High-Level Execution and Communication Support for Parallel Grid Applications in JGrid

Szabolcs Pota and Zoltan Juhasz

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

Abstract

This paper describes the high-level execution and communication support provided in JGrid, a service-oriented dynamic grid framework. One of its core services, the Compute Service, is the key component in creating dynamic computational grid systems that enable the execution of sequential and parallel interactive grid applications. A fundamental set of program execution modes supported by the service is described, then a programming model and its corresponding application programming interface is presented. The execution support of the service architecture is described in detail illustrating how remote evaluation and run-time task spawning are provided. The paper also shows in detail how task spawning and dynamic proxies can be used for a service-oriented communication mechanism for coarse-grain parallel grid applications.

1. Introduction

Traditionally Grid systems were created to connect geographically distributed resources for solving large computational problems. The computing resources were typically batch runtime systems running on clusters or parallel supercomputers. While the advent of service-oriented architectures is changing the grid landscape, the majority of production grids still only support this rather limited form of execution mode.

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

The goal of the JGrid project [1] is to investigate these problems and develop a novel service-oriented grid system that supports the above mentioned new features, provides a high-level, effective service-oriented programming model for developers. In the computational domain, it aims to create a dynamic computational service fabric in which applications can discover and use computing resources on-demand.

In this paper, we introduce one of its core services, the Compute Service, that enables the execution of sequential and parallel interactive grid applications. A fundamental set of program execution modes supported by the service is described, then a programming model and its corresponding application programming interface is presented. The execution support of the service architecture is described in detail illustrating how remote evaluation and run-time task spawning are provided.

The outline of the paper is as follows. In Section 2, a short overview of previous results related to our paper is given. Section 3 describes the functionality, execution modes and architecture of the JGrid Compute Service. Section 4 explains how the programming model interacts with the underlying infrastructure of the Compute Service covering remote evaluation and dynamic task spawning. Section 5 describes a high-level service-oriented communication mechanism for parallel grid applications and illustrates its use in a sample parallel image processing application. Section 6 summarizes our experience gained in using the JGrid Compute Service for executing sequential and parallel programs in various application areas. The paper ends with a summary of the results and planned future improvements.

2. Related Work

Next-generation grid systems, their architecture, functionality and programming questions are in the forefront of current grid research. While the current preference is for the OGSA/Globus-type Web Service technology based Grid architecture, due to its lack of

1-4244-0054-6/06/$20.00 ©2006 IEEE

high-level programming support, Java-centric systems are demonstrating alternative ways and strategies for building grid systems.

Early Java metacomputing systems (e.g. SuperWeb [2], Javelin [3]) relied on Java RMI but the inflexibility and scalability problems of RMI limited their applicability at a global scale. Several groups concentrated on creating a Java implementation of the MPI message passing standard aiming to create a platform-independent communication fabric that can be used effectively in Java computing grids mpiJava [4], MPIJ [5], JavaMPI [6]. The most successful implementations are MPJ [7] and MPJ/Ibis [8].

Several researchers designed and developed novel Java-based metacomputing systems, such as the Harness system [9] and its successor H2O [10] and ProActive [11]. The dynamic discovery support of Jini Technology [12] is central in several grid systems, such as ICENI [13], JISGA [14], JGrid [1], aiming to create more dynamic and service-oriented environments. 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.

3. Compute Service

The Compute Service is the key entity in supporting the execution of sequential and parallel grid applications within the JGrid framework. It allows clients to execute Java programs using virtualized remote resources, which can represent single, multi-processor computers or clusters. 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, in fact, is a special Java runtime system that (*i*) enables clients to execute programs in a secure and controlled way, and (*ii*) acts as core building block in dynamic grids that can execute interactive grid applications. This way, it complements our batch execution service (JGrid Batch Service) that integrates traditional batch execution functionality in JGrid using a service-oriented interface to batch execution systems (e.g. Sun Grid Engine [15] or Condor [16]).

3.1. Supported Program Execution Modes

Our aim during the design of the Compute Service was to create a universal compute engine that supports the majority of parallel and grid programming models and can be accessed via a single API instead of relying on the integration of several existing tools.

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 that are accessed via remote method invocation, and (*iv*) MPI-like message passing. This makes the Compute Service suitable for a wide range of sequential and parallel grid applications.

Synchronous remote evaluation is the fundamental form of remote task execution. A client task object containing data and executable code is serialized and transported to the Compute Service for execution. The executable code is downloaded to the service automatically relying on the mobile code support of the Java Platform, and the computed result is returned to the client upon successful execution. The synchronous remote evaluation is a blocking operation; consequently, it is most suitable for relatively short single or multi-threaded programs.

Asynchronous remote evaluation facilitates the execution of long running tasks. The task object is sent to the service similarly to the synchronous remote evaluation mode, but the result is returned as an asynchronous remote event. This mode of execution is very suitable for parameter sweep or master-worker type of applications with independent tasks, or graphical applications in which the user interface needs to be refreshed regularly without continuous polling.

