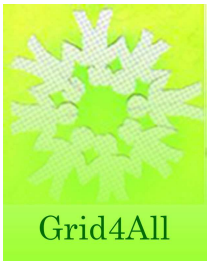


Project no. 034567

Grid4All

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

D1.1: State of art of overlays, component models, grids, and autonomic management Requirements and initial design of infrastructure

Due date of deliverable: June 1, 2007
Actual submission date: June 27, 2007

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

Organisation name of lead contractor for this deliverable: SICS (Swedish Institute of Computer Science) and KTH (Royal Institute of Technology)

Release: 1

Authors and contributors: Per Brand (SICS), Joel Höglund (SICS), Konstantin Popov (SICS), Ahmad Al-Shishtawy (KTH), Vladimir Vlassov (KTH), Jean-Bernard Stefani(INRIA), Noel de Palma (INRIA), Fabienne Boyer (INRIA), Nikos Parlvanzas (INRIA)

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
DHT	Distributed Hash Table
DKS	Distributed K-ary System
EMS	Execution Management Services
GRAM	Grid Resource Allocation and Management
GSI	Grid Security Infrastructure
GT	Globus Toolkit
JSDL	Job Submission Description Language
MDS	Monitoring and Discovery System
OGF	Open Grid Forum
OGSA	Open Grid Service Architecture
PKI	Public Key Infrastructure
QoS	Quality of Service
RM	Resource Management
SAGA	Simple Api for Grid Applications
SAML	Security Assertion Mark-up Language
SLA	Service Level Agreement
SOA	Service Oriented Architecture
VO	Virtual Organization
W3C	World Wide Web Consortium
WSA	Web Service Addressing
WSDL	Web Service Description Language
WSDM	Web Service Distributed Management
WSRF	Web Service Resource Framework
WSS	Web Service Security
XACML	Extensible Access Control Mark-up Language
XML	Extensible Markup Language

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

The primary purpose of the infrastructure (or overlay services) being developed in WP1 in the Grid4All project is to support the management of dynamic virtual organizations (VOs). We use management in a very wide sense which includes resource management (including discovery and monitoring), application/component/data management (including reconfiguration and deployment), and member management (including member monitoring and control).

The VOs that we target are Internet-based and dynamic along two dimensions. The first dimension is *churn*, indicating that the identities of the individual members of the VO and the resources that they bring into the VO are constantly changing. The totality of resources and members, however, remains roughly the same. The second dimension is that of VO *evolution*, indicating that the number or amount of resources, components, services and members also changes. Due to the high rate of system change and the fact that we target non-professional collaborations a high degree of autonomic management (self-management) is mandatory. The infrastructure must thus not only support management but autonomic management as well.

Our approach is to combine, integrate and extend recent results in autonomic computing together with structured peer-to-peer systems (overlays). Overlays (such as DHTs) are known to deal well with churn and are largely self-managing, and have shown their utility for data-centric services. The challenge here is to extend this to more sophisticated services and to couple them to VO management. Progress has also been made in the area of autonomic management where tools and methods have been developed for at a high level specifying architectures (ADLs) and behaviours to achieve self-healing, self-tuning, self-configuration and self-protection. The challenge here is to extend this to dynamic Internet-based VOs and systems, which in our approach is where the overlay-based support services come into play.

Our state of the art survey spans a wide area. From the preceding paragraph it is clear that overlay networks, structured peer-to-peer and DHTs need to be covered. Furthermore, we describe the state of the art as regards peer-to-peer services (e.g. publish/subscribe) built on top of basic overlays. The area of autonomic management is also covered. As we are developing lightweight Grid infrastructure for dynamic VOs the state-of-the-art as regards grid services is relevant. A major consideration here is that traditional Grids assume a more static VO structure, have different assumptions about reliability of individual machines, and can allow themselves more centralized implementations. In the dynamic VOs that we are targeting it is normal to create and manage applications and services inside the VO making use of VO-wide resources. The VO is thus not only managing monolithic services but also individual components, repairing and re-configuring upon need. We therefore need to consider state-of-the-art of component models and autonomic management at that level.

One can divide the control-based approach to management into four aspects, 1) sensing – monitoring, 2) actuation – control, 3) decision – management logic) and 4) the execution platform. In our design the infrastructure supports management by dealing with three of these four aspects, all excluding point 3 - decision. It provides sensor and actuation services, for monitoring, control and discovery of individual system elements as well as aggregates thereof. It also provides the platform for management logic execution so as to allow multiple management nodes, load-balancing, and dynamic delegation. This design allows for considerable reuse of previous results from autonomic computing in cluster-based systems where the focus has been on the management logic (as with a suitable component model sensing and actuation is fairly straightforward).

There are a number of requirements that follow from the proposed division of labour between the infrastructure and the higher-level management, mainly being developed in WP2. They range from fairly straightforward requirements, such as robustness and completeness of the sensor/actuation services to ones that are very specific to highly dynamic VOs. Naively extending autonomic management based on monitoring and control of individual system elements from fairly static cluster environments to dynamic VOs would result in an overwhelming flood of status information and require a continuous very detailed stream of commands to adjust the system in the face of massive churn. Not only might this exceed bandwidth/storage

limitations, but the complexity of this micro-management would make automation of the management difficult. This leads to requirements that stipulate that the sensor/actuation services must be able to provide a more abstract view of the system, by, for instance, being able to deal with aggregations.

Another important requirement for an Internet-based system is that sensing is done in a publish/subscribe manner (if there is no interest in a churn event then it is not reported) and that the sensing services include discovery. The infrastructure services provide a location transparent substrate, so that a piece of management logic (together with its internal references to system elements) may be dynamically delegated. Multiple management nodes will access the same VO-wide infrastructure. A final requirement (one that we have to date only taken a small step towards) is that this delegation can be made intelligent in the sense that a piece of management logic may be placed close to the system elements that it is monitoring.

We have identified three critical overlay/infrastructure services for VO management, two of which are described in this report. They come with a well-defined APIs (both syntactic and semantic). One deals with resources and the other with components. Basic resources include compute cycles on a single machine and storage on a single machine. More generally, resources are useful system entities that come in two flavours; used, or free. Software components are handled by the component service which is concerned with deployment (which causes a resource state change from free to used), as well as configuration and bindings. The word component is here used loosely and allocating storage for data services also acts through the component service.

The peer-to-peer core, at the heart of the described infrastructure overlay services, is called Niche, and is based on DKS, a DHT developed at SICS and KTH. In the context of this work DKS had a number of advantages compared to other DHTs, specifically stronger guarantees (responsible node always found in non-partitioned systems) and symmetric replication. It has been extended along several dimensions, including provisions for various types of event handlers and constrained mutable data to handle sensor subscriptions.

This is work in progress, to date a number of infrastructure service functions have been designed, and some of these have been implemented in the first prototype. A major driving force for the more abstract service functions is that there is a clear advantage (in terms of lower messaging overhead, better decentralization, robustness etc.) in realizing them as overlay services as compared to abstractions used exclusively at the management nodes.

2 Introduction

2.1 Work package and Task Context

Work package 1 aims to build a lightweight grid infrastructure suitable for Grid4All scenarios based on structured overlay networks (structured peer-to-peer). Grid4All scenarios are Internet-based and characterized by high dynamicity (in amount/identity of resources) and non-professional users and administrators. We aim to leverage and combine recent research results in structured overlay networks, component models and self-managing systems, adding and extending to the state-of-the-art where necessary to realize our goal.

In this deliverable we give an overview of the state-of-the-art in the relevant fields and then go on, in that context to identify the requirements for a Grid4All infrastructure and relate them to the state-of-the-art. Finally we present an initial design of the infrastructure, and its interaction with higher-level services and management as being developed in other work packages.

This report (deliverable 1.1) reflects work done in task 1.1 (overlay infrastructure) 1.2 (component framework), and to a small extent task 1.3 (overlay service composition) and thus covers much of the workpackage. Work done in task 1.4 (replication and consistency models) will in due course be described in deliverable 1.3. While the final task of the work package (1.5 – integration) has not yet commenced.

2.2 Organization of the Deliverable

In chapter 3 we describe the goals of the project, in general, and work package 1, in particular.

We characterize dynamic virtual organizations (VOs) and the associated management challenge. We also briefly describe the dividing line between the role of the infrastructure being developed in WP1 and the other management functions being developed in WP2.

The next part of this deliverable consists of a number of state-of-the-art chapters:

4. Overlays
5. Grid Services
6. Web and Grid Security Services
7. Peer-to-peer based Services
8. Component Models
9. Autonomic Management

The relevant state-of-the-art covers a wide area. In Grid4All we are trying to develop a lightweight Grid infrastructure for dynamic VOs so the state-of-the-art as regards grid services is relevant (chapters 5 and 6) A major consideration here is that traditional Grids assume a more static VO structure, have different assumptions about reliability of individual machines, and can allow themselves more centralized implementations.

Furthermore we propose to base our infrastructure on structured overlays and therefore need to cover the state-of-the-art on overlay networks, like DHTs (chapter 3). As we aim to build infrastructure services on top of overlays, we also discuss the state-of-art on higher level services build on top of overlays (chapter 6).

In the dynamic VOs that we are targeting it is normal to create and manage applications and services inside the VO making use of VO-wide resources. The VO is thus not only managing monolithic services but also individual components, repairing and re-configuring upon need. We therefore need to consider state-of-the-art of component models (chapter 7).

Finally, in the state-of-art section, as Grid4All VOs are characterized by high rates of change we believe it crucial that VO management, supported by the infrastructure, can be highly automated. This includes managing complex applications consisting of multiple components running on different and potential volatile machines. In chapter 8, we summarize state-of-the-art in autonomic management.

Finally, in chapter 10 we present our initial but incomplete design of the overlay infrastructure, with particular focus on managing resource and components. In the DOW the design and specification of the overlay infrastructure is due in deliverable 1.2 (preliminary report of m24) and 1.5 (final report of m30).

3 Goals and Context

3.1 Goals

The primary purpose of the overlay services being designed in the Grid4All project is to support the management of dynamic virtual organizations (VOs). We use management in a very wide sense which includes

- resource management – including discovery and monitoring
- application/component/data management – including (re-)configuration, deployment
- member management – including member monitoring and control

To further clarify the management support issues we need to define and describe the following:

1. What are the characteristics of the systems, dynamic VOs, that are to be managed?
2. What does the management do? Is it human or autonomic management?
3. What do we mean by support, and what is the dividing line between the support service itself and the management functionality?
4. What are the requirements of the support services? Who manages the support services?

3.1.1 VOs and their characteristics

We define a virtual organization to be a management domain that controls, coordinates and aggregates services and resources provided by other management domains (called members) according to some given policy. This definition clearly allows for hierarchies in that VO may manage a collection of VOs, and so on, recursively. However the notion of member also includes such basic management domains as a home pc and its owner. Depending on the policy VOs may be thought of as an organization that pursues the common goal of the members, or as a general framework within which groups of members can form long and short-term collaborations based on non-VO wide common goals and interests.

VOs may be both dynamic and Internet-based. The dynamism can be seen along two dimensions. The first dimension is churn, which in the context of the VO means that the identities of the individual members of the VO and the resources that they bring into the VO are constantly changing, while totality of resources and members remains roughly the same. The second dimension is VO evolution, indicating that the number or amount of resources and members also changes. There are also aspects of both churn and evolution regarding the types of services that the VO is running on behalf of members, the member usage of those services, and VO policies.

We take a peer-to-peer view in that members, in general, both provide resources and services to the VO, and make use of the provided resources and services. Members are thus both producers and consumers of resources. However policy may dictate that some members may only produce (provide) or only consume. Policy may also dictate that some class of member should provide during part of the day but is free to consume at other times (a Grid4All school simulation scenario might dictate that contributed home machines are prevented from consuming resources during school hours).

3.1.2 VO management

Within the framework set by VO policy, members provide resources and services to the VO. VO management monitors, aggregates and present these resources and services to the VO members. Services can also be created within the VO making use of the aggregated resources. The VO management is thus responsible for deploying and managing an application that makes use of aggregated computation and/or

storage facilities. Note that managing these applications, presented to members as services, in the face of churn (and to a lesser extent, evolution) will, in general, require frequent management interventions.

We can divide the concerns of VO management into four aspects.

1. *Resources*: resources here are defined as that which members bring into the VO. It represents such basic things as computation power and data storage. The VO must be able to, upon need, aggregate needed resources.
2. *Components*: components here are defined as the constituent parts of services that are created in the VO by utilization of aggregated resources. For instance, the VO may deploy a distributed application consisting of several components on a number of member machines. Part of management may be to maintain the application in the face of individual machine failure or disconnection. 'Components' are used here in a very generic sense, and might include for instance storage 'components' used in constructing a data repository service.
3. *Members*: In general, members come and go, and this is monitored by the VO management. Policy dictates the type of member roles that exist with the VO. With a given role there are both obligations and privileges; it is part of member monitoring to check that members do meet their obligations.
4. *Services*: The VO management ensures that the services that are provided by the VO are made available or published in such a way as member can discover them.

A scenario illustrating the four aspects is as follows: member M joins the VO (aspect 3) and provides the resource R during school hours (aspect 1). Sometimes the VO makes use of the resource R and deploys some component C on R (aspect 2) as part of an application A. It also publishes and wraps A (aspect 4) so that other members can discover and use A.

In principle management may be either human or autonomic. However, in the context of dynamic VOs with high rates of churn continuously forcing the redeployments of many components, a significant part of management will need to be autonomic.

3.1.3 Management Support

The control approach of management distinguishing between three aspects of management

1. Sensing: the ability to sense or observe the state of system and system elements. In general observation may be active (triggered by the observer) or passive (triggered by the element).
2. Actuation: the ability to control and affect the system elements.
3. Decision: the logic or intelligence that given good knowledge of the system elements (provided by sensing) decides or plans actions (done by actuation) to ensure continued proper operation of the system.

Note that point 3) is the most open-ended. The decision or management logic may range from simple rules (of the type if A is sensed then actuate with B) to sophisticated AI techniques (e.g. reinforcement learning).

At *first approximation* the dividing line between the support service (i.e. the infrastructure) and the rest of the management is that sensing and actuation is done by the infrastructure while the management logic is done elsewhere. The support services are thus *sensor* and *actuation* services. Sensing in the Grid4All context is used in a general sense and includes discovery (e.g. resource discovery) as well as sensing state changes of known system elements. Sensing and actuation may work on aggregated system elements as well as individual system elements.

The classical division of control-based management into sensing, actuation and decision, ignores the question of the computational platform, the question of where in a distributed system management logic is actually run. The infrastructure can help here by offering a suitable platform with a view to help management

with scalability, load-balancing and fault-tolerance. This aspect of the infrastructure will become clearer in chapter 10.

In summary we have dynamic system characterized by high levels of churn and evolution. There are four types of system elements, resources, components, services and members. We can divide management into the four aspects sensing, actuation, management logic, and the computational platform. The infrastructure is responsible for three of these four aspects (all but the management logic). Our goal is then to develop an infrastructure that provides *good* sensing, actuation, and management computational services.

4 Overlays

The following chapter will briefly present the relevant issues and state of the art relating techniques for peer-to-peer based overlay networks.

4.1 Introduction to Peer-to-peer Overlays

A peer-to-peer system is a collection of nodes or peers. Each peer can play the role of both a client and a server. It is like a community of members where each member provides some service to the community and can also consume services provided by others. An advantage of peer-to-peer systems over traditional client-server systems is that peer-to-peer systems distribute the functionality of the server among peers increasing the scalability and the robustness of the system. Managing large-scale distributed systems is much more challenging than managing client-server systems. To solve this problem, peer-to-peer systems should support self-management to reconfigure themselves without human intervention. Self-management consists of four main elements: self-configuration, self-tuning, self-healing, and self-protection. These four elements will be discussed later in this document.

Usually the peers in a peer-to-peer system are equal. That means that peers have the same capabilities. Sometimes it is useful to have some peers with more capabilities than the others to increase the performance of the systems. These peers are called super-peers. The choice of super-peers can for example be based on machines with more powerful resources than other peers, or on nodes belonging to members which are considered trusted within the system.

Peer-to-peer systems can be classified into two main categories: structured and unstructured systems. This classification is based on how peers are organized and how they communicate with each other.

4.1.1 Unstructured Peer-to-Peer Overlays

Peers in a peer-to-peer system connect with each other in a specific manner forming an overlay network over the physical network. In the case of a fully unstructured peer-to-peer network, there are no rules on how nodes connect with each other. Each peer selects a number of random peers to be its neighbors and connect with them. This results in a random topology for the overlay network. Gnutella [GPS02] is one of the most popular unstructured peer-to-peer systems available today.

In a network without known structure, the only way to find a resource (like a file) is to use flooding broadcast where a peer sends a message to all its neighbors which forward the message to their neighbors. This continues recursively with a certain time-to-live (TTL) until the resource is found or the search fails. Since flooding is not a scalable technique this causes problems in large networks.

On the other hand, the lack of structure has some advantages. Peers need to keep only local information about its neighbors so peers joining and leaving have only local impact. Peers not directly connected with the joining/leaving node will not be affected. This increases the self-configuration and self-healing properties of unstructured peer-to-peer systems. Due to this local knowledge, self-tuning becomes less efficient in unstructured peer-to-peer systems.

4.1.2 Structured Peer-to-Peer Overlays

In structured peer-to-peer systems peers connect with each other following rigid rules. Each peer is assigned a unique ID, in the following called NodeID, and connects with some specific peers according to a known protocol. Resources are also assigned IDs from the same ID-space through some hash function. All IDs are from the same ID-space. This structure makes it easy to find a peer or a resource using its ID. According to the structure of the overlay each node becomes responsible to a part of this ID space. Any new node that

joins the system gets an ID and takes the responsibility for a part of the ID space. The node also learns about its neighbors in the system. This enables structured peer-to-peer systems to do so called Key Based Routing (KBR) [DZD03]. KBR means that given a key (that is also selected from the same ID space) the overlay can efficiently rout a message to the node responsible for this ID using the knowledge of neighbors. This routing is generally done in O(log N) overlay hops in a network consisting of N nodes given that each node knows about (log N) neighbors.

KBR enables the creation of Distributed Hash Tables (DHT). This allows the system to store (Key, Value) pairs by storing the Value at the node in the overlay responsible for its Key. Later it is possible to lookup the Value of any Key the same way. These Put/Get operations can be done from any node in the system by routing requests to the responsible nodes.

Efficient and scalable search comes at the cost of the need to maintain the structure of the system. A peer joining/leaving affect O(log N) other peers. The problem of churn appears in structured peer-to-peer systems when peers join and leave continuously. Self-configuration and self-healing become more challenging in structured peer-to-peer systems. Self-tuning (load balancing) is achieved by using a uniform hash function to assign resource IDs.

Chord [SMK01], Pastry [RD01], and DKS [AEB03] are some of the structured peer-to-peer systems available today. Among the open research issues are how to efficiently merge a partitioned network of two overlays back into one overlay when the underlying physical network heals. See for instance [SGH07].

4.1.3 Administrative Autonomy

The NodeIDs and item keys are randomized in most systems. If the IDs are calculated using a uniform hash function, it has the advantages of achieving load balancing and robustness. On the other hand this randomization does not give control over the placement of items. One cannot control which node will be responsible for the key of a specific item. In some situations this can not be accepted. For example if the overlay crosses multiple organizations each organization may want its items to be stored only on its own nodes and control access for other organizations. This is called content locality. It is also possible that an organization wants the routing path to stay within the organization when routing local messages. This is called path locality.

One approach to achieve content locality and path locality is to use multiple overlays [MD04] in a hierarchy that reflects the organizational hierarchy as shown in Figure 1.

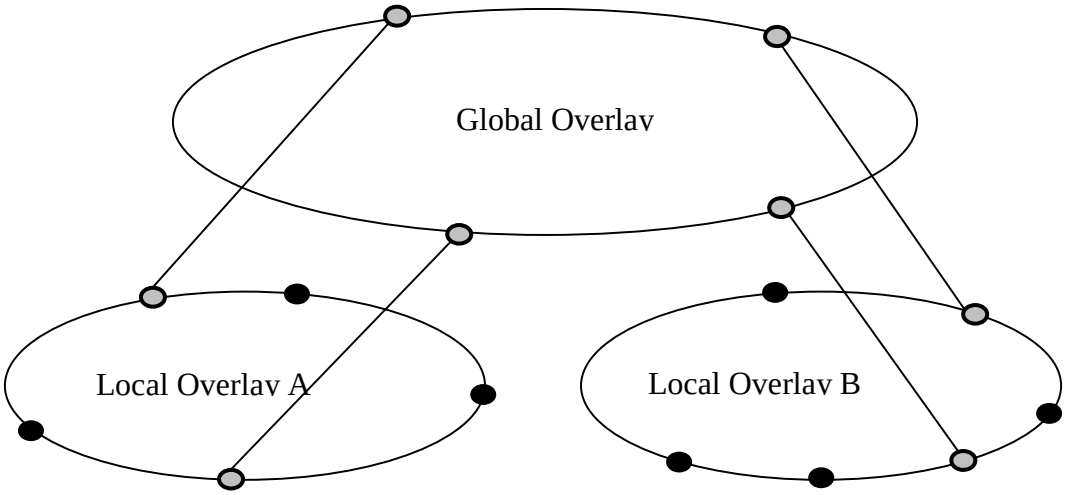


Figure 1

Each organization maintains its own overlay of its choice such as Pastry, Chord or DKS. The only constraint is that the protocol used should support KBR. The local overlay assures content locality and path locality. A common global overlay links the organizations together. The KBR allows overlays with different protocols and parameters to interact together as shown in Figure 2.

