ENABLING SELF-MANAGEMENT OF COMPONENT BASED DISTRIBUTED APPLICATIONS *

Ahmad Al-Shishtawy,[1] Joel Höglund,[2] Konstantin Popov,[2]
Nikos Parlavantzas,[3] Vladimir Vlassov,[1] and Per Brand[2]

[1] *Royal Institute of Technology (KTH), Stockholm, Sweden*
{ahmadas,vladv}@kth.se

[2] *Swedish Institute of Computer Science (SICS), Stockholm, Sweden*
{kost,joel,perbrand}@sics.se

[3] *INRIA, Grenoble, France*
nikolaos.parlavantzas@inria.fr

Abstract Deploying and managing distributed applications in dynamic Grid environments requires a high degree of autonomous management. Programming autonomous management in turn requires programming environment support and higher level abstractions to become feasible. We present a framework for programming self-managing component-based distributed applications. The framework enables the separation of application's functional and non-functional (self-*) parts. The framework extends the Fractal component model by the component group abstraction and one-to-any and one-to-all bindings between components and groups. The framework supports a network-transparent view of system architecture simplifying designing application self-* code. The framework provides a concise and expressive API for self-* code. The implementation of the framework relies on scalability and robustness of the Niche structured p2p overlay network. We have also developed a distributed file storage service to illustrate and evaluate our framework.

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

*This research is supported by the FP6 Project Grid4All funded by the European Commission (Contract IST-2006-034567) and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

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 resources.

The autonomic computing initiative [11] advocates self-configuring, self-healing, self-optimizing and self-protecting (self-* thereafter) systems as a way to reduce the management costs of such applications. Architecture-based self-* management [10] of component-based applications [5] have been shown useful for self-repair of applications running on clusters [3].

We present a design of a component management platform supporting self-* applications for community-based Grids, and illustrate it with an application. Community-based Grids are envisioned to 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. Our application, a storage service, is intentionally simple from the functional point of view, but it can self-heal, self-configure and self-optimize itself.

Our framework separates application functional and self-* code. We provide a programming model and a matching API for developing application-specific self-* behaviours. The self-* code is organized as a network of *management elements* (MEs) interacting through events. The self-* code *senses* changes in the environment by means of events generated by the management platform or by application specific sensors. The MEs can *actuate* changes in the architecture – add, remove and reconfigure components and bindings between them. Applications using our framework rely on external resource management providing discovery and allocation services.

Our framework supports an extension of the Fractal component model [5]. We introduce the concept of component groups and bindings to groups. This results in "one-to-all" and "one-to-any" communication patterns, which support scalable, fault-tolerant and self-healing applications [4]. For functional code, a group of components acts as a single entity. Group membership management is provided by the self-* code 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 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 (e.g. because of churn) affecting neither the source component nor other elements of the destination's group.

The management platform is self-organizing and self-healing upon churn. It is implemented on the Niche overlay network [4] providing for reliable communication and lookup, and for sensing behaviours provided to self-* code.

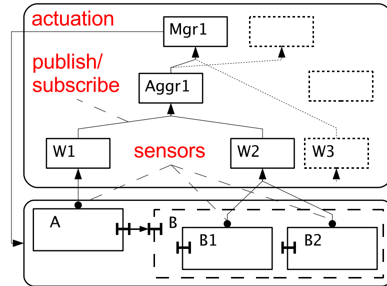

Figure 1. Application Architecture.

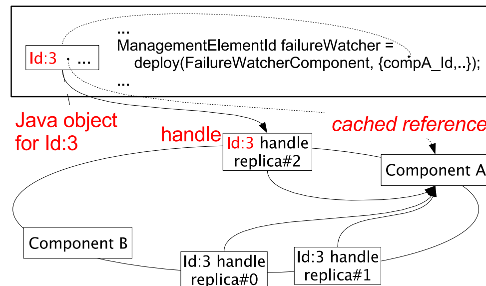

Figure 2. Ids and Handlers.

Our first contribution is a simple yet expressive self-* management framework. The framework supports a network-transparent view of system architecture, which simplifies reasoning about and designing application self-* code. In particular, it facilitates migration of components and management elements caused by resource churn. Our second contribution is the implementation model for our churn-tolerant management platform that leverages the self-* properties of a structured overlay network.

We do not aim at a general model for ensuring coherency and convergence of distributed self-* management. We believe, however, that our framework is general enough for arbitrary self-management control loops. Our example application demonstrates also that these properties are attainable in practice.

