A Remote Job Execution Service for Service-Oriented Grids

Neeran Karnik, Girish Chafle, Sunil Chandra
IBM India Research Laboratory, New Delhi, India
Email: {kneeran, cgirish, csunil}@in.ibm.com

Abstract

With the recently proposed integration of Grid and Web services technologies in the *Open Grid Services Architecture* (OGSA), computing resources in a Grid can be shared via a web services interface. The current *Globus* toolkit allows users to submit jobs to remote nodes for execution. The OGSA grid will also need a similar job execution capability – a service that abstracts the underlying compute resources and makes them available to grid users. This paper describes such a *computing service* which allows users to submit jobs for execution and provides the raw computing power. We discuss the architecture and design of the *Remote Job Execution* service, and specify its interfaces. Our design enables the RJE to use existing technologies to create a secure, resource-constrained sandbox for running a client job, potentially with QoS guarantees.

1 Introduction

Grid computing has been defined as flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources – referred to as *virtual organizations* [9]. It takes its name from the notion of linking computers across the Internet into a "grid", analogous to electricity grids for sharing power.

Grid computing has been in use since the mid-1990s as a high performance distributed computing environment for scientific and engineering applications [7]. The Globus Toolkit [6] is a prominent example of the middleware in current use. Grid computing is now being explored by the commercial world, to serve as an e-business infrastructure capable of integrating distributed, heterogeneous resources to deliver the desired qualities of service. Just as the Worldwide Web began as a technology for scientific collaboration and was adopted for e-business applications, Grid technologies are expected to follow a similar trajectory [8]. Further, it is argued that Grid technologies are important for commercial computing not primarily as a means of enhancing capability, but rather as a solution to new challenges of constructing reliable, scalable, and secure distributed systems [8].

Recently there has been a proposal to marry the Grid with Web Services technologies [11] to create a service-oriented architecture for the Grid [8, 22]. This Open Grid Services Architecture (*OGSA*) views a Grid as an extensible set of *Grid Services* that may be aggregated in various ways to meet the needs of virtual organizations. While Globus defined the protocols required for interoperability, OGSA focuses on the nature of services that respond to protocol messages.

The primary purpose of the Globus middleware [6, 5] is to let users submit jobs to remote nodes in a grid for execution. This is usually done by specifying a program to be executed. The code is either pre-installed at the remote node, or is accessible (given a URL) via FTP or the GASS service[1]. In contrast, in a service-oriented grid such as OGSA, grid nodes provide canned functionality to users, exposing only a service endpoint. The interface, and to a certain extent, the semantics of the service are described using a standard language (e.g. WSDL[4]), and it is incumbent upon a user to discover the available services using one or more service registries. There is no guarantee however, that an application developer/integrator will find services of the type (s)he needs. Therefore in some situations, they may have to write their own custom services to act as components of their applications. They may not however possess (or wish to maintain) the computing facilities needed for running such services or applications in a robust, reliable manner. Therefore, a remote job submission capability similar to that in Globus would still be needed – a service that abstracts underlying *computing resources* and makes them available to grid users. The *Remote Job Execution Service* (RJE), which is the focus of this paper provides such a computing service within the OGSA framework, potentially with QoS guarantees.

RJE provides the capability to remotely deploy services and/or applications. It could thus be used as a *programmatic* interface for server farms that otherwise rent out computing hardware after *offline* negotiation and configuration. It can also be used by service providers to perform dynamic load balancing in the face of fluctuating workloads. Application-level resource managers may use such computing services to dynamically reposition the application components within a network, so as to optimize network usage – for example, if a module of the application requires frequent access to a database, it may be deployed on to a grid node "near" the database. The RJE service can also be extended to act as a host for mobile agents[17].

We discuss the architecture of RJE in Section 2, followed by the high-level design issues in Section 3. We then describe the service interface in Section 4 and briefly discuss related work in Section 5. Finally we present some conclusions and our plans for future work, in Section 6.

2 Architecture of the RJE

Figure 1 shows the architecture of a generic remote job execution service for the OGSA.

