

# The Gridkit Resource Management Frameworks

Geoff Coulson<sup>1</sup>, Wei Cai<sup>1</sup>, Paul Grace<sup>1</sup>, Gordon Blair<sup>1</sup>, Laurent Mathy<sup>1</sup>, Wai Kit Yeung<sup>1</sup>

<sup>1</sup>Computing Department, Lancaster University, UK

[geoff, w.cai, gracep, gordon, laurent, yeungwk]@comp.lancs.ac.uk

## ABSTRACT

*Traditional resource discovery and management systems in Grid Computing tend to be coarse-grained, have fixed static policies and deal exclusively with concrete resource entities e.g. CPUs, memory bytes. In this paper, we present the resource discovery and resource management frameworks that forms part of our Gridkit middleware. These frameworks are underpinned by an overlay network-based communications infrastructure which allows sophisticated and dynamically changeable resource discovery. In addition, we describe how our resource frameworks manage both coarse-grained and fine-grained resources, and support abstract and pluggable task description to better support end-to-end quality of service.*

## Keywords

Grid middleware; service-oriented architecture; resource discovery; resource management; overlay networks

## 1. Introduction

A Grid execution environment consists of a number of interconnected computational nodes with contrasting capabilities (e.g. different hardware and operating systems). These must simultaneously support multiple distributed applications that must be mapped onto subsets of these nodes in such a way that their execution constraints are met. Furthermore, the resources required by each application vary over the applications' lifetime, as does the resource availability on each node. It is the function of Grid resource management middleware to appropriately map applications to nodes according to their resource requirements, and to dynamically manage the resourcing of applications as they execute.

The major player in Grid resource management is Globus [Globus,03], which employs a multi-level resource management architecture consisting of a 'broker' (which maps high-level resource requirements, expressed in a language called RSL, to more concrete requirements; and also finds appropriate computational nodes on which to execute the application) and a 'co-allocator' (which initiates the execution of appropriate parts of the application on appropriate nodes). While this scheme is in successful use, it has a number of

limitations. In particular, it is coarse-grained in the sense that resource specifications tend to deal with whole machines (or at best processes) rather than with fine-grained resources such as threads, buffer pools, connections etc. In addition, it lacks any support for run-time adaptation—the resources allocated at application launch-time cannot be adjusted during runtime.

Other proposals attempt to alleviate these limitations. For example, in ERDoS [Chatterjee,99] a per-node manager (referred to as a 'resource agent') monitors resource usage and reports exceptions back to a global 'system manager'. The latter can make an appropriate decision on how and what to adapt (resulting in a corresponding command being sent to the resource agent) according to the execution of a so-called 'benefit function'. While this approach goes further than Globus, it has the following drawbacks: the resource agent is per-node rather than per-application so it is difficult to reconcile the conflicting needs of multiple applications; the benefit functions installed in the system are relatively static (they can be changed, but only by restarting the system); and there is no local autonomy at the resource agent level—all decisions must be made centrally (this is also true of GRMS [Huang,98]).

This latter point is an example of a wider tendency in existing systems: they tend to be static and to prescribe fixed policies and mechanisms. For example, Globus has only a single mechanism for discovering computational nodes, which involves looking up entries in a static database. ERDoS does better than this in supporting on-the-fly querying of a fixed set of available hosts (thus obtaining more up to date information); but there is no scope to employ other possible mechanisms such as peer-to-peer resource discovery [Pallickara,03].

A final point is that existing systems don't offer any consistent notion of an *abstract* resource: they deal exclusively with concrete entities such as CPUs, memory bytes etc. This makes it difficult to map from application-centric notions of resource (e.g. "I need 3 matrix containers of type X, 1 buffer pool of type Y, a scheduler for EDF threads, and a Java virtual machine") to the notion of 'resource' that the system understands. It should ideally be possible to deal with such abstract notions of resource (e.g. in terms of allocating, releasing and adjusting their allocation) in a

consistent manner.

In this paper we present our resource management design, which plays a fundamental role in an overarching Grid middleware platform known as Gridkit. A key aspect of Gridkit is its utilisation of *plug-in overlay networks* to underpin its communication mechanisms. Overlay networks [El-Sayed,03] are virtual networks that ‘overlay’ the physical network infrastructure with some value added functionality or semantic (which can, in this case, support particular modes of resource discovery). For example, a distributed hash-table [Rowstron,01] can be used to underpin peer-to-peer resource discovery. We believe that providing a plug-in framework for overlays is a powerful and general way of supporting a rich set of discovery types to complement resource management, rather than singular, static discovery policies.