2. The Management Framework

An application in the framework consists of a component-based implementation of the application's functional specification (the lower part of Fig. 1), and an implementation of the application's self-* behaviors (the upper part). The management platform provides for component deployment and communication, and supports sensing of component status.

Self-* code in our management framework consists of *management elements* (MEs), which we subdivide into watchers (`W1`, `W2` .. on Fig. 1), aggregators (`Aggr1`) and managers (`Mgr1`), depending on their roles in the self-* code. MEs are stateful entities that subscribe to and receive events from *sensors* and other MEs. Sensors are either component-specific and developed by the programmer, or provided by the management framework itself such as component failure sensors. MEs can manipulate the architecture using the management *actuation* API [3] implemented by the framework. The API provides in particular functions to deploy and interconnect components.

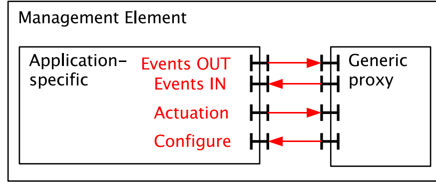

Figure 3. Structure of MEs.

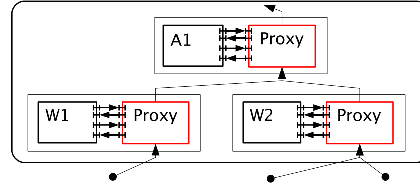

Figure 4. Composition of MEs.

Elements of the architecture – components, bindings, MEs, subscriptions, etc. – are identified by unique *identifiers* (IDs). Information about an architecture element is kept in a *handle* that is unique for the given ID, see Fig. 2. The actuation API is defined in terms of IDs. IDs are introduced by DCMS API calls that deploy components, construct bindings between components and subscriptions between MEs. IDs are specified when operations are to be performed on architecture elements, like deallocating a component. Handles are destroyed (become invalid) as a side effect of destruction operation of their architecture elements. Handles to architecture elements are implemented by *sets of network references* described below. Within a ME, handles are represented by an object that can cache information from the handle. On Fig. 2, handle object for `id:3` used by the `deploy` actuation API call caches the location of `id:3`.

An ME consists of an application-specific component and an instance of the generic proxy component, see Fig. 3. ME proxies provide for communication between MEs, see Fig. 4, and enable the programmer to control the management architecture transparently to individual MEs. Sensors have a similar two-part structure.

The management framework enables 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 eliminates 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.

A **Set of Network References**, SNR [4], is a primitive data abstraction that is used to associate a *name* with a set of *references*. SNRs are stored under their names on the structured overlay network. SNR references are used to access elements in the system and can be either direct or indirect. Direct references contain the location of an entity, and indirect references refer to other SNRs

by names and need to be resolved before use. SNRs can be cached by clients improving access time. The framework recognizes out-of-date references and refreshes cache contents when needed.

Groups are implemented using SNRs 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 name is resolved to its SNR, and one or more of the group references are used to send the message depending on the type of the binding. SNRs also enable mobility of elements pointed to by the references. MEs 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. Changing the references of an SNR will transparently deploy/undeploy sensors for the corresponding elements.

SNRs can be replicated providing for reliable storage of application architecture. The SRN replication provides eventual consistency of SNR replicas, but transient inconsistencies are allowed. Similarly to handling of SNR caching, the framework recognizes out-of-date SNR references and repeats SNR access whenever necessary.

3. Implementation and evaluation

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

3.1 Application functional design

A YASS instance consists out of *front-end components* which are deployed on user machines and *storage components* Fig. 5. Storage components are composed of *file components* representing files. The ovals in Fig. 5 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 is sent to an arbitrary storage component (one-to-any binding) that will find some r different storage components, where r is the file's replication degree, with enough free space to store a file replica. These replicas together will form a *file group* containing the r dynamically created new file

