

The Gridkit Distributed Resource Management Framework

Wei Cai, Geoff Coulson, Paul Grace, Gordon Blair,
Laurent Mathy, Wai-Kit Yeung

Computing Department, Lancaster University, UK
w.cai@comp.lancs.ac.uk

Abstract. Traditionally, distributed resource management/ scheduling systems for the Grid (e.g. Globus/ GRAM/ Condor-G) have tended to deal with *coarse-grained* and *concrete* resource types (e.g. compute nodes and disks), to be *statically configured and non-extensible*, and to be *non-adaptive* at runtime. In this paper, we present a new resource management framework that tries to overcome these limitations. The framework, which is part of our 'Gridkit' middle-ware platform, uniformly accommodates an extensible set of resource types that may be both *fine-grained* (such as threads and TCP/IP connections), and *abstract* (i.e. represent application-level concepts such as matrix containers). In addition, it is *highly configurable and extensible* in terms of pluggable strategies, and supports flexible *runtime adaptation* to fluctuating application demand and resource availability. It thus comprises a *QoS driven* and potentially *autonomic* resource management facility. A key contribution is the notion of *tasks* which enable resource requirements to be expressed orthogonally to the structure of the application. This allows intuitive application-level QoS/ resource specification, highly flexible mappings of applications to available distributed infrastructures, and also facilitates autonomic adaptation.

1. Introduction

The task of a Grid resource management/ scheduling system is to appropriately map a set of distributed applications onto an underlying Grid infrastructure that consists of a diverse set of interconnected computational nodes. This is not an easy task if application demands are to be met without wasting resources. Ideally, applications should share nodes to maximise the exploitation of scarce resources, and the environment should dynamically adapt the mapping and resourcing of applications to reflect fluctuating runtime application demand and resource availability.

Presently, the major player in Grid resource management is Globus/ GRAM [Globus,03], which employs an architecture consisting of 'brokers' and 'co-allocators'. Brokers map high-level resource requirements, expressed in RSL, to concrete requirements. They also locate suitable computational nodes on which to execute the application. Co-allocators then allocate nodes, and initiate the execution of appropriate parts of the application on them.

While this scheme is in successful use, it has a number of limitations. First, it deals only with *coarse-grained* resource specifications (e.g. whole machines or at best processes) rather than fine-grained resources like threads, buffer pools, or TCP/IP

connections. Second, it deals only with *concrete* resource specification and doesn't support abstract resources that are meaningful to particular applications (e.g. matrix containers or EDF schedulers). Third, it is *statically-configured and non-extensible*—i.e., its behaviour in many dimensions can only be changed—if at all—by restarting the system (examples: the strategy used to find computational nodes, the set of resource types understood, the policy used to decompose distributed applications). Finally, it is *non-adaptive*—i.e., the resources allocated at application launch-time cannot be adjusted at runtime.

These limitations are not peculiar to Globus/ GRAM—they are shared, at least in part, by most other distributed resource management proposals (see section 4). In this paper we present a new resource management framework that attempts to address these limitations and thereby improve both application-level flexibility and infrastructure-level resource exploitation. The remainder of the paper is structured as follows. Section 2 provides brief background on the 'Gridkit' middleware platform in which our framework is embedded. Subsequently, section 3 offers a detailed description of the framework itself, section 4 discusses related work, and section 5 concludes.

2. Background on the Gridkit Middleware Platform

The Gridkit middleware platform [Grace,04] is intended to provide flexible high-level support to complex and 'advanced' Grid applications such as forest fire management and environmental informatics systems. As well as traditional Grid functionality, such applications involve elements such as sensor network infrastructure, mobility, and collaborative visualisation.

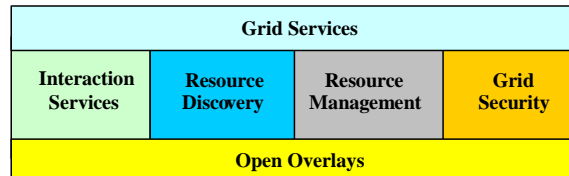


Fig. 1. The Gridkit Architecture.

As illustrated in figure 1, Gridkit places a *Grid Services* layer, which is presented in terms of web services, on top of four orthogonal domains of generic middleware support. These, in turn, are layered on top of an *Open Overlays* layer which abstracts the diversity of underlying communications support mechanisms in a consistent manner whether or not the underlying physical network supports a given communications service (e.g. multicast).

