

PARALLEL PROGRAM EXECUTION SUPPORT IN THE JGRID SYSTEM

Szabolcs Pota¹, Gergely Sipos², Zoltan Juhasz^{1,3} and Peter Kacsuk²

¹*Department of Information Systems, University of Veszprem, Hungary*

²*Laboratory of Parallel and Distributed Systems, MTA-SZTAKI, Budapest, Hungary*

³*Department of Computer Science, University of Exeter, United Kingdom*

pota@irt.vein.hu, sipos@sztaki.hu, juhasz@irt.vein.hu, kacsuk@sztaki.hu

Abstract Service-oriented grid systems will need to support a wide variety of sequential and parallel applications relying on interactive or batch execution in a dynamic environment. In this paper we describe the execution support JGrid, a Jini-based grid infrastructure, provides for parallel programs.

Keywords: service-oriented grid, Java, Jini, parallel execution, JGrid

1. Introduction

Future service-oriented grid systems, in which users access application and system services via well-defined interfaces, will need to support a more diverse set of execution modes than those found in traditional batch execution systems. As the use of grid spreads to various application domains, some services will rely on immediate and interactive program execution, some will need to reserve resources for a period of time, while some others will need a varying set of processors. In addition to the various ways of executing programs, service-oriented grids will need to adequately address several non-computational issues such as programming language support, legacy system integration, service-oriented vs. traditional execution, security, etc.

In this paper, we show how the JGrid [1] system – a Java/Jini [2] based service-oriented grid system – meets these requirements and provides support for various program execution modes. In Section 2 of the paper, we discuss the most important requirements and constraints for grid systems. Section 3 is the core of the paper; it provides an overview of the Batch execution service that facilitates batch-oriented program execution, and describes the Compute Service that can execute Java tasks. In Section 4 we summarise our results, then close the paper with conclusions and discussion on future work.

2. Grid Requirements

Service-orientation provides a higher level of abstraction than resource-oriented grid models; consequently, the range of applications and uses of service-oriented grids are wider than that of computational grids. During the design of the JGrid system, our aim was to create a dynamic, Java and Jini based service-oriented grid environment that is flexible enough to cater for the various requirements of future grid applications.

Even if one restricts the treatment to computational grids only, there is a set of conflicting requirements to be aware of. Users would like to use various *programming languages* that suit their needs and personal preferences while enjoying platform independence and reliable execution. Interactive as well as batch *execution modes* should be available for *sequential* and *parallel programs*. In addition to the execution mode, a set of interprocess communication models need to be supported (shared memory, message passing, client-server). Also, there are large differences in users' and service providers' *attitude to grid development*; some are willing to develop new programs and services, others want to use their existing, non-grid systems and applications with no or little modification. Therefore, *integration* support for legacy systems and user programs is inevitable.

3. Parallel execution support in JGrid

In this section we describe how the JGrid system provides parallel execution support and at the same time meets the aforementioned requirements concentrating on (i) language, (ii) interprocess communication, (iii) programming model and (iv) execution mode.

During the design of the JGrid system, our aim was to provide as much flexibility in the system as possible and not to prescribe the use of a particular programming language, execution mode, and the like. To achieve this aim, we have decided to create two different types of computational services. The Batch Execution and Compute services complement each other in providing the users of JGrid with a range of choices in programming languages, execution modes, interprocess communication modes.

As we describe in the remaining part of this section in detail, the Batch Service is a Jini front end service that integrates available job execution environments into the JGrid system. This service allows one to discover legacy batch execution environments and use them to run sequential or parallel legacy user programs written in any programming language.

Batch execution is not a solution to all problems however. Interactive execution, co-allocation, interaction with the grid are areas where batch systems have shortcomings. The Compute Service thus is special runtime system developed for executing Java tasks with maximum support for grid execution, including

Table 1. Support in JGrid

<i>Category</i>	<i>Compute Service</i>	<i>Batch Service</i>
Programming language	Java	any
Execution Mode	interactive, batch	primarily batch
Communication	shared mem, msg passing, client/server, tuple space	runtime dependent (shared mem, msg passing)
Legacy user programs	no	yes
Legacy service integration	no	yes
Application domain	any	mainly HPC

parallel program execution, co-allocation, cooperation with grid schedulers. Table 1 illustrates the properties of the two services.

