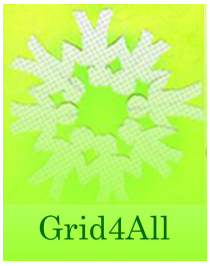

Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)

Thematic Priority 2: Information Society Technologies

D 6.7A Self-Management with Niche

Due date of deliverable: 30 June 2009

Actual submission date: 25 July 2009

Start date of project: 1 June 2006 Duration: 37 months

Organisation name of lead contractor for this deliverable: SICS

Revision [1]

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
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)	

Table of Contents

Abbreviations used in this document

Abbreviation / acronym	Description
VO	Virtual Organization
DCMS	Distributed Component Management System

Grid4All list of participants

Role	Participant N°	Participant name	Participant 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. Executive Summary

Niche is a general-purpose distributed component management system (DCMS) used to develop, deploy and execute self-managing distributed applications or services in all kinds of environments, including very dynamic ones with volatile resources. Niche is both a component-based programming model that includes management aspects as well as a VO-wide distributed run-time environment. Niche is especially designed to enable application developers to design and develop complex distributed applications that will run and manage themselves in a 'democratic Grid'. By a democratic Grid we mean collaborations that wholly, or partly, are characterized by extending into the edge of the Internet. The resources that are shared are low-end and volatile, and both users and system administrators are non-professionals as regards computer technology. Typical driving scenarios involve schools, NGOs, and home users, coming together to form virtual organizations. Members need only to agree on policies to cover such issues as which applications to scale down or stop upon resource contention. After deployment the application manages itself, completely without human intervention. During the application lifetime the application is transparently recovering from failure, and tuning and reconfiguring itself on environmental changes such as resource availability or load.

Achieving self-management in a dynamic environment characterized by volatile resources and high churn (leaves, failures and joins of machines) is challenging. State-of-the-art techniques for self-management in clusters are not suitable. One challenge is to develop a churn-tolerant, efficient sensing and actuation infrastructure through which management senses failures, load and other important environmental changes, and actuates or manipulates the configuration to heal or tune the application. Another challenge is to avoid the management bottleneck as the overall management load for a single application depends on both the size of the system and the volatility of the environment. The standard mechanism of a single management node will introduce a bottleneck, both in terms of management processing, but also in terms of bandwidth.

Niche is a distributed component management system designed to deal with these challenges. Niche demonstrates scalability, robustness and churn-tolerance. Niche supports decentralized management, and the sensing/actuation infrastructure has features to reduce messaging.

2. Introduction

Niche is a general-purpose distributed component management system (DCMS) used to develop, deploy and execute self-managing distributed applications or services in all kinds of environments, including very dynamic ones with volatile resources. Niche is both a component-based programming model that includes management aspects as well as a VO-wide distributed run-time environment [4, 7,8,9].

Niche provides a programming environment that is especially designed to enable application developers to design and develop complex distributed applications that will run and manage themselves in a 'democratic Grid'. By a democratic Grid we mean collaborations that wholly, or partly, are characterized by extending into the edge of the Internet. The resources that are shared are low-end and volatile, and both users and system administrators are non-professionals as regards computer technology. Typical driving scenarios involve schools, NGOs, and home users, coming together to form virtual organizations (VOs) to collaborate for shorter or longer periods. The vision is that having formed such a VO and having installed the VO-wide Niche runtime environment, the members can then download or otherwise obtain application software that has been developed using Niche, and then install and deploy the software with virtually no effort. Members need only to agree on policies to cover such issues as which applications to scale down or stop upon resource contention. After deployment the application manages itself, completely without human intervention, excepting, of course, policy changes such as changing priorities of the various applications running in the VO, stopping and deinstalling the application. During the application lifetime the application is transparently recovering from failure, and tuning and reconfiguring itself on environmental changes such as resource availability or load. This cannot be done today (in volatile environments), i.e. is beyond the state-of-the-art, except for single machine applications and the most trivial of distributed applications, e.g. client/server.

The rest of the paper is organized as follows. In section 3we briefly describe what self-management is and why it is important. In section 4 we discuss the challenges that are involved in achieving self-management in dynamic and volatile environments. In section 5 Niche is described, beginning with features that are common in the field, and then moving onto less common and novel features. Our test and demonstrator applications are also described, as well as the current limitations of the Niche prototype implementation. Finally, in section 6 we conclude.