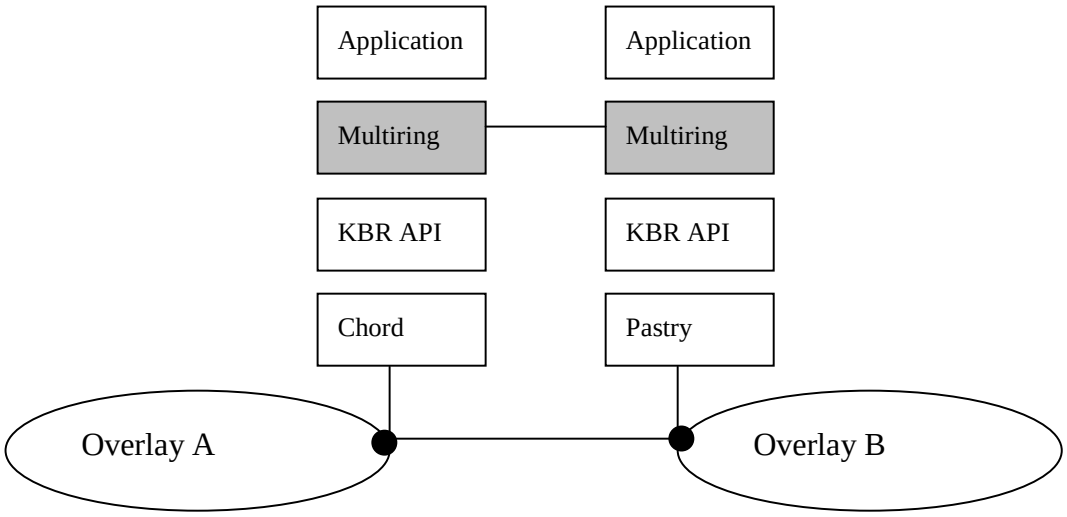


Figure 2

Routing and lookup across organizations are performed as follows. Each local overlay has a unique ID that distinguishes it from others. The global overlay has an ID of all zeros. Each node in an organization joins both local overlay and global overlay with the same NodeID unless the node has limited connection such as being behind NAT or firewall. In that case it only joins the local overlay. Nodes in both local and global overlays are called gateways. The gateway nodes then announce themselves to others by subscribing to

any-cast group with ID equals to the overlay ID they are gateway to. A gateway node will register to any-cast group in the global overlay with ID equals its local overlay ID and same time register to local overlay to the any-cast group with ID of all zeros.

Organizations store their items on the local overlay. If an organization wants some items to be available to other organization it stores a special redirection Item in the global overlay that consists of a Key equals to the Items Key and a value equals to the OverlayID where the real value is stored. Redirection items are the only items that can be stored in the global overlay. This allows organizations to perform global lookups and find organizations responsible for specific Items.

Now routing will use both OverlayID and Key to rout a message to the responsible node. If the OverlayID is not known it will be looked up from the global overlay. When a node receives a message that needs routing it does the following: first it checks if the OverlayID of the message matches one of the OverlayIDs that the node is member in. If it matches, this node is a gateway for this overlay and the node just forwards the message in the overlay normally from this point. If not then this node need to locate a gateway for the target overlay. This is done using any-cast. If the node is in the global overlay then the any-cast is forwarded to the group with ID that matches the target OverlayID. If the node is in local overlay then first forward any-cast to group ID of all zeros (global overlay) and then proceeds as described above in the global overlay case.

Besides solving the problem of content locality and path locality, this approach also solves other problems. In large overlays that cross multiple organizations it is difficult to agree on the same set of protocols and parameters. With multiple overlays each organization can select the most suitable protocols and fine tune its overlay according to application needs. Also this approach solves the problems that appear in peer-to-peer overlays because of limited connectivity caused by firewalls and NAT. Gateway nodes with full connectivity can rout messages to outside of the organization and vice versa.

The concept of content and path locality was introduced by SkipNet [HJS03]. But the presented approach can be more suitable for Grid4All because besides achieving content and path locality it also provide additional functionality such as integrating organizations with different requirements, efficient multicast on sub-rings, and balancing load separately on each ring.

4.1.4 Peer-to-Peer Conclusion

Both structured and unstructured overlays provide a useful communication substrate upon which useful services can be built such as broadcast, multicast, and lookup operations. From the point of view of the Grid4All project requirements, structured overlays are more attractive because the constraints on their structure makes them more robust and efficient and can provide stronger guarantees. It has also been also shown that structured overlays can emulate the functionality of unstructured overlays with comparable or even better performance [CCR05]. It will be further investigated to which extent administrative autonomy should be supported.

5 Grid Services

This section presents state of the art in Grid middleware and services, namely in related specifications (e.g. OGSA) and standards (e.g. WSRF, WSD, WSDM) as well as in implementations of Grid services (e.g. Globus Toolkit 4, and gLite). Also included is a discussion on whether some of the standards, specifications and implementations can be used to develop and/or provide overlay services that enable self-management in ad-hoc (dynamic) Grids considered in the Grid4All project. We shortly analyse whether existing standards and implementations of monitoring, discovery and resource/VO management services fulfil major requirements to the overlay services specified in Chapter 2, namely, robustness and being self-manageable.

Grid Computing is the field of computer science dealing with Grid systems. Grid is a SOA distributed system built on the Internet that enables integration (sharing), virtualization and management of distributed services and resources across department and organizational boundaries in a secure, efficient manner; provides seamless access to services and resources and supports forming Virtual Organizations –dynamic federations of heterogeneous organizational entities sharing data, metadata, processing and security infrastructure.

Motivation for Grids:

- To achieve greater performance and throughput put by aggregating resources from different organizations, e.g. forming Virtual Organizations (VOs)
- To aggregate and to provide seamless access to resources (CPUs, storage, applications) within VOs

According to the three point grid checklist by Ian Foster [Fos02], “A grid is a system that:

(1) coordinates resources that are not subject to centralized control…; (2) … using standard, open, general-purpose protocols and interfaces…; (3) … to deliver nontrivial qualities of service”. Fundamental challenge in building Grids is to integrate resource in a secure, standardized and efficient way.

In general the Grid architecture is formed of the following layers [FKT02] listed below in order from lower to the upper.

- *Fabric layer* provides actual resources to the Grid, e.g. storage, computers, networks, code repositories, catalogues, etc. It implements operations on the resources requested by higher levels, and must provide a sort-of inquiry & resource management mechanism; Example of operations performed in this layer include: starting and monitoring processes; putting and getting files; network transfers and access to code repositories, CVS and databases.
- *Connectivity layer* provides communication (transporting data between fabric layer resources), authentication and authorization (verifying identity and rights of users and resources). In respect to security, the connectivity layer provides integration of VO-level security with various local security solutions. This layer provides connectivity API from the resource (upper) to the fabric (lower) layers.
- *Resource layer* deals with management of and access to individual resources, e.g. secure negotiation, initiation, monitoring, accounting on individual resources. At this layer, each available grid resource has its specific well-defined API. Global state is either not known or ignored (not used) at this layer.
- *Collective layer* deals with global resources and interactions across collections of resources. It provides resource discovery, brokering, monitoring and security. It is formed of components and services that deal with the coordination of multiple grid resources. Examples of components and services at this level include directory services; discovery services; monitoring and diagnostics services; co-allocation, scheduling, and brokering services; workload management systems and collaboration frameworks; data replication services; community accounting and payment services; collaboration services.
- *Applications laye*r includes user applications running in a VO environment. The applications use APIs from underlying layers (resource and collective layers) in the Grid architecture.

A traditional assumption for Grids is that the infrastructure is built on reliable, stable and (highly)-available resources and services provided by VO members forming the Grid. This means that the Grid infrastructure services are reliable, robust and (highly)-available. Even if resources and services are offered by external providers, an SLA-based mechanism allows controlling quality of the services provided. The assumption on robust and stable (static) Grid infrastructure and (highly) available services deployed in the infrastructure is not realistic for ad-hoc Grids considered in the Grid4All project. An ad-hoc Grid is built on resources voluntarily donated by ordinary users and small organizations forming VOs to share the resources. The pool of the donated resource as well as VO members in the ad-hoc Grid can be highly dynamic due to the churn (i.e. resources and members can frequently join and leave the VO) and due to VO evolution (i.e. the number of members can increase or decrease). The ad-hoc Grid infrastructure and services deployed on it should be able to tolerate churn and manage VO evolution. To be able to rapidly react on this dynamism the ad-hoc Grid infrastructure should be self-managing.

5.1 Specifications, Recommendations and Standards

Open Grid Services Architecture, OGSA, is a set of specification documents that define the overall architecture, and services to be provided in Grid environments, i.e. services and components common for Grids and Grid applications [OGSA]. OGSA specifies high-level components and services, e.g. VO management, security, resource management, data service, etc. The word “Open” in OGSA means compliant to open set of standards and extendable, as standard protocols enable interoperability at the communication level whereas standard APIs enable interoperability at the service level. OGSA adopts the Service-Oriented Architecture (SOA) using service orientation to virtualize resources that become available as (web-)services. This allows grid services to be developed and described using standards such as WSRF (Web-Service Resource Framework) and WSDL (Web-Service Description Language); and to be accessed using standards such as XML and SOAP so that a grid environment supports services which are provided and consumed via standard protocols.

OGSA defines the following groups of grid services to be provided and used in a typical grid environment:

- Execution management services, e.g. a job management service;
- Data management services, e.g. data access and integration;
- Resource management services, including resource discovery and monitoring
- Information services;
- Security services;
- Self-management services;
- VO management services.

The above list of OGSA services is not standardized although most of implementations of the above services in existing Grid environments are compliant to OASIS standards such as WSRF, WSN and WSDM. Most of the services might have different names in particular grid environments. Many of the OGSA services are implemented in existing Grid environments and toolkits such as Globus Toolkit (GT4) [GT] [Fos06] and gLite [gLite].

OGSA requires *stateful* services in order to discover, manage and utilize grid resources and services. Currently, there are two competing sets of specifications related to (stateful) web-services, notification mechanisms and resource management:

1. WSRF (Web Services Resource Framework) and related technologies WSDM (Web-Services Distributed Management) and WSN (Wes-Services Notification) standardized by OASIS;
2. A set of WS-* specifications which is under development (some of specifications have been already submitted to W3C) by a consortium of leading industry partners including BEA Systems Inc., Computer Associates Int., HP, IBM, Intel, Microsoft, Sun, Systinet, TIBCO Software, and others.

The goal of all the above specifications is to define concepts and to standardize properties, semantics and message formats of operations (WS portTypes) related to resources and services, events and management. The specifications are rather consortium recommendations that the WS and Grid communities agree upon.

5.1.1 WSRF, WSDM, and WSN Standards

Web Services Resource Framework (WSRF) [WSRF] is a generic and open framework for modelling and accessing stateful resources using Web services. The WSRF specification is one of the specifications (recommendations) standardized by OASIS [OASIS]. The development of WSRF has been motivated by an ongoing trend in the grid community towards providing web service interfaces to grid resources in order to make them available in the same interoperable way as ordinary web services.

In order to merge stateful grid resources with stateless web services, web services should be extended with a notion of *state*. To model a stateful web-service (i.e. resource), WSRF defines a concept of *WS-Resource* that is the composition of a resource and a web-service through which the resource can be accessed. A state of the WS-resource is modelled by its *Resource Properties* which are exposed to a client as a set of name-value pairs. The resource properties can be presented to a client in the form of a Resource Properties Document, or in some custom application-specific form. A web-service associated with the resource provides interfaces to access (get/put), to update, and to insert and to delete resource properties. The WS-resource can also include interfaces to manage lifetime of the resource and to monitor its state by means of pulling or notification callbacks. A client can subscribe for notification of lifetime events, resource properties changes. A reference to a WS-Resource is represented by its endpoint reference (EPR).

The WSRF standard also includes the Web Services Service Group (WS-ServiceGroup) specification that defines a concept of *Service Group* that represents a collection of web-services or WS-Resources. The Service Group's members are modelled as *entry WS-Resources*. Each entry has the ServiceGroupEntry interface to directly access the entry. Entries can be removed and new entries can be added to the collection. Constraints on membership are defined in the MembershipContentRule resource property of a Service Group. Service Groups can be used to build registries of resources and services, e.g. the GT4 Index service.

WSRF is considered and used together with other two sets of specifications standardized by OASIS, namely Web Services Notification, WSN, [WSN] and Web Services Distributed Management, WSDM [WSDM].

- WSRF includes five specifications: WS-Resource, WS-ResourceProperties, WS-ResourceLifetime, WS-ServiceGroup, WS-BaseFaults;
- WSN includes three specifications: WS-BaseNotification, WS-BrokeredNotification, and WS-Topics;
- WSDM includes Management Using Web Services (WSDM-MUWS) and Management of Web Services (WSDM-MOWS).

The three above standards define a set of web-services technologies related to WS-resource access (WSRF), monitoring (WSN) and management (WSDM). Each of the specifications provides definitions of concepts, describing architecture, semantics of operations, port-types, messages, schemas, etc. These technologies are interrelated to each other: WSN is using WSRF; whereas WSDM is using both WSRF and WSN. All of the specified technologies use the three core web-services technologies specified by W3C: SOAP, WSDL and WS-Addressing (WSA).

The ***WS-Notification (WSN)*** standard specifies functionalities and interfaces for publish-subscribe notification mechanisms that enable event-driven services. WSN introduces concepts of Notification, Subscription, Notification Producer, Notification Consumer and Notification Broker. WSN uses WSRF to model the above concepts. Operations of the Notification Producer interface are used to subscribe for notifications from a given producer or to pull the producer for the notification messages; whereas the Notification Consumer interface is used to notify the consumer according to the subscription. The Subscription WS-resource represents a subscription. Its interface is used to control a lifetime of the

subscription, e.g. to cancel, to suspend or to resume the subscription. In addition to ordinary notification, WSN defines support for notification filtering based on topics, on message content or producer state. Filtering is defined by a consumer when subscribing for notifications. WSN also includes support for notification message brokering, which allows the notification producer and notification consumer to be separated by intermediates – notification brokers. Thus, the WSN standard defines support for developing publish/subscribe notification mechanisms, event-driven Grid components and services, e.g. services for resource monitoring and management.

WS Distributed Management (WSDM) is another OASIS WS standard related to WSRF and WSN [WSDM]. WSDM defines interfaces for managing a wide range of resources, from computers, network routers, printers to small electronic devices. WSDM complements WSRF by specifying how stateful resources should exchange management information. The WSDM standard consists of two specification, Management Using Web Services, MUWS, and Management of Web Services, MOWS. The latter uses and extends the former to define management capabilities for web services.

WSDM introduces concepts of *manageable resource*, *manageability consumer* (i.e. a resource management service), and *manageability capability* that is a set of properties, operations, and events, enabling a resource to be managed in a particular way. The standard defines several manageability capabilities such as the resource identity, state (and last state transition), operational status (e.g. available, partially available, unavailable), configuration. The manageability capabilities can be accessed (inspected or/and altered) by a manageability consumer, e.g. resource manager. According to WSDM, a manageability consumer can interact with a manageable resource in three ways: (1) the consumer can retrieve management information; (2) the consumer can alter the state and/or the behavior of the resource by changing the resource's management information; (3) the resource can inform the manageability consumer about events by means of the subscribe/notify mechanism specified in WSN. WSDM also defines the Advertisement capability that supports notification on the creation or the destruction of a manageable resource. For example, a resource registry (implemented as Service Group) can provide the resource creation and destruction information (notification) via its Advertisement capability.

5.1.2 WS-* Specifications

An alternative, more recent set of specifications called WS-* comes from Distributed Management Task Force (DMTF), which is a consortium of leading IT-related companies including IBM, Sun, Microsoft and others [DMTF]. The members of DMTF are trying to agree upon set of new standards which then would supersede the current standards such as WSRF, WSN, and WSDM. In a joint white paper HP, Microsoft and Intel [Cli+06] stated that the work would require about two years, and a common set of specifications might be available by summer 2008.

As stated in [Cli+06], WS-* specifications cover the following common functionalities: (1) "Resources: the ability to create, read, update and delete information using Web services; (2) Events: the ability to connect Web services together using an event driven architecture based on publish and subscribe; (3) Management: providing a Web service model for building system and application management solutions, focusing on resource management." The above functionalities covered by WS-* specifications are very similar to those covered by WSRF together with WSN, and WSDM. It is expected that the former will replace the latter. Therefore it could be worth trying to support functionalities likely to be included in the joint WS-* standards. The WS-* specifications are not considered and discussed in details here, as they are still under development. However, Grid4All will pay attention to development of the WS-* specifications and standards.

5.2 Some Existing Implementations of Grid Services

This section provides a brief survey of Grid services provided in Globus Toolkit 4 and gLite middleware that define state of the art in Grid computing.

5.2.1 Globus Toolkit 4

Globus Toolkit 4 (GT4) [GT] defines state of the art in Grid computing. Implementation of GT4 services meets requirements of OGSA and is compliant to the relevant OASIS standards, namely WSRF, WSN and WSDM. Globus Toolkit (GT) is an open source software toolkit for building grids with services written in a combination of C and Java. The C components run on UNIX platforms, including Linux; whereas the Java-only portions may be run on any platform with a Java SDK. GT4 implements WSRF specifications and provides many OGSA services, e.g. execution management, data management, information management, and security. GT4 includes three sets of components: (i) containers – Java, Python and C containers – for hosting GT4 and user-developed services; (ii) predefined OGSA services most of which are WS-I and WSRF compliant as implemented on top of Java WS core; (iii) Client libraries that enable client programs in Java, C and Python to invoke operations on both GT4 and user-developed services.

A GT container implements SOAP over HTTP as a message transport protocol; transport-level and WS-Security message-level security for all communications; WS-Addressing, WSRF, and WS-Notification functionality; WS-Resources with information about services deployed in the container. The GT4 container can host WSRF services of Grid infrastructure provided by GT4 (such as GRAM, MDS, and RFT) as well as custom Web services and WSRF services deployed by the user. Clients can use the GT4 Index service as well as GT4 container administration interfaces to determine deployed services.

Gt4 provides the following groups of services, components and systems related to execution managements, resource management, data management, and security.

Grid Resource Allocation & Management (GRAM) enables execution management and allows submitting, scheduling, monitoring and control of remote jobs executed in the Grid. For simple tasks, GRAM can use the Unix fork command to create a process for job execution at the target resource. GRAM provides a uniform, flexible interface to different backend job scheduling systems, such as Condor or PBS, via scheduler-specific GRAM adapters. GRAM can be used for negotiate access to clusters, creation of virtual machines, establishment of virtual networks, etc. It is usually deployed together with Delegation (to provide proxy credential) and RFT services (for file staging). GRAM language support includes Resource Specification Language. It is also worth mentioning OGF efforts towards unified interface to Grid execution management systems, namely, Jobs Submission Description Language (JSDL) [JSDL] and SAGA (Simple API for Grid Apps) [SAGA]. The former aims to facilitate job submission; whereas the latter aims to unify job submission and execution control API for different execution environments.

Monitoring and Discovery System (MDS) collects and provides information about the available resources on the Grid and their status. MDS is based on several components such as *MDS-Index* (a registry similar to UDDI), *MDS-Trigger* (that collects data from various sources and takes actions based on those data), and Aggregator Framework (that provides a unified mechanism used by the Index and Trigger services to collect data). Typically, MDS is structured as a hierarchical system, in which some components (aggregator index, trigger, WebMDS,) should be deployed centrally to a VO, while others (e.g. indexes) should be deployed on individual resources. It is worth mentioning that Globus MDS supports self-organizing capability, e.g. auto-registration of select GT4 services in a container at the container startup, and upstream or downstream auto-registrations specified in a configuration file. This feature simplifies building of the hierarchical monitoring and discovery system.

GT4 includes several data management services and systems such as *Data Access and Integration* (OGSA-DAI) that allows access and integration of data from multiple data sources (databases, XML files and ordinary file systems); *Replica Location service* (RLS); *Reliable File Transfer* (RFT) that supports reliable data movement based on GridFTP.

GT4 support for security includes the Globus security framework, *Grid Security Infrastructure* (GSI), which in 2005 has been called “a de facto standard for Grid security” [Bak+05], and the *Community Authorization Service* (CAS). GT4 Security supports message-level security mechanisms (complaint with the WS-Security standard and the WS-SecureConversation specification) and transport-level security mechanisms based on

TLS. GT4 security provides public-key-based authentication using credentials as well as delegation of credentials to a hosting environment using single sign-on and proxy certificates (short lived certificates issued by users). GT4 security provides an extensible authorization framework that allows different authorization schemes based on security standards such as SAML (Security Assertions Markup Language) and XACML (eXtensible Access Control Markup Language) policy language. Use of the above standards enables policy-based authorization. Additional security aspects are discussed in the following chapter.

Most of the GT4 services are WSRF-compliant, i.e. they are provided as WS-Resources that, in addition to specific functional and control portTypes, can include the standard WSRF resource property query, lifetime management and subscription/notification interfaces. WS-services (provided by GT4 and developed by the users) can be deployed and provisioned in distributed GT4 containers that form a distributed Grid environment. Services can be deployed and provisioned independently, or they can be deployed and configured (composed) to work together in a concerted way. The services are registered in the MDS that allows services to discover each other as well as a client to discover a service and to obtain its EPR. A service interacts with its clients and other services via the web-services protocol stack (the SOAP stack).

5.2.2 gLite Middleware

gLite [gLite] is another existing Grid environment that defines state of the art in Grid computing, specifically, in production Grids. Like GT4 (OGSA), gLite also follows the service oriented architecture approach. gLite includes Grid middleware services (a.k.a. Foundation Grid Middleware), which should be deployed on the EGEE (Enabling Grids for E-sciencE [EGEE]) Gird infrastructure. gLite also provide high-level grid services (e.g. visualization), which are rather optional and are supposed to help the users building their computing infrastructure. gLite middleware is a middleware stack that combines components developed in various Grid projects such as Condor and Globus, and extended by EGEE developed services. In gLite service instances can belong to multiple VOs.. The gLite middleware services are grouped in the following five service groups [Lau+06].

- Security services provide authentication and authorization based on X.509 certificates, certificate proxies and delegation. Security support in gLite also includes *Virtual Organization Membership Service* (VOMS) [Alf+04] that maintains a member database and issues signed attributes to identified users. This enables VO policy to be applied in authorization.
- Information and monitoring services are based on R-GMA (Relational Grid Monitoring Architecture) [R-GMA] that defines three types of components Consumers, Producers and a directory service/registry. Producers produce information, Consumers consumes the information, and Registry helps Consumers to lookup and to locate Producers. Information and monitoring enables job monitoring.
- Job management services include Computing Elements, the Workload Management System (WMS), and Job Provenance Service and the Package Manager. A computing element virtualizes a computing resource (e.g. a cluster), provides information on the resource and a common interface to submit and control jobs on the resource.
- A set of data services includes Storage Elements, File and Replica Catalogs, and Data Management. A storage element virtualizes a storage resource (e.g. a disk server) and provides an interface to the resource. The catalog services maintain location and other metadata (e.g. ACLs) on data (files and replicas); whereas data transfer services allows reliable and efficient data movement between storage elements. The gLite data services provide for the user an abstraction of a Global file system.