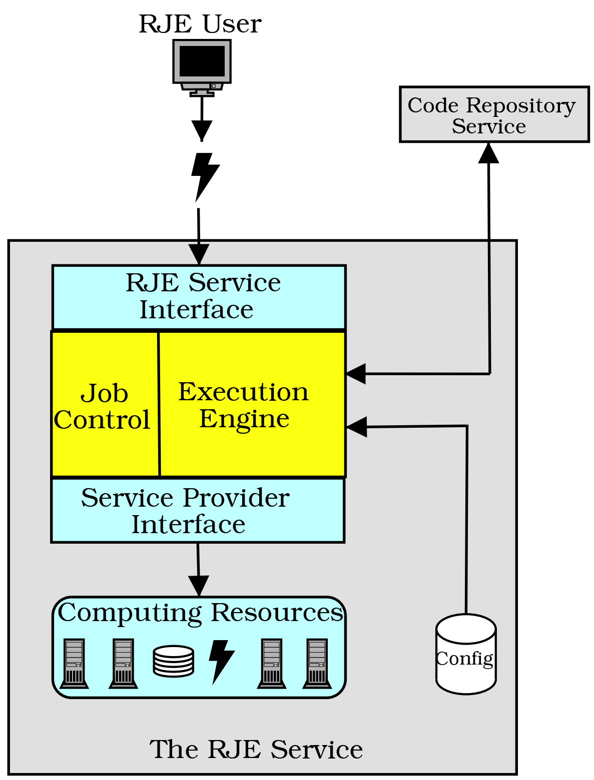

Figure 1: Architecture of the RJE Service

The RJE service runs atop a set of computing resources. These could be as simple as a single PC-class server running Linux, or as complex as a server farm with a heterogeneous mix of high-end servers, mainframe machines, storage networks, etc. The user accesses RJE using its *Service Interface*, an OGSA-compliant API that includes various operations for submitting jobs, fetching their output, monitoring and controlling them, etc. The RJE has a code *Execution Engine* that provides the user's job with a thread of execution – the basic unit of computing power – along with *controlled* access to the underlying resources. The Execution Engine must have the ability to:

- load the user code (from an external *Code Repository* if necessary),
- configure a protection domain (a *sandbox*[10, 3, 15]) to safely execute the user code with limits on its resource usage, and
- spawn a thread of control onto the underlying hardware, which may involve making a load-balancing decision to choose amongst available machines.

The RJE's *Job Control* module is responsible for tracking each job being hosted by the service. It contains the internal data structures necessary for maintaining status information on the jobs, their resource consumption, their metered usage charges, etc. It also responds to status queries or control commands issued by the user via the RJE service interface.

Before it can execute the user-submitted (foreign) code, the RJE service must take certain steps to protect itself. For example, it must set up a sandbox that limits the user code's *access* to system resources, perhaps by trapping certain sensitive system calls and interposing its own access control policy on such calls. In addition, it may choose to impose limits on the resource *consumption* by a user's job, based on pre-defined service-level agreements – for example, the job may not use any more than 20% of its CPU, nor more than 100MB of disk space. Such controls can also help protect against denial of service attacks, and against malfunctioning code. However, the mechanisms required for creating such resource-constrained sandboxes are very system-specific. We cannot therefore mandate any particular mechanisms in specifying the architecture of a generic RJE service. Instead, we define a *Service Provider Interface* (SPI) that abstracts out the commonly available resource-control operations. The RJE invokes these SPI operations in order to configure a sandbox. The underlying platform that provides the RJE service must implement the SPI operations using locally available mechanisms. It is possible that some of these operations cannot be implemented on a particular platform. The RJE's access control and resource usage policies must then be carefully designed to take such limitations (and the potential threats that result) into account. These policies, as well as configuration information about the underlying computing resources, are stored in a *Config* database as shown in Fig. 1.

We anticipate that in many situations, the user would prefer not to send the entire code for a job to the RJE for execution. Instead, (s)he may request that the code be downloaded as needed from a *Code Repository Service*. Such a service could provide standard applications or libraries from certified software developers, apart from temporarily hosting user code when necessary. The RJE provider is also more likely to trust code supplied by a well-known code repository, and thus give the user's job a less-constrained sandbox in which to run.