The disadvantage of remote evaluation is that every method call requires the transfer of executable code. In such applications, where only the input parameter changes between successive method calls, this mechanism creates unnecessary overheads. Creating and deploying specific remote computational server objects could solve this problem but leads to a fixed configuration and contradicts the fundamental concepts of grid computing.

The *task spawning* execution mode of the service provides an efficient and elegant solution for dynamically creating remote processes in the Grid. Clients, in run-time, can create and deploy tasks in the Compute Service as remote server objects, and then interact with them via remote method calls. Using task spawning, executable code is transferred only once. With this mechanism, clients can turn the Compute Service into an arbitrary server executing application-specific computational tasks. The key mechanism enabling this execution mode is dynamic proxy generation described in detail in Section 4. The dynamic proxy can also be used for inter-task communication between spawned objects creating a high-level object-oriented communication model for parallel applications.

When Compute Services represent nodes of a cluster with low-latency, high-speed interconnect, a low-level, message-passing communication mode is more suitable for creating high-performance parallel applications. To

support these environments, the Compute Service provides MPI-style *message passing* communication primitives based on the MPJ (Message Passing interface for Java) API recommended by the Message Passing Group of the Java Grande Forum [17].

3.2. Service Architecture

After looking at the execution modes, we describe the internal architecture of the Compute Service. As shown in Fig. 1, the core of the service is a special thread pool in which client programs (tasks) execute. A task is received by the Task Manager that prepares the task for execution: access control checks, object unmarshalling, resource allocation and task adapter generation.

Tasks are not executed directly in a thread of the Thread Pool but wrapped in `Runnable` adapter objects that manage the entire lifecycle of the task. The Adapter is responsible for staging input files, returning output to the task submitter, handling exceptions and providing information to the internal monitoring subsystem. It also handles task control commands (cancel, suspend, resume), and provides host context to the running task. After initialization, the Task Manager places the adapters into the thread-pool for execution.

The minimum and maximum number of the threads within the pool is configurable, allowing the execution of one or more tasks simultaneously. The service uses its own security manager that ensures that tasks execute in isolation, so they cannot access resources of other tasks.

Initially, tasks are in the ready-to-run state and the *Scheduler* decides upon their exact execution order. The modular architecture of the Compute Service allows one to use schedulers most suitable for the given resource or workload. By default, tasks are scheduled by the built-in thread scheduler of the JVM. A prototype Lottery scheduler is also implemented to allow proportional share scheduling of the CPU among the tasks. Batch execution functionality can be achieved using an FCFS scheduler.

The *Monitor* keeps track of important events within the service in order to provide detailed information to the service providers.

The *Lease Management and Task Control Module* is responsible for automatic resource management. Each task is executed under a lease, i.e. it can only use service resources for allotted time duration. If the client cannot renew the lease due to some system failure (client error, broken network connection, etc.) before its expiration deadline, the task is cancelled and the resources it used are freed.

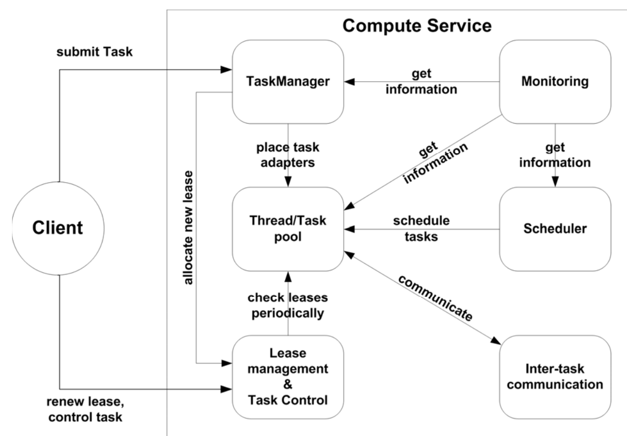

Figure 1. Internal architecture of the Compute Service.

The *Inter-task communication* module is an interface to the lower level transport layer that allows task objects executing at different locations to communicate via remote method invocations or message passing.

4. Programming Models and Their Execution Support

This section describes the programming models used in the Compute Service and illustrates how the execution of programs created with these models is supported by the Compute Service architecture. Using code examples we show the use of the application programming interface, and the internals of the corresponding execution mechanism.

The JGrid system is a Jini-based service-oriented grid, consequently each service is represented to the client program (and the developer) as a Java interface. The Compute Service interface is shown in Fig. 2. The two `execute()` methods represent the synchronous and asynchronous remote evaluation, with the `allocate()` method one can create message passing parallel applications, while the `spawn()` method is used for creating remote tasks.