5.3 Discussion (Conclusions)

The three OASIS standards described in 5.1 – WSRF (Web Services Resource Framework), WSN (Web Services Notification), and WSDM (Web Services Distributed Management) – define support related to resources, events, and resource management, respectively. The concepts and operations (protocols) defined in the standards can be used in development of self-managing distributed systems such as dynamic (ad-hoc) Grids, which are in the focus of the Grid4All project. Even though the above standards are defined

for the web-services protocol stack, the concepts and operations defined in the standards can drive development of a Grid infrastructure and services that are based on other protocol stacks, e.g. Java RMI.

Grid4All uses the Fractal component model for design of ad-hoc Grid infrastructure and services on overlay networks. In our view, WSRF together with WSN and WSDM provide sufficient support for implementing the Fractal model. WSRF defines a set of interfaces to access resource properties as well as provides ability to define custom functional interfaces for components. WS-resource properties and ServiceGroups can be used to hold references to nested components and to support component bindings and component groups. Lifetime management operations (in WSRF), subscribe-notify operations (in WSN) and distributed resource management operations (in WSDM) can be used to construct component management interfaces for the distributed sensor/actuator management mechanism of the Fractal model.

From the OGSA perspective execution management services are services which deal with all aspects of managing jobs, where jobs should be seen as both legacy jobs and running applications designed specifically for the target environment. From the point of view of the Grid user who has computational jobs to submit, some EMS might be all she needs to interact with (maybe not directly but via some Web-based portal interface). Based on that viewpoint, EMS are the services that create the business values of the VO. Taking into account importance of the execution management services, a Grid environment should provide a language support as well as convenient and easy to use APIs and user interfaces to facilitate job submission, monitoring and control. How to present an easy-to-use API for application developers is on the list of challenges a framework for dynamic VOs should address. There are some efforts in OGF, e.g. Job Submission Description Language [JSDL] and Simple API for Grid Applications [SAGA] to achieve this, however, a user interface to execution services is still rather complex. For example, the JSDL description requires the user to know and specify detailed resource requirements of her job. While it can make the exact location of the execution transparent to the client, it does not offer virtualization in the sense of hiding details of the hosting environment or the number of machines running the job. It also has no support for considering policy constraints in job planning and only very limited support for security bindings. For further discussion of the OGSA approach see also [Hog07].

Monitoring and resource discovery services are essential Grid services. Typically, the monitoring and discovery system within a VO has a hierarchical structure with a centralized resource repository on top, which keeps track of all available resources available in the VO. Grid4All scenarios, which assume highly dynamic VOs, require these services to be able to tolerate churn, and quickly react on VO (de)evolution. For the sake of scalability and high availability, the monitoring and discovery system can be structured as a peer-to-peer system, in which each peer is a top index of a separate hierarchical resource repository covering a subset of the VO resource.

Data management services in GT4 (OGSA-DAI, Replica Location Service and data transfer services) and in gLite (Storage Elements, File and Replica catalogs, and data transfer service) provide efficient support for data management in Grids and used by other services that require data access or/and transfer. Most (if not all) of the services are developed to be complaint to WS standards and specifications, in particular, be accessed via the web-services protocol stack (the SOAP stack) and therefore most of the services proving remote access to data source suffer of low performance. We can conclude that both of the Grid environments considered in 5.2, namely, Globus Toolkit 4 and gLite, do not provide sufficient support for self-management of Grid infrastructure services, i.e. most services are not self-managing. For example, the environments do not provide support for automatic handover of a service when a node leaves the Grid or fails. Some services do not have control interfaces, i.e. they are not manageable or have limited manageability. Existing monitoring services (e.g. GT4 MDS) can be used to provide sensing and actuating support for the sensor/actuator-based self-management in Grids, because they provide a subscribe/notify mechanism as well as an event-driven control (actuating) mechanism (e.g. GT4 Trigger Service). However the existing monitoring and management services do not guarantee robustness and availability in a dynamic (ad-hoc) Grid, as they are not able to tolerate churn. This motivates our research on the sensor and actuating overlay services for self-management in dynamic Grids.

6 Web and Grid Security Services

When computer systems move from being limited in geographical and administrative scope to Internet scale, potentially encompassing more than one administrative domain, new security problems arise. While many of the issues have been addressed in scientific and enterprise grid settings, the work to achieve grid-like security functionality on peer-to-peer overlays has only started and is mainly at research stage. The Grid4All project will not solve the problems related to distributed security; instead it must be possible to incorporate existing solutions into the infrastructure proposed by the project. Therefore the purpose of the following section is to outline the SoA in the security areas that the Grid4All infrastructure must be able to relate to.

Some security issues are purely technical, but even if all the technical solutions are in place, the security infrastructure is no more secure than the organizational setting it is placed in. Therefore issues of trust, and mechanisms to support building trusts are relevant, especially for cross domain grid solutions.

Requirements

The security requirements will be very different for different scenarios. Some of the core requirements which all or almost all scenarios will need are:

Protection of the resources. The VO infrastructure should allow mechanisms for authorization to be used, so that the machines of individual contributors are protected against unauthorized access. (For the higher level VO management services also issues of sandboxing are relevant.)

- Privacy. The VO should allow mechanisms for secure communication and storage to be used, so that messaging and user data can be kept private.
- Requirements regarding trust can be addressed by trusted delegation and reputation mechanisms, requirements regarding interoperability can be addressed by following web service standards and requirements regarding distributed authorization can be addressed by solutions such as the Community Authorization Service.

Since the Grid4All infrastructure is meant to be used in dynamic scenarios with heterogeneous resources, also the security services must be able to meet the following requirements:

- Robust design. It must be possible to implement the security services so that they can deliver service despite of resource churn.
- Lightweight. Specifically in the scenarios where there are no dedicated resources for VO management, the overhead due to security processing on the individual peers must be kept low.
- Easy to use. To be able to provide the desired “4All” functionality, the security management should not depend on the existence of experienced it personnel.

6.1 Web and Grid standards for security

6.1.1 Web service standards

The web service security specification, WS-Security or WSS, is a OASIS standard which provides “ability to send security tokens as part of a message, message integrity, and message confidentiality” by defining extensions to SOAP messages. [Nad06]

Separate WSS profiles define how to attach different security tokens such as X.509 certificates, or Kerberos tickets to SOAP messages. These tokens could be used by potentially different components in a system to achieve authentication, or authorization, since certificates can be used to embed authorization attributes.

Message integrity can be achieved through message signatures, which allows the receiver to check whether the data has been changed. [XML02]. Message confidentiality and thereby secure communication can be provided through defining how different encryption mechanisms should be used. Either message security,

transport security or combinations of both could be used, although transport security, which encrypts the entire communication, cannot be used if the message is supposed to be routed by intermediates. [Nad06]

A standard way for how to construct, encode and sign the relevant security tokens can be achieved by using the Security Assertion Mark-up Language, SAML, which is designed to communicate identity and authorization tokens between an identity provider and a service provider which requires the tokens to be presented [Dem05].

Constraints on how security tokens should be handled and delegation of trust can be specified through the WS-Trust specification [Nad+05] which expands the basic WSS specification. The specification is discussed by a OASIS working group but is no standard. For several companies to work together they can further specify how to handle security tokens between themselves through the WS-Federation specification which is suggested by prominent companies but with no standard status [NK+06].

6.1.2 Grid standards

The Globus security framework, Grid Security Infrastructure, GSI, is a de facto standard for grid security. The current version of GSI for Globus toolkit 4 enables building grids which support security operations following the web service security specifications [Wel+05]. Also the UNICORE middleware development is moving more slowly towards web service standards [RM05].

Delegation of authority can be represented by proxy certificates, a special form of the X.509 standard, supported by the GSI. [Wel+04]. Expressing and enforcing the policy regulating who, how or when delegation can be done is an active research area with so far no fixed solution, although the policy language XACML has been extended to support delegation, as shown in [Sei+05].

6.1.3 Applicability to Grid4All scenarios

In order to be provide privacy and interoperability, the Grid4All framework should either implement the WS security standards itself, of it should be easily extendible so that Grid4All services can communicate using the established protocols.

The robustness and the ease of use is not dictated by the WS standards per se, but is the consequence of well made implementations. Regarding the overhead, there has been concerns about SOAP based web service standards causing unacceptable overhead to grid implementations [RM05]. There might therefore be scenarios where more light weight protocols can be used within the system whereas services used to communicate with other systems follow known standards.

6.2 Implementations of security services

6.2.1 Authentication

The standard grid security solutions provided by gLite, GT and UNICORN all assume that users are equipped with X.509 credentials, which are used to authenticate the user. GT4 allows less strict authentication using only username/password. Certificates can also be used to authenticate services. The need for issuing and maintaining certificates requires organizations to create and maintain a PKI, which requires additional software besides the core grid middleware and security expertise to maintain it [Wel+05].

6.2.2 Authorization

The problem of authentication in a traditional centralized manner is clearly addressed by the use of certificates within a PKI. Authorization provides a more complex problem, as it involves all users and all accessible resources in a system which might span multiple organizations.

Traditional grid authorization systems require all resource owners to maintain accurate access lists of all users that should be given access to a resource. With a large number of changing users, this becomes

infeasible. One solution to deal with the number of individual users is to let all the physical members of one organization share the same user account on the resource side. This has been used in scientific grid communities where all users can be considered to have the same needs (i.e. running large scale computations on remote clusters). In more complex VOs, it is considered too coarse grained to give all users the same privileges regardless of the tasks they need to achieve. Instead system which allow more complex role based access policies are created, where the access decision can depend on a number of user and environment attributes.

Most modern authorization frameworks can be mapped to an authorization model where the attributes needed to evaluate a request is either pushed to the policy decision point, or pulled on demand. In the push model, the service requester has to present all the attributes directly when making the service request. Benefits of the push model include a greater level of user control of which attributes to reveal to which service provider. Drawbacks include more work for the user to manage her own attributes, and more complex request constructions on the client side. Using the pull model, the users have no control of which attributes are demanded by the service provider, but no user attribute management needs to be done. Also hybrid models are possible, where some attributes are pushed and others can be pulled on demand if needed.

Existing solutions for authorization in grids include Virtual Organization Membership Service (VOMS), Community Authorisation Server (CAS), Shibboleth/GridShib and “PrivilEge and Role Management Infrastructure Standards”, PERMIS [Alf+04] [Bar+06] [PERMIS]. They are all possible to integrate with the Globus toolkit. VOMS also works with gLite and will be adopted to work with UNICORE. [DS06]. They all initially required policies written in their own specific policy languages, but they are being adapted to use some combination of XACML and SAML. [PERMIS]

VOMS

VOMS works as a member database which stores user information and issues signed attributes to identified users. It lacks the capability to act as a PDP and must be used together with some policy framework, such as PERMIS to achieve policy enforcement. Since VOMS delegates the attribute certificates to the user, when a service requests authorization information, the user can control which attributes that should be revealed. A main disadvantage of VOMS is that it uses a non standard credential format, which requires adoption of the PDP at the resource side. [Alf+04].

CAS

The Community Authorisation Server, CAS, is capable of member management and providing authorization decisions. The final authorization decision is delegated to the CAS system. CAS only records capabilities, not groups or roles. When a request reaches the resource provider, it is already authorized. This requires the resource provider to fully trust the CAS. Promoters of alternative solutions criticise that the final authorization decision is taken from the resource provider. In other solutions, the resource provider still has the final decision in allowing access, independently of reached VO agreements. [Alf+04]

GridShib

GridShib is a solution to combine GT and Shibboleth. Shibboleth is a web service solution to enable single sign in and attribute based authorization. The user only has to authenticate herself once with her home identity provider, which then can provide security attributes to service providers in the same trust domain. The GridShib project defines extensions which make it possible to request attributes from Shibboleth attribute authorities, based on the X.509 distinguished name of the requesting user. The developers hope to have the solution integrated in the next GT release. [Bar+06].

PERMIS

PERMIS uses a single attribute certificate repository, which leaves less control to the user whether to reveal certain attributes to a service demanding them [Alf+04]. While it has its own policy language, it is being adopted to work with XACML [PERMIS]. It has comparable good tool support, i.e. a policy editor with at least some graphical support has been developed [Cha+].

An integrated solution

That there are no common solution for authorization has been recognized as a large challenge by the Globus developers [Wel+05]. A suggest solution tries to encompass the different authorization solutions into a common framework. In [Lan+06] an infrastructure is described where authorization requests carry attributes that can be evaluated by a number of different policy decisions points, corresponding to existing systems. Also old fashioned access control lists can be modelled using PDP interfaces. To bridge two security domains, a new master PDP is needed to forward requests to the appropriate PDPs, evaluate the results and make a final decision. This is related to the work being done with GridShib to provide a more fine grained attribute based authorization for the next GT release (planned as GT 4.2) [Bar+06].

These solutions for providing attribute based authorization focus on the provisioning on secure attribute tokens, and leave some other issues aside. One assumption made is that anyone who wish to request a particular resource has some means to find out which specific policy language to use for the request, and which attributes to provide. This illustrates the need both for general policy languages to express conditions on resource access rights, and for policy container formats for advertising (and eventually negotiate) the policy which a resource or service provider wishes to enforce on incoming requests.

6.2.3 Auxiliary services – user and credential management

Some of the above mentioned authorization services also functions as membership service, which needs to be present in a VO scenario. The chosen membership service must provide basis for accountability. Depending on the demands of the VO, tracking the actions of users might be an important basis for accounting and trust through reputation building.

An important part in maintaining a PKI is to make it easy for users to handle their credentials in a secure manner. One tool used in many grid solutions is the open source credential management software MyProxy. [Bar+06, MyProxy]. Still, grid access is not necessarily easy for ordinary users. Additional software to relieve users of security considerations has been developed. One example is GAMA, Grid Account Management Architecture, which can work together with MyProxy and CAS to provide a portal for user authentication and account management [Mue06].

6.2.4 Applicability to Grid4All scenarios

The above described services are all primarily implemented for traditional grid systems. This means they are less robust, more heavy weight and less easy to use as would be the optimal for Grid4All scenarios. For the cases where security requirements are strict, and insufficient resources are available within the VO, the desired security services could be provided by trusted third parties. This emphasis the need for the Grid4All infrastructure to provide the suitable extension points so that existing solutions can be plugged in, or be provided by external providers.

7 Peer-to-peer based services

While the services presented in the section on existing implementations of grid services meet functional requirements of VO services, they lack the non-functional capabilities needed to meet the Grid4All requirements. This lack of capabilities has been addressed by peer-to-peer based solutions. In the following section representative examples of state of the art concerning peer-to-peer based solutions to services for VO management are presented.

7.1 Data services in peer-to-peer systems

Many peer-to-peer systems, such as networks for file sharing or applications for distributed storage, are stand alone data services themselves. Because of the good scalability and fault tolerant properties of peer-to-peer networks, efforts have been done to provide event/notify-services and publish/subscribe systems built on overlay networks, although few of the solutions can yet be seen as reusable services. There is a sliding scale between simple event/notify-systems and compound query services.

7.1.1 Data storage services

An important group of peer-to-peer based applications are distributed storages such as OceanStore [Rhe+03] built on Tapestry, or the backup system MyriadStore [ST06]. While those systems currently function as stand alone applications rather than interoperable services, they could be prime candidates for being extended into more general purpose data storage services.

Besides the need to perform standard DHT maintenances tasks, peer-to-peer storage services have to manage the storing and reconstruction of files from separate file blocks which might be widely distributed. For example, if a file is split on a high number of nodes and the systems wants to give some guarantees on reading latencies, issues of locality aware block placement becomes very relevant. This is a prime example of conflicting requirements since structured peer-to-peer networks achieve load balancing benefits if the responsibilities of nodes are randomly assigned, but randomness might cause related file blocks to be placed on opposite sides of the globe. The issue of locality aware peer-to-peer networks is addressed for example in [Har+03], where names instead of hashed identifiers, as in DHTs, are used to arrange data and nodes in the overlay.

7.1.2 Event based systems and publish/subscribe for overlay networks

An event based system reports events triggered by some state change of a resource. Potentially the amount of events generated in a system is very large, and subscribers might only be interested in a small subset of all of them. To ease the burden on the subscribers and the network, events can be filtered before reaching their final destination. There are two main schemes for subscription filtering, topic based (or subject based) and content based. [Eug+03].

Topic based subscriptions

By topic based subscriptions, the system can arrange the subscribers in interest groups and multicast relevant events within the group. This approach is closely related to research in achieving efficient multicast in different network topologies.

If the topics are broad, the filtering on the system level is made easy, but causes more network traffic and load on the subscribers, who have to do the final filtering themselves. On the other hand, if the topics are narrow the system becomes more complex, and events might have to be published within multiple groups. The grouping of topics can be facilitated by the existence of a topics hierarchy. Either event can be published once with a general tag, and be forwarded to all subscribers interested in related subtopics, potentially delivering notifications the subscriber is not interested in. Or the reverse, events can be published with several subtopic descriptors, and be forwarded to subscribers interested in that specific topic, or related

more general topics. This requires forming and maintaining a potentially complex semantic structure, which offset the benefits of topic based subscriptions as the simple alternative to content based subscriptions.

Content based subscriptions

The alternative, content based subscriptions, require a more complex design, since the notification messages have to be parsed for their actual content. The subscribers can define the content patterns they are interested in, and the system will do the filtering [Eug+03].

A common interest of a subscriber is to be notified on “notable” events, such as a stock value or a temperature reading changing outside some predefined range. In those scenarios it is needed that the system can handle ranges of attribute values in the subscriptions rather than only exact values.

Depending on the type of system, the subscribers might be able to register interest in arbitrary attributes, or might be confined to a set of known attributes. If the publishers can describe their resources in any way they want, or if the system is meant to be used with many different types of content, where the possible attribute values cannot be known in advance, there must be support for arbitrary subscriptions. For instance query languages normally assume queries can be made with arbitrary attributes, which means that any event based system that wants to support query languages such as SPARQL, needs to allow arbitrary attributes.

For those solutions where the number of attributes must be known in advance, the management system must be able to impose restrictions on the form of events and subscriptions. Each event must be modelled to enable the predefined form of mapping to the subscriptions.

Peer-to-peer based implementations

In small scale networks, well maintained company LANs or large grids with dedicated and reliable machines, centralized solutions for event based system might work well. In a large heterogeneous milieu, bottlenecks and partial failures are obstacles that must be overcome.

Event based system have their own workshop in the Distributed Computing Systems conference, which can serve as a resource for related reading. (The list of papers can be found on [DEBS3-6]). A theoretical work on principles for publish/subscribe is found in [Eug+03]. In the following some representative examples of publish/subscribe systems build on peer-to-peer networks are briefly reviewed.

Scribe

Scribe is “a large-scale, decentralized application-level multicast infrastructure” which is built upon Pastry [Row+01]. Any peer can register a topic, which other peers can subscribed to, or publish events within. Their design features access control by allowing the node registering a topic to set credentials, which will be required by anyone that wants to publish or subscribe to events with that topic.

By default, Scribe only gives best effort delivery of events with no event delivery order specified. Scribe is included as a component in the Dermi, “Decentralized Event Remote Method Invocation” peer-to-peer middleware, which is available under a LGPL-like license. [DERMI]

NaradaBrokering

The NaradaBrokering Project is an open message oriented middleware system which offers publish/subscribe functionality [NB]. It builds a hierarchical system of brokers which can be used to interconnect clients and services interested in publishing and receiving events. In theory, nothing prevents one machine from acting as client, server and broker simultaneously. Still, NaradaBrokering is not a peer-to-peer system itself, but is designed for supporting the interconnection of several peer groups with each other and with external resources. The peer groups could be communicating using JXTA protocols internally.

The brokers can filter subscriptions on published events on a range patterns, including regular expressions and sql queries. The framework supports WS-Eventing, reliable messaging and different security mechanisms including authorized publication and encrypted communication. [Pal+03].

Atlas

Two query algorithms which enables querying data stored in basic RDF triplets in a DHT are presented in [LIK05]. Each RDF triple will be hashed and stored on nodes responsible for the hash of the subject, predicate and object. In the same way that individual triplets of published descriptions are stored on different

nodes, queries will be split in as many sub-queries as there are triplets. The single query chain algorithm will assign one node for each sub query. If a matching triple arrives to one responsible node, it will forward the partial result to the next node in the chain. The query chain will carry the identifier of the subscribing node. The last node will know when the query is answered, and return the matching bindings. If a query generates many matches, nodes in a single query chain might get overloaded. The multiple query chains algorithm tries to spread the query processing load by starting separate chains for each different matching subject.

Some drawbacks of are: The papers do not describe any way to avoid overloading nodes responsible for popular keywords. They do not address the issue of range queries, which makes a number of interesting subscriptions impossible to make.

Edutella-based

Another RDF query design proposed within the Ontogrid project is a pub/sub system built upon the Edutella design where super-peers are organized in a hypercube structure where ordinary peers join through the super peers. [Chi+04]. It is assumed that the resources mentioned in subscription can be sent between peers, and stored temporarily at super-peers if the recipient is unavailable. The system does not hash any information, instead the super-peers build plain text tables of advertisements and subscriptions, and forward relevant parts of the subscriptions to all super-peer neighbours it knows. This way it is relatively easy to achieve range query abilities, but subscription data has to be duplicated across all super peers. Not using DHTs they have to make the strong assumption of available, non failing super-peers. It is unclear what would happen if one of the super-peers fails.

Meghdoot

Meghdoot is a content based publish/subscribe system which assumes a fixed attribute space. [Gup+04] With n attributes, each with a bound domain, a 2n dimensional space is created. This space is mapped to a CAN-based DHT, where different nodes take responsibility for different regions of the space. Both subscriptions and events are mapped to single points in the multi dimensional plane. Through calculations it can be determined which neighbouring zones that events need to be propagated to in order to reach related subscriptions. To avoid node overload, zones with many subscriptions can be split on more nodes while zones that are on the “transport route“ for generated events can be duplicated. They describe how to achieve replication of subscriptions. Their experiments show the algorithm to be successful in scalability and load balancing, but fault tolerance is not demonstrated.