The paper particularly focuses on the architecture of our application-focused resource management framework, which notably integrates global coarse-grained resource management with the management of local end-system fine-grained resources (e.g. threads, buffers etc.) to better support end-to-end quality of service (QoS) requirements.

## 2. Background on the Gridkit Middleware

The Gridkit middleware is intended to provide highly configurable support for complex Grid applications involving sensor network infrastructure, mobility and collaborative visualisation. Examples are forest fire management and environmental informatics systems [CSWM,04]. As illustrated in figure 1, the aim of Gridkit is to provide support in each of four ‘domains’ that we identify as key in underlying the provision of the Grid services that underpin such applications. These are as follows:

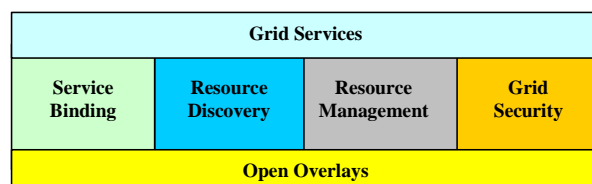


Figure 1. The Scope of Gridkit

- *Service binding.* This area provides sophisticated communication services beyond SOAP: i.e., support for QoS management, and for different interaction types such as publish-subscribe, multicast, streaming etc. This aspect of Gridkit is already discussed in the literature [Coulson,04].

- *Resource discovery.* This provides service, and more generally, resource, discovery services, allowing for the use of multiple discovery technologies to maximise the flexibility available to applications. Examples of alternative technologies are SLP or UPnP for more traditional service discovery, GRAM for CPU discovery in a Grid context, and peer-to-peer protocols for more general resource discovery.
- *Resource management.* This comprises both coarse-grained distributed resource management as currently provided by services such as GRAM, and fine-grained local resource management (e.g. of channels, threads, buffers etc) that is required to build end-to-end QoS.
- *Grid security.* This supports secure communication between participating nodes orthogonally to the interaction types in use.

In this paper we focus on the resource discovery and resource management domains.

Gridkit follows the OpenORB philosophy of adaptive systems development [Blair,01]; i.e. it is a marriage of *i) components*, *ii) reflection* and *iii) component frameworks*. Components are used as the building blocks of the architecture (and applications); reflection then allows inspection and dynamic reconfiguration of the system and application; and finally, component frameworks (hereafter, CFs) provide coarser-grained structure, support ‘plug-in’ configurability, and maintain architectural integrity. The four domains of middleware functionality shown in figure 1 are implemented in Gridkit as independent, horizontal, CFs each of which is highly configurable and reconfigurable. As such, they are directly available to application services, and can also be combined to provide more complex middleware capabilities. In particular, as shown in this paper, the resource management and discovery frameworks can be combined to provide a complete implementation of a resource management system.

A general view of the CF model is shown in figure 2. Here, a CF is an OpenCOM component [Clarke,01] that maintains an internal architecture (i.e. a representation of the topology of its internal components or CFs—components in OpenCOM are potentially composite). A meta-architecture interface (*ICFMetaArchitecture*) provides operations to inspect and reconfigure the component configuration. Finally, different validation strategies can be plugged into the *IAccept* receptacle (a ‘receptacle’ is our term for what is often called a *required* interface) so that when a change is made to the CF’s structure, the plug-in checking strategy is executed, and if invalid the CF rolls back to its previous state. By default, the internal

sub-component topology is checked against a set of XML-based architectural descriptions of valid component configurations.

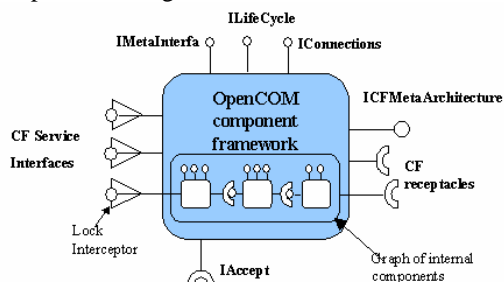


Figure 2. The Component Framework Model

Underpinning the four domains is the Overlay CF (see figure 3), the role of which is to uniformly abstract network-level communication support provided by various overlay network configurations. The benefit of this is that it allows us to treat the diversity of communications support mechanisms in a consistent manner whether or not the underlying physical network supports the required mechanism (the special case of a standard IP network is handled by viewing it as a ‘null’ overlay). We have previously identified how sophisticated interaction and discovery mechanisms (e.g. content distribution, peer-to-peer resource discovery and reliable multicast) can ideally be supported by configurable and reconfigurable overlay instantiations [Grace,04].