In more detail, the four middleware domains addressed by Gridkit are as follows:

1. *Interaction services.* This domain provides sophisticated and extensible application-layer communication services beyond SOAP: i.e., support for quality of service (QoS) configurable interactions, and for pluggable 'interaction types'

such as publish-subscribe, multicast, streaming etc. More detail on this aspect of Gridkit is available in [Grace,04].

2. *Resource discovery*. This offers generic and extensible resource and service discovery services. It supports the use of multiple discovery technologies to maximise the flexibility available to applications. Examples of supported technologies are SLP or UPnP for traditional service discovery, Globus MDS for CPU discovery, and peer-to-peer protocols for general resource discovery.
3. *Resource management*. This domain, which is the focus of the present paper, provides comprehensive distributed resource management and scheduling support for Grid applications.
4. *Grid security*. This supports secure communication between participating nodes orthogonally to the interaction types in use.

The construction of Gridkit follows the OpenORB architectural approach [Blair,01] which is targeted at building dynamic and adaptive systems software. The general idea is as follows: a lightweight *component technology* called OpenCOM [Clarke,01] provides building-blocks for constructing systems by composition (as advocated, e.g., by Szyperski [Szyperski,98]). *Reflection* then provides means to discover the structure and behaviour of component compositions, and to adapt and extend them at runtime. And, finally, *component frameworks* have the role of imposing structure on component compositions, and ensuring architectural integrity during periods of adaptation. These frameworks are typically structured to accept 'plug-in' components that tailor and extend their functionality. Each of the boxes in Fig. 1 is implemented in this way. As far as we know Gridkit is unique in applying such a component-based approach to both the middleware and the application layer of a Grid environment (most component-based systems, e.g. [Furmento,02], address only applications).

3. The Resource Management Framework

3.1 Overview and Application Model

The distributed resource management design is realised as a component framework in which several areas of functionality are 'pluggable' as detailed below. At the most abstract level (see Fig. 2a), the framework is separated into two parts: *i*) a *global resource manager*, which coordinates resource management over multiple computational nodes, and *ii*) a *local resource manager*, which manages resources in an individual computational node. In addition, it operates over two distinct phases: *i*) an initial *resource co-allocation* phase, and *ii*) a subsequent *run-time resource management* phase that can perform dynamic reconfiguration of resources in response to evolving application requirements and fluctuating resource availability in the infrastructure.

The framework requires that applications are structured and described as a graph of potentially-distributed OpenCOM components. In more detail, the description submit-

ted to the framework by an application deployer (see Fig. 2b) is as follows¹:

1. a set of top-level² OpenCOM components that encapsulate the various pieces of application functionality;
2. a set of QoS-annotated *tasks* (see below) which, among other things, express the required QoS of different areas of the application;
3. a set of QoS-annotated *binding templates* (see below) that represent bindings between the interfaces of the application components, and thus capture the abstract topology of the application as a graph;
4. a component/ task *mapping*—i.e. a list of components associated with each task.

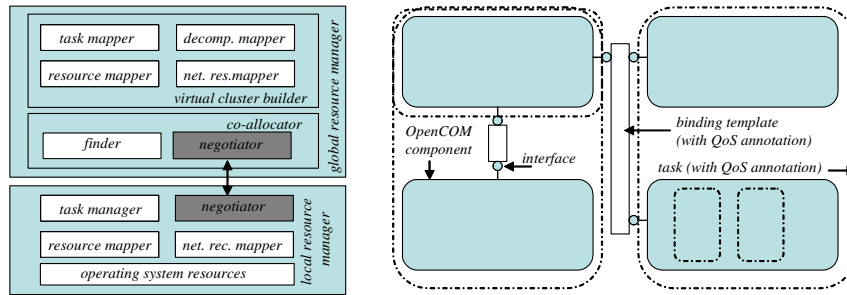


Fig. 2. a) Overall architecture. b) Elements of an application description.

