The Benefits of Java and Jini in the JGrid System

Szabolcs Pota and Zoltan Juhasz

University of Veszprem
Dept. of Information Systems
Veszprem, Hungary
{pota,juhasz}@irt.vein.hu

Abstract

The Java language and platform have been considered by many as natural candidate for creating grid systems. The platform-independent runtime environment, safe and high-level language and its built-in support for networking and security are very valuable features. Despite its potential and the many proof-of-concept systems developed, the grid community is turning to web services technology as its implementation base. In this paper, we show that Java, by joining forces with Jini Technology can provide a very appealing technology base for highly dynamic grid systems. The key properties of Java and Jini technology are examined with reference to their role in grids. Then, the JGrid Jini-based service-oriented grid system is overviewed describing its key concepts, services and how it extends Jini to address some of the unique requirements of grid systems.

1. Introduction

Future service-oriented grid systems will need to provide dynamic service discovery, support for interactive applications, more effective mechanisms to interconnect and orchestrate multiple services to solve complex problems, and the ability to integrate and collaborate with non-computational services.

Java and Jini [1] have many unique and useful characteristics that make them an ideal candidate for grid systems. While Java is definitely the language of the Internet, due to the many legacy applications and systems, and fear of using one single language, it is not fully accepted in the grid community. Web Services technology is preferred on this basis, but interestingly, it helps spread the use of Java as Java is the primary language of choice in implementing Web services.

The goals of the JGrid project [2] are to examine and demonstrate the advantages of Java and Jini in the grid technology area, and to develop a novel Jini-based service-oriented grid system that supports discovery, interactive applications, service composition, and provides a high-level, effective service-oriented programming model for developers.

In this paper, we overview those features of Java and Jini that can be valuable in creating grid systems and show that Java, by joining forces with Jini Technology can provide a very appealing technology base for highly dynamic grid systems. The use and value of these features are illustrated in the JGrid system.

The outline of the paper is as follows. In section II, a short overview of previous results related to our paper is given. Section III introduces the main features of Jini Technology and shows how these features can be used as building blocks in a future grid technology. Section IV explains how the JGrid system extends the capabilities of Jini to create an infrastructure for service-oriented grid systems. It also provides a brief overview of the key concepts and services of the system. Section V illustrates with a list of demos and proof-of-concept applications that JGrid can be used in a wide range of application domains and can be easily extended with new services. The paper ends with conclusions.

2. Related Work

Service-oriented grid systems, their architecture, functionality and programming are central to current grid research. While the current technology preference for Grid systems is Web Services, due to the lack of high-level programming support of Web Services, Java-centric systems are demonstrating alternative ways and strategies for building grid systems.

Early Java metacomputing systems (e.g. SuperWeb [3], Javelin [4]) relied on Java applets and RMI [5] but the inflexibility and scalability problems of RMI limited their applicability at a global scale. Several of these systems, such as the Harness system [6] and its successor H2O [7] and ProActive [8], successfully demonstrate the advantages of Java, although they are not general grid environments.

The advantages of dynamic discovery and other unique features of Jini Technology have been already used and their value demonstrated in several grid-related projects, such as ICENI [9], JISGA [10], CoABS [11], ZENTURIO [12], aiming to create more dynamic and service-oriented Grid environments.

3. Jini Technology as a Basis for Grid Systems

Service-orientation has become the accepted model for creating next-generation distributed (including Grid) systems. Although Web Services Technology is becoming dominant as the implementation basis for Grid systems, it is not the only possible technology to use. In this section we examine how various features of Jini provide useful support for Grid systems.

3.1. Service-Oriented Architecture

Jini [1] is a true and mature service-oriented technology, released in 1999 by Sun Microsystems Inc. It is built on the Java platform, which implicitly provides many advantages for Grid systems. The Java Virtual Machine creates a unified, platform-independent environment on top of the widely heterogeneous set of Grid resources. It provides a proven programming model and development methodology used by millions of developers. It has built-in networking support and security architecture that can be extended to meet Grid requirements.

A Jini system consists of services and service clients. Since Jini was designed to create a federation of program components that operate automatically, services are described by Java interfaces as a natural choice. This also facilitates loose coupling of clients and services in a sense that service capabilities are separated from the concrete implementations.