3. Background

The benefits of self-managing applications apply in all kinds of environments, and not only in a democratic Grid environment. The alternative is management by humans, which is costly, error-prone, and slow. In the well-known IBM Autonomic Computing Initiative [2] the axes of self-management were self-configuration, self-healing, self-tuning and self-protection. Today, there is a considerable body of work in the area, most of it geared to clusters.

Application management in a distributed setting consists of two parts. First, there is the initial deployment and configuration, where individual components are shipped, deployed, and initialized at suitable nodes, then the components are bound to each other as dictated by the application architecture, and the application can start working. Second, there is dynamic reconfiguration where a running application needs to be reconfigured. This is usually due to environmental changes, such as change of load, the state of other applications sharing the same infrastructure, node failure, node leave (either owner rescinding the sharing of his resource, or controlled shutdown), but might also be due to software errors or policy changes. All the tasks in initial configuration may also be present in dynamic reconfiguration. For instance, increasing the number of nodes in a given tier will involve discovering suitable resources, deploying and initializing components on those resources and binding them appropriately. However, dynamic reconfiguration generally involves more, because firstly, the application is running and disruption must be kept to a minimum, and secondly, management must be able to manipulate running components and existing bindings. In general, in dynamic reconfiguration, there are constraints on the order in which configuration change actions are taken unlike initial configuration when the configuration can be built first and components are only activated when this has been completed, starting the application.

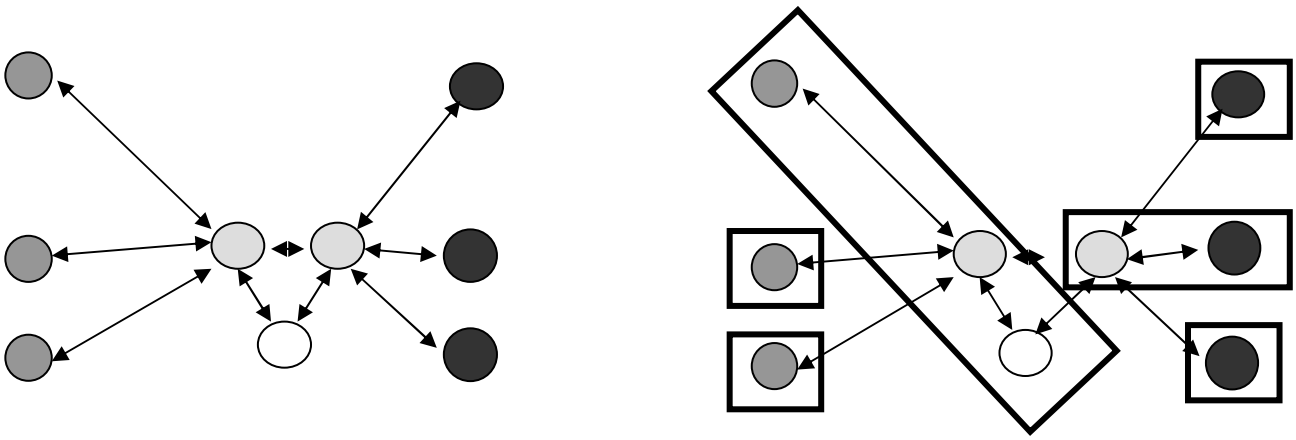

fig 1. Abstract (left) and concrete (right) view of a configuration

A configuration may be seen as a graph, where the nodes are components and the links are bindings. Components need suitable resources to host them, and we can complete the picture by adding the mapping of components onto physical resources. In figure 1 this is illustrated. On the left we show the graph only, the abstract configuration, while on the right the concrete configuration is shown. The bindings that cross resource boundaries will upon use involve remote invocations, while those that do not can be invoked locally. Reconfiguration may involve a change in the concrete configuration only or in both the abstract and concrete