Tasks are abstractions of ‘activities’ or ‘units of work’ which are meaningful at the application level and which can be decorated with application-oriented QoS annotations (the precise form of the QoS annotations is discussed below). Importantly, the definition of the tasks that comprise an application is *orthogonal* to the structure of the application itself in terms of components. Thus, in some cases (see Fig. 2b) a single task may span a set of (cooperating) components, while in others a single component may host multiple independent tasks (also, tasks may overlap, as shown). Examples: *i*) a ‘transcode stream’ task could be realised as a set of components that cooperate to transcode a media stream (e.g. buffering, compressing, encoding etc.); *ii*) multiple instances of an ‘access database’ task could be encapsulated within a single component that deals with concurrent database access.

The orthogonality of tasks and component structure facilitates QoS specification that is meaningful at the application level. Thus, in the ‘transcode stream’ case, the QoS specification is attached to the entire user-visible task rather than to microcosmic aspects such as buffering etc. This orthogonality also offers a useful separation of concerns between writing an application and specifying its QoS. Note that tasks, as well as serving as units of QoS specification, also have a runtime representation that is used in ongoing resource management during application execution. This aspect is

¹ This information is packaged as an XML schema which we do not present here due to space constraints.
² Components in OpenCOM may be *composite*—i.e. (recursively) composed of sub-components. In this paper, for simplicity, we only consider the special case of non-composite components. Essentially, composites are dealt with by applying the resource framework recursively.

discussed in section 3.2.3.

Binding templates serve as abstract placeholders for inter-component communication bindings. A wide and extensible set of binding templates is supported including request-reply, multicast, publish-subscribe, workflow etc (see Fig. 2b). Binding templates are specified in terms of roles and message ordering constraints as described in detail in [Parlavantzas,03] and, like tasks, can be decorated with QoS annotations. Each binding template can be realised by (mapped to) a potentially wide range of concrete technologies (as supported by the interaction services module—see Fig. 1). For example, for components that will share a common node, a request-reply template could be realised as ‘vtables’ or as local IPC links; or, where the components will run on separate nodes, as TCP/IP connections or VME links. The precise mapping is selected by the decomposer as discussed in section 3.2.1.

Rather than define a fixed set of QoS parameters for use in QoS annotations, we support, for reasons of generality and extensibility, an extensible set of *QoS ontologies* that can be defined for specific areas of applicability. Each QoS ontology is a ‘pluggable’ entity that defines a vocabulary for QoS annotations. For example, a QoS ontology suitable for use with the ‘transcode stream’ task mentioned above might offer the following vocabulary: *frame_size, frame_rate: integer*. On the other hand, a simple QoS ontology for a request-reply binding template might offer: *connectivity: {high-speed, medium-speed, low-speed}*. As well as specifying a vocabulary, QoS ontologies are responsible for mapping QoS annotations to underlying pools of resources that are dedicated to individual tasks and bindings as discussed in the next section.

3.2 The Framework in Detail

The goal of the resource management framework is to place the application’s constituent components on some appropriate set of physical computational nodes, and then to manage their ongoing execution. This involves the following steps which are discussed in the three sub-sections below: *i*) mapping components to configurations of virtual nodes called *virtual clusters*, *ii*) finding and negotiating the use of appropriate physical nodes and interconnects with which to underpin virtual clusters, and *iii*) maintaining the QoS of the application at runtime in the face of fluctuating resource needs and resource provision.

3.2.1 Mapping Components to Virtual Clusters

Given an application description as defined in section 3.1, the *virtual cluster builder* (see Fig. 2a) is responsible for generating a list of candidate virtual clusters, each of which represents a possible physical infrastructure that could viably support the application, assuming the latter was decomposed in some appropriate way. Virtual clusters could, for example, represent a set of processes on a machine, CPUs on a VME bus, islands of VME clusters, or machines randomly located in the global Internet.

The first step is that the *task mapper* derives the resource requirements of each of the application's tasks. To do this, it relies on the above-mentioned pluggable QoS ontologies which map QoS specifications expressed in their particular vocabularies to lower-level *resource ontologies*. As an example, the 'transcode stream' QoS ontology discussed in section 3.1 might map to a 'buffer processing' resource ontology with a vocabulary of *buffer_pool_size*, and *number_of_high_priority_threads*. Resource ontologies (which are hosted by the *resource mapper*) work analogously to QoS ontologies, but at a lower level. It is also intended that they be general than a typical QoS ontology's so that they can underpin multiple QoS ontologies. The lower-level entities to which their vocabularies map are either concrete OS-level resources, or the vocabularies of further resource ontologies. So, extending the above example, the 'buffer processing' resource ontology might map to some quantity of memory, and a pool of threads with a given OS-level priority.