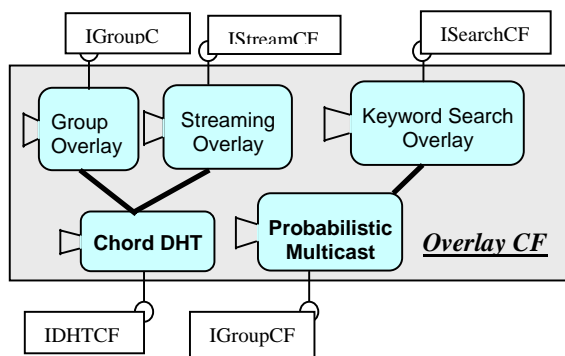


Figure 3. The Overlay Framework Architecture

Figure 3 illustrates the general architecture of the Overlay CF, which allows for multiple overlays to be configured simultaneously, and for there to be dependencies between different overlays. For example, two overlays are shown depending on the Chord [Stoica,01] DHT CF, whereas the keyword search overlay operates atop a completely separate overlay. Each overlay in the framework exposes its interface to the outside, allowing it to be used by higher-level frameworks e.g. the resource discovery framework. A key feature of the Overlay CF architecture is that it

does not enforce a fixed layered architecture; overlays can depend upon one another in arbitrary combinations. However, we must clearly ensure that configurations are *sensible*. The overlay CF achieves this by maintaining a set of architectural rules defined in XML, which describe sensible configurations. In addition, the CF adopts common interfaces that can be used by multiple plug in components. For example, the *IGroupCF* interface can be offered by a range of alternative underlying implementations, e.g. featuring either the probabilistic or DHT based multicast components.

### 3. The Resource Management Framework

Our resource management design is realised as a (distributed) CF in which several areas of functionality are ‘pluggable’ as detailed below. At the most abstract level, there are two parts to the framework (see figures 4 and 5 respectively): *i) global resource management* (i.e. coordinating resource management over multiple computational nodes) and *ii) local resource management* (i.e. managing resource allocation and usage in individual computational nodes). In addition, our framework operates over two distinct phases: *i) an initial resource allocation phase*, and *ii) a subsequent run-time resource management phase* that comprehends dynamic reconfiguration of resources in response to evolving application requirements, and in response to fluctuating resource availability in the infrastructure.

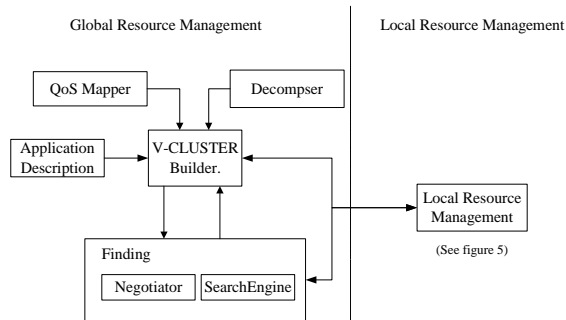
The framework first requires that applications are structured in terms of a set of potentially-distributed components. These components are written according to the above-mentioned OpenCOM component model. An application description which can be submitted to the resource management framework for execution consists of the following:

- a set of top-level<sup>1</sup> components that comprise the application;
- a set of associations, or *bindings*, between interfaces and receptacles of these components that capture the abstract topology of the application;
- a set of so-called *tasks* (see below) which, among other things, express the required QoS of different parts of the application;
- a *mapping* of tasks to components.

The set of top-level components and bindings

<sup>1</sup> As previously mentioned, components in OpenCOM are potentially *composite*—i.e. they can be (recursively) composed of sub-components. In this paper, for the sake of simplicity, we only consider the special case of non-composite components. Essentially, composite components are dealt with by applying the resource framework recursively.

together comprise the compositional structure of the application. The bindings are annotated with QoS requirements; for example, annotations could be taken from the following simple QoS ontology: {*high-speed connectivity*, *medium-speed connectivity*, *low-speed connectivity*}. These annotations are used later when the various components of the application are mapped to physical computational nodes. As explained below, the QoS ontology used is a pluggable element of the framework.



