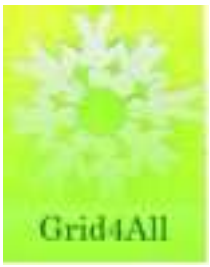

Project no. 034567

Grid4All

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

Specification & Initial Prototype of Overlay Services in Accordance with D1.4

Due date of deliverable: 11th July 2008.

Actual submission date: 11th July 2008.

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

Organisation name of lead contractor for this deliverable: (SICS, KTH, INRIA)

Revision: Draft of 11th July 2008

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)	√

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 resources, such as community-based Grids. Such 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. Application management by humans would render many small and simple applications economically infeasible to run in such environments.

The autonomic computing initiative [8] advocates self-configuring, self-healing, self-optimizing and self-protecting (self-* thereafter) systems as a way to reduce the management costs of such applications. With the architectural approach to self-* management [7], autonomous services have hierarchical architecture where each element is autonomous on its own. Autonomous elements also need to provide certain introspection interfaces like status enquiry that enable higher-level elements in the service architecture to implement their own autonomous behaviours. Architecture-based self-* management of component-based applications has been shown useful for self-repair of legacy applications. In particular, in [3] authors show how their cluster-based component management system allows to wrap elements of legacy software into components of the Fractal component models [5] and then making the legacy software self-healing by means of an explicit and configurable feedback control loop structure.

This document introduces a distributed component management service (DCMS) and its application programming interface (API) that support self-* applications for community-based Grids. DCMS intends to reduce the cost of deployment and run-time management of applications by allowing to program application self-* behaviours that do not require intervention by a human operator.

Our framework separates functional and self-* application code. The functional code of applications is developed in an extended 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, 1]. 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.

Our framework provides 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) communicating by events. Application-specific parts of MEs are components implemented by the application developer. The self-* code *senses* changes in the environment by means of events generated by DCMS or by application specific sensors. MEs can *actuate* changes in the architecture – add, remove and reconfigure components and bindings between them. Architecture elements have identifiers (*Id"s*) that are used by MEs to sense and manipulate the application architecture. Id"s are network-transparent, thus management elements can be executed on any computing node in the system. The framework allows the programmer to control the location of management elements on the network which can improve the performance of self-management and simplify handling of failures of nodes hosting management elements.

We distinguish the following types of management elements: *watchers*, *aggregators* and

managers. 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.

Applications using our framework rely on external resource management providing discovery and allocation services. DCMS does not manage resources on its own. In the context of Grid4All, resource management must be provided by VO resource management services. DCMS API functions related to component life-cycle management are defined in terms of *resources* that are used when a component is deployed and executed.

DCMS is self-organizing and -healing upon churn. It is implemented on the Niche overlay services [4] providing for reliable communication and lookup, and for sensing behaviours provided to self-* code. Thus, the DCMS framework allows application developers to exploit the self-* properties of structured overlay networks using a simple programming model that specifically targets designing application self-* code, and hides unneccessary details of management the P2P infrastructure.

The first contribution of our work is a simple yet expressive self-* management framework. The framework is network-transparent yet network-aware. The framework relieves the application programmer from low-level details of managing the functional and non-functional parts of applications. In particular, it provides the `deploy`/`undeploy` and component sensing abstractions. Network-transparency enables to program both functional and non-functional parts of applications independently of particular deployment scenarios. In particular, the framework supports a network-transparent view of system architecture, which simplifies reasoning about and designing application self-* code. Network-transparency also enables application-transparent migration of MEs: DCMS can move MEs when their nodes are about to leave, and when new resources join achieving better load balance. The framework facilitates programming scalable applications since both functional and non-functional parts are distributed. We believe also that the framework and our implementation of it can be extended for flexible and application-transparent replication of management elements, effectively giving the programmer a conceptually simple platform for programming robust self-management code. Finally, controlling co-location of architecture elements gives the programmer a useful degree of network awareness.

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 [6]. Our example application demonstrates usability of our approach in practice.

We proceed to gradually introduce DCMS and its concepts 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 DCMS API document, the YASS example, and the discussion of features, limitations and future DCMS extensions. The presentation here is informal, and particular syntax of examples can stray from the existing DCMS and YASS code for the sake of presentation clarity.