3.1 The Batch Execution Service

The Batch Execution Service provides a JGrid service interface to traditional job execution environments, such as LSF, Condor, SGE, etc. This interface allows us to integrate legacy batch systems into the service-oriented grid and allow users to execute legacy programs in a uniform, runtime-independent manner.

Due to the modular design of the wrapper service, various batch systems can be integrated. The advantage of this approach is that neither providers nor clients have to develop new software from scratch, they can use well-tested legacy resource managers and user programs. The use of this wrapper service also has the advantage that new grid functionality (e.g. resource reservation, monitoring, connection to other grid services), normally not available in the native runtime environments, can be added to the system.

In the rest of Section 3.1, the structure and operation of one particular implementation of the Batch Execution Service, an interface to the Condor [3] environment is described.

3.1.1 Internal Structure. As shown in Figure 1, the overall batch service consists of the native job runtime system and the front end JGrid wrapper service. The batch runtime includes the Condor job manager and N cluster nodes. In addition, each node also runs a local Mercury monitor [4] that receives runtime information from properly instrumented user programs. The local monitors are connected to a master monitor service that in turn combines local monitoring information and exports it to the client on request. Figure 1 also shows a JGrid information service entity and a client, indicating the other required components for proper operation.

The resulting infrastructure allows a client to dynamically discover the available Condor pools of the network, submit jobs into these pools, remotely manage

the execution of the submitted jobs, as well as monitor the running applications on-line.

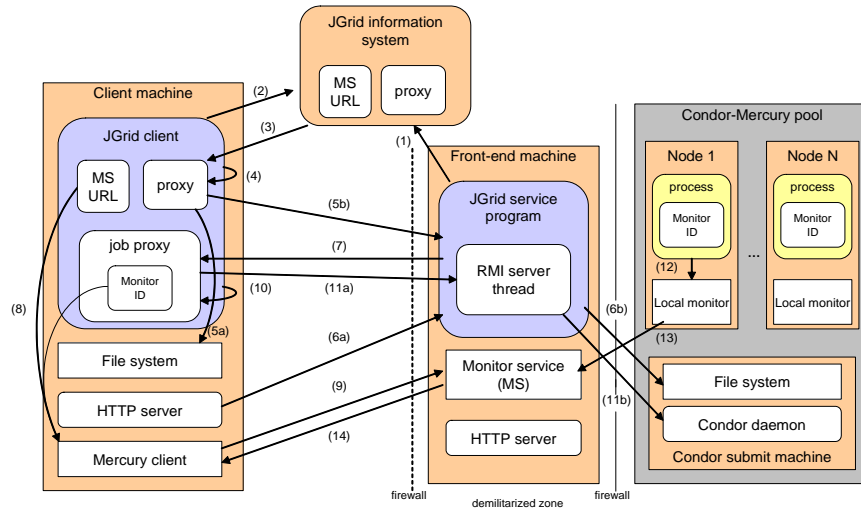


Figure 1. Structure and operation of the Batch Execution Service.

3.1.2 Service operation. The responsibilities of the components of the service are as follows. The JGrid service wrapper performs registration within the JGrid environment, exports the proxy object that is used by a client to access the service and forwards requests to the Condor job manager. Once the job is received, the Condor job manager starts its normal tasks of locating idle resources from within the pool, managing these resources and the execution of the job. If application monitoring is required, the Mercury monitoring system is used to perform job monitoring. The detailed flow of execution is as follows:

- 1 Upon start-up, the Batch Execution Service discovers the JGrid information system and registers a proxy along with important service attributes describing e.g. the performance, number of processors, supported message passing environments, etc.
- 2 The client can discover the service by sending an appropriate service template containing the Batch service interface and required attribute values to the information system.
- 3 The result of a successful lookup operation results in the client receiving the proxy-attribute pair of the service.
- 4 The client submits the job by calling appropriate methods on the service proxy. It specifies as method arguments the directory of the job in the

local file system, a URL through which this directory can be accessed, and every necessary piece of information required to execute the job (command line parameters, input files, name of the executable, etc.).