configurations. Note, that we concentrate on the more interesting and challenging aspects of reconfiguration and ignore reconfigurations that leave the graph unchanged but only change the way in which components work by changing component attributes. For instance, in these dynamic environments, a resource may announce that it is leaving and a new resource will be located and the component moved. In this case only the concrete configuration changed. Alternatively, when there is an increase in the number of service components in a tier this will change the abstract (and concrete) configuration by adding a new node and the appropriate bindings. Another example is when a resource fails. If we disregard the transient broken configuration, where the failed component is no longer present in the configuration and the bindings that existed to it are broken, an identical abstract configuration will eventually be created, differing only the resource mapping. In general, an application architecture consists of a set of suitable abstract configurations with associated information as to the resource requirements of components. The actual environment will determine which one is best to deploy or to reconfigure towards. Note that in figure 1 only the top-level components are shown. At a finer level of detail there are many more components, but for our needs we can ignore components that are always co-located and bound exclusively to co-located components. Note, that we ignore only those that are always co-located (in all configurations). There are components might be co-located in some concrete configurations (when a sufficient capable resource is available) but not in others. In the figure, on the right, a configuration is shown with one machine hosting 3 components; in another concrete configuration they might be mapped to different machines. .

4. Challenges

Achieving self-management in a dynamic environment such as a democratic Grid, characterized by volatile resources and high churn (leaves, failures and joins of machines) is challenging. State-of-the-art techniques for self-management in clusters are not suitable. The challenges are:

1) Discovering and utilizing free resources.
2) Churn-tolerant, efficient and robust sensing and actuation infrastructure
3) Avoiding management bottleneck and single-point-of-failure
4) Scale

In the democratic Grid scenario resources are extremely volatile. This volatility is only partly related to churn. Machines, and in particular home machines, may be shared within a VO, but may be at any time removed when the owner needs the machine for other purposes. Owners may remove the machine, entirely, or partly by diminishing the sharing slice of the home. Also, this kind of heterogeneous environment over low-end resources and non-professional owners we cannot expect the same level of reliability in the individual machines. Demanding applications may require more resources than are available in the VO and additional resources then need to be obtained quickly from a market or an external provider. These new resources need to be integrated with existing resources to allow applications to run over the aggregated resources. We expect, at least sometimes, that the VO finds itself hard pressed to find resources, and therefore will need to make use of all potential resources, even the most transient. A democratic Grid VO does not have the luxury of overprovisioning, and may be working close to available capacity so that even smaller changes of load in one application may trigger a reconfiguration as other applications need to be ramped up or down depending on the relative priorities of the applications (according to VO policy). We see the need for a VO-wide infrastructure where volatile resources can efficiently be discovered and utilized. This infrastructure itself also needs to be self-managing.

The sensing and actuation infrastructure needs to be efficient. The demand for efficiency rules out, at least as the main mechanism, a probing monitoring approach. Instead the publish/subscribe paradigm needs to be used. The sensing and actuation infrastructure must be robust and churn-tolerant. Sensing events must be delivered (at least once) to subscribing management elements, irrespective of failure events, and irrespective of whether or not the management element has moved. In the Grid4All environment it is quite normal for a management element to move from machine to machine during the lifetime of the application as resources leave and join the VO.

It is important that management does not become the bottleneck. For the moment, let us disregard the question of failure of management nodes. The overall management load for a single application depends on both the size of the system (i.e. number of nodes in the configuration graph) and the volatility of the environment. It may well be that a Grid4All environment of a few hundred nodes could generate as many events per time unit as a large data centre. The standard mechanism of a single management node will introduce a bottleneck (both in terms of management processing, but also in terms of bandwidth). Decentralization of management is, we believe, the key to solving this problem. Of course, decentralization of management introduces design and synchronization issues, There are issues on how to design management that requires minimal synchronization between the manager nodes (as clearly, if there is too much necessary interaction between manager nodes then this will become the bottleneck). However, it is clear that the component management system must not assume or demand any kind of centralization. The issue of failure of management nodes in centralized and decentralized solutions are, on the other hand, not that different. (Of course, with decentralized approach, only parts of management fail). If management elements are stateless fault-recovery is relatively easy. If they are stateful, some form of replication can be used for fault-tolerance (e.g. hot standby in a cluster).