3 Design Issues

Given the architecture outlined above, we now consider several issues that arise in the design of a remote job execution service.

The Code: When the user submits "a job" for execution, how is the code provided? There are several possibilities:

1. The code is sent as part of the service request.
2. The code is pre-installed on the RJE service.

3. The code is pre-installed on a third-party Code Repository service.

Case 1 is perhaps the obvious way for a user to supply the code. An RJE service may choose to support this by providing an operation in its interface, that takes the code as an input parameter. There are however a couple of drawbacks. The RJE must trust the user sufficiently to execute code supplied by him/her in this manner. Also, the integrity of the service request message must be guaranteed using some mechanism such as message digests – this allows the RJE to detect any tampering or corruption of the submitted code while it is in transit.

Case 2 is the easiest for an RJE provider. The RJE advertises itself as being capable of executing a specific set of applications, and the user merely specifies which program to execute using some naming scheme such as a file-system path. However, it may be argued that this reduces the RJE to a collection of independent OGSA services – one for each application that it provides – and thus the RJE does not remain a 'pure' job hosting service.

Case 3 is in our view, the most likely in practice. Using techniques such as code signing[18], a code repository could make certain assertions about the authorship, integrity and trustworthiness of a program. The code repository is an OGSA service with a published interface of its own. The user provides the grid service handle (GSH)[22] of the code repository, along with a name for the program to be executed, using a naming scheme such as URNs[19]. The RJE service, based on its local policy, decides whether it trusts the specified repository. If so, it downloads the code and executes it.

As explained earlier, Case 2 need not be supported by a generic job execution service. In designing the RJE service interface therefore, we choose to support only Cases 1 and 3. We also assume that the code submitted is either self-contained, or is capable of finding and loading any dependencies. For example, if the job is in the form of native code, then all the libraries are either statically linked or supplied with the code. If the job is a Java program, then all the required classes are packaged in the submitted jar file.

Inputs and Outputs: The submitted job usually requires certain input parameters, and produces output results and/or errors. How should this data be communicated between the RJE service and its clients?

Programs usually take input data in four forms – command-line arguments, environment variables, input files and interactive input. The user can supply command-line arguments and environment variables as part of the job submission request. The input files can similarly be part of the request. Alternatively, fully-qualified path names of the input files may be specified. RJE must download these files using a protocol such as GridFTP[20] before executing the job. A third possibility is that the user deposits the input files on an OGSA-compliant grid storage service, analogous to GASS[1] in Globus. Then, either the RJE can download these from the storage service and provide them to the job before it executes, or the job itself may have the logic to access the data when needed, from the storage service. Interactive input is job-specific and is beyond the scope of the RJE specification. The job may use any communication mechanism (modulo the RJE's policies) to acquire the input directly from the user.

Programs may provide output in different forms – printed onto the terminal, stored in files, sent over the network, etc. The RJE captures terminal output as well as any error messages in a file, apart from output files created by the job. When the job terminates, the user can request these output files via the service API. Network communication with the user's site may of course occur during job execution, subject to the RJE's policies as before.

Job Monitoring and Control: During a job's execution, particularly for long-running jobs, the user may wish to monitor its status and progress. One way is for the user to explicitly request job status. The RJE should, after authenticating the caller as the owner of the job, supply such details as the current status of the job (*running, terminated, suspended for I/O, killed, ...*) as well as its cumulative resource consumption. Alternatively, the user may register an OGSA NotificationSink[22] with the RJE, to which the RJE sends asynchronous notifications whenever the job status changes. This allows the user to decide whether the job is making progress, needs more resources, or perhaps should be terminated prematurely. The RJE service interface must offer such job control operations as well. The RJE may also provide status information on the service as a whole, such as its current load, how many jobs are active, how much capacity is available, etc.

Job Confinement and Resource Control: As discussed in Sec. 2, the RJE must execute the user's job in a confined environment such that its access to resources can be limited, both in terms of *which* resources it can use, and *how much* of those it can consume. Several platform-specific solutions are available for this purpose[15, 3, 2, 13, 21]. Rather than specifying any particular sandboxing or resource management solution, the RJE merely expects a specific SPI to be provided by these solutions. The SPI allows the RJE to demand certain services from the underlying platform – access control, resource limits, resource allocation guarantees, etc. An extensible vocabulary may have to be defined for naming the resources being controlled, and for use in the RJE's policies.