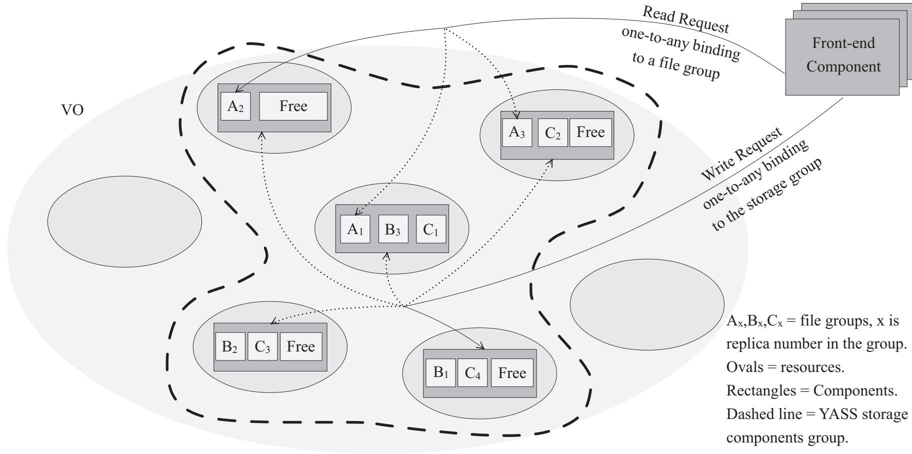

Figure 5. YASS Functional Part

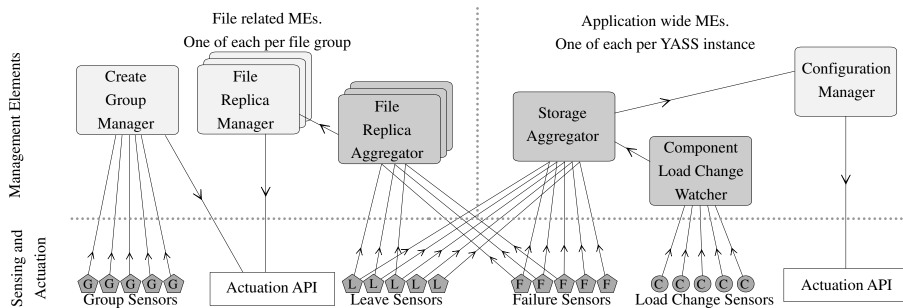

Figure 6. YASS Non-Functional Part

components. The user will then use a one-to-all binding to send the file in parallel to the r replicas in the file group. Read requests can be sent to any of the r file components in the group using the one-to-any binding between the front-end and the file group.

3.2 Application non-functional design

Configuration of application self-management. The Fig. 6 shows the architecture of the watchers, aggregators and managers used by the application.

Associated with the group of storage components is a system-wide Storage-aggregator created at service deployment time, which is subscribed to leave- and failure-events which involve any of the storage components. It is also

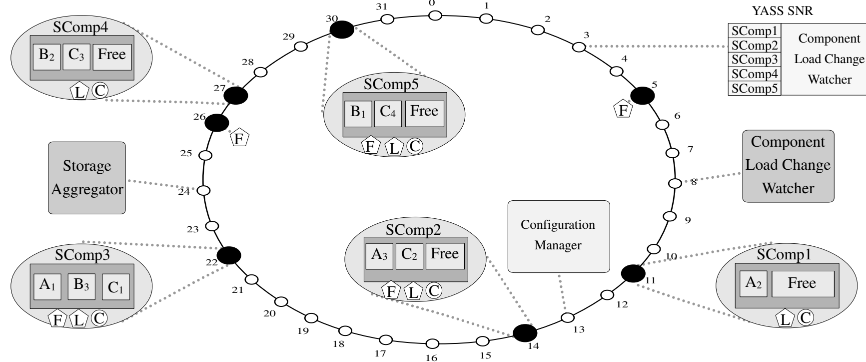

Figure 7. Parts of the YASS application deployed on the management infrastructure.

subscribed to a Load-watcher which triggers events in case of high system load. The Storage-aggregator can trigger StorageAvailabilityChange-events, which the Configuration-manager is subscribed to.

When new file-groups are formed by the functional part of the application, the management infrastructure propagates group-creation events to the Create-Group-manager which initiates a FileReplica-aggregator and a FileReplica-manager for the new group. The new FileReplica-aggregator is subscribed to resource leave- and resource fail-events of the resources associated with the new file group.

3.3 Test-cases and initial evaluation

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

Fig. 7 shows the management elements associated with the group of storage components. The black circles represent physical nodes in the P2P overlay Id space. Architectural entities (e.g. SNR and MEs) are mapped to ids. Each physical node is responsible for Ids between its predecessor and itself including itself. As there is always a physical node responsible for an id, each entity will be mapped to one of the nodes in the system. For instance the *Configuration Manager* is mapped to id 13, which is the responsibility of the node with id 14 which means it will be executed there.

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

Resource leave. An infrastructure sensor signals that a resource is about to leave. For each file stored at the leaving resource, the associated FileReplica-aggregator is notified and issues a replicaChange-event which is forwarded to the FileReplica-manager. The FileReplica-manager uses the one-to-any binding of the file-group to issue a FindNewReplica-event to any of the components in the group.