2 Application Architecture with DCMS

An application in the framework consists of a component-based implementation of the application's functional specification (the lower part of Figure 1), and an implementation of the application's self-* behaviors (the upper part). DCMS provides functionality for component management and communication which is used by user-written implementation of self-* behaviors.

Self-* code in our management framework consists of *management elements* (MEs) developed by the programmer, as described in Section 3. MEs are stateful entities that subscribe to and receive events from *sensors* and other MEs. Sensors are either application-specific and developed by the programmer, or provided by DCMS itself such as component failure sensors. MEs can manipulate the architecture using the management *actuation* API implemented by DCMS and introduced in this document. The API provides in particular functions to deploy and interconnect functional components and MEs.

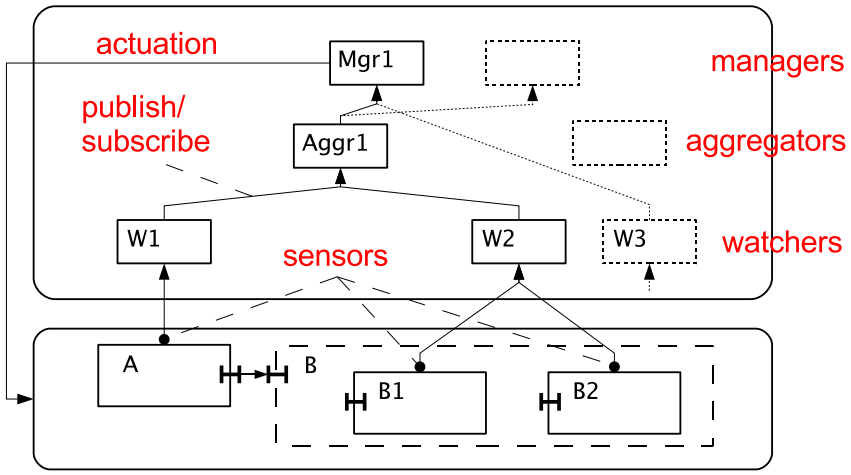

Figure 1: Application Architecture with DCMS.

We subdivide MEs into watchers (`W1`, `W2` .. on Figure 1), aggregators (`Aggr1`) and managers (`Mgr1`), depending on their role in the self-* code. Watchers monitor the status of individual architectural elements, or groups of similar elements. A watcher is a stateful entity that is connected 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. Managers use the information to decide on and execute the changes in the architecture.

3 Management Elements and Sensors in DCMS

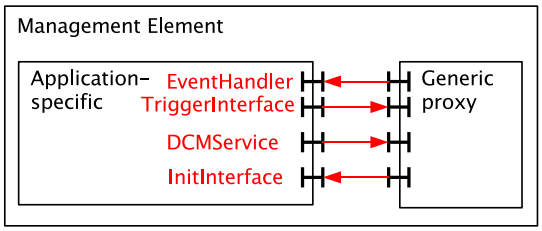

Figure 2: Structure of MEs.

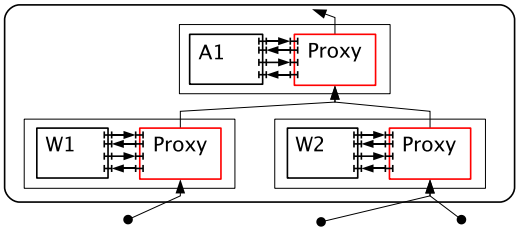

Figure 3: Composition of MEs.

An ME consists of an application-specific Fractal component [5] and an instance of the generic *proxy* component, see Figure 2. Application-specific ME components are developed by application programmer, and the proxy component is provided by DCMS. In our Java-based DCMS prototype, application-specific ME components are implemented as Java classes which manipulate the application architecture using DCMS API. DCMS and application-specific ME components share certain data structures that identify elements of the architecture, as discussed in Section 4. ME proxies provide for communication between MEs, see Figure 3. When a ME is to be deployed, the application developer specifies the implementation of the application-specific ME component, and DCMS takes care about the rest: finding a suitable computer among those interconnected by DCMS, creating the ME proxy and connecting the the application-specific ME component to the proxy. Hosting and executing MEs is taxed to a set of physical nodes executing DCMS, typically VO members. DCMS attempts to evenly balance the load of ME hosting.