3.2. Dynamic Networking

Large distributed systems are dynamic by default. Components may disappear due to shut-down or network/computer failure and appear at any moment. Jini explicitly supports this dynamism via its service *discovery*, *join* and *lookup* protocols. Each service registers in a special service called *Lookup Service*, with a Java object. The registration signals the existence of the service in the Jini network. Clients also use the Lookup Service to find suitable services. This is similar to searching for web services in a UDDI directory. Note that both clients and services need to know the lookup service, as this represents the entry point to the Jini system. An entity can use either multicast or unicast discovery, depending on whether or not it knows the address of the lookup service, respectively.

An added benefit of using Java is that Jini service references are represented by Java objects. These reference objects are embedded in proxy objects which, in turn, are part of the registration objects stored in the Lookup Service upon registration. Hence, when a client finds a suitable service, the result of the lookup operation is the proxy object of the service that automatically provides the required service invocation mechanism.

3.3. Mobile Code and the Proxy Object

The combined use of the proxy pattern, mobile code and dynamic class loading in Jini has many advantages. Since the proxy is downloaded from the service at run-time, it is always up-to-date and no prior installation is necessary. The proxy hides communication and implementation details from the client. Since the proxy is provided by the service and communication is private to the proxy and the service, the service implementer can use the protocol best suited to the given situation (raw socket, RMI, HTTP, etc.). The proxy can also provide the basis for various service execution strategies: (*i*) the service can run entirely on the remote server with the proxy merely delegating requests, (*ii*) the service can run entirely in the proxy on the client, or (*iii*) the proxy and the service can distribute service functionality by executing service code in the proxy and the service as well. The last pattern (smart proxy) could facilitate e.g. intelligent error handling and recovery, support preserving state between method calls, reduce communication overhead or provide (perhaps limited functionality) operation in disconnected mode.

Since the proxy implements the well-known service interface, the Jini client accesses services via method calls on Java objects. Interfaces can evolve by subclassing prior interfaces. Clients using the old interface will be able to use new services, albeit with the old functionality only. It must be emphasised that it is up to the service implementer to decide how to delegate the method calls to the service; e.g. via Remote Method Invocation or proprietary protocols over TCP/IP or HTTP. To the client, they all appear identical – Java method calls on the proxy. Another advantage of the proxy pattern is that it benefits from the Java exception handling mechanism. Communication, service or proxy problems can be reported to the client via well-defined Java exceptions.

Jini services can provide user interfaces as well as programmatic ones. The Jini ServiceUI specification describes a standard way to attach user interface objects to the service proxy that can be downloaded from the service dynamically on an on-demand basis. This also provides for the use of multi-modal interfaces, providing support for users and devices with different capabilities to use services.

3.4. Security

Since the proxy is the connection point between the client and the service, it is where the client and service administrative domains meet. Consequently, this is where security problems arise. The Jini security model builds upon the J2SE 1.4 security model (language and type safety, byte code verification, protection domains) and provides extensions required in distributed systems: network security, security constraints, dynamic permission granting, proxy trust checking, and method level access control. Network security is achieved with a new customisable RMI implementation called JERI (Jini Extensible Remote Invocation) that makes each layer of the protocol stack configurable, e.g. different transport protocols can be used to transfer remote method calls (e.g. TCP, SSL, HTTP, HTTPS).

3.5. Distributed Programming Model

In large systems hardware and software failures may be common. Such systems call for special programming mechanisms to maintain system health and stability. Jini provides the concept of leasing to create a self-healing system. A *lease* represents a time-based grant to a resource. For instance, a service registering in the lookup service receives a lease for that registration. If the service renews the lease before that lease expires, the registration object remains in the lookup service. If, for any reason, the service cannot renew its lease, the lookup service assumes the service to be faulty; hence it removes the service's registration entry. This frees up lookup service resources used by the service registration, and prevents clients from downloading the proxy of a non-existing service. The lease concept can be extended and applied to any interaction between objects. That includes the use of leases between clients and services. The client of an interactive application can obtain a lease for the use of a remote service. If that client disconnects from the service, even when that disconnection is the result of a network or client fault, the lack of the client's lease renewal will cause the service to stop the running computation and release the allocated resources.

Jini also provides a *distributed event mechanism*, as well as support for distributed transactions. Event notifications are an important requirement for grid systems where there is a very clear need for stateful services. Grid services and their clients can rely on the distributed event mechanism for notifications of particular state changes. Jini *distributed transactions* are provided as a framework, where service implementers can specify the actual implementation of the transaction. That mechanism allows services to comply with a set of known interfaces, and to perform operations under transactions, if required.

