNRs are used to support sensing through associating watchers with SNRs. Adding a watcher to an SNR will result in sensors being deployed for each element associated with the SNR. Thus all elements will be sensed by the watchers

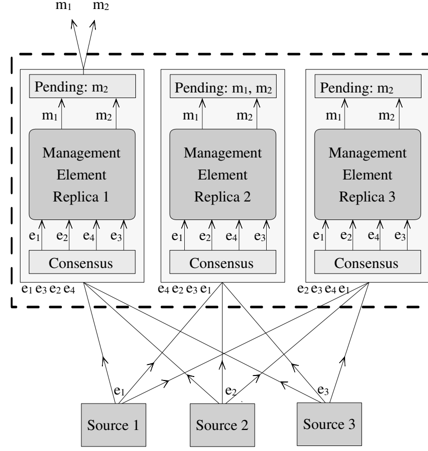

Figure 2. Management element wrappers.

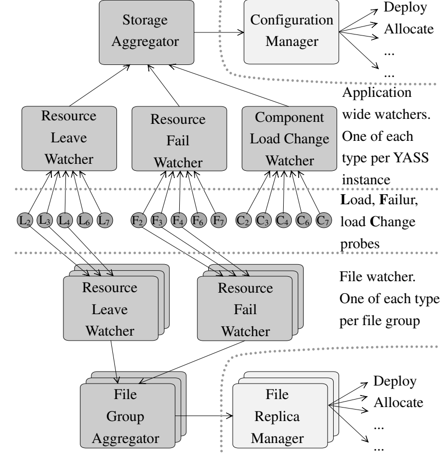

Figure 3. YASS sensors, watchers, aggregators and managers.

associated with the SNR. Changing the references of an SNR will transparently deploy/undeploy sensors for the corresponding elements.

Robustness of Management Elements. Our framework supports failure-tolerance of self-* code by providing a mechanism ensuring that *(a)* events are not lost during delivery and processed exactly once, and *(b)* all actuation commands generated by the managers are executed exactly once. The self-* code developer can decide whether a particular ME has to be failure-tolerant.

We achieve fault-tolerance through replication of MEs and reliable messaging for delivering events and commands. MEs are encapsulated by MEWs as shown in Fig. 2. Replication of MEs is transparent from the ME's point of view, and ME is a black box from the MEW's point of view. A group of MEWs each hosting an instance of the same ME form a replica group of the ME. MEWs in the replica group communicate with each other. MEWs serve two purposes. First, MEWs order events coming to MEs such that all MEs in the replica group observe the same order of events. This is achieved by a consensus protocol running by MEWs in the replica group. Given the same input all ME replicas will generate the same events/commands and will have the same state if any. Second, MEWs intercept all outgoing events and actuation commands such that the replica group behaves as one single ME. A MEW has a queue of pending outgoing events and commands issued by MEs and 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. This mechanism guarantees also reliable message delivery.

MEs uses the fail recovery model. In the case of a failure/leave of a resource where a ME is deployed, it is the responsibility of the infrastructure to bring