Sub-2-Sub

The Sub-2-Sub solution assumes an unstructured peer-to-peer network. The system uses an epidemic-based algorithm that clusters peers which have made similar subscriptions. All attributes are modelled as real numbers, or ranges of real numbers. The attribute space is assumed to be limited and all events must specify values for all allowed attributes. One node is responsible for one single subscription. If a user makes more than one subscription, it will be represented by multiple nodes participating in multiple overlays. With this model, nodes can exchange subscription information and see whether their interests are overlapping, or otherwise calculate the euclidean distance between the subscription values they are interested in. When any peer hears of an event, it will spread it among the peers it knows that are close in the attribute interest space. The presented forwarding method guarantees that only interested nodes will be reached by published events, but depending on the churn not all subscribers might be reached by relevant events [Vou+06].

P2P-ToPSS

The developers of the P2P-ToPSS system experiment with range queries for DHT networks, with limited number of nodes [MJ05]. As in the Sub-2-Sub solution, the attribute space is known and fixed, and all attributes must have known finite domains. This way, the attribute space can be modelled as a binary string. Different nodes will be responsible for subsets of the string space, indexed as a binary search tree. As each publication or subscription can be represented by a binary string, they can be mapped to corresponding nodes. To enable range queries, a subscription might have to be mapped to a set of nodes. Their experiments show that the design caused heavy load on the nodes in small networks (up to 30 nodes) if the subscriptions are “coarse grained”. (Since any range with enough distance between upper and lower bound is coarse grained, subscriptions of the type (anyValue >= fixedNumber) are considered coarse grained.)

Semantic p2p pub/sub

The paper [CF05] presents a similar design to Sub-2-Sub where peers are grouped based on interest. The forwarding algorithm is very simple; “forward an event if you are interested in it yourself”. Only statistical predictions on how many subscribers that will receive an relevant event can be given. In their evaluation, with 5000 peers, the number of false positives – interested subscribers not reached – is around 5%. They claim their grouping algorithm works with XML attributes, but do not present any details on how the attribute values must be constrained.

7.1.3 Applicability to Grid4All scenarios

The storage services described in “Data storage services” could be seen as functionality which does not need to be provided by the VO infrastructure itself, but rather could be deployed as services of their own. Therefore it should be possible to deploy the most suitable distributed storage depending on the scenario at hand, that choice should not be made within the Grid4All project.

More closely tied to the VO infrastructure is the publish/subscribe mechanism used to enable the event based architecture. While topic based systems as Scribe are relatively easy to implement on top of DHTs, they suffer from the drawbacks of topic based systems, which in Scribe is further emphasised by the lack of topic hierarchies. The alternatives are content based subscription systems such as Meghdoot, Sub-2-sub or P2P-ToPSS, which might be suitable for resource management tasks, where a limited set of attributes is enough to model the resources of interest, and the attribute set is known in advance. Systems such as Atlas or the Edutella-adoption which allow an unconstrained attribute space might be useful to provide metadata services where range queries are unneeded.

Even though event delivery guarantees might not be needed in all Grid4All scenarios, the publish/subscribe system should be able to given them if needed. Therefore systems as P2P-ToPSS which trade delivery guarantees for low message overhead cannot be used.

It is worth noting that besides the ambitious NaradaBrokering system, there seems to be no other peer-to-peer system that has implemented the web service standards, WS-Notification or WS-Eventing. What is more, the bulk of the papers do not refer to them as relevant related works. While some papers touch upon issues of authorization (for instance the Edutella-based), it is done in a non-standard manner, again with the exception of the NaradaBrokering system.

7.2 Resource management for peer-to-peer systems

Peer-to-peer systems largely yet lack the infrastructural abstractions that have been developed for grids and are not available as separate services. Nevertheless the issues have been addressed in various peer-to-peer systems.

7.2.1 Resource management for distributed computing

For cycle sharing applications such as XtremWeb or BOINC [Cap+05] [BOINC] the resource management is mainly centralized. Because of the low demands on interaction with central management services, the resource management on the client side can be kept light-weighted. One way of keeping management simple is by ignore issues of job handover on node leaves. For example the XtremWeb architecture does not support graceful node leaves. If a node leaves during a computation, the work is not redirected. Instead, the system provides enough redundancy through duplicating the subtasks. Further ideas on resource management for computational peers can be found in [CBL04] [TZ06]. While some of the papers outline general resource management services for peer-to-peer networks, the suggested systems are not (yet) publicly available.

7.2.2 Generalized peer-to-peer resource management

Besides the specialised solutions there are attempts to describe resource management for peer-to-peer networks in more general terms. In [Fri+03] the authors describe a simple resource management framework layer between the application and the peer-to-peer communication layer. The management layer offer basic resource register and query services which reassembles a stripped down MDS. Their work is extended to a system for managing (relatively static) business data. Besides this implementation by the same researchers, there seem to be no other implementation done based on their design.

An attempt to provide standardised general resource management support for peer-to-peer overlays is made in the JXTA-SOAP project, which aims at providing WSRF support for JXTA built networks [JXTA].

7.2.3 Applicability to Grid4All scenarios

While ideas can be taken from specific solutions, peer-to-peer based resource management functionality has not yet reached the stage of maturity where it can be reused as separate services. Comparisons of systems is hindered by the lack of standards or agreed upon terminology for resource management capabilities for peer-to-peer systems. The JXTA-SOAP project should be checked to see if they produce something which might be reusable.

7.3 Security in peer-to-peer networks

While security in grids has been tightly coupled with issues of identification and certificates, security in overlay networks has often been related to building trust where no certificates can be known to be available. Other issues are providing privacy through anonymity, resistance to denial of service and encryption for shared storages. With the ongoing interest of enabling more functionality in peer-to-peer networks, some newer peer-to-peer solutions try to address the traditional grid security issues of authentication and authorisation. There are numerous suggested solutions to different subtopics of the security issues in peer-to-peer networks, but they have not yet reached the maturity to be standardized.

7.3.1 Enabling techniques

A central issue in establishing secure communication is agreeing on keys to use for encryption. There are solutions for how to extend the classical Diffie-Hellman key exchange for groups, which can be used in peer-to-peer overlay networks. The algorithm in [STW00] has been analysed, implemented as a library and used in other research projects [Cliques]. Since the algorithm uses a shared secret key, issues of peer trust and changing keys when a peer no longer is trusted, need to be addressed.

For building trust in peer groups several reputation mechanisms have been proposed. A non-complex design is suggested in [AS01]. The peers judgments of each other are stored in a known order, similar to a DHT, but without the hashing of values. Any peer that wants to calculate the trust value for a peer can easily retrieve all complaints made by other peers. A more complex solution is found in [WV03] where a Bayesian network-based trust model is used to enable peers to consider different aspects of the peers trustworthiness depending on the task at hand.

If specifying delegation is hard in traditional grids, it is even harder in peer-to-peer based networks where no traditional credential authorities exist. One attempt to combine trust management and delegation is done in [RY06], where delegation certificates are created, signed and evaluated in a way similar to the web of trust-approach. Their solution is still pseudo-centralized since it assumes the presence of a peer group leader which helps the other peers in assessing the level of trust in each other.

7.3.2 Security in JXTA

JXTA is a set of “protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a peer-to-peer manner” [JXTA]. The

communication in JXTA is focused around peer groups where certain super-peers have more responsibilities than so called edge peers.

The protocols support building a PKI on top of JXTA peer group overlays, which then can be used for both authentication and various forms of authorization. Basic authentication services are included in the main reference implementation. [Alt03]

As a complement to traditional X.509 certificates JXTA supports the use of Poblano, a reputation mechanism where trust between peers in calculated over a period of time based on peer interaction [CY03]. This trust metric can either be used where no certificate authority is available, or as a backup if connection to an authority is temporarily lost.

7.3.3 Applicability to Grid4All scenarios

None of these solutions tries to be a complete security solution. Instead they can be seen as indications that research is active in the security for peer-to-peer based networks area. Since the development has not reached the level of maturity where functionality can be provided through well-defined services, the Grid4All infrastructure cannot make assumptions about specific interfaces. The conclusion is the same as for the relation to the more mature grid security services – the Grid4All framework should provide the necessary hooks and handlers to make it easy to adapt be able to make out-calls to auxiliary services. This is particularly important for the areas where traditional solutions are insufficient, for example to provide reputation mechanisms for overlay networks.

8 Component Models

This section first identifies requirements for component models targeted at building Internet-scale distributed systems. The section then discusses a selection of commercial and research component models and evaluates them based on the requirements. Finally, an overview of the evaluation is presented.

8.1.1 Requirements

Component models for building dynamic, Internet-scale distributed systems should address the following requirements:

Reconfigurability. The component model should support static and dynamic changes in individual components and component systems in a flexible and easy-to-use way. Reconfigurability is essential for accommodating the diversity and dynamism of the environments in which Internet-scale distributed systems are deployed.

Extensibility with regards to reconfiguration facilities. The model should support an extensible set of reconfiguration facilities that control a variety of system aspects, such as component distribution or replication. Extensible reconfiguration facilities increase the applicability of the component model while providing consistency for application developers.

Support for hierarchical composition. The model should support defining composite components that integrate smaller components. Hierarchical composition is a key means of managing the complexity of building and evolving component systems.

Extensibility with regards to binding types. The model should enable components to interact using an extensible set of “binding types”. Binding types are interaction paradigms such as remote method invocation, messaging queuing, or group communication. This feature increases the range of application communication needs that can be met by the component model, thus increasing its applicability and value.

The following sections present a set of representative component models and evaluate them based on these requirements.

8.1.2 CORBA component model (CCM) and Enterprise JavaBeans (EJB)

The *CORBA component model* (CCM) is a component technology that supports components distributed over different machines [OMG02]. CCM extends the basic CORBA object model to support the concept of *CORBA components*, which interact through interfaces and events and execute within run-time execution environments called *containers*. Containers are responsible for providing infrastructure services to a hosted component, such as transactions, persistency, security, events, and lifecycle management. Components can access these services both *explicitly*, through invoking interfaces on the container and receiving invocations on their interfaces, and *implicitly*, through having the container intercept incoming calls on the component and perform pre- and post-processing (e.g., starting transactions). This architecture simplifies component development because it allows developers to concentrate mainly on functional concerns while containers manage extra-functional concerns. The container behaviour is configured declaratively using *component descriptors*, which specify requirements of components on container-provided services. For example, a component descriptor may specify the transaction policies and security rights for a particular operation of the component. Descriptors are set statically, before component deployment.

Enterprise JavaBeans (EJB) is a component technology with a container-based architecture similar to that of CCM [Sun06]. In fact, CCM was explicitly designed to provide close correspondence with version 1.1 of EJB in order to facilitate integration between the two technologies. Similarly to CCM, the EJB container provides a fixed set of middleware services.

Evaluation

CCM and EJB provide limited support for reconfigurability. Their containers provide a fixed set of services with a fixed range of configurations selected statically. CCM supports extension with respect to a small number of platform elements—notably, transport protocols and interceptors—but extension components can only be integrated statically. CCM and EJB provide no support for adding new binding types beyond remote method invocation and event delivery. Hierarchical composition is unsupported. Specifically, CCM allows grouping interconnected components into “assemblies”, but these assemblies are not components and cannot be further composed.

8.1.3 Microsoft COM and .Net

COM is a component technology that supports in-memory interoperation between independently developed components, possibly written in different programming languages [MS04a]. Interoperation relies on a binary standard that defines how interfaces are represented in memory and how their operations are invoked dynamically. Over time, the COM technology has grown to include:

- support for invoking methods on COM components that reside in different processes and machines;
- an extension of COM, known as *COM+,* that provides components with a set of declaratively-configured services, such as transactions, security, events, and synchronisation

COM+ has a container-based architecture that employs interception and declarative requirements on container-provided services. Similarly to CCM and EJB, the set of services is fixed and their configuration is static.

The *.Net framework* is the latest, programming language-neutral component technology introduced by Microsoft [MS04b]. It consists of a virtual machine, called the common language runtime (CLR), and a set of supporting APIs. The *remoting framework* is the part of .Net concerned with interacting with remote objects and provides rich configuration and extension facilities, such as custom proxies, pluggable communication channels (e.g., TCP and HTTP channels) and pluggable formatters (e.g., binary and XML formatters). Moreover, similarly to COM+, the remoting framework defines a container-based architecture for transparently providing services to objects. Specifically, services are provided by *contexts*, corresponding to the container abstraction, which host one or more objects. The context within which an object resides depends on *context attributes* associated with the object’s class; the context attributes express requirements on context-provided services. Importantly, unlike COM+, the set of services is extensible. Custom services are realised as sets of *message sinks* that intercept and manipulate cross-context invocations. Custom services are configured through custom context attributes.

Evaluation

COM and .Net provide no support for extension with respect to binding types. Hierarchical composition is lacking. Because of its extensible container-provided services, .Net provides stronger support for reconfiguration than similar container-based technologies, such as COM+, CCM and EJB. However, dynamic flexibility is still lacking in .Net. While one can configure aspects of the remoting framework at various times, the resulting configurations remain normally static. For example, the set of context services provided to an object and their properties cannot be changed after object instantiation. COM provides no support for dynamic changes.

8.1.4 OpenCOM

OpenCOM is a lightweight, efficient, language-independent component model for building adaptable component systems [CBC01, CBG04]. OpenCOM relies on the following fundamental concepts: component, interface, receptacle, binding, and capsule. *Components* are units of functionality that expose multiple interfaces and receptacles. *Interfaces* are units of service provision, and *receptacles* are units of service requirement. *Bindings* are associations between a single receptacle and a single interface. *Capsules* are runtime environments where components are deployed. Apart from these fundamental concepts, OpenCOM supports a set of three “reflective meta-models” that enable inspecting and adapting different system aspects at run-time. The *interface meta-model* provides meta-information about components and interfaces. The *interception meta-model* supports injecting code that runs before or after invocations on a specific interface.

The *architecture meta-model* supports maintaining and manipulating the topology of a system in terms of components and bindings.

OpenCOM-based systems employ the *component framework* (CF) notion in order to reduce the design complexity and to manage the dynamic configurability [CBC02]. CFs are reusable component architectures targeted at specific domains (e.g., building protocol stacks); they define rules and interfaces that constrain the interactions between a set of *plug-in* components. CFs are explicitly represented by composite components that accept the plug-ins as subcomponents and handle their static and dynamic configuration. CFs expose easy to use and consistent reconfiguration interfaces by exploiting their domain-specific knowledge and built-in constraints. A useful CF is the binding CF, which supports developing and integrating new binding types while exposing a simple programming model for using them [PCB03]. This CF also supports dynamically adapting bindings, which are explicitly represented as objects.

OpenCOM has been implemented on various deployment environments (e.g., PCs, PDAs, and network processors) and has been used to build various types of system software, such as middleware for networked embedded systems, and programmable networking software. *Gridkit* is an OpenCOM-based middleware platform for grid computing [CGB06]. Gridkit consists of two component frameworks: the interaction CF and the overlay CF. The *interaction CF* supports 'pluggable' binding types (it is a variation of the binding CF mentioned earlier). The *overlay CF* supports pluggable overlay networks, such as Chord, Scribe, and probabilistic multicast. Interaction paradigms can be automatically configured to use different overlays depending on the available network infrastructure and environmental conditions. For example, group communication may use a Scribe implementation on a fixed network and a probabilistic multicast implementation on an ad-hoc network.

Evaluation

OpenCOM provides strong support for reconfiguration owing to its reflective meta-models. Using these meta-models, multiple aspects of the component system are available for dynamic modification and extension. Extending and customizing reconfiguration facilities are supported through CF-specific reconfiguration. However, the CF concept is not a first-class part of the component model. Similarly, hierarchical composition requires using CFs, and thus it is not natively supported by OpenCOM. OpenCOM allows extensibility with respect to binding types. Such extensibility is explicitly supported by the binding and interaction CFs.

8.1.5 Fractal

Fractal is a general, extensible component model for implementing, deploying, and managing complex software systems [BCL06]. Its main features are support for hierarchical composition, support for sharing, and support for open reflective facilities.

Fractal components are runtime entities that expose multiple interfaces of two types: *client interfaces* that emit operation invocations and *server interfaces* that accept them. Interfaces are connected through communication paths, called *bindings*. Fractal distinguishes *primitive bindings* from *composite bindings*, which contain a set of primitive bindings and other components. Similarly, Fractal distinguishes *primitive* components from *composite* components, formed by hierarchically assembling other components (called sub-components). Hierarchical composition is a key Fractal feature that helps managing the complexity of understanding and developing component systems. Another original Fractal feature is *sharing*, which allows a component to be contained in multiple other components. Sharing is particularly useful for modelling access to low-level system resources.

Each Fractal component is made of two parts: the *membrane*, which provides interfaces to introspect and reconfigure its internal features, and the *content*, which consists of a finite set of sub-components. The membrane is composed of several controllers that superpose control behaviour to the sub-components. Fractal supports open reflective facilities in the sense that it allows arbitrary (including user-defined) forms of controllers. Nevertheless, Fractal does define a useful set of four controllers. The *attribute controller* supports configuring component properties. The *binding controller* supports binding and unbinding client interfaces to server interfaces. The *content controller* supports listing, adding, and removing sub-components. The *life-cycle controller* supports starting and stopping the execution of a component.

Fractal includes an extensible architecture description language (ADL) for specifying configurations comprising components, their composition relationships, and their bindings. The reference Fractal implementation is in Java, but there are multiple implementations for multiple programming environments. Fractal has been applied in various domains, such as building asynchronous middleware, operating systems, and GUIs.

Evaluation

Fractal provides strong support for reconfigurability through endowing components with controllers. A key advantage of Fractal is that its reconfiguration facilities are extensible. Moreover, hierarchical composition forms a first-class part of the model. Fractal allows extension with respect to binding types through employing composite bindings. However, the model provides no particular support for such extensions.

8.1.6 Common Component Architecture (CCA)

The *Common Component Architecture* (CCA) is a component model for high-performance scientific computing developed by the CCA Forum [CCA04]. CCA components are connected through *provides* and *requires* ports corresponding to incoming and outgoing invocations. One can connect, add, or remove ports at any time. Ports are typed by interfaces described in the scientific interface definition language (SIDL), which provides constructs necessary for scientific computing, such as complex numbers and dynamically-allocated, multi-dimensional arrays. There are various implementations of CCA for different computing environments. Ccaffeine is an implementation that targets Single Program Multiple Data (SPMD) parallel applications. Ccaffeine-based components interact within a single process using CCA ports; parallel instances of Ccaffeine-based components interact across different processes using a separate programming model, typically MPI. XCAT3 is another implementation that supports components distributed over different address spaces; the components are accessible as collections of Grid services compliant to OGSI (Open Grid Services Infrastructure). CCA has a broad community of users and has been applied to a wide range of application domains, including global climate modelling, combustion simulation, material science, and quantum chemistry.

Evaluation

CCA supports manipulating connections and ports dynamically. However, reconfiguration facilities are not extensible. Moreover, the model lacks support for hierarchical composition and for extensible binding types.

8.1.7 ProActive/Fractal and Grid Component Model (GCM)

ProActive/Fractal is a component model for developing grid applications [BCM03]. The model conforms to the general Fractal model and extends it with a number of features that specifically target grid programming. The model is implemented on top of the ProActive library, a Java middleware platform for parallel, distributed, and concurrent programming [PA07].

The ProActive/Fractal model extends Fractal in the following ways. Primitive components are specialised to become *active* objects (i.e., objects with their own thread of control) that are remotely-accessible in a transparent way. Composite components may thus contain multiple active objects and may be distributed over different hosts. Method invocations are typically asynchronous and provide automatic synchronisation based on *future* objects. However, invocations are synchronous in the case that the method declares exceptions or returns a primitive type or final class. ProActive/Fractal provides first-class support for multi-participant communication through the notion of *collective interfaces*. There are two types of collective interfaces—multicast interfaces and gathercast interfaces—represented as new kinds of interface cardinalities in Fractal. *Multicast interfaces* transform a single invocation to multiple invocations forwarded to a set of connected server interfaces. The invocation parameters can be distributed in different ways, and the invocation result is always a list of results. *Gathercast interfaces* transform invocations from multiple connected client interfaces to a single invocation. A gathercast interface may define different ways to synchronise incoming invocations, to aggregate incoming parameters, and to redistribute return values.

ProActive/Fractal also provides support for configurable component deployment based on *virtual node*s and *deployment descriptors*. Virtual nodes are logical locations of components used in the source code or the ADL. The virtual nodes are then mapped to the physical infrastructure using XML deployment descriptors. The descriptors contain the mapping of virtual nodes to running Java Virtual Machines (JVMs), and the process of creating or acquiring those JVMs (e.g., using ssh to launch a JVM on a remote host, or using the PBS job management system to allocate multiple JVMs on a cluster). The descriptors enable deploying a component system on various infrastructures without having to modify the source code or the ADL description of the system.

The ProActive/Fractal implementation builds on the ProActive library, which inherently supports active objects and asynchronous method invocation. Multicast interfaces are implemented using ProActive typed groups, which enable invoking multiple objects with compatible type, dynamically generating a group of results. Component deployment builds directly on the descriptor-based deployment framework in the ProActive library. There is currently limited experience in using ProActive/Fractal in real-world systems. Most notably, the component model has been applied to reengineer an existing high performance numerical solver to become a component-based application [PMG07]. The application was deployed on a large-scale grid (more than 300 processors on 4 remote clusters), and the experimental results demonstrated that componentisation has no adverse effect on performance.