Following task and resource mapping, the *decomposition mapper* (with help from the network resource mapper—which is the network counterpart of the above-described resource mapper) generates a list of virtual cluster definitions that could potentially support the required resources according to various possible application decompositions. Rather than prescribe a fixed policy for decomposition, the decomposer accepts plug-in *decomposition heuristics* which are principally guided by three factors:

1. the resource requirements of each task (as derived above);
2. the set of available realisations of the binding templates specified in the application description;
3. the QoS annotations on the binding templates.

As an example, the goal of a particular decomposition heuristic might be to try to employ as few nodes as possible, and to depend on no more than low-speed connectivity between these nodes. This assumes the availability of well-resourced nodes that are each capable of supporting several tasks (it is usually not possible to split a task over more than one node as all of a task's components need to have access to its dedicated resources). The binding template information comes into play when it must be decided how to distribute the various tasks over multiple nodes. Likely 'lines of cleavage' can be identified where the components participating in one task are connected to components in an adjacent task only by a small number of bindings with 'relaxed' QoS requirements. For example, given the illustrative QoS ontology of section 3.1, a simple decomposition heuristic might try to map two adjacent tasks whose inter-task bindings were all "low speed", to two virtual nodes with a minimal interconnect (e.g. the global Internet). On the other hand, tasks whose inter-task bindings were "medium speed" or high speed" might need to be mapped to a common virtual node (or perhaps to two virtual nodes in a tightly-coupled cluster).

Overall, then, the output of the virtual cluster builder is a set of candidate virtual clusters, each of which is a template for a possible physical infrastructure that could adequately support the application. One of these candidate virtual clusters will eventually be concretized to an appropriate infrastructure of physical clusters/ nodes/ interconnects etc. as explained below. When this concretization has been successfully

performed, the rest of the candidate virtual clusters are discarded; however, the ‘winning’ virtual cluster is retained as a first-class entity throughout the subsequent runtime of the application as it plays a role in runtime adaptation (see section 3.2.3).

3.2.2 Finding and Negotiating Resources

Having derived a set of candidate virtual clusters, the *co-allocator* (see Fig. 2) is next invoked to locate appropriate sets of physical computational nodes with which to concretize candidate virtual clusters, and to negotiate the co-allocation of resources on these. The *finder*, which is part of Gridkit’s resource discovery framework (see Fig. 1), is pluggable in terms of the mechanisms it uses for finding nodes: for example, a peer-to-peer search protocol could be used [Pallickara,03]; 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 à la Condor-G [Frey,01]. Having located a set of potentially suitable nodes, the co-allocator’s *negotiator* confirms (or otherwise) the suitability of candidate physical nodes in supporting nodes from the virtual cluster. This is done by negotiating with each candidate’s *local resource manager* (see Fig. 2a). Negotiation assumes that both negotiating parties share common resource ontologies. Both end-system resources and the necessary network resources required to underpin the concrete binding technologies selected by the decomposition mapper are considered in the negotiation.

The local resource management architecture is an extension of the work reported in [Duran,02]. In outline, the *task manager* is responsible for allocating resource pools to each of the application’s tasks that will run on the machine, and for managing tasks at runtime (see below). The size and kind of these resource pools are determined by the resource mappers as discussed above. Separate instances of the task manager are created for each application running on the node.

3.2.3 Managing QoS at Runtime

At runtime, all application requests for resources (e.g. memory allocation, thread creation, or requests for abstract resources such as matrix containers that are understood by the framework) trap to the currently executing (runtime) task—i.e. the task with which the currently executing thread is associated [Duran,02]. Requests are then satisfied from the resource pools that were dedicated to the task when it was first instantiated.

QoS adaptation is triggered when the per-application task manager observes resource consumption or availability in some task falling outside acceptable thresholds that were determined during the QoS/ resource mapping phase. At this point, the task manager consults a plugged-in policy which specifies one or more of the following actions in some specified order:

- transfer resources from other tasks owned by the application;
- request more resources from the local operating system;
- negotiate with other task managers on the same machine (i.e. those associated with other applications) to borrow resources; or
- report to the parent virtual cluster, asking the latter to find new computational

nodes on which to re-execute the affected task.