**Figure 4. The Resource Management Framework**

Of course, the above does not yet address the QoS/resource requirements of the components themselves. This is done using the notion of *tasks*. Tasks are application-specific logically-separable units of work (e.g. “media transcoding”, “matrix multiplication”, “pipeline processing”) to which it is meaningful to attach application-level QoS specifications; they also function as the unit of computation to which resources are allocated and resource usage is charged. Importantly, the structure of the set of tasks that comprise an application is *orthogonal* to the structure of the application in terms of components (unlike the case of ‘units of work’ in ERDoS [Chatterjee,99]). That is, in some cases a single task may span a set of (cooperating) components, and in other cases a single component may host multiple tasks. Where tasks span multiple components separate QoS annotations are provided for each per-component ‘sub-task’ in case the respective components eventually execute on separate physical nodes (see below). Although task structure is orthogonal to application structure we can associate the two structures simply by listing, for each component, the set of tasks that the component participates in. This is the ‘mapping of tasks to components’ mentioned above. This mapping eventually yields (see below) the aggregate resource requirement of each component.

The ultimate goal of the framework is to appropriately place the application’s constituent components on some specific set of physical computational nodes. It is the framework’s job *i)* to map components to nodes, *ii)* to ensure that each component’s tasks are adequately resourced by its supporting nodes, and *iii)* to maintain the resourcing of

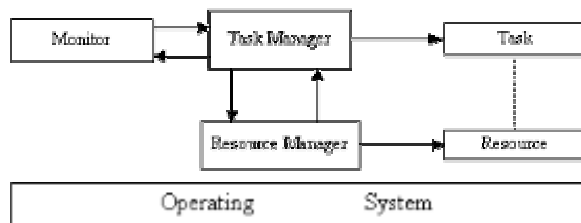
the application at runtime as resource needs and resource provision fluctuate.

The way in which the framework operates is illustrated in figure 4. Initially, the application description (packaged in an XML format) is passed to the framework’s *virtual cluster builder* function. This, in cooperation with the associated *QoS mapper* and *decomposer* functions, analyses the structure of the application into components and bindings, and proposes possible mappings to ‘virtual clusters’ of virtual computational nodes. The QoS mapper is used to map the QoS annotations on the application’s bindings and tasks to resource requirements, whereas the decomposer is used to influence the mapping of application components onto nodes. The output of the virtual cluster builder is a set of candidate ‘virtual cluster mappings’, each of which represents a possible mapping of the application onto a ‘virtual cluster’. In turn, each virtual cluster is a template for a possible physical cluster of machines of a certain specification interconnected by links of a certain specification. One of these candidate virtual clusters is subsequently mapped to physical clusters/ nodes/ interconnects as explained below. When this happens, the rest of the candidate virtual clusters are discarded; the ‘winning’ virtual cluster is retained throughout the subsequent runtime of the application as it plays a role in runtime adaptation (see below).

In more detail, the QoS mapper function accepts plug-in *QoS mappers* which are defined on a per-application-domain basis. QoS mappers define a ‘QoS ontology’ that is meaningful for their associated domain, together with mappings from the ontology to a corresponding ‘resource ontology’. For example, a domain of media transcoding applications might define QoS parameters such as “throughput in frames per second”, “latency”, and “acceptable frame degradation”, together with mappings from these parameters to a resource ontology that comprehends concepts such as “buffer pool size”, “number of high-priority threads” etc. Resource ontologies are also pluggable, but each is expected to be much more general than the average QoS ontology. Similarly, the decomposer accepts plug-in *decomposition policies* that employ application-domain-specific heuristics to identify ‘good’ decompositions. For example, given the above example of a simple QoS ontology for bindings, a simple decomposition policy might try to map components that are bound with “high-speed connectivity” onto a single physical node (bindings here would be implemented as subroutine calls or local IPC). Similarly, it might map components that are bound with “medium-speed connectivity” onto nodes in a common physical cluster, and components that are bound with “low-speed connectivity” to nodes

arbitrarily located in the Internet. We believe that implementing QoS mapping and decomposition as plug-ins is crucial in providing a sufficiently general and extensible architecture. For example, an application domain for which communication latency was of extreme importance could define a considerably more sophisticated decomposition policy than the above.