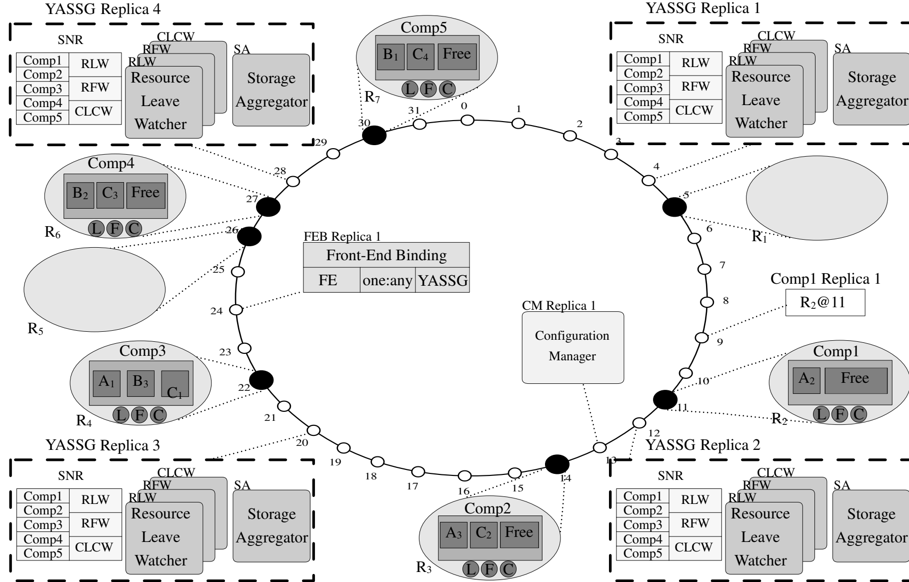

Figure 4. Parts of the YASS application deployed on the management infrastructure. The replication of the main component group, labelled *YASSG*, is shown. The filled black circles represent physical nodes. Architectural entities are mapped to ids. As there is always a physical node responsible for a certain id, each entity will be mapped to one of the nodes in the system. For instance the *YASSG replica one* is mapped to id 4, which is the responsibility of the node with id 5. Practically that means the management elements associated with the replica will be executed on node 5.

the ME back. This is done by redeploying it and restoring any associated state in the case of a stateful ME. Non-ME components, such as sensors and application's components, use the fail stop model. If such a component crash then the infrastructure is not responsible for restoring it. The restoration of those components should be configured through management components. For none-MEs we assume pseudo reliable message delivery. As long as the source does not crash, a message will eventually reach its destination.

5. Application Self-* behaviours

Configuration of application self-management. The Fig. 3 shows the architecture of the watchers, aggregators and managers used by the application. In the following description "one" is used to mean "functionally one", since for each described management element there will be as many instances as the replication degree prescribes.

Associated with the YASSG are the following system-wide watchers created at service initialization time: one ResourceLeave-watcher, one ResourceFail-watcher and one ComponentLoad-watcher. Subscribed to all of them is the Storage-aggregator for the whole application. The Storage-aggregator can trig-

ger StorageAvailabilityChange-events, which the Configuration-managers is subscribed to. The Configuration-managers can also query the ComponentLoad-watcher to be informed about components with low load. Fig. 4 shows the collocation of the management elements associated with the YASSG.

When new components are created by the functional parts of the application, the management infrastructure can run scripts which initiates corresponding new MEs. This is how the following watchers for file groups are created. Associated with each file group is one ResourceLeave-watcher and one ResourceFail-watcher, which are created dynamically at the same time as the file group is created. Subscribed to both of them is a FileGroup-aggregator, which can trigger ReplicaChange-events. Subscribed to the FileGroup-aggregator is the FileReplica-manager. When MEs are created together with the new file-group, the management infrastructure registers the watchers to the existing sensors responsible for monitoring for resource leaves and resource failures of the resources associated with the file group.

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

Resource leave. A sensor signals that a resource is about to leave. For each file stored at the leaving resource, the associated ResourceLeave-watcher is notified and issues a resourceLeave-event. The event is transformed by the FileGroup-aggregator to a replicaChange-event which is forwarded to the FileReplica-manager. The FileReplica-manager uses the one-to-any binding of the file group to issue a FindNewReplica-command to any of the involved components. When a new replica is instantiated the FileReplica-manager signals to the leaving resource that it is free to leave.

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

Application Self-configuration. With self-configuration we mean the ability to adapt the system in the face of dynamism, thereby maintaining its capability to meet existing functional requirements. This is achieved by monitoring the total amount of allocated storage. The Storage-aggregator is initialized with the amount of available resources at initial deployment time. Thereafter each resource leave and resource failure is captured by the main resource watchers, and propagated to the Storage-aggregator. If the total amount of allocated resources drops below the given requirements, the Storage-aggregator issues a storageAvailabilityChange-event, indicating the availability is critically low, which is received and processed by the Configuration-manager. The

Listing 1.1. Pseudocode for parts of the Storage-aggregator

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

Configuration-manager will try to find and allocate a unused resource to deploy a new storage component, which then is added to the group of components. Parts of the Storage-aggregator and Configuration-manager pseudocode is shown in Listing 1.1, demonstrating how the stateful information is kept by the aggregator and updated through sensing events, while the actuation commands are initiated by the manager.