To support the latter action, the parent virtual cluster is also equipped with a plug-in policy capability to guide its subsequent actions.

A final avenue for QoS adaptation is an application-wide QoS renegotiation requested at the level of the global resource manager. In such cases, a revised application description is submitted and the mapping process is (selectively) restarted to attempt to meet the revised QoS requirements. In the absence of migratable components, this inevitably involves restarting certain application components. A deeper discussion of the fundamental research on which our runtime QoS management approach builds is available in [Blair,00].

3.3 Deployment Issues

The modular and configurable nature of the framework enables several deployment options in networked environments. One is to run the global resource manager on a small number of gateway machines, and to run the local resource manager on a much larger set of machines that are willing to contribute towards executing applications. Another option is to run both the global and local managers on some or all machines. Here, the global managers can interoperate in a peer-to-peer style in which an original application description is decomposed by global managers into sub-descriptions that can be delegated to other global managers (and so on recursively). This can result in faster deployment and also facilitate cooperation between peer virtual organisations. Beyond these two major options refinements are possible which leverage the underlying dynamic loading capabilities offered by OpenCOM [Clarke,01]). For example, one can refine the first option to support just-in-time instantiation of the local manager on nodes as they are discovered by the co-allocator. Or, one can refine the second by dynamically instantiating the global functions on discovered nodes that know of distributed resources in their area but which do not wish to directly contribute themselves.

At a more fine-grained level, it is necessary to consider the deployment of the various plug-ins that are accepted by the framework. We would *not* expect these to be written and installed on a casual basis by end users. Rather they would be the province of Grid infrastructure experts who also have application-domain knowledge. Key plug-ins such as QoS and resource ontologies and decomposition heuristics would be expected to be produced relatively rarely, and be well documented and widely used by application developers. In other words, the role of these plugs-in would primarily be to facilitate the *evolution* of the resource management framework as it finds itself operating in new infrastructure environments and new application areas.

4. Related Work

Apart from the Globus/ GRAM approach discussed in section 1, several researchers have attempted to alleviate the limitations identified in this paper. Condor-G

[Frey,01] has been extensively used in the Globus context and provides a substantial instantiation of Globus/ GRAM. However, Condor-G supports only coarse-grained and concrete resource types, is statically configured and non-extensible, and has serious limitations in terms of adaptation: all it can do is migrate or restart jobs in the case of failures. ERDOS [Chatterjee,99], has similar limitations in terms of resource types and configuration. In terms of adaptation it has a per-machine local ‘resource’ agent which monitors execution and reports exceptions to a global ‘system manager’. However, this has the following drawbacks: *i*) adaptation is coarse-grained in resource managers are per-machine rather than per-application (so that conflicting needs between applications are difficult to reconcile), and *ii*) local resource managers have no autonomy: all adaptation decisions are centrally made. Another feature of ERDOS is its notion of ‘units of work’. These are superficially similar to our tasks; but in fact ERDOS units of work are equivalent to ‘components’: there is no orthogonality between the two concept as there is in our approach. GRMS [Huang,98] is a system with similar benefits and limitations to ERDOS. It is of interest in supporting on-the-fly querying of a fixed set of available hosts, thus obtaining more up to date information than Condor-G. However, there is no scope to configure other possible mechanisms such as peer-to-peer resource discovery.

More recently, [Kumar,03] describes a resource management framework for ‘interactive Grids’. This still suffers from resource type and lack of configurability limitations, but it does have a more sophisticated local resource management scheme that features agents such as resource sensors for monitoring purposes and an enforcement agent for fulfilling QoS specifications. However, the work is limited in not supporting decomposition: each application can only run on a single computational node. A final piece of related work is research on the automatic decomposition of applications into workflows [Cicerre,04]. This takes an approach related to our decomposition mapper, but it lacks any support in the other areas addressed by our work or by the other systems discussed above.

In sum, the state of the art can fairly be characterised as lacking in *i*) fine-grained and abstract notions of QoS specification and resource provision, *ii*) plug-in configurability and extensibility, and *iii*) fine-grained runtime adaptation.