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

Application Self-configuration. With self-configuration we mean the ability to adapt the system in the face of dynamism, thereby maintaining its capability to meet functional requirements. This is achieved by monitoring the total amount of allocated storage. The Storage-aggregator is initialized with the amount of available resources at deployment time and updates the state in case of resource leaves or failures. If the total amount of allocated resources drops below given requirements, the Storage-aggregator issues a storageAvailabilityChange-event which is processed by the Configuration-manager. The Configuration-manager will try to find an unused resource (via the external resource management service) to deploy a new storage component, which is added to the group of components. Parts of the Storage-aggregator and Configuration-manager pseudocode is shown in Listing 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. In addition to the two above described test-cases we have also designed but not fully tested application self-optimization. With self-optimization we mean the ability to adapt the system so that it, besides meeting functional requirements, also meets additional non-functional requirements such as efficiency. This is achieved by using the ComponentLoad-watcher to gather information on the total system load, in terms of used storage. The storage components report their load changes, using application specific load sensors. These load-change events are delivered to the Storage-aggregator. The aggregator will be able to determine when the total utilization is critically high, in which case a StorageAvailabilityChange-event is generated and processed by the Configuration-manager in the same way as described in the self-configuration section. If utilization drops below a given threshold, and the amount of allocated resources is above initial requirements, a

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
```

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_description, new_resource)
      add_to_group(new_component, component_group)
   elseif total_storage > initial_requirement and new_load < low_limit then
     least_loaded_component = component_load_watcher.get_least_loaded()
     least_loaded_resource = least_loaded_component.get_resource()
     trigger(resourceLeaveEvent(least_loaded_resource))
   end
```

storageAvailabilityChange-event is generated. In this case the event indicates that the availability is higher than needed, which will cause the Configuration-manager to query the ComponentLoad-watcher for the least loaded storage component, and instruct it to deallocate itself, thereby freeing the resource. Parts of the Configuration-manager pseudocode is shown in Listing 1.2, demonstrating how the number of storage components can be adjusted upon need.

4. Related Work

Our work builds on the technical work on the Jade component-management system [3]. Jade utilizes the Java RMI, and is limited to cluster environments as it relies on small and bounded communication latencies between nodes.

As the work here suggests a particular implementation 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.

Autonomic Management. The vision of autonomic management as presented in [11] 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 [6]. 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.

Relevant complementary work include work on checkpointing in distributed environments. Here recent work on Cliques [8]can be mentioned, where worker nodes help store checkpoints in a distributed fashion to reduce load on managers which then only deal with group management. Such methods could 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 [7], allow dynamic flexibility, but their associated infrastructure lacks support for operation in dynamic environments.

The Grid Component Model, GCM [9], 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.

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 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. *Behavioural skeletons* [1] 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 [2]. 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, so groups can not be shared and reused. 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 [13], which extends the DERMI [12] 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 abstractions such as watchers and event handlers, and the support for self-healing and issues of consistency are only partially addressed.

5. Future Work

Currently we are working on the management element wrapper abstraction. This abstraction adds fault-tolerance to the self-* code by enabling ME replication. The goal of the management element wrapper is to provide consistency between the replicated ME in a transparent way and to restore the replication degree if one of the replicas fails. Without this support from the framework, the user can still have self-* fault-tolerance by explicitly implementing it as a part of the application's non-functional code. The basic idea is that the management element wrapper adds a consistency layer between the replicated ME from one side and the sensors/actuators from the other side. This layer provides a uniform view of the events/actions for both sides.

Currently the we use a simple architecture description language (ADL) only covering application functional behaviours. We hope to extend this to also cover non-functional aspects.

We are also evaluating different aspects of our framework such as the overhead of our management framework in terms of network traffic and the time need execute self-* code. Another important aspect is to analyse the effect of churn on the self-* code.

Finally we would like to evaluate our framework using applications with more complex self-* behaviours.

6. Conclusions

The proposed management framework enables development of distributed component based applications with self-* behaviours which are independent from application's functional code, yet can interact with it when necessary. The framework provides a small set of abstractions that facilitate fault-tolerant application management. The framework leverages the self-* properties of the structured overlay network which it is built upon. We used our component management framework to design a self-managing application to be used in dynamic Grid environments. Our implementation shows the feasibility of the framework.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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 2008*, pages 15–26, 2008.

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

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

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

[12] 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.

[13] 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.