3.6. Legacy Integration Support

Jini is considered by many as pure Java system. This may be the reason why it has been neglected by the grid community. The Jini specification does not mandate that Jini services must be implemented in Java. One of the advantages of using service proxies is that the proxy hides service implementation to such level that the programming language used for service implementation becomes irrelevant.

Jini services can be implemented in any programming language. The role of the proxy in this case is to use a suitable private protocol that is acceptable to the service implementation, e.g. a raw socket, and communicate method call and data transfer information over that private protocol to the non-Java backend. This mechanism facilitates simple integration of legacy, non-Java services into a Jini-based grid. Special helper services exist in the Jini reference implementation that provide discovery, registration and lease management for non-Java service implementations.

Restricted devices that cannot run Java code or not the required version (J2SE 1.4) can use the Surrogate Architecture in which a surrogate host performs Jini-related tasks for the device.

4. The JGrid System

The aim of the JGrid project is to demonstrate the benefits of Java and Jini in grid systems. The JGrid system is the result of this research and development effort; it is a service-oriented grid framework building on and extending the capabilities of Jini.

The Jini properties examined in the previous section lay down the foundations for creating dynamic grid systems where services, as well as protocols, can operate without major maintenance downtimes and evolve without disrupting the system's operation. However, Jini has been primarily designed for small client-service assemblies operating on a LAN.

The JGrid system extends Jini to create a Grid system spanning administrative domains and introduces higher level core and application-level grid services that can be used as building blocks for large grid systems. The JGrid project provides a complete dynamic service-oriented grid infrastructure including wide-area service discovery, security support, core computational service (batch, compute and storage services) and a high-level programming API for interacting with the services. In the rest of this section we describe these additions.

4.1. Service Discovery

Jini lookup services do not provide wide-area discovery naturally. Some routers do not forward multicast packets; unicast discovery does not provide spontaneous discovery as lookup service addresses must be known. It is also a potential problem if too many lookup services exist (how to find and iterate over them), and if many lookup or application service proxies must be downloaded by a client.

JGrid includes a wide-area discovery system that is a distributed hierarchical overlay network connecting dispersed lookup services trying to overcome these problems. The entry point to the discovery system is the *Grid Access Point* (GAP) service that provides lookup operations for clients and service announcements on the grid for services. The assumption is that there is a GAP service near to the client that can be discovered via multicast discovery, and once this has been done, a client wishing to discover remote services can issue a query to this local GAP in a way similar to the standard Jini lookup semantics. The query is then propagated through the discovery network and reaches those lookup services that can return the requested services.

This mode of operation is achieved by *Router Service*s that connect to the GAP and form the routing overlay network. To avoid network flooding, router services only hold aggregate service information, which facilitates content-based query routing. This mechanism allows lookup messages to be sent only to those lookup services that have potentially matching services. Routers at different level in the hierarchy store information at different representation level. Routers discover their neighbouring routers in the topology via either multicast or unicast discovery. Depending on the configuration, various levels of self-organisation and reliability can be achieved.

4.2. Security

Building on the Jini 2.0 security architecture, JGrid provides secure access to and communication with services. The JGrid security is based on public key security infrastructure combined with role-based access control. This is provided by two JGrid core services; *Authentication* and *Registration* services.

The Authentication service is responsible for logging in users to the grid; authenticate them with the appropriate (configurable) authentication method. Once the user is authenticated, the authentication service issues a short-term X.509 certificate that will be used in subsequent service method calls.

Clients authenticate services by checking service certificates. Trusted services are accessed via JERI method calls that transfer the short-term certificate issued by the authentication service. When the target service receives the method call, it checks whether the client is authorised to execute the method. This is done by contacting the registration service associated with the target service. If the user has the required role in the registration service database, the method call will execute otherwise a security exception is thrown to the caller.

The main advantage of this architecture is that users only have to log-in to the grid once, and individual services do not need to store user data separately for each user. Services only need to define access roles and assign these roles to users and store it in the registration service.

Future versions of JGrid will support the federation of authentication services and registration services, creating a single sign-on system, similar to Liberty but adopted to the Jini world.

4.3. Core Computational Services

The original aim of developing the JGrid system was to create a dynamic Jini-based computational grid. The most important functionalities (processing and storage) have been abstracted out as Jini services. This section describes the Compute and Batch services used for executing Java and non-Java programs and the Storage service to access remote files.

Compute Service. We believe that using the same, platform-independent programming language has enormous advantages over other approaches. Developers do not need to be concerned by the target architecture, whether it is compiled correctly to that platform, or libraries are deployed. Also, the number of potential target boxes for large computational applications increases with several orders of magnitude.