- 5 The proxy archives the job into a JAR file, then sends the URL of this file to the front end service.
- 6 The front end service downloads the JAR file through the client HTTP server, then extracts it into the file system of a submitter node of the Condor pool.
- 7 As a result of the submit request, the client receives a proxy object representing the submitted job. This proxy is in effect a handle to the job, it can be used to suspend or cancel the job referenced by it. The proxy also carries the job ID the Mercury monitoring subsystem uses for job identification.
- 8 The client obtains the monitor ID then passes it - together with the MS URL it obtained from the information system earlier - to the Mercury client.
- 9 The Mercury client subscribes for receiving the trace information of the job.
- 10 After the successful subscription, the remote job can be physically started with a method call on the job proxy.
- 11 The proxy instructs the remote front end service to start the job, which then submits it with a secure native call to the Condor subsystem. Depending on the specified message passing mode, the parallel program will execute under the PVM or MPI universe. Sequential jobs can run under the Vanilla, Condor or Java universe.
- 12 The local monitors start receiving trace events from the running processes.
- 13 The local monitor forwards the monitoring data to the master monitor service
- 14 The master monitor service sends the global monitoring data to the interested client.

Once the job execution is finished, the client can download the result files via the job proxy via other method calls. The files then will be extracted to the specified location of the local filesystem as specified by the client.

It is important to note that the Java front end hides all internal implementation details, thus clients can use a uniform service interface to execute, manage and

monitor jobs in various environments. In addition, the wrapper service can provide additional. grid-related functionality not available in traditional batch execution systems.

3.2 The Compute Service

Our aim with the Compute Service is to develop a dynamic Grid execution runtime system that enables one to create and execute dynamic grid applications. This requires the ability to execute sequential and parallel interactive and batch applications, support reliable execution using checkpointing and migration, as well as enable the execution of evolving and malleable [5] programs in a wide area grid environment.

Malleable applications are naturally suited to Grid execution as they can adapt to a dynamically changing grid resource pool. The execution of these applications, however, requires strong interaction between the application and the grid; thus, suitable grid middleware and application programming models are required.

Java is a natural choice for this type of execution due to its platform independence, mobile code support and security, and since traditional batch systems do not fully support dynamic, grid-aware execution, it seemed natural to create a new type of execution service for this purpose. We envisage that grid resource brokers or meta-schedulers will be able to use the Compute Service as computation nodes.

3.2.1 Task Execution. The Compute Service, effectively, is a remote JVM exported out as a Jini service. Tasks sent for execution to the service are executed within threads that are controlled by an internal thread pool. Tasks are executed in isolation, thus one task cannot interfere with another task from a different client or application.

Clients have several choices for executing tasks on the compute service. The simplest form is remote evaluation, in which the client sends the executable object to the service in a synchronous or asynchronous `execute()` method call. If the task is sequential, it will execute in one thread of the pool. If it uses several threads, on single CPU machines it will run concurrently, on shared memory parallel computers it will run in parallel.

A more complex form of execution is remote process creation, in which case the object sent by the client will be spawned as a remote object and a dynamic proxy created via reflection, implementing the `TaskControl` and other client-specified interfaces, is returned to the client. This mechanism allows clients e.g. to upload the code to the Compute Service only once and call various methods on this object successively. This proxy will have a major role in parallel execution as shown later in this section.

A single instance of the Compute Service cannot handle a distributed memory parallel computer and export it into the grid. To solve this problem we created a ClusterManager service that implements the same interface as the Compute Service, hence appears to clients as another Compute Service instance, but upon receiving tasks, it forwards them to particular nodes of the cluster. It is also possible to create a hierarchy of managers e.g. for connecting and controlling a set of clusters of an institution.