Having derived a set of candidate virtual cluster mappings, the *finder* function (see figure 4) is next invoked to locate an appropriate set of physical computational nodes and to negotiate resource allocation on these. The *search engine* function, which is part of the resource discovery CF, is pluggable in terms of the mechanisms it uses for finding nodes: for example, a peer-to-peer search protocol supported by the Overlay CF could be used; or alternatively, one could choose a simpler strategy such as querying a fixed set of available hosts for their current loading, or even querying a static central database of nodes. Having located a set of potentially suitable nodes, the finder's *negotiator* confirms (or otherwise) the suitability of subsets of nodes in supporting the various mappings proposed by the virtual cluster builder. This is done by negotiating with the local resource management function (see below) of each candidate node to determine whether or not it has sufficient resources to underpin one or more of the virtual cluster's virtual nodes (negotiation assumes that the negotiator and the candidate node share a common resource ontology). The association between application components and physical nodes is not necessarily 1:1: several components can be mapped to a single node if the latter has sufficient resource provision.



**Figure 5. Local Resource Management**

Finally, the *local resource management* function (see figure 5) operates on each of the physical nodes that have been identified by the above process. There is one instance of this function for each application running on the node (except, of course, for the local operating system which is shared by all local resource management instances). Our local resource management architecture is an extension of the work reported in [Duran,02]. In outline (see figure 5), a local *task manager* is responsible for allocating resources to each of the supported sub-application's tasks (or sub-

tasks). In addition, there is a *resource manager* which is responsible for allocating and managing individual resource types. The resource manager takes pluggable resource ontologies which determine the set of resources available and also determine how these resources map to physical resources in the local operating system environment.

The local resource management function is additionally responsible for triggering *runtime resource adaptation*. This is done by a generic *monitor* that observes resource consumption and availability, and raises an exception when this falls below acceptable levels (in general, QoS ontologies are expected to support the specification of tolerable operating ranges, and these map to corresponding ranges of acceptable levels of resource). When such an exception is raised, the task manager takes appropriate measures. For example, it may ask the local resource manager to request more resources from the local system; or it may transfer resources from non-critical tasks to critical tasks; or it may communicate with other task managers on the same machine to have them degrade their QoS level, thus releasing some resources; or it may report to its parent virtual cluster, asking the latter to find new computational nodes on which to migrate or re-execute the affected components. To support this flexibility both the local resource management and the virtual cluster are configurable with pluggable adaptation policies along the lines proposed in [Blair,00].

## 4. Evaluation

The implementation of our resource framework is still incomplete and we are therefore not yet in a position to evaluate it empirically. Instead, we offer a functional comparison of our framework with the Globus resource management framework. This serves to evaluate *i)* the extent to which our framework conforms to existing practice, and *ii)* the additional functionality and flexibility offered by the framework.

First, regarding application and QoS/ resource decomposition, Globus uses *application brokers* to refine high-level resource requirements into more detailed low-level requirements. In our framework the corresponding functions are carried out by the QoS mapper and the decomposer. These offer a richer and more flexible set of possibilities than Globus. For example, our plug-in QoS and resource ontologies allow us to support QoS/ resource abstractions that are simultaneously close to the application, and tailorable to an extensible range of application domains. Globus does have the flexibility to change application brokers; but not without restarting the system. In addition, Globus supports only a single language or notation for expressing resource requirements (i.e. RSL) whereas

our framework can accept any number of pluggable notations, including RSL. More fundamentally, our use of the task abstraction allows us to specify and manage QoS/ resources in a much finer grained way than Globus. Globus typically operates in terms of coarse grained units like “CPUs with X Mbytes of RAM” whereas we can precisely specify arbitrarily fine-grained application-defined resources. This both increases the probability that applications will execute as required, and avoids wasting resources through over-allocation.

Second, our framework offers a more flexible approach to locating suitable computational nodes than does Globus. In Globus, as mentioned above, resource brokers query a central database to locate nodes that meet their requirements. In our framework, however, a range of alternative ‘finding’ strategies can be plugged into the resource discovery framework, including the Globus strategy. Again, Globus can be viewed as a special case of our more general architecture.

Third, in terms of local resource management, our design again performs all the functions of GRAM but adds benefits in terms of flexibility (e.g., the possibility of plugging in new resource ontologies) and the fine grained control made possible by the use of the task abstraction.

Finally, our framework integrates runtime resource adaptation which is not supported at all in Globus.

## 5. Conclusions and Future Work

In this paper, we have briefly described two self-contained areas of our research (resource discovery and resource management) that are closely related, and as such support comprehensive resource management as part of a larger Grid Middleware. Our design recognises that resource discovery must be dynamic in nature, a single policy will not maximise resource discovery, nor is it applicable to different environmental contexts. Therefore, the configurable resource discovery framework allows multiple discovery solutions to operate side-by-side, and change over time. Similarly, by underpinning these with overlay networks we are able to suitably scale the approach e.g. from discovering resources that are geographically widespread, to those operating in wireless or sensor networks.