Describing the RJE Service: Like any other OGSA service an RJE service must describe itself in a registry, to allow service discovery and selection. The service could publish information about its computing resources – its peak processing power, the amount of memory available, the type and version of the operating system on which jobs are hosted, etc. It may also publish its resource usage policies – e.g. limits on CPU usage, disk space, degree of concurrency, etc. Further, it may periodically refresh its current status using metrics such as the current CPU usage, number of active jobs, amount of free memory, etc. Such information could be used by a user (or more likely, a job scheduler working on behalf of the user[5]) to select an appropriate instance of RJE for deploying a job.

An alternative would be to publish quality of service (QoS) guarantees that the RJE provides. It appears difficult however, to define specific metrics that can be guaranteed. For example, execution time cannot be guaranteed since the RJE has no knowledge of the user's code. Job completion cannot be guaranteed either, since the job may contain bugs or may be terminated prematurely by the user. The RJE can only ensure that the job will get a fair share of the resources in case of contention. Here, fair share need not mean equal share. The job's share may depend on parameters contained in a service-level agreement with the user, such as the job's priority, what the user is willing to pay for job completion, etc. Each RJE service's description should then indicate such resource allocation policies as well, to allow the user to make an informed choice.

Implementing an Execution Engine: Given the heterogeneity of computing platforms in a grid, and the variety of code execution containers possible (native code, Java classes, EJBs...), different RJE implementations will be needed. There appears to be little commonality amongst these, that could be abstracted into an RJE specification or design. Each will use different mechanisms for setting up the environment, creating the secure sandbox, limiting access to the resources and executing the user code. We plan to create two prototype implementations of the execution engine – one for native (binary) code, and the other for Java code. Each of these will be hosted on two different platforms – Linux, and IBM's AIX, making use of certain platform-specific solutions for access control and resource usage limiting.

4 The RJE Interfaces

We now describe two interfaces – the service API exposed by the RJE to its clients, and the SPI used by it to access platform-specific resource control mechanisms. Together, these interfaces may be deemed to constitute the *specification* of a generic RJE service for OGSA grids. Implementations of RJE may differ widely as discussed above, but they should conform to these abstract interfaces.

4.1 The API

The RJE Service Interface consists of several operations that can be categorized as follows. Each of the following categories would constitute a separate *portType*[22] (refer Table 1).

portType	Operation	Input	Output
JobSubmission	runJob	JobName, Environment[1], Arguments[1], Inputfiles[1], CodeRepositoryGSH, DataStoreGSH[1], NotificationSinkGSH[1]	JobHandle
JobSubmission	runJob	JobName, Environment[1], Arguments[1], Inputfiles[1], Codefile, DataStoreGSH[1], NotificationSinkGSH[1]	JobHandle
OutputRetrieval	getOutput	JobHandle	JobOutput
JobMonitoring	getStatus	JobHandle	JobStatus
JobControl	speedupJob	JobHandle, Percentage	JobStatus
JobControl	slowdownJob	JobHandle, Percentage	JobStatus
JobControl	killJob	JobHandle	JobStatus
JobControl	suspendJob	JobHandle	JobStatus
JobControl	resumeJob	JobHandle	JobStatus

Table 1: The RJE serviceType

- **Job Submission:** The client uses this API to submit a job execution request to RJE. The alternatives for submitting code and input data were discussed in Section 3. In addition, the client optionally specifies where a notification should be sent when a significant state change occurs, such as the completion of the job. The RJE returns a job handle that can be used for further inquiries and/or instructions for this job. If the RJE cannot execute the submitted job for some reason – perhaps because it is overloaded, or its security policy does not allow the user to submit code directly – a null value is returned instead.

- **Output Retrieval:** When a job terminates, the client uses this API to retrieve its results and/or errors. The client supplies the job handle that was returned during job submission. The RJE sends back terminal output captured as a file, errors captured in another file and any output files generated by the job.