The *Grid Component Model* (GCM) is a component model currently under development within CoreGRID, the European Network of Excellence in grid and peer-to-peer technologies [GCM07]. Similarly to ProActive/Fractal, the model is defined as an extension of Fractal that specifically targets grid infrastructures. GCM draws significantly on ProActive/Fractal but includes a wider range of features. Specifically, the main features of GCM are as follows. The model allows component communication with various semantics. The default semantics is asynchronous method invocation, but additional semantics may be supported in the future (e.g., communication via streams). Similarly to ProActive/Fractal, GCM supports collective interfaces (i.e., multicast and gathercast interfaces) as well as deployment based on virtual nodes. GCM allows dynamic controllers; that is, it allows controllers to be treated as components that can be added or removed dynamically. The model provides basic support for *autonomic components* by defining simple Fractal controllers that expose autonomic behaviour. Specifically, components may exhibit two levels of autonomic control: low-level, passive autonomic control and high-level, active autonomic control. The former indicates that components support inspecting and adapting their current status; the associated controllers have operations to retrieve and execute a set of autonomic operations. The latter indicates that components can manage themselves given high-level goals. The associated controllers have operations to accept QoS contracts and to signal QoS violations. GCM is also expected to specify component packaging in the form of XML documents. The GCM specification is not finalised yet. Prototype implementations are currently being investigated within the CoreGRID and GridCOMP projects.

Evaluation

Since ProActive/Fractal and GCM are based on Fractal, they inherit its main advantages; namely, extensible reconfiguration facilities and support for hierarchical composition. The GCM support for autonomic components is minimal as the model provides no guidance for implementing actual autonomic controllers. Similarly to Fractal, both models allow extension with respect to binding types but provide little support for it. Moreover, the built-in support for multicast and gathercast interfaces complicates such extension. Specifically, adding support for new binding types can take two forms. First, it can take the form of defining new kinds of interfaces analogous to collective interfaces. However, modifying the publicly available component model specification every time binding types are introduced or customised is costly and impractical. Second, adding support for new binding types can employ existing Fractal features, such as composite bindings. However, this leads to distinct ways for modelling binding types, imposing complexity on component developers.

8.1.8 p2pCM

p2pCM is a component model for developing distributed applications on top of structured overlay networks [PGM05]. The model builds on a distributed object middleware platform called *DERMI*, which is outlined next.

The main features of DERMI are: multiple invocation abstractions, decentralised object location, and distributed interception. DERMI supports both one-to-one and one-to-many invocation abstractions. *One-to-one* invocations can be direct invocations or "hopped" invocations. The former are standard invocations in which the object reference contains a physical address. The latter are invocations in which the object reference is a key that is mapped to a physical node using the overlay network. The advantage of hopped invocations is that they allow the physical object address to change (e.g., in the case of node failures). *One-to-many* invocations can be divided into three types—multicall, anycall, and manycall invocations. The *multicall* abstraction enables a set of objects to be gathered in a group, and an invocation to be delivered to all group members. The *anycall* abstraction is a variation of multicall that enables an invocation to be delivered to a single, nearby member of the group. The *manycall* abstraction is another variation that enables an invocation to be delivered to several group members; specifically, the invocation is routed to group members until a condition is satisfied.

The *decentralised object location* service enables objects to be designated by human-readable, URI-style names. Name-to-object associations are stored using the fault-tolerant DHT facility of the overlay network. The *distributed interception* service enables dynamically injecting functionality into the path of one-to-many invocations without changing application code. DERMI is implemented on top of FreePastry, the open-source implementation of Pastry and Scribe.

p2pCM extends DERMI with support for the component concept. A p2pCM component is a package of a DERMI object implementation and related metadata. The main features of the model are decentralised component deployment, adaptive component activation, and lifecycle management. Decentralised component deployment enables storing and retrieving components using the DHT facility of the overlay network. Component instances may be replicated on multiple nodes. When a client requests a component instance, the system returns a reference to the closest replica. If no such replica is active, the system instantiates the component on the local node. A lightweight container manages the lifecycle of component instances and supports passivation and adaptive activation. *Passivation* involves removing an instance from the local node when it is unused for a specified amount of time. *Adaptive activation* involves creating additional replicas of the instance when the instance is overwhelmed with invocations.

The p2pCM model is currently being used within Snap, a platform that supports deployment of J2EE-compatible web applications on P2P networks [PGM07]. The Snap project is a member the OW2 consortium (formerly ObjectWeb).

Evaluation

p2pCM provides limited support for reconfiguration in the form of component passivation and adaptive activation. However, reconfiguration facilities cannot be customised or extended, and hierarchical composition is unsupported. The distinguishing feature of the model is that it provides a range of binding types implemented on top of structured overlay networks—namely, hopped, multicall, anycall, and manycall method invocations. However, no support for defining further binding types is provided.

8.1.9 Hadas

Hadas is a component model for encapsulating geographically dispersed services [BSHL01]. Hadas includes a composition model that facilitates assembling encapsulated and possibly active services into bigger applications, while providing the administrative autonomy of sites and individual components. The component model allows for the adjustment of structure and behaviour of components to changes in their environment. The composition model includes a set of protocols that enables dynamic deployment of components in distributed environments, as well as dynamic reconfiguration through reflective stubs termed Ambassadors. There is a full-featured implementation of Hadas, including tools for creation, deployment, and composition of Hadas components. Hadas addresses the following requirements on the software design of encapsulated geographically dispersed services: (a) dynamic deployment, allowing to deal with different environments at deployment time, (b) dynamic adaptability, allowing to deal with changing environments and application requirements, and (c) autonomy-sensitive deployment and adaptability, allowing to deploy applications in distributed environments partitioned by different authorities.

The Hadas programming model distinguishes between the low-level infrastructure layer, the component development layer, and the application composition layer. Hadas components follow the so-called "mutable reflective component model". Higher-level programming uniformly access and manipulate only Hadas components. The configuration API is used to deploy and establish application-level connections between components following the Hadas "configuration model". The adaptation API is used to adapt components during their execution, either by themselves or by third-party components. Hadas components are fully reflective, i.e. they support both dynamic introspection and evolution. Introspection is provided by built-in methods that return detailed descriptions of components. Each Hadas component is split into two sections, "fixed" and "extensible", of which the latter can change at runtime. Components are "self-adapting", where components modify themselves through dedicated built-in metamethods capable of structural and behavioral changes. The two-level invocation scheme is used to invoke modified methods.

The Hadas configuration model is built on the notions of "sites", "ambassadors", and "visitors". Each site contains a collection of Hadas components. A site is represented by a site object (a Hadas component on its own right) that is used for establishing communication links with other sites. Site connectivity is "on-demand", contributing to autonomy and scalability. Similarly, prior to invoking operations on remote components, component-level interconnection must be established. An "ambassador" is a representative of a component that is dynamically deployed on a remote site. Ambassadors serve both as an adaptive remote reference and as an interoperability handler. Ambassadors can form hierarchies, addressing the scalability issues. Ambassadors are implement using the "visitor" general-purpose component type that posses mobility capabilities.

Evaluation

Hadas supports dynamic deployment and composition of distributed applications. Applications can be dynamically reconfigured using the combination of facilities of the mutable reflective component model, and the configuration model dealing with component interconnections. Specifically, component behaviour can be modified within certain margins, and the set of components and their interconnections can be changed at runtime. Using these facilities, additional reconfiguration facilities and binding types can be programmed. However, the extensibility of reconfiguration facilities is limited by the degree of network awareness provided by a particular implementation of the infrastructure layer, and extensibility with regards to binding types can be limited in practice by the type and performance characteristics of bindings provided by the low-level infrastructure layer. Hierarchical component composition can be implemented in Hadas, but it is not directly provided by it.

8.1.10 Evaluation summary

The following table summarises the preceding evaluation of component models with respect to the requirements. In the table, the extent to which requirements are satisfied can be high (+), medium (+/-), or low (-).

	Reconfigurability	**Extensibility with reconfiguration facilities**	**Support for hierarchical Composition**	**Extensibility with binding types**
CORBA / EJB	-	-	-	-
COM / .Net	+/-	-	-	-
OpenCOM	+	+/-	-	+
Fractal	+	+	+	+
CCA	+	-	-	-
ProActive/Fractal / GCM	+	+	+	+/-
P2PCM	+/-	-	-	-
Hadas	+/-	+/-	-	+/-

9 Autonomic Management Systems

This part of the document describes the related work on autonomic management systems. The first section presents the motivations and objectives of these systems. The architectural patterns that are commonly used in the software design of these systems are overviewed in Section 9.2. Then the following sections focus on a particular autonomic aspect, including self-configuration, self-repair, self-sizing and self-protection. Finally, a conclusion is given in the last section of the chapter.

9.1.1 Motivations

Because of the increasing complexity of networked computer systems, autonomic computing [KC03], which aims to build self-managed and self-healing systems, has become an important research agenda. The software architecture of distributed systems is becoming more and more elaborated, involving several tiers and combining different legacy middleware technologies (Web servers, EJB servers, databases, etc.). Consequently, the management of such systems, which include tasks such as system deployment, configuration, reparation in case of failures, etc., is particularly error-prone and costly in terms of resources of the following type:

- Human resources are required because most management tasks are at the moment performed by human administrators. Moreover, events such as failures must be rapidly managed in order to minimize the impacts on the system availability, which implies to have human administrators waiting for these.
- Material resources are required, because the general solution adopted to ensure a system's availability despite failures or incidents relies on an overbooking of resources (such as nodes).

To improve this situation, autonomic management aims at performing administration tasks without (or with minimal) human intervention. This trend has emerged as an important research agenda in the face of the ever increasing complexity and pervasiveness of distributed applications. Autonomic management has the following goals:

- Improve the system's reactivity to its changing environment: events such as failures or variations in an application's load are tracked by the autonomic management system which immediately reacts by adapting the system to improve its processing. Examples of reactions are: a node reparation, a better sizing of the resources allocated to a given sub-system, or the reconfiguration of a system's parameters.

- Optimize the use of resources: an autonomic management system allows to bring down human resources (administrators) because human interventions are limited to the management of exceptional situations, as well as it allows to reduce the use of material resources by continuously adjusting these to the application's need.

- Reduce the management errors: distributed applications are administered through complex and specific management interfaces and functions, which is error-prone. An autonomic management system aims at simplifying this task by providing a uniform management interface.

9.1.2 Objectives

This section defines more precisely the functions that are expected from an autonomic management system. From a global point of view, such systems are intended to provide a support for facilitating the management of a distributed application. The management tasks include but are not limited to:

- Installing the application's software on a set of nodes.
- Configuring the application's software architecture.
- Deploying the application's components on a set of nodes.

- Repairing the application in case of failures appearing in the environment or in the application itself.
- Adjusting the use of resources.

More precisely, an autonomic management system aims at providing these tasks in an automatic way [KC03], leading it to support self-*autonomic properties:

- Self-configuring: denotes the ability of a system to be installed, configured and deployed without human intervention.
- Self-repairing: denotes the ability of a system to observe and to repair itself in case of failures, without human intervention.
- Self-sizing: denotes the ability of a system to adjust its use of resources without human intervention.
- Self-protecting: denotes the ability of a system to protect itself against malicious actions without human intervention.

To gain these autonomic properties, a common requirement is that the system must have the knowledge of itself: from what components is it composed, what are their current state, etc. Moreover, the system needs to observe itself through relevant probing functions, as well as it requires the ability to adapt itself to take into account the observed events.

9.2 General architecture

A common approach to autonomic computing, called the *control loop* approach, views the functioning of an autonomic system as follow.

- The overall management goals are expressed at a high level, and their translation into technical terms is performed by the management system.
- The management system observes and monitors the managed system; the observation may be active (triggered by the observer) or passive (triggered by the observed element).
- On the base of observation results, the management system takes appropriate steps to ensure that the preset goals are met. This may entails both preventive and defensive actions, and may necessitate some degree of planning.

In this approach, an autonomic system is governed as a feedback control loop involving Managed Elements (as presented in 9.2.1) and autonomic managers (described in 9.2.2).

9.2.1 Managed Element

The building block of an autonomic system is a Managed Element (ME). Fig. 1 illustrates the structure of a ME, which is composed of one (or more) managed resources. A resource is a material or software element that is required by an application, such as computers, hard disks, databases, files, servers, etc. Any managed resource is *wrapped* into a ME which provides a uniform management interface overshadowing its specific management interface. This interface covers two aspects: probing the managed resource through methods called *sensors*, and acting on this resource through methods called *actuators*.

A sensor aims at observing a ME in order to give information on its current state. The communication of the information can be initiated by an autonomic manager, or it can be done in response to a given event. For instance, in the case the managed resource is a computer, the use of the CPU is a measure which can be probed and periodically communicated to the interested autonomic managers.

The actuator interface allows the ME to be modified from a management point of view. The aspects which can be manipulated through an actuator interface depend on the type of the managed resources. However, common management aspects include the properties, the relationships, or the lifecycle of the ME.

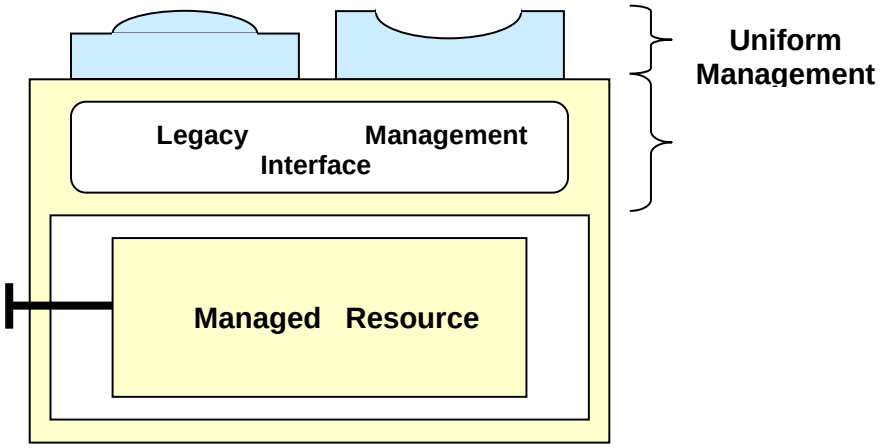


Figure 1 A Managed Element (ME)

9.2.2 Autonomic Manager

An autonomic manager (AM) is a software component in charge of ensuring a given autonomic behaviour on a given set of ME (Managed Elements). Generally, a given ME can be under the control of several AMs. An AM is built through a control loop involving the four main following parts, as illustrated in Fig. 2.

- Detection: this part is in charge of detecting some events concerning the controlled ME.
- Analysis: this part analyses the current system state after the detection of an event and decides if some actions have to be taken.
- Planning: this part typically proposes a workflow of the actions that should be executed in reaction to a detected event.
- Execution: this part is in charge of executing the workflow resulting of the previous task. This workflow is composed of invocations of the actuator interfaces of the ME.

It should be noted that an AM can be considered as a ME in order to be under the control of a higher level autonomic manager. In this case, the AM should exhibit the uniform management interface expected from any ME.

The detection part collects information on the set of ME. This information can correspond to the current properties values of the ME, as well as it can exhibit performances measurements, or the current relationships with some other ME. While some kind of information is relatively stable (e.g., relationships), other information can present a strong variability (e.g., performances measurements). In general, the detection part has the ability to save this kind of information in a common information base (e.g., a database).

The analysis part has to take some decisions to manage to the detected events, according to a given politic. Different politics can be considered, and registered in the information base, which gives the ability to the system to dynamically adapt an AM's politic.

The planning part builds a workflow of the actions to be executed. Once again, this building may be subject to different strategies (e.g., sequential / parallel workflows).

Finally, the execution part has to execute the workflow resulting from the previous step. This part can also update the information base to keep some history of the executed actions.

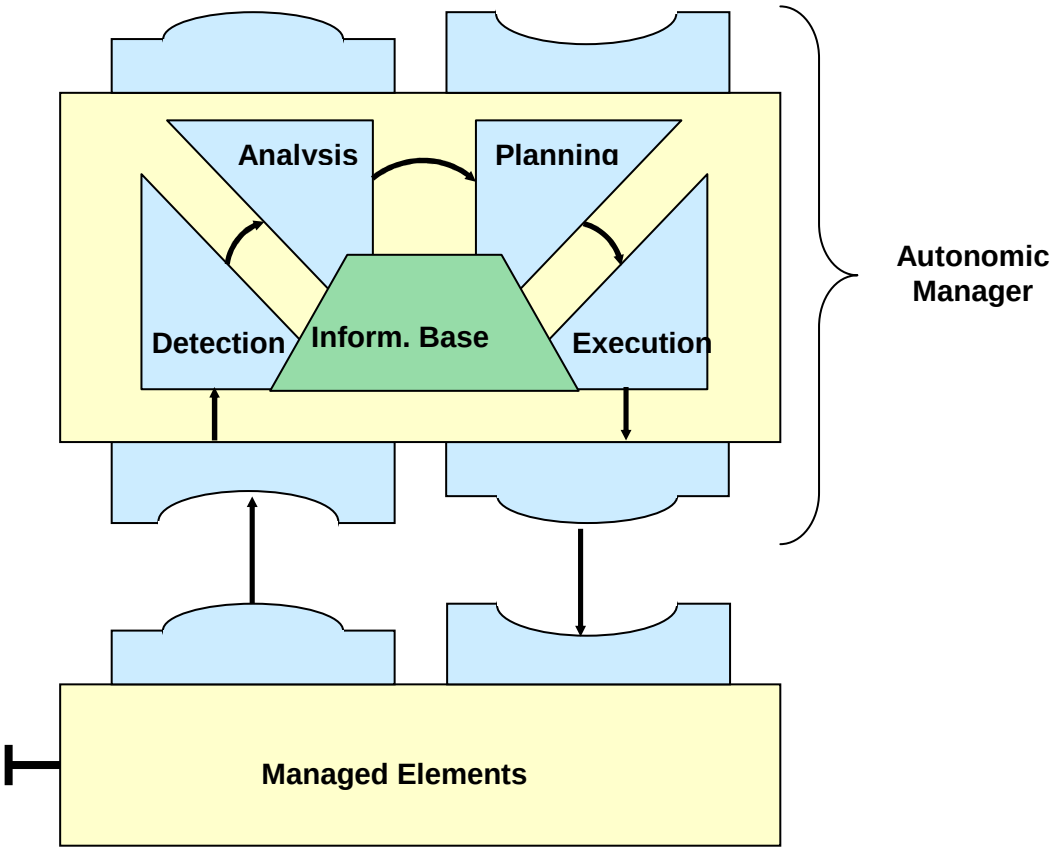


Figure 2. The basic structure of a Control Loop.

9.2.3 Hierarchical Autonomic Managers

Hierarchical Autonomic Managers is the solution presented by IBM [KC03] to maintain autonomic properties on an overall distributed system. The problem is to maintain several autonomic properties on a set of recovering managed resources, which leads to coordination problems. Fig 3 illustrates this hierarchical organization.

The Managed Elements (ME) are at the bottom of the organization. They are under the control of one or more AMs. These AMs are themselves under the control of a higher level AM (coordinator AM) which applies a conflict management policy (e.g., give priority to the repair manager except when this would degrade the quality of service of a specified client). At the top of the organization is an AM which provides a management interface to a human administrator. This interface allows the administrator to have a global knowledge of the system state, as well as it allows him to adapt the politics and the strategies of the different AMs.

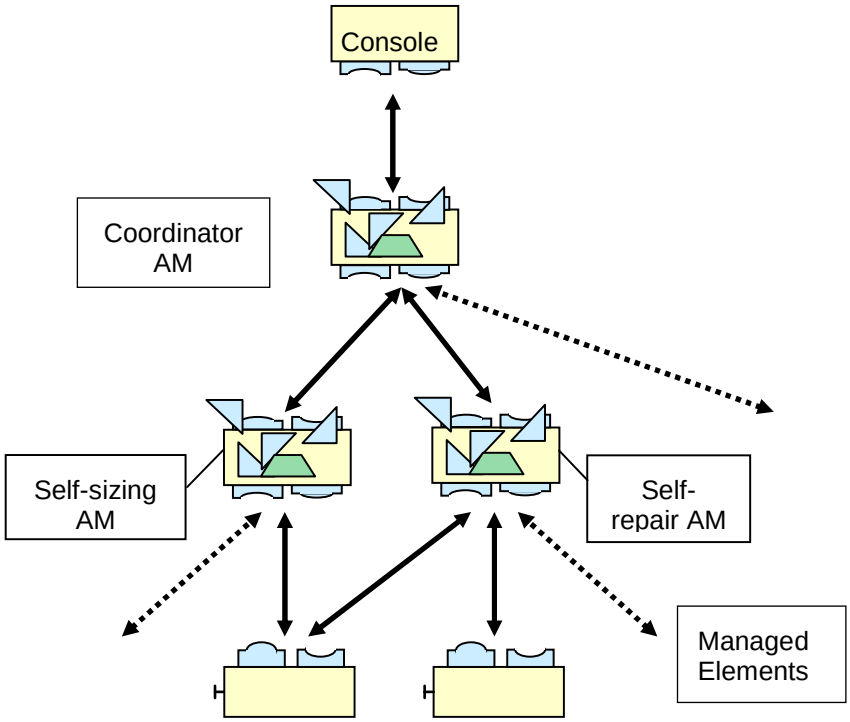


Figure 3 Hierarchical Autonomic Managers.

9.3 Autonomic Behaviors

This section gives details on the usual autonomic properties considered by autonomic systems and summarizes the related work associated to these properties.

9.3.1 Self-Configuration

Self-configuration gives the ability to a system to configure itself according to the properties of its execution environment without human intervention. An example of self-configuration is the automatic adjustment of the size of a thread pool of a server according to both the capacities of the machine on which it must be deployed and the expected server load.

The configuration task may include setting the parameters of an application, generating configuration files, as well as looking for services or resources required by the application. Concerning this last point, it should be pointed that a large number of services providing a same function may be available in the environment. The selection of an appropriate service required by an application may have to take into account several aspects, such as heterogeneity, performance, protection or security management. This may lead to the use of a registry service providing a high level standardized interface to register and to lookup services.

Self-configuration can also be applied at execution time. This should allow an application to support dynamic exchanges of services according to changes in the environment. It should also permit to adjust some parameters such as the size of a thread pool (if the application supports such a dynamic change), in order to match with the dynamic load of the application.

Wang. & al. [HS02] proposes a new P2P architecture associated to a self-configuration mechanism allowing machines to connect themselves on the overlay network in a relevant fashion. This mechanism takes care of the heterogeneity of the machines performances. It allows the overlay network to be configured without having access to a global knowledge of its state. A hierarchical structure is built with the machines presenting high performance capacities (called index machines). When a machine connects itself to the overlay network, an assessment of its performance capabilities is performed. This determines the number of connections in which the incoming machine should be involved. This machine should then find available connections by making requests to index machines. At the end, if the incoming machine is connected to machines having lesser performance capabilities than it, it becomes an index machine.