Additionally, we have presented our approach to resource management. This architecture promotes the integration of both fine-grained and coarse-grained resources to fully support end-to-end QoS guarantees. Furthermore, to better support individual applications the framework is reconfigurable to allow application-specific QoS mappers to be plugged into the framework, and hence tailor the resource management

to the application type.

In our future work, we plan to further evaluate the functional and performance properties of the resource frameworks within a number of visualisation applications, particularly in the areas of forest fires and environmental informatics. Furthermore, we plan to explore the self-management and automatic reconfiguration across the domains of Gridkit. This will build on the inherent openness of the (component-based) platform but will require additional CFs that deal with areas such as monitoring, recovery strategy selection, and recovery strategy deployment. We have carried out initial explorations in this area [Blair,02], but resource management and overlay networks will provide a challenging context for these ideas.

## REFERENCES

- [Blair,00] Blair, L., Blair, G.S., Andersen, A., Coulson, G., Sanchez Ganedo, D., “Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform”, IEE Proceedings Software, Vol 147, No 1, pp. 13-21, 2000.
- [Blair,01] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K., “The Design and Implementation of OpenORB v2”, IEEE DS Online, Special Issue on Reflective Middleware, Vol. 2, No. 6, 2001.
- [Blair,02] Blair, G.S., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira R., Parlavantzas, N., “Reflection, Self-Awareness and Self-Healing in OpenORB”, Proc. ACM Sigsoft Workshop on Self-Healing Systems (WOSS’02), Charleston, USA, Nov 2002.
- [Chatterjee,99] Saurav Chatterjee, Bikash Sabata, Michael Brown “Adaptive QoS Support for Distributed, Java-based Application” In Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), St-Malo, France, 1999.
- [Clarke,01] Clark, M., Blair, G.S., Coulson, G., Parlavantzas, N., “An Efficient Component Model for the Construction of Adaptive Middleware”, Proc. IFIP Middleware 2001, Heidelberg, Germany, Nov. 2000.
- [Coulson,04] Coulson, G., Grace, P., Blair, G.S., Mathy, L., Duce, D., Cooper, C., Yeung, W.K., Cai, W., “Towards a Component-based Middleware Framework for Configurable and Reconfigurable Grid Computing”, Proc. Workshop on Emerging Technologies for Next Generation Grid (ETNGRID-2004), Modena, Italy, June 2004.
- [CSWM,04] Centre for Sustainable Water Management. <http://www.swm.lancs.ac.uk/>
- [Duran,02] Duran-Limon, H., Blair G.S., “Reconfiguration of Resources in Middleware”, 7<sup>th</sup> IEEE International Symposium on Object-oriented Real-time Dependable Systems (WORDS 2002), San Diego, CA, January 2002
- [El-Sayad,03] El-Sayed, A., Roca, V., Mathy, L., “A Survey of Proposals for an Alternative Group Communication Service”, IEEE Network, Vol 17, No 1, pp46-51, Jan 03.

[Globus,03] The Globus Project, "Resource Management: The Globus Perspective", GlobusWORLD 2003, presentation at GlobusWORLD 2003, available at <http://www.globus.org/>, 2003.

[Grace,04] Grace, P., Coulson, G., Blair, G., Mathy, L., Yeung, W.K., Cai, W, Duce, D., Cooper, C., "GRIDKIT: Pluggable Overlay Networks for Grid Computing", submitted to Distributed Objects and Applications (DOA'04), June 2004.

[Huang,98] Huang, J., Wang, Y., Cao, F. "On Developing Distributed Middleware Service for QoS- and Criticality-Based Resource Negotiation and Adaptation", The special issue on operating systems and Services, Journal of Real-Time Systems,1998.

[Pallickara,03] Pallickara, S., Fox, G., "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids", Proc. IFIP/ACM/USENIX Middleware 03, Rio de Janeiro, Brazil, April 2003.

[Rowstron,01] Rowstron, A., Druschel, P., "Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems", Proc. IFIP Middleware 2001, Heidelberg, Germany, Nov, 2001.

[Stoica,01] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., Balakarishnan, H., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications", Proc. ACM SIG-COMM, San Diego, 2001.