Recent advances in virtual machine optimization technology make Java a strong contender in numerical computation as well. SciMark 2.0 numerical benchmark results show that Java 1.4 and 1.5 HotSpot virtual machines achieve performance close to optimized C. Our experiments show that on average 95% of C performance is achieved, with Java outperforming C in several benchmark categories (Monte Carlo, SOR).

The compute service therefore is one of the most important services in JGrid. Its role is to create a distributed Java execution environment in which sequential and parallel Java programs (objects) can be executed transparently at remote services representing single, multi-processor computers or clusters. In essence, it is similar to a distributed Java Virtual Machine, but transparency is achieved at the application not at the JVM level.

The Compute Service was designed to support dynamic grid applications that can adapt to changes in the number and quality of resources, detect and react to execution errors or environment failures in a highly heterogeneous environment.

The service offers four different types of execution modes: (*i*) synchronous remote evaluation, (*ii*) asynchronous remote evaluation, (*iii*) process spawning that creates dynamic server objects accessible via remote method invocation, and (*iv*) parallel execution using MPI-like message passing. Common in all execution modes that a Java task object is sent to the Compute service at run-time that will execute in the service's thread pool under the control of the service task scheduler.

These execution modes support interactive grid applications as the client is connected to these dynamically 'outsourced' tasks during execution, hence, clients can communicate with the task when required. This enables one to create long-running interactive simulations (e.g. man-in-the-loop systems), collaborative experiments and visualization applications, etc.

One of the important properties of the compute service is that it can coexist with any other application running on the host computer. As a result, it can be configured to use spare CPU cycles in organisations where a dedicated compute server farm is not available. The compute service threads run below normal priority and give way to user payload immediately while continuing the task at a reduced performance.

Batch Service. The role of the Batch service is to provide traditional batch job execution facilities for users. The Batch service enables the integration of legacy batch runtime systems (such as Sun Grid Engine [13], Condor [14]) into a Jini service community as well as allows users execute non-Java programs. The batch service also provides a user-friendly ServiceUI-based graphical interface for job submission and management, and can be integrated into more complex execution environments due to its programmatic interface.

All files required for execution must be available on a specified file server for download. This, typically, is an HTTP server, although a JGrid Storage service can be used either. The batch service receives a job description template, downloads the executable from the file service if necessary then submits the job on behalf of the user to the local batch runtime environment. The results of the execution are stored on an HTTP server and the user receives a URL pointing to the location of the results.

Internally, the batch service is a Jini service wrapper. It receives requests from the user, and forwards them in the appropriate format to the batch environment. The current implementation of the batch service relies on the DRMAA (Distributed Resource Management) – a Global Grid Forum – specification that provides a uniform programming interface to various batch execution environments.

Storage Service. The Storage service represents a remote file system as a Jini service. It enables clients to perform file and directory operations on a remote system. The key difference between the storage service and an ordinary file service is that it not only is a download/upload service; it provides programmatic file operations such as read/write on remote files.

Storage service users use a hierarchical set of proxies to perform operations. The main service proxy provides access to a user's own storage space that is the root of the user file system. Directory proxies represent the next level of the hierarchy, while file proxies are at the lowest level. When a file is selected, its proxy is downloaded, opened and presented to the client as a remote file stream on which read/write operations can be performed transparently. This feature enables clients to access remote files just like local ones. Users can also create delegated file and directory proxies that allow other services to read or write files on behalf of their owner. The use of hierarchical proxies illustrates how proxies can be used to partition service functionality.

5. Applications

Since JGrid is a service-oriented Grid system, it is not merely a computation grid. Anything can be represented as a service and hence can become part of the JGrid service grid.

Consequently, as part of our project, we developed several proof-of-concept applications to demonstrate the applicability of the system, the development methodology, the dynamic and reliable operation.

5.1. Computational Demos

We used JGrid to execute a biological application performing pair-wise alignment of biological sequences. The core computations used algorithms from the BioJava library [15] dedicated to provide a Java framework for processing biological data. The application demonstrated the reliability of JGrid in executing long running numerical applications, interaction of the Compute and Storage services during execution, and also showed that appropriate computational performance can be achieved in a Java-based computational environment.

In another case study, a financial parallel Monte Carlo simulation was developed using the master-worker paradigm. The client application behaved as the master process allocating computational tasks asynchronously to Compute Services discovered dynamically at run time, as well as collecting and evaluating the results.