As outlined DCMS API documentation, proxies will enable the programmer to control the management architecture transparently to individual MEs, and will enable the programmer to transparently and selectively replicate MEs for the sake of robustness of self-management code.

Application-specific ME components can have the following client interfaces (see also Figure 2):

- `TriggerInterface` interface with the `trigger` method used to emit events generated by the management element
- `DCMService` interface that provides DCMS 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` mehtod used when a management event arrives to the ME
- `InitInterface` interface used to (re)configure the management elements

The application-specific part of a ME can have further client or server interfaces which can be bound to functional components in the application, under certain restrictions discussed in DCMS API documentation.

Sensors have a similar two-part structure, see Figure 4. Application-specific sensor components interact with components being sensed (component `A` in the Figure). Application-specific sensor components can use the following client interface:

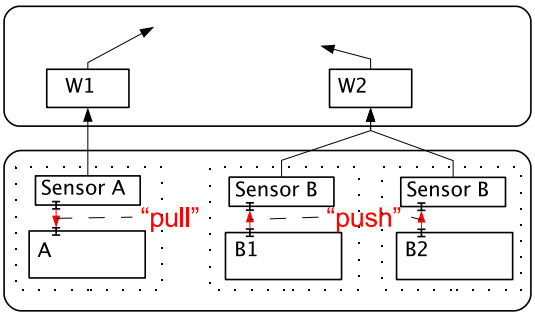

Figure 4: Structure of Application-Specific Sensors.

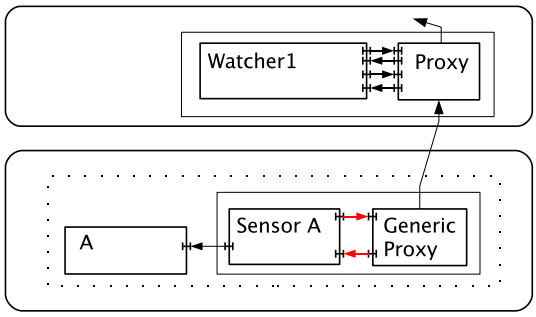

Figure 5: Composition of Application-Specific Sensors.

- `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

Similarly to MEs, when a sensor is to be deployed, the application developer specifies the implementation of the application-specific sensor component, and DCMS takes care about the rest: it locates the component for which the sensor is to be deployed, deploys both parts of the sensor and interconnects them (see Figure 5). Using the facilities of the Fractal component model [5], the application developer also specifies two lists of interfaces – for information "pull" and "push" between the sensor and the component being sensed (see Figure 4), and DCMS uses this information to connect the named application-specific sensor component interfaces to matching interfaces of the component being sensed.

4 Architecture Representation in Self-Management Code

Elements of the architecture – components, bindings, groups and MEs – are identified by unique *identifiers* (Id"s). Self-* code receives information about status of architecture elements and manipulates them using the Id"s. In our Java-based prototype of DCMS, Id"s are represented in self-* code as certain Java objects. We discuss the implementation and performance characteristics of our DCMS prototype in Section 9.

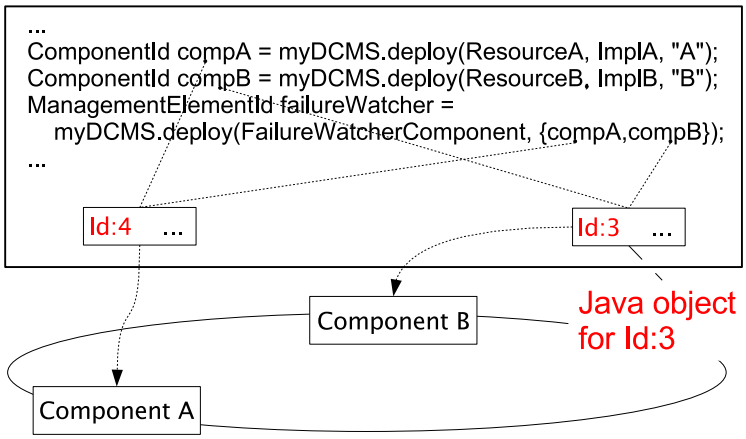

Figure 6: Application Architecture in Self-* Code.