The major building blocks of the Compute Service are the task manager, the executing thread pool and the scheduler. The service was designed in a service-oriented manner, thus interchangeable scheduling modules implementing different policies can be configured to be used by the service.

3.2.2 Executing Parallel Applications. There are several approaches to executing parallel programs using Compute Services. If a client discovers a multi-processor Compute Service, it can run a multi-threaded application in parallel. If the client looks up a number of single-processor Compute Services or a multi-processor one along with a Cluster attribute, it must rely on other interprocess communication mechanisms. Our system at the time of writing can support communication based on (i) MPI-like message passing primitives and (ii) high-level remote method calls. A third approach using JavaSpaces (a Linda-like tuple space implementation) is currently being integrated into the system.

Programmers familiar with MPI can use Java MPI method calls for communication. They are similar to mpiJava [6] and provided by the Compute Service as system calls. The Compute Service provides the implementation via system classes. Once the subtasks are allocated, processes are connected by logical channels. The Compute Service provides transparent mapping task rank numbers to physical addresses and logical channels to physical connections to route messages. The design allows one to create a wide-area parallel system.

For some applications, MPI message passing is too low-level. Hence, we also designed a high level object-oriented communication mechanism that allows application programmers to develop tasks that communicate via remote method calls. As mentioned earlier, as the result of remote process creation, the client receives a task control proxy. This proxy is a reference to the spawned task/process and can be passed to other tasks. Consequently, a set of remote tasks can be configured to store references to each other in an arbitrary way. Tasks then can call remote methods on other tasks to implement the communication method of their choice. This design results in a truly distributed object programming model.

4. Results

Both the Batch Execution Service and the Compute Service have been implemented and tests on an international testbed have been performed. The trial runs demonstrated (i) the ease with which our services can be discovered dynamically with JGrid, (ii) the simplicity of job submission to native batch environments via the Batch Execution Service, and the (iii) ability of the Compute Service to run tasks of wide-area parallel programs that use either MPI or remote method call based communication.

Further tests and evaluations are being conducted continuously to determine the reliability of our implementations and to determine the performance and overheads of the system, respectively.

5. Conclusions and Future Work

This paper described our approach to support computational application in dynamic, wide-area grid systems. The JGrid system is a dynamic, service-oriented grid infrastructure. The Batch Execution Service and the Compute Service are two core computational services in JGrid; the former provides access to legacy batch execution environments to run sequential and parallel programs without language restrictions, while the latter represents a special runtime environment that allows the execution of Java tasks using various interprocess communication mechanisms if necessary.

The system has demonstrated that with these facilities application programmers can create highly adaptable, dynamic, service-oriented applications. We continue our work with incorporating high-level grid scheduling, service brokers, migration and fault tolerance into the system.

References

- [1] The JGrid project: <http://pds.irt.vein.hu/jgrid>
- [2] Sun Microsystems, *Jini Technology Core Platform Specification*, <http://www.sun.com/jini/specs>.
- [3] M. J. Litzkow, M. Livny and M. W. Mutka, "Condor : A Hunter of Idle Workstations" *8th International Conference on Distributed Computing Systems (ICDCS '88)*, pp. 104-111, IEEE Computer Society Press, June 1988.
- [4] Z. Balaton, G. Gombás, "Resource and Job Monitoring in the Grid", *Proc. of the Euro-Par 2003 International Conference*, Klagenfurt, 2003.
- [5] D. G. Feitelson and L. Rudolph, "Parallel Job Scheduling: Issues and Approaches" *Lecture Notes in Computer Science*, Vol. 949, p. 1-??, 1995.
- [6] M. Baker, B. Carpenter, G. Fox and Sung Hoon Koo, "mpiJava: An Object-Oriented Java Interface to MPI", *Lecture Notes in Computer Science*, Vol. 1586, p. 748-??, 1999.