5.2. Application Services

An interesting prototype *Media Service* has been created for delivering on-demand streaming media delivery. Based on the Java Media Framework (JMF), this Jini service can stream media to clients. The benefit of this service is that it is completely transparent, its dynamic user interface supports media search; consequently, users can find and play various media contents very effectively. The user opens the main window in the service browser, specifies keywords of the requested media clip, then selects one from the results and starts the playback process. The media can be controlled similarly to a tape recorder.

An *Internet Radio Service* has also been developed that demonstrated the integration of Java, Jini and the Java Media Framework in creating a service that feeds live radio broadcast to clients.

Other examples of services include a live data source service, a collaboration service with stereo 3D user interface, and a dictating service to store recorded sound memos at remote locations. They successfully demonstrated that in a very straightforward manner JGrid can be extended with services to support various scientific and business applications.

6. Conclusions

In this paper we examined the advantages of using Java and Jini in creating grid systems. We argued that Java has an important role as a programming language and platform for grids and that Jini extends this environment with dynamic networking capabilities and a distributed service-oriented programming model that forms a strong base for grid technology.

We presented an overview of the JGrid system that aims to demonstrate the benefits of Jini in the grid area. We described the aims of the project, described how it extends Jini, what services it contains and how they can be used to create Grid applications.

Our current work, besides improving the system and developing new services/applications, focuses on the use of JGrid for business and health care applications.

Acknowledgments

The authors thank Krisztian Kuntner and Mark Magyarodi for the design and implementation of the discovery and security architecture of the JGrid environment as well as Gergely Sipos, Peter Kacsuk (MTA SZTAKI), Tamas Zsemlye (Sun Microsystems), Laszlo Lovei, Gabor Pecsy (Eotvos University) for their contribution to the JGrid project.

This work was supported in part by the Hungarian Ministry of Education under Grant IKTA-5 089/2002 and the National Office for Research and Technology Department of Commerce under Grant GVOP-3.1.1.-2004-05-0035/3.0. The generous support of Sun Microsystems, Inc. under their Academic Equipment Grant is gratefully acknowledged.

References

[1] J. Waldo and K. Arnold, *The Jini Specifications. Jini Technology Services*, Addison-Wesley, Reading, MA, USA, second edition, 2001.

[2] JGrid: A Jini-based Universal Service Grid, http://www.irt.vein.hu/jgrid.

[3] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman, "SuperWeb: Research Issues in Java-Based Global Computing," *Concurrency: Practice and Experience*, June 1997.

[4] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser , D. Wu, "Javelin: Internet-based parallel computing using Java," *Concurrency: Practice and Experience*, Dec 1998, vol. 9, No. 11, pp. 1139-1160

[5] Sun Microsystems. *Java Remote Method Invocation Specification*, JDK 1.1, http://java.sun.com/products/jdk/rmi_ed, 1997.

[6] M. Migliardi, V. Sunderam. "The Harness metacomputing framework," In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio (TX), USA, March 22-24 1999.

[7] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, V. Sunderam, "Towards self-organizing distributed computing frameworks: The H2O approach," *Parallel Processing Letters*, 2003, vol. 13, No. 2, pp. 273–290.

[8] D. Caromel, "ProActive Java Library for Parallel," *Distributed and Concurrent Programmming*, 2001, http://www-sop.inria.fr/oasis/ProActive/

[9] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington, "ICENI: An Open Grid Service Architecture Implemented with Jini," in *Proc SuperComputing 2002* (SC2002), Baltimore, MD, USA (2002).

[10] Y.Huang, "JISGA: A Jini-based Service-oriented Grid Architecture," *The International Journal of High Performance Computing Applications* 17 (2003) 317–327 ISSN 1094-3420.

[11] M. L. Kahn1, C. De. T. Cicalese1, "The CoABS Grid", *In Proc. First International Workshop on Radical Agent Concepts,* January 2002, vol. 2564, pp. 125-134.

[12] R. Prodan, T. Fahringer, "ZENTURIO: An Experiment Management System for Cluster and Grid Computing", *In Proc. of the IEEE International Conference on Cluster Computing,* 2002, p. 9

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

[14] D. H. J Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne, "A Worldwide Flock of Condors : Load Sharing among Workstation Clusters," *Journal on Future Generations of Computer Systems*, 1996, vol. 12.

[15] M. Pocock, T. Down, T. Hubbard, "BioJava: open source components for bioinformatics," *ACM SIGBIO Newsletter*, August 2000, vol. 20, No. 2, pp. 10-12.