- **Job Monitoring:** This API allows a client to enquire about the status and health of the job submitted. Given the job handle, the RJE sends back the job status and resource usage statistics. The job status is one of *Running, Terminated, Suspended, Killed, Waiting for I/O.* The resource usage metrics include:
  - CPU time consumed so far
  - Memory Usage (current and maximum)
  - Disk Space Usage (current and maximum)
  - Network I/O Usage (cumulative)

- **Job Control:** The client uses this API to control the execution of the job submitted. The client supplies the job handle and can request that the job be terminated, suspended or resumed. If it observes that the job is making slow progress, it may also request the RJE to speed up its execution (assuming the user is willing to pay extra). The RJE may honour this request if it has additional resources available, using any appropriate mechanism such as increasing the CPU share of the job, increasing its priority, providing more swap space, etc. The converse request – for slowing down a job – can also be supported.

[1] optional parameter

4.2 The SPI

We briefly introduced the SPI in Section 2. The SPI is designed to make the RJE platform-independent, as well as independent of specific *sandboxing* and *resource control* solutions used in its implementation. The RJE executes the client code in a secure and resource-controlled environment, using operating system support or third party solutions. The service provider may choose not to use sandboxing or resource control or both, depending upon the threat perception and trust in the user community.

We define two sets of SPIs: one for sandboxing and the other for resource control. The sandboxing SPI is derived from the scheme described by Goldberg et al. [10]. Their solution monitors and restricts a process' access to the platform resources by trapping OS-level system calls. The sandboxing SPI provides a mechanism for RJE to specify its access control policy to the sandboxing solution on its platform. For this purpose, we need a *language* using which the policy can be encoded. Let us assume that system calls are divided, based on their functionality, into different modules:

- *KernelData* : calls for accessing kernel data structures.
- *FileSystem* : file system related calls.
- *IPC* : calls for communication among processes on the same machine.
- *Network* : network access related calls.
- *Process* : process creation, control and monitoring related calls.
- *Environment* : system calls to set and access environment variables.

Each module may be further subdivided based on the functionality it provides e.g. *Network* can be further divided into *tcp, udp, icmp* etc. and a hierarchy of modules can thus be established. This gives us a naming system for identifying the objects in the authorization policy.

A typical implementation of RJE will have a *global* configuration of a sandbox – settings that apply to all the jobs it executes. This configuration can be further refined for each job, based on the client profile. RJE defines the following SPI for refining the sandbox configuration:

```
configureSandbox(JobHandle, Module, Permission, Params);
```

In this operation, the `JobHandle` is used to map the configuration to the actual OS process or thread executing the job. The `Module` is a fully qualified module name (e.g. *Network:tcp:portNum*), Permission is either *Allow* or *Deny*, and `Params` are parameters specific to the module (e.g. wildcard addresses like **.ibm.com:80* for the *Network:tcp* module, or paths like */usr/bin/** for the *FileSystem* module). Together, these parameters, the naming mechanisms, wildcard characters, etc. constitute the language used to specify access control policy.

The RJE again relies on OS or third-party solutions for per-job resource usage control. Its generic resource control SPI is used to configure the underlying platform based on its policies and agreements with the user. Our SPI specification is derived from [2, 13]. Using the SPI, the RJE may specify limits on a job's usage of various resources. For each resource, three limits may be specified:

- *Minimum* : The guaranteed minimum resource allocation for the job.
- *Soft maximum* : The maximum usage that can be allowed when there is contention for the resource. Naturally, it is possible for a job to get more resources than this limit in the absence of contention.
- *Hard maximum* : The maximum usage that can be allowed even if there is no contention for that resource. The job will never get more that this limit, even if it is the only active job on the system.

We define the following SPI operation for configuring resource limits of a job:

```
setResourceLimit(JobHandle, Resource, minimum, softMax, hardMax);
```

Again, the `JobHandle` correlates to the actual OS-level process. `Resource` identifies the resource being controlled – *CPU, Memory,* or *DiskIO* for example, and the three usage limits are specified as discussed above.