Chess & al. [CSW05] develops an architecture named Unity which aims at automatically assemble autonomic software elements being given functional objectives. Each autonomic element present in the environment is registered within a global registry, in terms of the services it provides. When a new autonomic element arrives in the environment, it firstly tries to find the services that are required by itself, before registering its provided services. A basic and essential service registered in the registry is the *configuration politics service*. It is supposed to give access to all the information required by an automatic element to perform its self-configuration.

The SWORD project aims at exploring techniques for managing resources in wide-area distributed systems [OAP05]. SWORD is a resource discovery tool that collects reports about available resources on nodes, and answers queries from users requesting nodes matching user-defined criteria (*e.g.,* load, free memory, or free disk space) or *inter-node* (*e.g.,* inter-node latency).

Self-configuration has also been explored in the context of replicated DBs [MJP04, JPA02]. One of the main challenges in replicated DBs is being able to maximise the performance of the cluster as a whole. This implies the ability to maximise the performance of each individual replica according to its current workload and configure the system at the global level to attain an even load across replicas. This is precisely the approach taken in [MJP04]. The performance of individual replicas is maximised through autonomic load control, whilst the performance of the system at a global level is maximised through dynamic load balancing.

9.3.2 Self-Repair

From a global point of view, the self-repair property provides a system with the ability to repair itself in case of failures appearing at any time in the application, in the environment or in the management system itself. It should be pointed that a self-repairing system does not intend to avoid failures. On the contrary, it is based on the observation that failures cannot be fully prevented.

The self-repair property is generally provided through one or several instances of a repair service, which internal organisation follows the control loop structure illustrated in the previous section. Several different kinds of failures can be considered by a self-repair service: software failures, hardware failures or failures due to the operator, transient failures, fail-stop failures, etc..

An important aspect is that error-masking techniques, which are widely used when designing distributed application, are not considered as repair solutions. Such techniques mainly use redundancy to provide an application's availability despite a fixed number of failures. In this context, self-repair of computing systems is investigated by the Recovery-Oriented Computing (ROC) initiative, a joint project of Berkeley and Stanford universities that aims at studying approaches for building highly-dependable Internet services [BS07]. In case of hardware fail-stop failures (e.g. node failures), replication enables to tolerate failures of individual nodes to provide high availability. However, this is not enough since failed replicas should be replaced, which is precisely the task that must be performed by the repair service.

Several approaches and protocols have been recently proposed to handle hardware failures by enabling online recovery of cluster-based J2EE servers [BBH05] or replicated database servers [SPS05, JPA02] without disrupting operation of the replicated servers during it.

Other approaches are also proposed to handle transient software failures, such as applying recursive reboot techniques to stop/restart failed components; this was experimented with centralised J2EE servers [CKF04]. Another approach to tackle such failures is to apply retry techniques as experimented with cluster-based J2EE servers [BPK05].

Self-repair from failures due to the operator can be obtained using undo techniques that allow the operator to recover from his mistakes, as applied to centralised e-mail servers [BP03].

It should be pointed that self-repairing an application implies that self-configuration capabilities be provided for this application, in order to be able to redeploy some software elements dynamically.

As said in the introduction, distributed applications have a complex software architecture, consisting of several interacting parts, each of which has a fair degree of intricacy. A repair service should take into account this architecture, because restarting or replacing an element in this architecture implies that the relations with the other elements be re-established. Such relations can correspond to connections such as effective communication channels between elements (e.g., TCP connections). They can also be represent a dependency between two elements (e.g., a shared file name attribute for instance). To be able to get the knowledge about the architecture of a distributed application, some projects [GCS03, BBH05] maintain an up to date view of this architecture during the application execution. Depending on the design choices, this view may be more or less expressive. In [GCS03], architectural constraints may be defined, allowing the evolution of an application's architecture to be controlled.

9.3.3 Self-Sizing

The self-sizing property allows a system to react to environmental variations in order to optimize its performances. The term environmental refers to all aspects which can intervene on an application processing: load variations, resources evolution, etc.

For cost reasons, several applications may have to run in the context of a shared execution environment like a Grid. In this context, the self-sizing of an application should take into account the sizing requirements of the others running applications. Two main principles are considered:

- The isolation principle ensures that a minimum set of resources is provided to an application independently of the requirements of the other applications. This minimal set of resources should be available when the application requires it. While it is not required by the application, it can be allocated to another application.
- The differentiation principle associates different priorities to different applications, with regard to their resource provisioning.

Self-sizing has been studied by several research projects, such as Q-Fabric, Cluster Reserves, OnCall and JADE.

The Q-Fabric project [KPG03] at the Georgia Institute of Technology provides more specifically support for quality management in order to provide end-users with the quality of service they require.

Cluster Reserves [ADZ00] is designed to take into account performance isolation constraints in Grid environments. On each machine, where several applications can run, a set of probes are deployed as well as a Resource Manager. The probes allow the Resource Manager to know about the consuming of resources for each application. Performance isolation is performed globally at the Grid level through a (global) matrix which centralizes all this consuming information.

OnCall [NCF04] implements an isolation mechanism where each application is ensured to be provisioned with a minimum number of machines. A probing mechanism allows detecting the over-provisioned applications, which can give some machines to under-provisioned applications for a given cost.

The JADE project at INRIA investigates the concepts and techniques for providing cluster-based J2EE systems with self-optimisation, such as dynamically increasing/decreasing the cluster size to improve application performance according to workload variations [TBB05].

9.3.4 Self-Protection

In order to prevent network intrusions, many methods have been developed, notably firewalls and intrusion detection systems (IDS). But these techniques have a number of limitations. First of all, most of these tools report abnormal behaviour (perhaps attacks) to administrators, who must then carry out a manual analysis of the problem. This analysis may take time and allows the pirate to freely exploit the flaw whereas a fast answer would have stopped the attack while it was still in an early stage. Moreover, current security tools can often only protect systems against known attacks and pirates are always a length ahead. Finally, security tools are very difficult to configure in a distributed computing environment.

The main tools and techniques currently used by security experts to fight against intrusions make the distinction between different functions (protection filters, detectors of suspicious activity, logging and backtracking of tools), although many available solutions integrate several of them.

Protection filters are used to restrict interactions among machines (or, more generally, distributed processes / resources), to a given set of limited, well-established set of patterns. For instance, a firewall acts as a network filter that checks if any given packet can be forwarded according to its related protocol, source/destination addresses and ports.

Detectors (or scanners) used to recognize malicious activity fall generally into two categories [Sun96, [DDW99]: (i) misuse intrusion detection and (ii) anomaly detection intrusion. The former approach compares the data packets passing through the detector with a library of patterns typical of known attacks, while the latter tries to spot irregular behaviour of the system. In addition, scanners can sometimes react themselves against the intrusion, but their action is usually limited in scope (block offending request / packet, quarantine suspect resource) and context (no coordination between the different servers). Thus human intervention is generally still required for further study and containment of the problem.

Forrest & al. [FHS96] proposes a solution for detecting malicious processes on Unix machines. This detection is based on a preliminary observation of an application under non-intrusive circumstances. During this stage, all kernel calls are registered in an ordered manner in a database. After this observation stage, the application execution is supervised in order to detect some sequences of kernel calls that do not follow the previously registered sequences.

The Vigilante system [CCC05] is an antivirus system where detectors are based on the immune system analogy and are able to find unknown viruses. Furthermore, when a new virus is found, its signature is spread across the network to all other protected computers.

Self-cleansing [HS02] is another solution to build self-protected software. This pessimistic approach makes the assumption that all intrusions cannot be detected and blocked. In fact, the system is considered to be compromised after a certain time. Hence, this approach periodically reinstalls a part of the system from a secure repository. However, this solution only applies to stateless components. When a computer is compromised, another important function is the ability to restore the system in a trusted state.

Finally, the Taser system [GPF05] provides a file system with a selective self-recovery capability. Taser logs all file system access for each process. If a process is compromised, Taser computes illegal access for each

file and is able to rollback illegal modification. However, if a dependency is found between an illegal and a legal access, Taser requires a human intervention.

9.4 Conclusion

Despite the number of projects working on the domain of autonomic management, the providing of a complete solution is still a challenging research area. A number of projects tackle a particular aspect of autonomic management (e.g. self-recovery [BS07], self-configuration [MJP04], etc.), and often propose solutions that cover only partially the overall requirements of the targeted aspect.

Some projects provide a more global management framework intended to support several self-* capabilities [GCS03, BBH05]. One main issue that is not completely solved by these systems concerns the coordination of multiple management services acting simultaneously on common software components. In particular, basic synchronization features such as component locking only allows a simplistic management of these situations. More complex solutions, allowing solving conflicts between contradictory requests, should be investigated.

Besides the different autonomic management aspects, a capital property of the autonomic feature is its applicability to legacy systems. Concerning this point, a grand challenge of autonomic management is to explore the feasibility of building wrappers onto legacy enterprise systems, externally, without any need to understand or modify the code [BBH05, KPG03].

Another aspect that is not completely mastered concerns the size of the environment in which the autonomic management functions should operate. While most solutions target medium-sized environments (e.g., clusters), the consideration of large scale environments such as grids or overlay networks is an emerging research area.

Finally, in addition to academic research on autonomic management, several industrial initiatives are conducted, such as IBM Autonomic Computing [KC03], Sun N1 Grid Engine 6 that provides resource management facilities [Sun07], and HP Adaptive Computing through the SmartFrog framework for self-configuration and deployment of computing systems [Sab06].

10 General Architecture

In chapter 3 we described the division of responsibility of the infrastructure and the higher-level management functions. There are four types of system elements: resources, components, services and members. Our dynamic VOs are characterized by high levels of churn and evolution. Potentially this applies to all four different types of system elements. This makes management of such systems challenging, and our goal is to provide an infrastructure that provides the necessary support.

In chapter 3 we divided management into the four aspects sensing, actuation, management logic, and the computational platform. The infrastructure is responsible for three of these four aspects (all but the management logic). We begin by considering the first two, sensing and actuation only.

In theory the management logic could be run on a single management node, and all interaction with other nodes would then take place through the sensing and actuation services. This (naive) simple model can be further clarified. The management works locally with a system representation of the system elements (member, components and resources). The management reasons about system elements without knowing where they are. The sensor service ensures that the system map is kept up to date (within the limits of network latency). Actuating done on the system element representations automatically triggers the appropriate actuation commands sent to the represented elements over the network.

This makes for an implicit overlay in our design, a logical network that connects all the nodes of the system. In our design, as we shall see, the overlay is quite explicit and henceforth we refer to the actuating and sensing support services as *overlay services*.

In this simple model all the other nodes in the system are ‘known’ to the sensor and actuating services. They are, in fact, all part of the same overlay. When nodes enter or leave the system they join/leave the overlay. When resources enter/leave the system they report this to their local overlay proxy, an action that may via a suitable message lead to a management action as some management rule is triggered. An actuating command such as deploying a component on a given resource will eventually reach the correct node and trigger over the client API the appropriate deployment.

These issues have been discussed to a greater detail in the paper and technical report [Bra+07].

10.1.1 Management support requirements

Requirement 1, complete

Our first requirement is that overlay services provide all the needed sensing and actuating information. Potentially all state changes in the system elements (resources, components and members) can be monitored. Up to the limits of network connectivity this also includes failure, which obviously cannot be reported by the failing system element itself. Furthermore the overlay services are able to deal with arbitrary system elements. In other words the overlay services are able to deal with heterogeneous resources.

Requirement 2, push and pull

The distinction between sensing as triggered by the observer or the system element (push/pull) is an important one in distributed systems as it illustrates a fundamental trade-off where the best solution depends upon the context. There are three aspects to the design of sensor services. Firstly it allows the management to actively observe the system or parts of the system. This includes discovery, i.e. trying to find specific types of system elements. Secondly, it allows for the installation (and deinstallation) of specific sensors. Examples of such sensors 1) tell me when an element of a certain type becomes available, and 2) tell me when a certain component or resource has failed or left the system. Thirdly, it reports to the management node when the installed sensors are triggered.

Requirement 3, multiple management nodes

There should be nothing that prevents the overlay from having multiple management nodes. It is up to the management logic to deal with the concurrency control issues when multiple management nodes act/sense on the same system elements. However, there are no restrictions within the overlay service per se.

We will make no assumptions about timings. The sensing and actuating model is asynchronous and successive actions are unordered. For example, two successive actuating commands may arrive in any order even if the destinations turn out to be co-located on the same node

Requirement 4, abstraction

The simple straightforward approach of sensing (and actuating) all system elements (via push or pull) from a single management node (or even a small set of management nodes) might not scale. In the considered dynamic VOs, this approach would result in an flood of status information and require a continuous detailed stream of commands to adjust the system in the face of high degrees of churn. Not only might this exceed bandwidth/storage limitations, but the complexity of this micro-management would make automation of the management difficult.

For this reason we also require that the overlay support services to be characterized by:

Hiding – sensors may ignore churn (e.g. one free resource replaced by another, or when a component has been moved from one node to another).

Aggregation – sensors detect aggregate properties/events rather than individual properties/events (e.g. when 5 free Linux resources become available). Also actuators may act on groups.

Abstraction – we have abstractions over resources, components, members and their relationships. One example is component placement abstractions (e.g. 3 components A, B, and C are to be deployed, A and B are close in the sense that they interact frequently – information the deployment uses when allocating appropriate resources).

Building abstractions over sensing/actuating is an open-ended activity. The abstractions are useful for simplifying management logic. However, this is not enough. An abstraction may either be an integral part of an overlay service or an abstraction that runs exclusively on a single management node (where it aggregates sensing events and/or actuation events and presents the result on a higher level). These two different design choices can be made functionally equivalent. Therefore our design criterion for incorporating abstractions into overlay services is that something can be gained non-functionally by doing so. By gain we mean that fewer messages need to be sent, the latency is lower, etc.

Requirement 5, intelligent delegation

Finally, also in the interests of scale we require the following:

Decentralization and delegation – delegation is natural in conjunction with hierarchical component models (e.g. deploying the self-management function of a subsystem of components on an appropriate resource). Also tasks of self-management may be split in aspect-oriented manner and decentralized (e.g. self-healing in one place, self-tuning in another).

Requirement 5 and 2 are related in that they both imply parallelization of management, however requirement 5 implies that the decision to delegate some management logic is made dynamically. One example would be when a management node becomes heavily loaded that some management functionality can be delegated, e.g. the node delegates logic associated with tuning to some other node, but keeps healing and configuration functionality in place.

Moreover, delegation can be made intelligent, in the sense that the delegated management logic may be placed optimally. In particular a management rule that is triggered by sensor events may be placed close to (or at) where the sensing events originate. Alternatively the management rule may be placed close to where actuation commands are to be sent to. In some cases determining optimality is complex, depending on latencies between nodes and probabilities of various kinds of sensing events. At the same time there are rules where the determination of optimal placement is easy, for example a management rule that triggers on a resource leaving the system (e.g. member going off-line) is best executed at the node containing the resource.

Our requirement is that the architecture is open-ended with respect to the intelligence of delegation, i.e. that we make no assumptions that limit such delegation.

10.1.2 Overlay service requirements

We have identified a number of requirements on overlay services that reflect the properties that DHT-based peer-to-peer systems have demonstrated. These may be seen as three non-functional properties of the overlay services (in addition to the previous functional five). We have:

Requirement 6, self-managing overlay

The overlay is in itself self-managing. The overlay topology is maintained in the face of nodes joining, leaving and failing. This has been demonstrated for DHTs assuming no network partitioning up to fairly high rates of churn. As the focus of our work is not to develop DHTs, a practical formulation of our requirement is that our overlay services are self-managing to the same extent as the best DHT-based systems.

Requirement 7, service robustness

Traditionally DHTs and other peer-to-peer systems offer a data storage service. Systems work with pairs of the form Key:Data, where the data is non-mutable. Robustness of this particular service may be characterized by that if data has been inserted it can always be found, even in the face of nodes leaving and failing. In DKS this is achieved by 1) suitable replication and 2) the details of topology maintenance (the higher the churn the larger degree of replication is needed).

The sensor and actuating overlay services offer other kinds of services. By robustness here we mean:

1. Within the limits of network connectivity a sensing event that takes place is always delivered to the management logic that has subscribed to it.
2. Within the limits of network connectivity an actuation command that takes place is always delivered. Clearly, if the actuated element has failed then the command cannot be delivered and if the actuated element fails after the command is given then it may or may not have be delivered.

Requirement 8, efficiency

We require the overlay services to be reasonably efficient in terms of the number of messages sent as well as latencies. Good efficiency measurements are open issues, but in many cases it is possible to compare the functionality of our overlay services with the functionality of peer-to-peer storage systems, in which case we require the best that DHTs provide.

Where there are fundamental trade-offs we should provide the means to tune according to application or environmental scenarios. For example, in DHTs there is a trade-off between topology maintenance overhead and routing table size, which go hand-in-hand, and lookup path length. The best choice depends on the rate of churn and available bandwidth.

Requirement 9, security

It should be possible to make the overlay services secure, although for performance reasons one might not always want to. This comes down to providing necessary hooks and handles for security mechanisms to be put into place. For example, communication between nodes may or may not be encrypted. Note that our concern here is security for the sensor and actuating services only; there are many security considerations on the level of the management logic particularly in conjunction with delegation which are outside this scope.

10.1.3 Intersection Requirements

In this subsection we need to consider some details of the resource and component models that the management logic uses. We need to ensure that there is a good match between the overlay services (as part of VO management) and the corresponding models. It is possible that we will, in the future, also need to consider the match between the membership model and the overlay services.

Resources and Components

We are not precise as to the definitions of components or resources (from the viewpoint of overlay services). Some further clarifications can be made:

A *resource* is a consumable entity needed for 'component' deployment. Resources represent physical capabilities of nodes, like the amount of available memory. Resources are in a given state. The most important are *free*, *used, leaving* and *failed*. These states are self-explanatory except for the state *leaving* which indicates that the used resource will soon leave the system and the system is given time to perform the necessary re-configuration. Note that if a machine is to be taken off-line then all used resources will enter the state *leaving*, but in the case a the machine owner suddenly needs machine resources for himself and hence can provide less to the VO may entail that only a subset of used resources enter the *leaving* state. Associated with a given resource is a description of the resource (e.g. memory, cpu, etc.)

Note the close analogy between placing data into a free storage resource and deploying a software component on a free computation resource. In both cases the resource enters the state used and we can say that the 'component' is deployed on the resource. In both cases the resource can later be freed when the software component is stopped and deallocated or when the data is deallocated.

For practical reasons we allow resources to be split and merged. The rationale behind this is to limit the number of free resources that need to be monitored together with optimal resource usage. For example, a component of size 10 is to be allocated on a resource of size 100, in which case the resource is split into two pieces of size 10 and 90. The size 10 resource will then be used to deploy the component.

The design is open-ended with respect to how much of the resource description is exposed to the resource overlay service. If all is exposed then the resource service can, in principle, match resource with component needs by itself, if none is exposed then all matching must be done outside the overlay service and discovery may require contacting all nodes. Our requirement can be formulated as follows:

Requirement 10, resource modelling

The overlay services need to be partially aware of resource properties so as to enhance discovery performance.

10.1.4 Software Component Model

Maintaining a distributed application or service with components on many machines in a dynamic VO is challenging. Leaving aside failure, the resources that the application is running on may leave the system, in which case it is up to the management to move the component to another resource. The (software) component model must therefore include provisions for moving (i.e. passivating, moving and restarting) components. We therefore see that management will need to trigger the conversion of a running component

into data, and then use this data to recreate the component at another resource. It is clearly advantageous to hold the data close to resources, and avoid unnecessarily sending the data back to the management node. This exemplifies our final requirement:

Requirement 11, component modelling

The overlay services need to be partially aware of the software component model/properties so as to enhance re-configuration and other management activities.

The component sensor and actuating services need not be aware of all component properties. The component actuating service locates the given component and passes the given, but for the overlay service completely opaque command, to the component, collects the answer (also opaque) and passes it back to the management node.

Other examples of software component model concepts that the component overlay service needs to be aware of are binding, deployment, stop and start. Of particular importance is binding, which can be seen as the process of forming relationships between components.

Requirement 12, software component model

The (software) component model should capture those properties that will enable full use of the potential of the overlay services.

The concept of bindings is one feature of component models that needs to be re-examined and loosened. Typically (primitive) bindings between components are one-to-one low level links formed by the appropriate operations at both ends. This needs to be replaced by a rich flora of different types of bindings. Examples are one-to-many (broadcast communication), one-to-a-group (SOA style where any will do), bindings between named components versus located components.

From the functional point-of-view the rich flora of binding types is programmable, but this would not be equivalent as regards non-functional properties. They need to be made primitive in the sense that the component actuating service is aware of them and can handle them optimally. For instance, a multicast binding, through the overlay can use an overlay multicast to realize the binding (whereas the best that can be achieved with programming is repeated unicast by the programmed binding abstraction). Another important advantage is that when bindings are done through the overlay service that component on one end of the binding can be moved without requiring action at the other end.