5. Conclusions and Future Work

We have discussed an approach to distributed resource management in the context of the Gridkit middleware platform. Our resource management framework promotes the use of application-tailored abstract QoS specification which drives the allocation and adaptation of fine-grained and application-defined abstract resources. Furthermore, to better support individual applications and deployment environments, the framework is capable of being flexibly configured, extended and run-time reconfigured by means of plug-ins in the following areas: *i*) QoS ontologies, *ii*) resource ontologies, *iii*) application decomposition heuristics, *iv*) resource adaptation policies, and *v*) resource location strategies. In addition, it supports flexible and fine-grained runtime adaptation. And, finally, it supports the task concept which enables highly flexible mappings of applications onto distributed infrastructures, and also facilitates

runtime adaptation. We believe that these features provide a foundation for a future QoS-driven and potentially autonomic resource management facility for the Grid.

We now have an extensive implementation of Gridkit that includes many key aspects of the resource management framework. In particular, we have a mature Open Overlays framework which is populated with the following plug-in overlays: tree-based multicast, DHT-based key routing, and Gnutella-like search. And we have populated the Resource Discovery framework with a set of protocols which can be exploited by the finder as discussed in section 3.2.2 (specifically, we have UDDI, Jini, uPnP, and JXTA). We have also populated the Interaction Services domain with a range of binding templates (together with underlying protocols) including request-reply with QoS, publish-subscribe, OGSA-DAI-based data sharing, and media streaming. In addition, we have a prototype implementation of the task mapper and underlying runtime adaptation functionality, together with simple QoS and resource ontologies, and are currently integrating this into Gridkit proper. We are currently developing a simple decomposition mapper and are exploring decomposition heuristics that assume the following infrastructures: *i*) a single high-speed cluster, and *ii*) a set of islands of clusters.

Future work is planned on two fronts: first we will exercise and evaluate our framework by using it to support existing Grid applications. These include a distributed visualisation scenario from our partners at Oxford Brookes University, and a dynamic computationally-steered chemistry application from the RealityGrid project [Brooke,03]. Second, we plan to explore autonomic self-management in Gridkit. This will build on the inherent openness of the (component-based) platform but will require additional frameworks that deal with areas such as monitoring, recovery strategy deployment, and recovery strategy selection. We have carried out initial explorations in this area and believe that Gridkit provides a highly promising 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.
- [Brooke,03] Brooke, J.M., Coveney, P.V., Harting, J., Jha, S., Pickles, S.M., Pinning R.L., Porter, A.R., "Computational Steering in RealityGrid", Proc. UK e-Science All Hands Meeting, 2003, <http://www.nesc.ac.uk/events/ahm2003/AHMCD>.
- [Chatterjee,99] Chatterjee, S., Sabata, B., Brown, M., "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.

- [Cicerre,04] Cicerre, F., Madeira, E., Buzato, L., "A Hierarchical Process Execution Support for Grid Computing", 2nd Intl Workshop on Middleware for Grid Computing, Toronto, Canada, October 2004.
- [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, November 2001.
- [Duran,02] Duran-Limon, H., Blair G.S., "Reconfiguration of Resources in Middleware", 7th IEEE International Symposium on Object-oriented Real-time Dependable Systems (WORDS 2002), San Diego, CA, January 2002
- [Frey,01] Frey, J., Tanenbaum, T., Livny, M., Foster, I., Tuecke, S., "Condor-G: A Computation Management Agent for Multi-Instructional Grids", Cluster Computing, Vol 5, pp237-246, 2001.
- [Furmento,02] Furmento, N., Mayer, A., McGough, S., Newhouse, S., Field, T., Darlington, J., "ICENI: Optimisation of Component Applications within a Grid Environment", Parallel Computing, Vol 28, No 12, pp1753-1772, 02.
- [Globus,03] The Globus Project, "Resource Management: The Globus Perspective", 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", to appear in Proc. 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", Special issue on Operating Systems and Services, Journal of Real-Time Systems, 1998.
- [Kumar,03] Kumar, R., Talwar, V., Basu, S., "A Resource Management Framework For Interactive Grids", 1st Intl Workshop on Middleware for Grid Computing, Rio de Janeiro, Brazil, June 2003.
- [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.
- [Parlavantzas,03] Parlavantzas, N., Coulson, G., Blair, G.S., "An Extensible Binding Framework for Component-Based Middleware", Proc. Enterprise Distributed Object Computing Conference (EDOC 2003), Brisbane, Australia, Sept. 2003.
- [Szyperski,98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.