On Figure 6 a snippet of self-* code from a ME is presented. 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 DCMS API calls that deploy functional components and MEs. On Figure, `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 6. Id"s are passed to DCMS API invocations when operations are to be performed on architecture elements, like deallocating a component. In the example, `compA` and `compB` are passed to the `deploy` DCMS API call that deploys a watcher that expects to receive Id"s of components to watch.

Note that Id"s are *network-transparent*: multiple MEs share the same Id"s even though they reside on different physical nodes. In the example, `failureWatcher` ME will in general be deployed on a different physical node from the one where the ME with the given part of self-* is executed, yet both MEs posses references to DCMS Id Java objects representing A and B. Different physical nodes necessarily have different Java objects representing the same DCMS Id, as discussed in Section 9. We continue illustrating the manipulation of the application architecture in Section 5 using the code for initial deployment as an example.

5 Initial Deployment of Applications

```
DCMService myDCMS;
ComponentId compA = myDCMS.deploy(ResourceA, ImplA, "A");
ComponentId compB = myDCMS.deploy(ResourceB, ImplB, "B");
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 7: Example of Self-Management Code with DCMS.

Initial deployment of applications is performed using the DCMS API. In the case the application functional architecture is specified using an architecture description language (ADL) specification, the application deployment service interprets the ADL specification and invokes corresponding DCMS API functions. DCMS ADL allows the programmer to specify the application architecture conveniently, concisely and compactly; it is described in the DCMS API documentation. The sequence of commands executed by DCMS can look like on Figure 7. In this example, `myDCMS` is an object that provides the DCMS API. The first `deploy` method deploys a component `A` implemented by `ImplA` on a resource `ResourceA`. We discuss the resource management in the next Section. `deploy` invocations contain also symbolic names of components in the application architecture, `A` and `B` in our example. If the application is deployed by the deployment service, the symbolic names of components are taken from the ADL specification. The symbolic names can be used to obtain the component Id by e.g. other management elements:

```
ComponentId compA = myDCMS.lookup("ApplicationPrefix"+"ComponentADLName");
```

Component Id"s can be also passed as arguments for deployment of management elements, as illustrated by the following call from the example above:

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

Here, the new manager `manager` will receive the argument list `{compA, compB}`. The argument list is not interpreted by DCMS itself.

Finally in our example, 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 DCMS `componentFailure` environment sensing events that are generated by DCMS when `compA` or `compB` fail:

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

6 Resource Discovery and Allocation

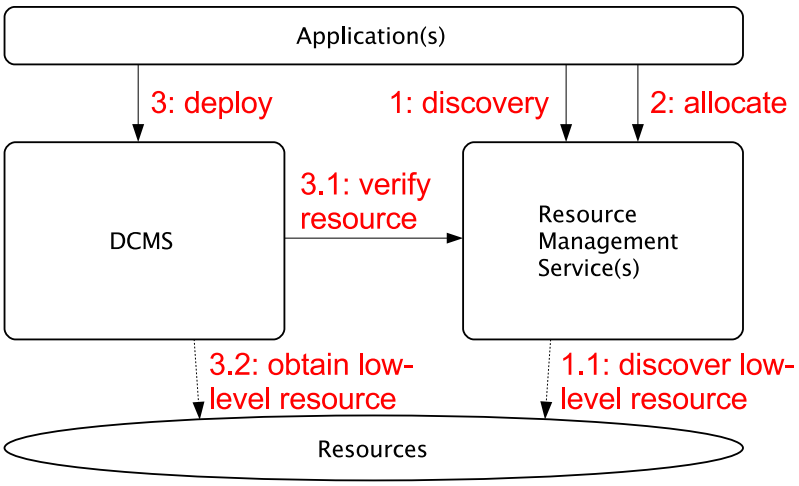

Figure 8: Resource Management with DCMS.

Resource management is out of the scope of DCMS. DCMS is supposed to be used together with one or several resource management services. Applications, DCMS 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 provided by resource management services to discover free resources on the network, see Figure 8. 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. The result of the “allocate” request is a `ResourceRef` object that represents a resource that is reserved for use by the calling application. One of the arguments of the “deploy” DCMS API function that serves component deployment (see Section 5) 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 DCMS verifies its resource usage with respect to `ResourceRef`”s specified for deployment of that component.

7 Groups and Group Sensing

DCMS supports group communication patterns through *one-to-any* and *one-to-all* bindings, which is an extension of the Fractal model [5]. In the current Java-based DCMS prototype, component bindings implement RMI. 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 affecting neither the source component nor other elements of the destination's group.

DCMS supports group communication through first-class *groups* created by means of the `createGroup` DCMS API call:

```
GroupId storageComponentGroupId = myDCMS.createGroup({compS1, compS2});
```

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

```
myDCMSInterface.bind(frontEndId, FILE_WRITE_CLIENT_INTERFACE, globalFileGroupId,
FILE_WRITE_SERVER_INTERFACE, ONE_TO_MANY);
```

Groups can be also 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. DCMS automatically deploys and removes sensors as the group membership changes. This behaviour is implemented using the SNR abstraction as described in Section 9.

To deploy a watcher and associate it with a group you need to specify ManagementDeployParameters as follows:

```
params = new ManagementDeployParameters();
params.describeWatcher(String className, String componentName,
                       Object[] initialArguments, NicheId GroupId)
myDCMSInterface.deploy(ManagementDeployParameters params,
                       IdentifierInterface destination);
```

The watcher, when initialized, must specify the sensor that DCMS will automatically deploy as follows:

```
myDeploySensorsInterface.deploySensor(String sensorClassName,
          String sensorEventClassName, Object[] sensorParameters,
          String[] clientInterfaces, String[] serverInterfaces);
```

8 Controlling Location of Management Elements

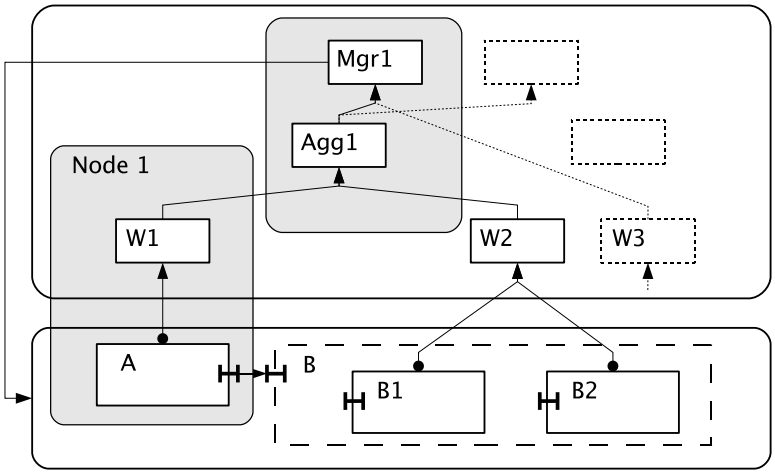

Figure 9: Co-location of MEs.

The application developer can control the location of MEs by specifying another architecture element to collocate with as an optional last parameter of the deployment call.

9 The Implementation Model of DCMS

DCMS implementation relies on structured overlay networking, overlay thereafter. The overlay is used by DCMS to implement bindings between components and message-passing between MEs, storage of architecture representation and also failure sensing.

At runtime, DCMS integrates several physical nodes using an overlay. On each physical node there is a local DCMS process that provides the DCMS API to applications. 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 DCMS also has an overlay identifier, and can be located and contacted using that identifier.

DCMS maintains several types of entities, in particular components of the application architecture and internal DCMS entities maintaining representation of the application’s architecture. Functional components are situated on specified physical nodes, while MEs and entities representing the architecture can be moved upon churn between physical nodes.

DCMS entities are identified by DCMS Id”s. DCMS Id”s are implemented using identifiers of the overlay. A DCMS Id contains an overlay identifier and a further local identifier. The local identifier allows to distinguish multiple entities assigned to the same overlay identifier. Note that DCMS Id”s act as both unique identifiers and addresses of entities in DCMS.

DCMS Id”s for entities situated on specific physical nodes both identify and address the entities. In particular, DCMS Id”s of components contain the overlay Id of the physical node with the component, and an identifier local to the node. The latter identifier allows the DCMS process on a node to distinguish between different components situated on the node.