10.2 Architecture

10.2.1 Introduction

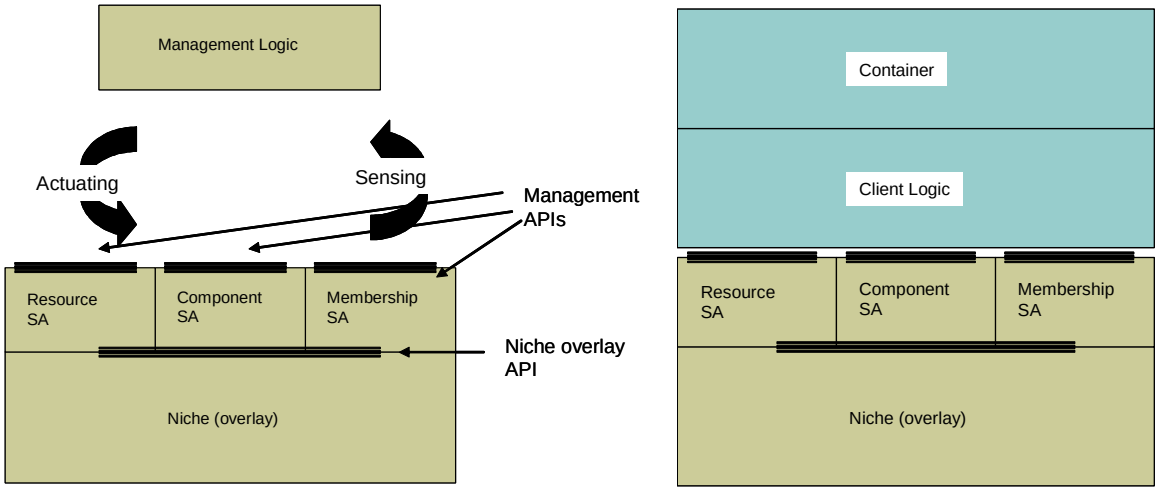
The figure on the left shows the system architecture at management nodes. The management logic senses and actuates through the three overlay services through well-specified management APIs.

1. Resource sensor and actuator service
2. Component/service sensor and actuator service
3. Member sensor and actuator service

The three services are reflected in three front-end subsystems that perform only a small amount of computation, packaging and bookkeeping. The largest subsystem is labelled *Niche*, the local representative of the *Niche* overlay on the management node. Communication with other nodes takes place exclusively in the Niche layer.

The figure on the right shows the system architecture at client nodes. The client logic component interacts with the three overlay services through the client API interfaces. Once again the three service components

(Resource SA, Component SA, Membership SA) perform only a small amount of computation, packaging and bookkeeping. The Niche component is the only component that communicates with other nodes in the VO. The container contains those resources, components that are allocated/located on the local machine on behalf of the VO. Examples of interactions that take place along the interface between the client logic and the overlay services are 1) notifications of resources joining and leaving the VO as the owning members withdraws and add resources to the VO and 2) components sending to remote components via bindings. Note that a physical node may be both a manager and a client, or a client only representing that it has no management privileges whatsoever. All machines within the VO are thus either clients or client/managers.



10.2.2 Resource Model

The Resource SA service monitors all the free resources of the entire VO, and asynchronously notifies its subscribers about the best-fitting free resources available according to the individual subscription criteria. (It may also be used to monitor the used resources of the entire VO).

When a free resource suitable for component deployment or re-deployment is found and allocated one of two things will take place. First, the resource may leave the state *free* and enter the state *used*. From the point-of-view of free resource monitoring the resource disappears from view. Alternatively the component only needs a part of that resource R in which case the resource R is split into two smaller pieces, one used and one free. The free piece keeps the name or handle R but its description changes to reflect its smaller size. The used piece gets a new name or handle R2.

The granularity of modelling the free resources of a physical machine is not set a priori. One possible policy is to ensure that there is always at most one free resource per physical machine; another is to divide the total resources from start in such a way as to avoid resource splitting altogether.

10.2.3 Management Context

It needs to be understood that the management APIs, the interface between management and the overlay services is at a fairly low level. We expect that the programming model for management is at a higher level. These high-level descriptions will by suitable tools get transformed/compiled into a low level **management assembler**. These tools and programming models are the province of WP2, but we will, for the sake of providing the context, briefly summarize two approaches.

ADLs: An architecture description language (ADL) is used to specify declaratively initial application deployment and simple self-configuration and self-healing behaviours of the system. Architectures are

specified in terms of components and bindings. Component descriptions include requirements and preferences for resources necessary for deployment of the component. Component descriptions state also component properties crucial for management logic, such as whether the component's state can be extracted into a data structure.

Abstractions over events and event handlers: Management aspects can be specified in terms of events and handlers and abstractions thereof. Events reflect status changes in the VO, such as availability of resources, and status changes of application components, such as failures. Event handlers evaluate the status of the application and the environment and can replace, add and remove application components and bindings.

The two approaches are analogous to that of developing high-level programming languages for uniprocessors. One approach, analogous to the ADL approach, takes a declarative view which by suitable emulation/interpretation can be realized on the non-declarative machine (e.g. as in functional or logic programming). The other view abstracts the underlying machine (e.g. C) while maintaining its imperative character. Both approaches have in common that errors are fewer or caught at compile time.

The management assembler works with mutable references to low-level entities such as resource and component handlers, entity and VO status watchers, and stateful event handlers. Later on we will illustrate management assembler with pseudo-code.

10.2.4 Management

The management logic assembler can be divided into the following different categories of instructions via their place in the management feedback loop, some are involved in sensing/monitoring the managed system and some are involved in actuating/commanding the managed system.

Events & event handlers: Events are the way in which sensing is realized. Typically the manager logic has an event handler that takes the appropriate action on receiving the event.

Sensor installation: This instruments the appropriate sensing. This class of operations may also be seen as publish directives. In this category are *discovery* operations, where the management asks the overlay services to find appropriate resources for it. Also in this category are *watchers* whereby the management asks the overlay services to monitor a specific entity (resource, service, component, and member). Note that if no watchers are installed on an entity then no information will be made available on that entity. The management can also stop any particular discovery or watching operation.

Triggers: This is an actuating part of the management logic whereby a command is sent to a specific entity or groups of entities (e.g. member, resource, component).

Activation of event handlers: Activating an event handler may be seen as a subscription. It also represents the dynamic part of actuation in that the body of a now activated event handler represents actuation operations that can now take place. Event handlers can be created and/or stopped. There may be more than one event handler listening to the same event (which may then be triggered in some arbitrary order). There may be no even handlers listing to some event in which case nothing happens.

Other: traditional programming constructs

10.3 Management API

In the following section the management API is described in an abstract form. We describe the more important functions of the resource and component overlay services.

10.3.1 Notation

We use the following types of identifiers, all which are VO-wide:

Resource identifiers (ridX, aridX) – an identifier of a resource, where aridX is allocated, ridX is not

Component identifiers (cidX) – an identifier of a component

Watcher identifiers (widX) – an identifier of a watcher
Binding identifier (bidX) – an identifier of a binding

There is a considerable amount of data-flow in the management functionality. For convenience we will make use of futures. A future is identified by a capital F in the variable name, for example ridF as a future for a resource identifier. A future when created represents a value that is not yet known but which will eventually be instantiated, given a value. For most futures one possible value is failed indicating that the operation failed.

10.3.2 Sensor Installation

Sensor installation is the way the management instructs the system about which events to monitor and report.

wid:discoverResource(requirements, comparison, tComparison, class, currentRes)

Installs a sensor which will continuously discover and report about the best matching resource that meets requirements with some restrictions, until explicitly stopped. The reported resources must be better than the resource described by currentRes. Resource candidates can be compared by the boolean function comparison. Discovery watchers in the same class are precluded from reporting the same resource. In addition the boolean threshold function tComparison must return true indicating that the new candidate is better by some margin compared with the current best fit. Wid is the watcher id which will be a parameter of events generated by the watcher.

widF:watchComponent(cid, componentParameter, comparison, tComparison)

Installs a sensor that monitors the componentParameter of component cid, with some threshold function given by tComparison.

widF:watchResource(rid, resourceParameter, comparison, tComparison)

Installs a sensor that monitors the resource rid and reports about state changes given by resourceParameter.

stopWatcher(wid)

Stops the sensor identified by wid, which is equivalent to cancelling all subscriptions.

10.3.3 Triggers

Triggers are actions by the management to change the state of a specific entity.

aridF:allocate(rid, specification)

Allocates the resource identified by rid according to specification and returns a future allocated resource, aridF. The future will either be instantiated to a valid id of a successfully allocated resource or become a failed future.

boolF:deallocate(arid)

Deallocates the allocated resource identified by arid.

cidF:deploy(component, arid)

Deploys the component specified by the parameter component on the allocated resource arid. It returns a future of the component id, cidF. The cidF may like allocated resource id become a failed future. The deploy trigger is overloaded. It is assumed that deploy as argument can take a checkpoint, image, code, url, etc.

fcidF:passivate(cidA)

Passivates the component identified by cidA, e.g. the component is stopped and serialized into data. The image of the component is stored in the system under the handle fcidF. What the image contains is application dependent, ranging from a complete serialized representation of the entire component state to, for instance a single state variable and a url to code.

boolF:start(cid) / stop(cid)

Starts or stops the component identified by cid.

gcidF:group(listOfComponentIds)

Groups a set of components into a group with the group id gcidF.

boolF:addToGroup(gid, cid) / removeFromGroup(gid, cid)

Adds or removes cid to the group identified by gid

bidF:bind(cid, gcid, bds, bdd, type)

Binds component cid to the group or single component gcid. The parameter type indicates the type of binding. Bindings are assumed to be uni-directional and asynchronous. There is a binding description on both the sending side, bds, and delivery side , bdd. They provide additional information, if needed, to properly connect to the component (e.g. specifying the port on the component).

boolF:unbind(bid)

Unbinds the binding identified by bid.

10.3.4 Events

Note that the events all have a watcher id, wid, argument. This argument is part of the pattern matching.

resourceReport(wid, oldState, newState)

The resourceReport event corresponds to the subscription watchResource. It reports that the resource has changed state from oldState to newState.

componentReport(wid, componentParameter, oldValue, newValue)

The componentReport event corresponds to the subscription watchComponent. It reports that the component parameter has changed enough to trigger the initially given threshold function.

discoveryReport(wid, rid, resourceDescription)

The discoveryReport event corresponds to the subscription discoverResource. It reports that a new resource is found which is better enough to trigger the initially given threshold function.

10.3.5 EventHandlers

Each watcher should have at least one event handler to process the generated events. An event handler has the form

upon event X(wid, es) **with** <attributes> **do**

This event handler is triggered by an exact match on both the event name, X, and the exact value of the id, wid. Es represent the other parameters given in the event, while attributes represents the initial attributes of the event rule. Note that as an event handler may be triggered arbitrary number of times, its attributes may need to be reset between invocations to ensure proper operation during subsequent invocations.

To activate or passivate event handlers, the following methods are used.

activateEventHandler(rule, wid, initAttributes)

Activates the event handler given by rule associated with the watcher wid. An event handler has attributes that are initialized as given by the list initAttributes.

passivateEventHandler(rule, wid)

Passivates the event handler given by rule.

10.3.6 Abstractions

We will also make use of the following abstraction. This abstraction can be programmed using discoverResource subscription and an appropriate event handler to set the value of the future.

ridF:oneShotdiscoverResource(Requirements, Comparison, Class)

discover the 'best' matching free resource **rid** that meets the **Requirements** and has the best fit. Resource candidates can be compared by the Boolean function **Comparison** which indicates which of two resource candidates is best. Discovery watchers in the same class (**Class**) are precluded from reporting the same resource. The resource id (**ridF**) is a future (but one that cannot fail). Note that it might be that there is no free resource in the VO that meets the requirements in which case the future remains a future.

10.4 Client side API

10.4.1 Client side logic initiated – downcalls

resourceJoin(lrid, description)

A new resource has joined the system. The RSA will convert the lrid to a rid before sending the information to the rest of the system. At the receiving end(s) the information will be found on demand through either resourceReport(s) or through a discoverResource call.

resourceLeave(lrid)

A resource is about to leave the the system. The RSA will convert the lrid to a rid before sending the information to the rest of the system. At the receiving end(s) the information will be reported through a resourceReport.

componentChange(lcid, description)

A (watched) component has made a state-transition that someone is interested in. The RSA will convert the lcid to a cid before sending the information to the rest of the system. At the receiving end(s) the information will be reported through a componentReport.

send(lbid, Object)

A component is making a call on an established binding. The call is directed through the CSA, and will end up as a deliver call on a node hosting the receiving component(s).

10.4.2 Overlay service initiated – upcalls

result:allocate(lrid, description)

The RSA has received a allocate request containing a rid from Niche. The RSA is responsible for converting the rid to a lrid before doing the upcall. The result might (newUsed_lrid, null) or (newUsed_lrid, newFree_ Description). In the case of null, the rid should be removed from the free resources. In the case of newFree_Description, the description of rid should be updated with newFree_Description. In both cases

(newUsed_lrid, description) will describe the used resource. The newUsed_rid is returned to the initiator of the allocate operation.

result:deallocate(lrid)

The RSA has received a deallocate request containing a rid from Niche. RSA will convert the rid to a lrid before doing the upcall. The result might be null or merge(lrid, otherPartOfMachine_lrid, newFree_ Description). In the case of null, the rid should be reported as a new free resource (this will lead to fragmentation). In the case of merge, the description of otherPartOfMachine_rid in should be updated with the newFree_Description. In both cases rid does no longer represent a used resource

lcid:deploy(lrid, componentDescription)

The CSA has received a deploy request containing a rid from Niche. The CSA will convert the rid to a lrid before doing the upcall.

bool:undeploy(lcid)

The CSA has received a undeploy request containing a cid from Niche. The CSA will convert the cid to a lcid before doing the upcall.

data:passivate(lcid)

The CSA has received a passivate request containing a cid from Niche. The CSA will convert the cid to a lcid before doing the upcall.

bool:start(lcid) / stop(lcid)

Ordinary life-cycle management interfaces.

lbid:bind(lcid, description)

The CSA has received a bind request containing a cid from Niche. The CSA will convert the cid to a lcid before doing the upcall. From the CSA point of view, this is a local operation which enables the component communication to be directed through send(lbid, Object)-calls.

bool:unbind(lbid)

The CSA has received a unbind request containing a bid from Niche. The CSA will convert the bid to a lbid before doing the upcall.

lwid:watchComponent(lcid, eventDescriptions)

The CSA has received a watchComponend request containing a cid from Niche. The CSA will convert the cid to a lcid before doing the upcall.

state:PollComponentState(lcid)

The CSA has received a periodicTimestamp request containing a cid from Niche. The CSA can be instructed to periodically poll the management proxy for the state of a locally deployed component.

deliver(lbid, Object)

The CSA has received a deliver request containing a bid from Niche, generated by a send from a component. CSA will convert the bid to a lbid before doing the upcall.

10.5 Use Case

In a existing VO the management decides to deploy an application consisting of one A component and 3 B components and where A is bound to any of the Bs. We have the following self* mechanisms

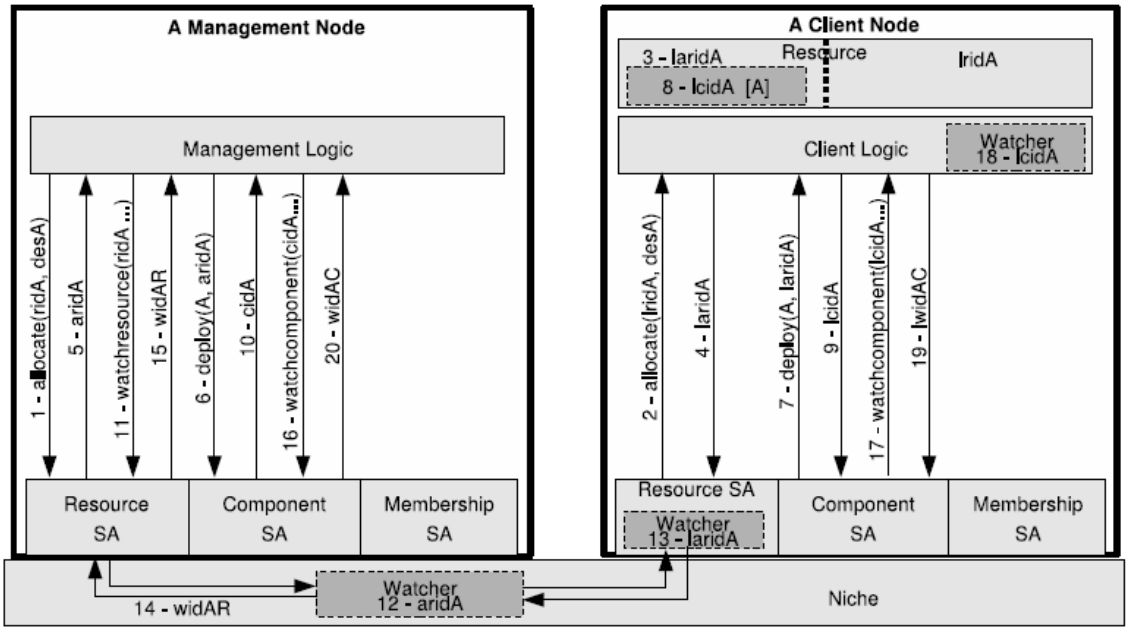
- Self-configuration (moving components in the face of resource joins and leaves)
- Self-healing (maintaining the number of Bs, checkpointing and restarting As, maintain bindings
- Self-tuning (adjust number of Bs according to component load – e.g. size of request queue)
- Self-protection (moving because of attacks (e.g. DDOS)

Note that many of the operations can fail but this is not shown in the use case (to deal with you would have to retry certain things).

10.5.1 Initial Deployment

A section of the initial deployment is shown below. These sections of code may wholly or partially be produced automatically with appropriate tool taking the ADL-description as input parameter. For the sake of the simplicity error handling is omitted.

```
ridA:oneShotDiscoverResource(RequirementsA, PreferencesA, 0)
% + the same for 3 Bs
desA := specifications(preferenceA, ridA)
% specifications is a function that produces an exact description of how much of the discovered
% resource is to be allocated
aridA:allocate(ridA, desA)
cidA: deploy(A,aridA)
% + similarly for 3 B components
gid:group([cidB1,cidB2,cidB3])
bid:bind(cidA,gid, BDesA,BDesB, random)
% random indicates one type of one-to-some binding
% We are now finished with deployment and will proceed with subscriptions and activation of event handlers
% to do the self-*. Some parameters are left out, when names are self-explanatory
widAR:watchResource(ridA, [used->fail, used->leaving])
activateEventHandler(self-configuration-leave, widAR, [bid, cidA, aridA, gid])
widA:discoverResource(requirementsA, preferencesA, thresholdA, 0, ridA)
activateEventHandler(self-configuration-join, widA, [bid, cidA, aridA, gid]
% periodic check-pointing needed for the rule providing self-healing for A
id:createUniqueId()
timeGenerate(checkPointing(id), timeInterval)
activateEventHandler(checkpointing, id)
activateEventHandler(self-healing, widAR, […])
for i:=1 to 3 do load[i]:=0
noB:=3
activateEventHandler(self-tuning, widB[1], […])
activateEventHandler(self-tuning, widB[2], […])
activateEventHandler(self-tuning, widB[3], […])
activateEventHandler(self-protection, widA, [ …])
```



10.5.2 Self-Configuration Rules

For simplicity in all the following rules the *waits* are implicit (futures as input parameters block the call until instantiated). Note that the rules are active until stopped so they may be fired many times. Once again we omit error-handling within rules.

self-configuration-join: % we have a better fit resource for A

upon event discoveryReport(wid, rid, RDes) **with** <bid, cidA, aridA, gId, PreferencesA> **do**

boolF: unbind(bid)
pcId: passivate(cidA)
boolF:deallocate(aridA)
aridA: allocate(rid, specification(PreferencesA, RDes))
cidA:deploy(pcId, aridA)
bid:bind(cidA, gId)

self-configuration-leaves

upon event resourceStateChange(widA, from, to) **with** <bid, cidA, aridA, gId > **do**

if from==used && to==leaving **then**

newRidA:discoverResource(RequirementsA, PreferencesA, 0)
% need to call discovery as up till now we have only subscribed
% to better resources than the one we currently are using
boolF:unbind(bid)
pcId:passivate(cidA)
boolF:deallocate(aridA)
aridA:allocate(newRidA, specification(PreferencesA, newRidA))
cidA:deploy(pcId, aridA)
bid:bind(cidA,gId)

In the above self-configuration rules that upon beginning the event handling that the attribute variable **arid** was instantiated to resource that component A is currently residing on. This attribute variable is used to

deallocate, but later on this attributed variable is set to the newly allocated resource. Note that arid and the other attribute variables have been appropriately reset at rule termination so that the rule can be fired again.

10.5.3 Self-healing

Checkpointing

checkId:takeAndStoreCheckpoint(cidA)

Self-healing

upon event resourceStateChange(widA, From,To) **with** … **do**
if from==used && to==failed **then**
newRidA:discoverResource(RequirementsA, PreferencesA, 0)
% the handler needs to call discovery, since the existing watcher only reports
% resources that are better than the failed one
aridA:allocate(newRidA, specification(PreferencesA,))
cidA:deploy(checkId,aridA)
bid:bind(cidA,gId)

10.5.4 Self-tuning

Self-tuning

upon event componentReport(widB[X], load, oldLoad, newLoad) **with** <gid ...> **do**
if loadMeasure(load[1..NoB])> HighLimit then % load is high we need another B
(newRidB, description):= discoverResource(requirementsB, preferencesB, 0)
newAridB:allocate(newRidB, specifications(preferencesB, description)
newCBid:deploy(B, newAridB)
addToGroup(newCBid, gid)
noB:=noB+1
update(load[])
elseif loadMeasure(load[1..NoB])<LowLimit then
if noB > 3 then
deallocate(lowestLoad(widB[]))
noB:=noB-1
update(load[])

10.5.5 Self-protection

Context: VO demands that users open port P. Niche messages are checked for authorization (e.g. shared key).

Self-protection
setup
widX:watchResource(rid, unauthorized, threshold)
event
upon event resourceReport(widX, unauthorized) with <…> do
<analogous to moving Bs and As because of leaving>

11 References

[ADZ00] M. Aron, P. Druschel, W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In : SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. New York, NY, USA : ACM Press, 2000. p. 90-101.

[AEB03] Alima, L.O.,El-Ansary, S., Brand, P., Haridi, S.: DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications, In The 3rd Int workshop CCGRID2003, Tokyo, Japan, (2003)

[Alf+04] R. Alfierif et al. “VOMS, an Authorization System for Virtual Organizations”. Springer Lecture Notes in Computer Science, Volume 2970, 2004

[ALR01] A. Avizienis, J.-C. Laprie, B. Randell. Fundamental Concepts of Dependability. Research Report N01145, LAAS-CNRS, avril 2001.

[ATIS00] Definition of Event, ATIS Telecom Glossary 2000
http://www.atis.org/tg2k/_event.html

[Bak+05] M Baker et al. Emerging Grid Standards. Computer 38, 4 (Apr. 2005), 43-50.