Application Self-optimization. With self-optimization we mean the ability to adapt the system so that it, besides meeting functional requirements, also meets additional non-functional requirements such as efficiency. This is achieved by using the ComponentLoad-watcher to gather information on the total system load, in terms of used storage. The storage components report their load change. These load reports are propagated to the Storage-aggregator. The aggregator will be able to determine when the total utilization is critically high, in which case a StorageAvailabilityChange-event is generated, which will be processed by the configuration manager in the same way as described in the self-configuration section. If utilization drops below a given threshold, and the total amount of allocated resources is above the initial requirements, another storageAvailabilityChange-event is generated. In this case the event indicates the availability is higher than needed, which will cause the configuration manager to query the ComponentLoad-watcher for the least loaded storage component, and instruct it to deallocate itself, thereby freeing the resource. Parts of the Configuration-manager pseudocode is shown in Listing 1.2, demonstrating how the number of storage components can be adjusted upon need.

6. Related Work

This work builds upon and extends the notation used in [2] where the concepts of watchers and event handlers for enabling self-* behaviours were introduced. The main extension in this paper is the notation of SNR:s as unifying architectural elements supported by the infrastructure.

As the work here presented suggests a particular implementational model for distributed component based programming, relevant related work can be found in research dealing specifically with autonomic computing in general and in research about component and programming models for distributed systems.

Listing 1.2. Pseudocode for parts of the Configuration-manager

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

We consider the area of distributed storage services only to the extent that we acknowledge there are functionally superior systems, but they are generally built and managed in more monolitical ways.

Autonomic Management. The vision of autonomic management as presented in [9] has given rise to a number of proposed solutions to aspects of the problem. Many solutions adds self-management support through the actions of a centralized self-manager. One suggested system which tries to add some support for the self-management of the management system itself is Unity [4]. Following the model proposed by Unity, self-healing and self-configuration are enabled by building applications where each system component is a autonomic element, responsible for its own self-management. Unity assumes cluster-like environments where the application nodes might fail, but the project only partly addresses the issue of self-management of the management infrastructure itself.

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 [5], allow dynamic flexibility, but their associated infrastructure lacks support for operation in volatile environments. The Grid Component Model, GCM [7], is a recent component model that specifically targets grid programming. GCM is defined as an extension of Fractal and its features include many-to-many communications with various semantics and autonomic components. The support for autonomic components is minimal. The model only defines simple “autonomic controllers” that embody autonomic behaviour and expose generic operations to retrieve and execute autonomic operations, to accept QoS contracts, and to signal QoS violations. The model provides no guidance for developing and composing actual controllers, that is, self-* code. Moreover, it does not prescribe a particular implementation model and mechanisms to ensure

the efficient operation of self-* code in large-scale environments. Thus, GCM can be seen as largely complementary to our work and thanks to the common ancestor, we believe that our results can be exploited within a future GCM implementation. A component model designed specifically for structured overlay networks and wide scale deployment is p2pCM, which extends the DERMI object middleware platform [11, 10]. The model provides replication of component instances, component lifecycle management and group communication, including anycall functionality to communicate with the closest instance of a component. The model does not offer higher level abstractions such as watchers and event handlers, and the support for self-healing and issues of consistency are only partially addressed.

Distributed storages and file systems. Our demo application does not try to compete on a functional basis with existing distributed storage systems. For the interested reader, one hash-based storage solutions which has proven useful in a real scenario is Dynamo, the storage system underlying the Amazon service infrastructure [6]. A recently suggested distributed file system which suggest loosening the consistency guarantees to instead provide higher performance is found in [12]. These system are functionally superior to our demo application, but they are written as separate services, and thus they do not show the separation of functional and non-functional concerns that we illustrate.

7. Conclusions

We used our component management framework to design a self-managing application to be used in highly dynamic Grid environment. The framework allows to develop application self-* behaviours as a distributed event-driven management application that is independent from application's functional code yet can interact with it when necessary. The framework provides a small set of abstractions that facilitate fault-tolerant application management, and fault-tolerant execution of the management code itself. The framework leverages the self-* properties of the structured overlay network which it is built upon.

Acknowledgments This research is supported by the European project Grid4All and the CoreGrid Network of Excellence.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, New York, NY, USA, 2007. ACM.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] J. Stribling, E. Sit, M.F. Kaashoek, J. Li, and R. Morris. Don't give up on distributed file systems. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS07)*, Bellevue, WA, February 2007.