Finally, there are many aspects of scale to consider. We have touched upon some of them in the preceding paragraphs, pointing out that we have to take into account the sheer number of environmental sensing events. Clearly the VO-wide resource discovery infrastructure needs to scale. But there are other issues to consider regarding scale and efficiency. We have used two approaches in dealing with these issues. The first, is keeping in mind, our decentralized model of management, is to couple as loosely as possible. In contrast to cluster management systems, not only do we avoid maintaining a centralized system map reflecting the ‘current state’ of the application configuration, we strive for the loosest coupling possible. In particular, management elements only receive event notifications for exactly those events that have been subscribed to. Secondly, we have tried to identify common management patterns, to see if they can be optimized (in terms of number of messages/events or hops) by supporting them directly in the platform as primitives, rather than as programmed abstractions when and if this makes for a difference in messaging or other overhead.

5. Niche

5.1 Common Features

There are a number of aspects of Niche that are fairly common in autonomic computing and we begin by describing these. Firstly, Niche supports the control loop paradigm is used where management logic in a continuous feedback loop senses changes in the environment and component status, reasons about those changes, and then, when needed, actuates, i.e. manipulates components and their bindings. A self-managing application can be divided into a functional part and a management part tied together by sensing and actuation.

The Niche programming model is Java-based. It is based on the Fractal components model [3] in which components are bound and interact functionally with each other using two kinds of interfaces: (1) server interfaces offered by the components; (2) and client interfaces used by the components. Components are interconnected by *bindings*: a client interface of one component is bound to a server interface of another component. Fractal allows nesting of components in composite components and sharing of components. Components have control (management) membranes, with introspection and intercession capabilities. It is through this control membrane, that components are started, stopped, configured. It is through this membrane that the components are passivated (as a prelude to component migration), and through which the component can report application-specific events to management (e.g. load). Fractal can be seen as defining a set of capabilities for functional components, it does not force application components to comply, but clearly the capabilities of the programmed components must match the needs of management. For instance, if the component is both stateful and not capable of passivation (or checkpointing) then management will not be able to transparently migrate the component.

5.2 Non-common to Novel Features

To deal with the first challenge Niche is a VO-wide infrastructure that loosely connects all physical resources, and provides for resource discovery by using a structured overlay. As Niche connects all the resources within the VO, different applications concurrently discover and allocate resources. (Concurrency might also exist within a single application as the required set of resources are discovered and allocated). Niche cooperates with component hosting container, which upon joining and leaving the VO, invokes Niche in a manner completely analogous to peer-to-peer systems (e.g. Chord). Currently Niche does not understand resource descriptions, merely hands them upon discovery to the hosting container to determine if it can match the request or not. Resource discovery in Niche is designed to work together with a resource management service (not part of Niche), to control discovery and allocation upon resource contention by different applications. Of course, overlay-based resource discovery mechanisms are common in peer-to-peer systems, but they are not common in integrated distributed application management systems.

Niche provides a sensing service for application managers, as part of its programming API. Application managers can subscribe to application-specific component events (e.g. load) or platform events (e.g. failure). Niche provides an actuation service, whereby managers can control and manipulate components and their bindings. For platform events (component leaves or failures) the program developer uses the subscription API needs only to specify the component identifier (which is VO-wide unique and which is supplies by Niche when the component is first deployed). For application-specific events the developer, in addition to ensuring that the appropriate manager subscribes to the event, needs to program the component so that it generates the event as expected (e.g. high load event). Note that if there are no subscriptions, and hence no interest, then no events or messages will be generated.

The control loops of the management part are built of distributed management elements, interacting through events delivered by the pub/sub mechanisms in Niche. Niche supports management elements of three types: *Watchers*, *Aggregators* (Aggr) and *Managers* (Mgr). A watcher monitors a part of the application, and can fire events when it finds some symptoms of management concern. Aggregators are used to collect, filter and analyse events from watchers. An aggregator can be programmed to analyse symptoms and to issue change requests to managers. Managers do planning and execution of change requests.