If no co-location constraints are specified for MEs and groups, DCMS assigns to them random overlay Id”s for the sake of load balancing. Since the location of MEs is controlled at runtime by DCMS, in particular to take advantage of stable nodes as indicated by users and/or resource management service, ME identities and ME locations are decoupled using *reference* entities, see Figure 10. For a given ME there is a unique DCMS reference entity, thus the Id of the reference entity is used as the identity of the ME. A DCMS reference contains a DCMS Id used as the actual location of the ME. DCMS processes usually cache the true actual locations of entity, as illustrated by the Figure. DCMS references also enable mobility of components. MEs can move components between resources, and by updating their references other elements can still find the components by their DCMS Id”s.

If MEs or groups are co-located with other DCMS entities, their DCMS Id”s are assigned as follows: the overlay Id is taken from the DCMS Id of the entity to be co-located with, and a fresh local identifier is chosen.[1]

Groups are implemented using *Set of Network References* (SNR) [4, 1] which is a primitive data abstraction that is used to associate a *name* with a set of *references*. SNRs can be thought of as DCMS reference entities containing multiple references. DCMS recognizes out-of-date references and refreshes cache contents when needed. 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

[1] This is actually an oversimplified model as it does not explain recursive co-location of entities, i.e. co-location with entities that are in turn co-located with some other entities.

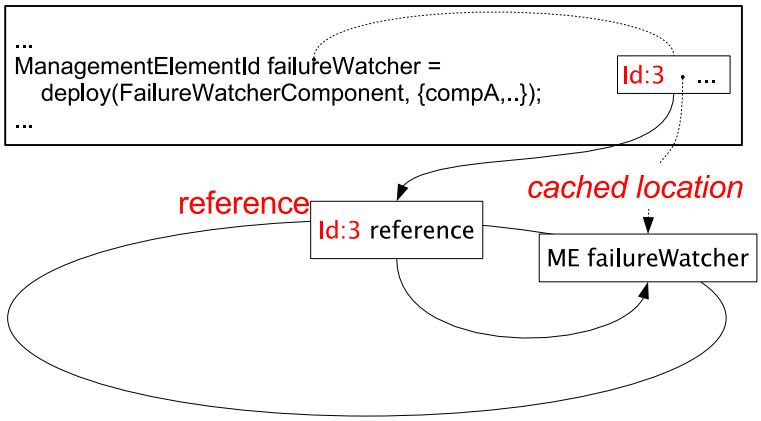

Figure 10: Id"s and References in self-* Code.

group's SNR. Changing group membership transparently causes deployment/unemployment of sensors for the corresponding elements.

10 **Conclusions**

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

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.

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

[7] 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.

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

Level of confidentiality and dissemination

By default, each document created within Grid4All is © Grid4All Consortium Members and should be considered confidential. Corresponding legal mentions are included in the document templates and should not be removed, unless a more restricted copyright applies (e.g. at subproject level, organisation level etc.).

In the Grid4All Description of Work (DoW), and in the future yearly updates of the 18-months implementation plan, all deliverables listed in Section 7.7 have a specific dissemination level. This dissemination level shall be mentioned in the document (a specific section for this is included in the template, both on the cover page and in the footer of each page).

The dissemination level can be defined for each document using one of the following codes:

PU = Public.
PP = Restricted to other programme participants (including the EC services).
RE = Restricted to a group specified by the Consortium (including the EC services).
CO = Confidential, only for members of the Consortium (including the EC services).
INT = Internal, only for members of the Consortium (excluding the EC services).

This level typically applies to internal working documents, meeting minutes etc., and cannot be used for contractual project deliverables.

It is possible to create later a public version of (part of) a restricted document, under the condition that the owners of the restricted document agree collectively in writing to release this public version. In this case, a new document code should be given so as to distinguish between the different versions.