5 Related Work

Our work is closely related to the *Globus Resource Allocation Manager* (GRAM)[5] – a component of the Globus Toolkit[6] used for resource management. It processes user requests for computing resources for job execution, allocates the required resources and manages active jobs. The RJE service outlined in this paper provides a service-oriented view of the GRAM functionality, in the OGSA context. The various portTypes defined for RJE map nicely to the GRAM API for submitting, monitoring and controlling jobs. In Globus, the programmer uses a resource specification language (RSL) to define a job and its resource requirements. In contrast, RJE interfaces can be used programmatically once the appropriate service instance has been determined, for submitting, monitoring and controlling jobs.

The concept of providing a container for executing code or components has been used often in the past. The J2EE[14] architecture for example defines specifications for Enterprise Java Beans, and components written as per these specifications can be deployed on EJB containers. Similarly, mobile agent frameworks usually include agent servers[16] that act as execution engines for mobile code. The RJE's Execution Engine is similar, in that it provides the basic compute resources to run user code. The security issues faced in its design are similar to those for mobile code containers[17]. However, unlike distributed object frameworks like CORBA[12] and J2EE, RJE doesn't itself need to provide any extra services – because each node participating in the OGSA grid already provides a set of core OGSA services. The job executing in the RJE container is free to use these services if necessary.

6 Conclusions and Future Work

We reported on an ongoing project aimed at providing a *computing service* to users in a service-oriented grid. We propose to abstract out low-level resources like cpu, memory and disk, using a service interface compliant with the emerging OGSA standard. We discussed the various issues involved in design and implementation of such a service. In the near future, we plan to create prototypes of the RJE service for running native code as well as Java programs submitted by users. In the process, we will also specify the RJE interfaces more rigorously using OGSA-defined extensions to WSDL. We plan to use existing technologies[10, 2, 3] to create a secure, resource-constrained sandbox for running jobs submitted by clients.

In addition, we are exploring extensions to the RJE for supporting mobile code[17]. A job submitted by a user may have the autonomy to decide that it needs to migrate to a different RJE. Its current host and desired destination RJEs must then provide the mechanisms necessary for such job migration – state capture, transfer, and re-creation at the destination. This will necessitate the expansion of the RJE's service interface, as well as new protocols to enable secure state and code transfer between RJE services.

References

[1] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[2] S. Castro, N. Tezulas, B. Yu, J. Berg, and H. Kim. *AIX 5L Workload Manager(WLM).* IBM Red Book SG245977, June 2001.

[3] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-constrained Sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[5] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer-Verlag LNCS, 1998.

[6] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[7] I. Foster and C. Kesselman, editors. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, July 1998.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/research/papers/ogsa.pdf, January 2002.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[10] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of 6th USENIX Security Symposium*, July 1996.

[11] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Sams, 2001.

[12] D. Harkey and R. Orfali. *Client/Server Programming with Java and CORBA*. John Wiley and Sons, March 1998.

[13] HP-UX Workload Manager white paper. http://www.hp.com/products1/unix/management/infolibrary/wlmwp.pdf, October 2001.

[14] Java 2 Platform Enterprise Edition. http://java.sun.com/products/j2ee.

[15] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the ISOC Symposium on Network and Distributed Systems Security (NDSS 2000)*, San Diego, USA, February 2000. The Internet Society.

[16] Neeran M. Karnik and Anand R. Tripathi. Agent Server Architecture for the Ajanta Mobile-Agent System. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 66–73, Las Vegas, USA, July 1998. CSREA Press.

[17] Neeran M. Karnik and Anand R. Tripathi. Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency*, 6(6):52–61, July–September 1998.

[18] John R. Michener and Tolga Acar. Managing System and Active-Content Integrity. *IEEE Computer*, 33(7):108–110, July 2000.

[19] R. Moats. RFC 2141: URN Syntax. http://www.cis.ohio-state.edu/htbin/rfc/, May 1997.

[20] The Globus Project. GridFTP: Universal Data Transfer for the Grid. Technical report, The Globus Project, September 2000. http://www.globus.org/datagrid/gridftp.html.

[21] Solaris Resource Manager 1.1 white paper. http://wwws.sun.com/software/whitepapers/wp-srm11/srm11wp.pdf.

[22] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. Grid Service Specification. http://www.globus.org/research/papers/gsspec.pdf, February 2002.