The sensing and actuation services are robust and churn-tolerant, and Niche itself is self-managing. Niche thus meets the second challenge. Niche achieves this by leveraging a structured overlay. The necessary information to relay events to subscribers (at least once) is stored with redundancy in the overlay. Upon subscription Niche creates the necessary primitives sensors that serve as the initial detection points. In some cases, sensors can be safely co-located with the entity whose behaviour is being monitored (e.g. a component leave event). In other cases, the sensors cannot be co-located. For instance, a crash of a machine will cause all the components (belonging to the same or different applications) being hosted on it to fail. Here the failure sensors need to be located on other nodes. Niche does all this transparently for the developer; the only thing the application developer must do is to use the Niche API to ensure that management elements subscribe to the events that it is programmed to handle, and that components are properly programmed to trigger application-specific events (e.g. load).

Niche allows for maximum decentralization of management. Management can be divided (i.e. parallelized) by aspects (e.g. self-healing, self-tuning), spatially, and hierarchically. A single application, in general, has many loosely synchronized managers. Niche supports the mobility of management elements (which might take place due to heavy load on the machine where the management element is hosted). Niche also provides the execution platform for these managers; they typically get assigned to different machines in the VO. There is some support for optimizing this placement of managers, and some support for replication of managers for fault-tolerance. Thus Niche meets, at least partly, the third challenge, avoiding the management bottleneck. The main reason for the ‘at least partly’ in the last sentence, is that more support for optimal placement of managers, taken into account network locality, will probably be needed (currently Niche recognizes only some special cases, like co-location). A vanilla replication mechanism is available, and work is ongoing on a robust replicated manager scheme based on the paxos algorithm, adapted to the Niche overlay.

An important feature is that all elements of the architecture, components, managers, etc., have VO-wide unique identifiers. Niche ensures that actuation messages reach the component and sensing events reach (at least once) the subscribed manager, even if the component or manager has moved. So, in the implicit connection between an architectural element and a manager both ends can move transparently. Also, managers do not need to be notified, when *other* events concerning the components that they are monitoring will occur. For instance, a manager that is responsible for self-healing does not need to be informed about component movement due to, for instance, a node shut-down. Indeed, in a dynamic Grid4All VO, system components might be reshuffled in the continuously changing resource pool for some time before a failure occurs and the self-healing manager is triggered. This is one illustration of the loose coupling that we used as a guiding principle in helping us to scale the system, in this case by avoiding unnecessary message sending.

5.3 Context of Niche

Niche is designed to work together with various VO Management tools, some of which have been developed in the Grid4All project. VO members (as opposed to application developers) will never interact directly with Niche. VO Management services will be used to deploy applications and set policies. Using those policies the resource management service will interact with Niche regulating the resources that the various applications managers (corresponding to different applications) can see and use.

Furthermore, although programming in Niche is on the level of Java, it is both possible and desirable to program management at a higher level and to considerable extent declaratively. Programs developed on this level would then be compiled down to Niche code (Niche API). One reason to do this is for expressiveness, and to increase programmer productivity. However, as the results of an evaluation of Niche [1] show, programmers did find Niche reasonably easy to use. A more important reason for developing declarative programming tools is to better avoid programming errors, not least because testing and debugging management code is difficult. A declarative approach makes for fewer programming errors, but also, such languages lend themselves more readily to program analysis. The management program can be checked, at development time, to ascertain that certain invariants are upheld. For instance, it would be useful to prove that irrespective of the path taken to arrive at a given configuration that the resultant configurations are isomorphic. For example, a given configuration could be an initial configuration, or reached by ramping down, in runtime, a larger configuration (with more components in a given tier or group), or reached by ramping up, in runtime, a smaller configuration.

Some work was done in Grid4All to develop such high-level management programming tools. Currently, initial configurations can be described in high-level ADL (Architecture Description Language) which is then compiled to Niche code (this is also available on the Niche site). High-level workflow tools have been developed but have not yet been integrated with Niche. More remains to be done in this area.

5.4 Applications

In order to test and demonstrate Niche, as well as to experiment with decentralization of management we developed two self-managing services. They are available on the Niche Web site [4] , together with the system and documentation. These services were YASS: Yet Another Storage Service; and YACS, Yet Another Computing Service. Each of the services has self-healing, self-turning and self-configuration capabilities. Each of services implements relatively simple self-management algorithms, which can be changed to be more sophisticated, while reusing existing sensing and actuation code of the services. These applications as well as the Niche system and documentation are available on the Web [4]

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

when the total free storage is below a specified threshold. In [6] we describe YASS and the lessons learned as regards how to design decentralized management.