[Ban06] T. Banks. “Web Services Resource Framework (WSRF) – Primer v1.2, Committee Draft 02”. OASIS, May 2006
http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf

[BBH05] S. Bouchenak, F. Boyer, D. Hagimont et al. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05). 2005. p. 13-24.

[BBH05] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. de Palma, V. Quema and J-B Stefani. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005), Orlando, FL, USA, October 2005.

[BCL06] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani. J.B., “The Fractal Component Model and its Support in Java”. Software - Practice and Experience (SP&E), 36(11-12):1257-1284, September 2006. special issue on “Experiences with Auto-adaptive and Reconfigurable Systems”

[BCM03] Baude, F., Caromel, C., and Morel, M., “From distributed objects to hierarchical grid components”. In International Symposium on Distributed Objects and Applications (DOA), Catania, Italy, volume 2888 of LNCS, pages 1226 – 1242. Springer, 2003

[BMW+06] Vaughn Bullard, Bryan Murray, Kirk Wilson, editors. “An Introduction to WSDM”, OASIS Committee Draft, February 2006
http://www.oasis-open.org/committees/download.php/16998/wsdm-1.0-intro-primer-cd-01.doc

[BOINC] Berkeley Open Infrastructure for Network Computing Site
http://boinc.berkeley.edu/

[BP03] A. Brown, D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. USENIX Annual Technical Conference, San Antonio, TX, USA, June 2003.

[BPK05] S. Bouchenak, N. De Palma, S. Krakowiak. Tolérance aux Fautes dans les Grappes d'Applications Internet. In 4ème Conférence Française sur les Systèmes d'Exploitation (CFSE 2005), Le Croisic, France, April 2005.

[Bra+07] Per Brand et al. “The Role of Overlay Services in a Self-Managing Framework for Dynamic Virtual Organizations“. Paper presented at CoreGRID Workshop, Greece June 2007 and CoreGRID technical report, TR-0094.

[BS07] Berkeley and Stanford Universities. The Berkeley/Stanford Recovery-Oriented (ROC) Project. http://roc.cs.berkeley.edu/

[BSHL01] Ben-Shaul, I., Holder, O., and Lavva, B. “Dynamic adaptation and deployment of distributed components in Hadas”. IEEE Transactions on Software Engineering, 27(9):769-787, 2001.

[Cal+04] Diego Calvanese et al. “Hyper: A Framework for Peer-to-Peer Data Integration on Grids”. In proceedings of first International IFIP Conference on Semantics of a Networked World, 2004 http://www.dis.uniroma1.it/pub/calvanes/calv-etal-ICSNW-2004.pdf

[Cap+05] Franck Cappello et al. “Computing on large-scale distributed systems: Xtrem Web architecture, programming models, security, tests and convergence with grid”. Future Generation Computer Systems. Volume 21, Issue 3, March 2005 http://www.lri.fr/~fedak/thesis/FGCS-2004.pdf

[CBC01] Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N., "An Efficient Component Model for the Construction of Adaptive Middleware", Proceedings of the IFIP / ACM International Conference on Distributed Systems Platforms (Middleware'2001), LNCS 2218, Heidelberg, Germany, November 2001, pp. 160.

[CBC02] Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N., "The Design of a Highly Configurable and Reconfigurable Middleware Platform", ACM Distributed Computing Journal, Vol 15, No 2, April 2002, pp 109-126.

[CBG04] Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., and Ueyama, J., "A Component Model for Building Systems Software," in Proceedings of IASTED Software Engineering and Applications (SEA'04), Cambridge MA, USA, November 2004.

[CBL04] A.J. Chakravarti, G. Baumgartner, M. Lauria. “The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network”. In proceedings of the International Conference on Autonomic Computing, May 2004
http://citeseer.ist.psu.edu/chakravarti04organic.html

[CCA04] CCA Forum Home Page. The Common Component Architecture Forum, 2004
http://www.cca-forum.org.

[CCC05] M. Costa, J. Crowsoft, M. Castro, A. Rowstron, L. Zhou, L. Zhang and P. Barham. Vigilante: end-to-end containment of Internet worms. In : Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP'05). 2005

[CCR05] Castro M., Costa M., and Rowstron A., "Debunking some myths about structured and unstructured overlays", NSDI'05, Boston, MA, USA, May 2005.

[CF05] Raphaël Chand, Pascal Felber. “Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks”. In proceedings of Euro-Par 2005
http://gyroweb.inria.fr/~viennot/enseignement/master/stages/EuropPar-2005.pdf

[CGB06] Coulson, G., Grace, P., Blair, G., Cai, W., Cooper, C., Duce, D., Mathy, L., Yeung, W.K., Porter, B., Sagar, M., and Li., W., "A component-based middleware framework for configurable and reconfigurable Grid computing", Concurrency and Computation: Practice and Experience, 18, pp. 865-874, 2006

[Chi+04] Paul Alexandru Chirita et al. “Publish/subscribe for rdf-based p2p networks”. In Proceedings of the 1st European Semantic Web Symposium, 2004
http://citeseer.ist.psu.edu/724915.html

[CKF04] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox. Microreboot – A Technique for Cheap Recovery. 6th Symposium on Operating Systems Design and Implementation (OSDI’04), San Francisco, CA, USA, December 2004.

[Cli+06] Kevin Cline, at el. Toward Converging Web Service Standards for Resources, Events, and Management. A joint white paper from Hewlett Packard Corporation, IBM Corporation, Intel Corporation, and Microsoft Corporation
http://msdn2.microsoft.com/en-us/library/aa480724.aspx

[CONV06] Toward Converging Web Service Standards for Resources, Events, and Management". A joint White Paper from Hewlett Packard, IBM, Intel and Microsoft Corporation http://xml.coverpages.org/ConvergingWS-ResourcesEventsManagement20060315.pdf

[Cor+05] Jason Cornpropst et al. "Proposal for a CIM mapping to WSDM". HP Developers Resource Central, January 2005

[Cov+06] Robin Cover, editor. "Standards for Automated Resource Management in the Computing Environment". Technology Report from Cover Pages Website, August 2006. http://xml.coverpages.org/computingResourceManagement.html

[CSW05] D.M. Chess, A. Segal, I. Whalley, S.R. White, Unity: experiences with a prototype autonomic computing system, Procs of Autonomic Computing, 2004. Date: 17-18 May 2004, page(s): 140-147

[DDW99] H. Debar, M. Dacier, and A. Wespi. Toward a taxonomy of intrusion-detection systems. In : Computer Networks. 31(9):805-822, 1999.

[DEBS3-6] International Workshop on Distributed Event-Based Systems. List of papers, 2003-2006 http://www.eecg.toronto.edu/debs03/dp.html http://serl.cs.colorado.edu/%7Ecarzanig/debs04/papers.html http://www.cs.queensu.ca/~dingel/debs05/papers.htm http://www.cs.waikato.ac.nz/~hinze/DEBS2006/papers.html

[DERMI] Dermi: Decentralized Event Remote Method Invocation. Project Website http://planet.urv.es/DERMI/

[DMTF] DMTF: Distributed Management Task Force, Inc. Website http://www.dmtf.org/

[DS06] Alistair Dunlop, Achim Streit. "Interoperability and Usability of Grid Infrastructures", OMII-Europe presentation, 2006 http://www.gridforumkorea.org/workshop/2006/w_data/12-1/OMIIEurope.pdf

[DZD03] Dabek F., Zhao B., Druschel P., Kubiatowicz J., and Stoical., "Towards a common API for structured peer-to-peer overlays". In Proc. IPTPS 2003, Berkeley, California, 2003.

[EGEE] EGEE, Enabling Grids for E-sciencE, Website http://www.eu-egee.org/

[Eug+03] Patrick Th. Eugster et al. "The Many Faces of Publish/Subscribe", ACM Computing Surveys, Volume 35, Number 2, June 2003 http://www.irisa.fr/paris/Biblio/Papers/Kermarrec/EugFelGueKer03ACMSurvey.pdf

[FHS96] S. Forrest, S. A. Hofmeyr, A. Somayaji et al. A Sense of Self for Unix Processes. In : Proceedings of the 1996 IEEE Symposium on Security and Privacy. Washington, DC, USA : IEEE Computer Society, 1996. p. 120-128.

[FHS97] S. Forrest, S. A. Hofmeyr, A. Somayaji. Computer Immunology. Communications of the ACM, 1997, Volume 40, No. 10, p. 88-96.

[FKT02] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". International Journal of Supercomputing Applications, Volume 15, no. 3, 2002 http://citeseer.ist.psu.edu/foster01anatomy.html

[Fos02] Ian Foster. "What is the Grid? A Three Point Checklist", GRIDtoday, Volume 1, Issue 6, July 2002 http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf

[Fos05] Ian Foster. "A Globus Primer. Or, Everything You Wanted to Know about Globus, but Were Afraid To Ask". Draft version 0.6, Globus Website, August 2005. http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf

[Fos06] I. Foster, Globus Toolkit Version 4: Software for Service-Oriented Systems, IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2006.

[Fri+03] Thomas Friese et al. “A Framework for Resource Management in Peer-to-Peer Networks”. Springer Lecture Notes in Computer Science, Volume 2591, 2003 http://www.old.netobjectdays.org/pdf/02/papers/node/0017.pdf

[GC03] A. G. Ganek, T. A. Corbi. The dawning of the autonomic computing era. IBM Systems Journal, 2003, Volume 42, No. 1 (jan.), p. 5-18.

[GCM07] Basic Features of the Grid Component Model, CoreGRID Deliverable, D.PM.04, March 2007

[GCS03] D. Garlan, S.-W. Cheng, B. Schmerl. Increasing System Dependability through Architecture-based Self-repair. In : Architecting Dependable Systems. de Lemos, Gacek, Romanovsky (Ed.), Springer-Verlag, 2003.

[gLite] gLite Website, http://glite.web.cern.ch/glite/

[GPF05] A. Goel, K. Po, K. Farhadi, Z. Li and E. De Lara. The Taser intrusion recovery system In : Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP'05),2005.

[GPS02] The Gnutella Protocol Specification, http://www.the-gdf.org/

[GT] Globus Toolkit Website, http://www.globus.org/toolkit/

[GTDOC] “Key Concepts for Common Runtime Components”, draft of Globus Toolkit 4.2 documentation http://www.globus.org/toolkit/docs/development/4.2-drafts/common/key/index.html

[Gup+04] Abhishek Gupta et al. “Meghdoot: Content-Based Publish/Subscribe over P2P Networks”. In proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, 2004 http://ikdesk4.inf.ethz.ch/ws06/Meghdoot_-_%20content-based%20publish-subscribe%20over%20P2P%20networks.pdf

[Han05] Jeff Hanson. “Event-driven services in SOA”, Java World Website, January 2005 http://www.javaworld.com/javaworld/jw-01-2005/jw-0131-soa.html?page=1

[Har+03] Nicholas J.A. Harvey et al. “SkipNet: A Scalable Overlay Network with Practical Locality Properties”. In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems, March 2003 http://research.microsoft.com/sn/Herald/papers/USITS/SkipNet-Usits.pdf

[Hog07] Joel Höglund, “Enabling Dynamic Virtual Organizations, Concepts, standards and challenges”, Master of Science Thesis, Royal Institute of Technology, ICT/ECS 2007-72.

[HS02] Y. Huang and A. Sood. Self-cleansing systems for intrusion containment In : Workshop on Self-Healing, Adaptative and self-MANaged systems (SHAMAN),2002.

[Hum+04] Marty Humphrey et al. “An Early Evaluation of WSRF and WS-Notification via WSRF.NET”. In proceedings of Fifth IEEE/ACM International Workshop on Grid Computing, 2004 http://citeseer.ist.psu.edu/655755.html

[Hum+05] Marty Humphrey et al. “State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations”. In proceedings of 14th IEEE International Symposium on High Performance Distributed Computing, July 2005 http://www.cs.virginia.edu/~humphrey/papers/WSRFComparison2005.pdf

[JPA02] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso. An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness. 21st IEEE Int. Conf. on Reliable Distributed Systems (SRDS 2002), pp. 150-159. Oct. 2002, Osaka, Japan.

[JSDL] Job Submission Description Language (JSDL) Specification, Version 1.0. Global Grid Forum (December 2005) http://www.gridforum.org/documents/GFD.56.pdf

[JXTA] JXTA-SOAP Project Site, http://soap.jxta.org/

[KC03] J. O. Kephart, D. M. Chess. The Vision of Autonomic Computing. IEEE Computer, 2003, Volume 36, No. 1 (jan.), p. 41-50.

[KPG03] G. Kaiser, J. Parekh, P. Gross, G. Valetto. Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. Autonomic Computing Workshop - 5th Annual International Workshop on Active Middleware Services (AMS 2003), Seattle, WA, USA, June 2003.

[Lau+06] E Laure, et al., Programming the Grid with gLite, The EGEE technical report EGEE-TR-2006-001, Geneva : CERN, 22 Mar 2006 . - 18 p. http://cdsweb.cern.ch/search?p=EGEE-TR-2006-001

[LIK05] Erietta Liarou, Stratos Idreos, Manolis Koubarakis. “Publish/Subscribe with RDF Data over Large Structured Overlay Networks”. In proceedings of the 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing, August 2005 http://www.intelligence.tuc.gr/~erietta/dbisp2p_05_Liarou.pdf

[Lor+04] Markus Lorch, editor. “Conceptual Grid Authorization Framework and Classification”, Global Grid Forum, GFD-I.038, November 2004. http://www.gridforum.org/documents/GFD.38.pdf

[Mac+05] F. Buchholz Maciel, editor. “Resource Management in OGSA”, Global Grid Forum, GFD-I.045, March 2005 http://www.ggf.org/documents/GFD.45.pdf

[MCC04] Matthew L. Massie, Brent N. Chun, David E. Culler. “The ganglia distributed monitoring system: design, implementation, and experience”. Parallel Computing, Volume 30, Issue 7, July 2004 http://ganglia.info/papers/science.pdf

[MD04] Mislove A. and Druschel P., "Providing Administrative Control and Autonomy in Structured Peer-to-Peer Overlays", International Workshop on Peer-to-Peer Systems (IPTPS), LNCS, Volume 3, 2004.

[Mic06] Brenda Michelson, “Event-Driven Architecture Overview”, Web Page February 2006 http://elementallinks.typepad.com/bmichelson/2006/02/eventdriven_arc.html

[MJ05] Vinod Muthusamy, Hans-Arno Jacobsen. “Small-Scale Peer-to-Peer Publish/Subscribe”. In proceedings of the Second International Workshop on Peer-to-Peer Knowledge Management, July 2005 http://p2pkm.org/downloads/Muthusamy2005.pdf

[MJP04] J. M. Milan-Franco, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive Distributed Middleware for Database Replication. ACM/IFIP/ USENIX Middleware Conference. Toronto (Canada). Oct. 2004

[MMR+06] Raymond McCollum, Bryan Murray, Brian Reistad, Microsoft, editors. “Web Services for Management (WS-Management)”, Preliminary DMTF release, April 2006 http://www.dmtf.org/standards/published_documents/DSP0226.pdf

[MS04a] Microsoft, COM Component Object Model Technologies, http://www.microsoft.com/com/default.mspx

[MS04b] Microsoft, .Net Home Page, http://www.microsoft.com/net

[NB] The NaradaBrokering Project @ Indiana University. Project Website http://www.naradabrokering.org/

[NCF04] J. Norris, K. Coleman, A. Fox et al. OnCall: Defeating Spikes with a Free-Market Application Cluster. In : Proceedings of the First International Conference on Autonomic Computing (ICAC'04). 2004

[OAP05] D. Oppenheimer, J. Albrecht, D. Patterson, A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. 14th IEEE Symposium on High Performance Distributed Computing (HPDC-14), Research Triangle Park, NC, USA, July 2005.
http://www.swordrd.org/

[OASIS] OASIS Standards and Other Approved Work, http://www.oasis-open.org/specs/

[OGFRM] “SCRM Standards Landscape”. Information from the Open Grid Forum working group Standards Development Organization Collaboration on Networked Resources Management, SourceForge Website
https://forge.gridforum.org/sf/wiki/do/viewPage/projects.scrmwiki/wiki/Landscape

[OGSA] Globus: OGSA - The Open Grid Services Architecture
http://www.globus.org/ogsa/

[OMG02] Object Management Group, CORBA Component Model v3.0, OMG Document formal/2002-06-65.

[PA07] ProActive Website, http://www.inria.fr/oasis/ProActive/

[Pal+03] Shrideep Pallickara et al. “A Framework for Secure End-to-End Delivery of Messages in Publish/Subscribe Systems”. In proceedings of the 7th IEEE/ACM International Conference on Grid Computing, September 2006
http://www.naradabrokering.org/papers/NB-Security.pdf

[Pal+03] Shrideep Pallickara, Geoffrey Fox. “A Scheme for Reliable Delivery of Events in Distributed Middleware Systems”. In proceedings of the IEEE International Conference on Autonomic Computing, May 2004
http://www.naradabrokering.org/papers/ICAC.pdf

[Pal+03] Shrideep Pallickara et al. “A Security Framework for Distributed Brokering Systems”, Indiana University Pervasive Technology Lab report, 2003
http://grids.ucs.indiana.edu/ptliupages/publications/NB-SecurityFramework_acmcss.pdf

[PCB03] Parlavantzas, N., Coulson, G., Blair, G.S., "An Extensible Binding Framework for Component-Based Middleware", Proceeding of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), Brisbane, Australia, September 16-19, 2003, pp 252-263.

[PGM05] Pairot, C., García, P., Mondéjar, R., and Gómez-Skarmeta. A.F., p2pCM: A Structured Peer-to-Peer Grid Component Model. 2nd International Workshop on Active and Programmable Grid Architectures and Components, LNCS, Volume 3516. Atlanta, USA, May 2005.

[PGM07] Pairot, C., García, P., and Mondéjar, R., Deploying Wide-Area Applications is a SNAP. IEEE Internet Computing Magazine, March/April 2007 (Volume 11, No. 2) pp. 72-79

[PMG07] Parlavantzas, N., Morel, M., Getov, V., Baude, F., Caromel, D., "Performance and Scalability of a Component-Based Grid Application", Proc of 9th Int. Worshop on Java for Parallel and Distributed Computing (in conjunction with the IEEE IPDPS conference), April 2007.

[RD01] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Middleware’01 (Nov. 2001).

[R-GMA] R-GMA: Relational Grid Monitoring Architecture, http://www.r-gma.org/

[Rhe+03] Sean Rhea et al. “Pond: the OceanStore Prototype”. Proceedings of the 2nd USENIX Conference on File and Storage Technologies, 2003
http://oceanstore.cs.berkeley.edu/publications/papers/pdf/fast2003-pond.pdf

[Row+01] Antony I. T. Rowstron et al. "SCRIBE: The design of a large-scale event notification infrastructure". In proceedings of the 3d International Workshop on Networked Group Communication, 2001
http://citeseer.ist.psu.edu/703666.html

[Sab06] R. Sabharwal,Grid Infrastructure Deployment using SmartFrog Technology, International conference on Networking and Services, 2006, page(s): 73-73

[SAGA] Simple API for Grid Apps RG (SAGA-RG), https://forge.gridforum.org/projects/saga-rg/

[SGH07] Tallat M. Shafaat, Ali Ghodsi, Seif Haridi. *Handling Partitioning and Mergers in Structured Overlay Networks, Seventh IEEE International Conference on Peer-to-Peer Computing (P2P'07), September, 2007, Ireland.

[SPS05] S. A. Sousa, J. Pereira, L. Soares, A. Correia Jr., L. Rocha, R. Oliveira, F. Moura. Testing the Dependability and Performance of Group Communication Based Database Replication Protocols IEEE/IFIP. Intl. Conf. Dependable Systems and Networks (DSN), 2005.

[ST06] Birgir Stefansson, Antonios Thodis. “MyriadStore: A Peer-to-Peer Backup System”. Master of Science Thesis, SICS Website, June 2006 http://myriadstore.sics.se/docs/mstorethesis.pdf

[Sto+01] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In SIGCOMM’01 (Aug. 2001).

[Sun06] Sun Microsystems, Enterprise JavaBeans Specification Version 3.0, http://java.sun.com/products/ejb/

[Sun07] Sun Microsystems. Sun N1 Grid Engine 6 http://www.sun.com/software/gridware

[Sun96] A. Sundaram. The An introduction to intrusion detection In : ACM crossroads Student Magazine 2(4):3-7,1996.

[TBB05] C. Taton, S. Bouchenak, F. Boyer, N. De Palma, D. Hagimont, A. Mos.SelfManageable Replicated Servers. Workshop on Design, Implementation, and Deployment ofDatabase Replication, in conjunction with the 31st International Conference on Very Large DataBases (VLDB 2005), Trondheim, Norway, August 2005.

[TZ06] Jia Tang, Minjie Zhang. “An Agent-based Peer-to-Peer Grid Computing Architecture: Convergence of Grid and Peer-to-Peer Computing”. In proceedings of 17th Australasian Database Conference, 2006
http://crpit.com/confpapers/CRPITV54Tang.pdf

[Vou+06] Spyros Voulgaris et al. “Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks”. INRIA technical report, 2006
http://citeseer.ist.psu.edu/voulgaris06subsub.html

[WSDM] Web Services Distributed Management (WSDM) v1.1, http://www.oasis-open.org/specs/index.php#wsdmv1.1

[WSMDS] “GT 4.0 WS MDS: Cluster Monitoring Information and the GLUE Resource Property”, Globus Toolkit documentation
http://www.globus.org/toolkit/docs/4.0/info/key/gluerp.html

[WSN] Web Services Notification (WSN) v1.3, http://www.oasis-open.org/specs/index.php#wsnv1.3

[WSRF] Web Services Resource Framework (WSRF) v1.2, http://www.oasis-open.org/specs/index.php#wsrfv1.2

[WSRFNET] WSRF.NET, Project Site http://www.cs.virginia.edu/~gsw2c/wsrf.net.html

[Zha02] Xuehai Zhang. “A Performance Study of Two Grid Information Systems: Measurement, Analysis and Comparison”. Master thesis, Department of Computer Science, University of Chicago, November 2002
http://people.cs.uchicago.edu/~hai/pubs/msthesis2002.pdf