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

5.5 Limitations

Niche is a research prototype and the current implementation (June 2009) has its limitations. Firstly, there are a number of engineering gaps and inconveniences, i.e. things that we know how to do better but did not have the resources to do so. One chapter of the Niche documentation is dedicated to listing these. Examples are an unhealthy reliance on the boot-node, to garbage collection issues, to some omitted caching and other optimization measures. Furthermore the coupling to VO Management tools is incomplete, in particular to the resource management service and to use Niche it is necessary to work around this. Finally, we do not make use of virtual machine technology (e.g. Xen), though Niche and the VO Mgt tools would greatly benefit from this, in particular to enable for fine-grained resource control and to achieve application isolation.

6. Conclusions

In the Grid4All vision non-professionals should be able to form a VO to collaborate and share their resources. In that VO, they should be able to install, deploy, use and manage non-trivial distributed applications with minimal effort. This requires that the applications are self-managing. Furthermore, in these environments, resources are at their most volatile, and therefore common mechanisms to achieve self-management will not work. A single management node that continuously monitors the entire system and keeps an up-to-date system map introduces not only a single-point-of-failure but more importantly a bottleneck. The more dynamic and volatile the system is the more environmental events will be generated.

Niche is a distributed component management system designed for such scenarios, i.e. for a democratic Grid. Niche demonstrates scalability and churn-tolerance. Niche supports decentralized management, and the sensing/actuation infrastructure has features to reduce messaging.

The essential and distinguishing feature of Niche is that it is a platform to support self-management in systems with volatile resources. We were guided by our democratic Grid scenarios, but Niche is just as relevant for edge and hybrid clouds. It has been pointed out [5] that it makes economic sense for businesses to integrate their own infrastructure with a cloud, generally running their own infrastructure at full capacity and using the absolute minimum of cloud resources for peak and excess demand. This introduces the kind of volatility with cloud resources being grabbed upon need and then being released as soon as possible, that Niche was designed to handle. Furthermore, full resource utilization as opposed to overprovisioning often introduces resource volatility, from the viewpoint of the individual application managers, even when there is little volatility in the VO as a whole. This is because even small changes in the load or needs of one application triggers a reallocation of resources when some applications are ramped up and others down. Thus it is possible, maybe even likely, that Niche and the principles upon which it was built are relevant to all systems with high resource volatility, including very large-scale data centres.

7. References

1. Qualitative Evaluation of Niche, in Deliverable 5.3 Grid4All project

2. See for instance, http://www-01.ibm.com/software/tivoli/autonomic/ or http://en.wikipedia.org/wiki/Autonomic_Computing

3. Fractal Web Site, lhttp://fractal.ow2.org/

4. Niche Web Site, http://niche.sics.se

5. Mark Burgess, “The Cloud Minders”, in ;login, the USENIX Magazine, April 2009

6.Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, Seif Haridi, “A Design Methodology for Self-Management in Distributed Environments” - accepted for the 2009 IEEE International Symposium on Scientific and Engineering Computing (SEC-09), Vancouver, Canada, August 29-31, 2009

7. N. De Palma, K. Popov, N. Parlavantzas, P. Brand, and V. Vlassov, “Tools for Architecture Based Autonomic Systems”Fifth International Conference on Autonomic and Autonomous Systems, 2009. ICAS '09. 20-25 April 2009 Page(s):313 - 320

8. Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov, Per Brand, “Distributed Control Loop Patterns for Managing Distributed Applications”
The workshop on Decentralized Self Management for Grids, P2P, and User Communities, 21 Oct 2008, Isola di San Servolo (Venice), Italy In: 2008 2-nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, 2008, IEEE Computer Society, pp. 260-265, Oct. 2008

9. Ahmad Al-Shishtawy, Joel Hoglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov and Per Brand, “Enabling Self-Management of Component Based Distributed Applications”
In: From Grids to Service and Pervasive Computing. Proc. of the CoreGRID symposium 2008, Las Palmas de Gran Canaria, Spain, August 2008. pp. 163-174, CoreGRID Series. Springer.

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.