Before using a service, the client must discover it using the Jini service discovery mechanism. In fact, the client discovers a Lookup Service in which it will search for services suited to its needs. A Jini service is described by its functionality (Java interface) and other, non-functional descriptors (attribute objects) that can provide other important information about the service (e.g. location, number and performance of processors, size of memory, etc.). This allows a client to discover and select only those services whose resources are suitable for the execution of the given task. By default Compute Services

```
interface ComputeService {
  public Object execute(Task t)
     throws RemoteException, ...;
  public TaskControl execute(Task t,
          RemoteEventListener result,
          long leaseDuration)
    throws RemoteException,...;
  public TaskAllocation allocate(
       Uuid AppID, Task task,
       TaskDescriptor desc,
       long leaseDuration)
    throws RemoteException,...;
  public Object spawn(Task task,
       long leaseDuration)
    throws RemoteException,...;
}
```

Figure 2. Description of the Compute Service in Java.

use attributes that advertise their number of processors, processor architecture, maximum memory size, physical location, software and vendor information.

Lines 1-12 of the code sample in Fig 3 show how to obtain a compute service reference via discovery. First a service template is constructed (lines 1-7) that consists of a unique id, a set of service interfaces and a set of attributes, and used for finding matching services [14]. In our example, only the service interface is specified implying that resource attributes are ignored. The result of the discovery step (lines 8-10) is a service proxy that implements the service interface (lines 11-12) and will delegate service method calls to the remote service.

4.1. Synchronous Remote Evaluation

Once a Compute Service reference is available in the program, the client can start executing its tasks. Our first example (Fig. 4) demonstrates the synchronous remote evaluation. In Line 1 a task object is instantiated that implements the mandatory `Task` interface. This is then sent and executed on the remote compute service using the synchronous execute method call (Lines 3-4). For space limitations we omitted exception handling

```
1. ServiceID id = null;
2. Class[] serviceTypes =
3.   new Class[] {ComputeService.class};
4. Entry[] attr = null;
5. ServiceTemplate template =
6.   new ServiceTemplate(id,
7.                        serviceTypes,attr);
8.  ServiceItem item =
9.   discoveryManager.lookup(template,
10.                             null);
11. ComputeService computeService =
12.   (ComputeService) item.service;
```

Figure 3. Programmatic discovery of the Compute Service.

```
1. Task task = new MyTask();
2. try {
3.   Double result =
4.     (Double)computeService.execute(task);
5.   System.out.println("Result: "+result);
6. }catch(Exception ex) {
7.   /* handle exceptions */
8. }
```

Figure 4. Executing a task using Synchronous Remote Evaluation.

On invoking the `execute` method, the task object is serialized and transferred to remote service transparently. The transport protocol between the client and the service, the authentication and authorization of clients are completely hidden to the programmer which greatly simplifies grid application development. The programmer can focus on the application tasks instead of system programming issues. On the service side, the arriving task is unmarshalled, then a worker thread is allocated from the thread-pool to the task, execution starts immediately, and the remote call is blocked until the execution is finished. The result is received as the return value of the method call.

4.2. Asynchronous Remote Evaluation

There are slightly more steps when executing a task asynchronously. As it is shown in Fig 5, first a remote event listener is created that will receive the result and exported so that it could accept remote method invocations (event delivery) from the Compute Service (lines 1-5). The Jini object export mechanism can be configured to use a suitable transport protocol between the proxy and the backend service object (TCP/IP, HTTP, SSL, HTTPS). The proxy of the exported listener object is then passed to the asynchronous `execute()` method (lines 9-11). The task instantiation is identical to the synchronous case.

The third parameter is the requested lease duration. The lease can be renewed via the control proxy object (line 12) returned as the result of the successful task submission (line 8). The controller can also be used to monitor the state of the running task, suspend or cancel the task.

Lines 1-12 of Fig. 6 depict a simple remote event listener class with a single `notify()` method. When the execution of the task is finished in the service, a remote event is created containing the result object, and sent back to the client by invoking the `notify()` method. The result then is extracted from the event (lines 5-6).

The execution of the task in the service is similar to the synchronous remote evaluation except the result in this

```
1. RemoteEventListener listener =
2.   new ResultListener();
3. RemoteEventListener listenerProxy =
4.   (RemoteEventListener)exporter.
5.                          export(listener);
6. Task task = new MyTask();
7.  try {
8.   TaskControl controller =
9.     computeService.execute(task,
10.                           listenerProxy,
11.                             60*1000);
12.   leaseMgr.renewFor(controller.
13.                               getLease())
14. } catch (Exception) {/* handle ex.*/}
```

Figure 5. Executing a task using Asynchronous Remote Evaluation.

case is sent back by the `TaskAdapter` object, and the `execute()` method will return immediately and the `ResultListener` object will wait for the arrival of the result in a separate thread.

4.3. Task Spawning

Consider an interactive rendering task where the client drives the rendering process with parameters calculated at run-time. A rendering task spawned on a Compute Service with which the client can interact can provide and elegant solution.

The code example in Fig. 7 outlines the structure and execution flow of this program. The renderer task (line 1) is spawned on the Compute Service (lines 3-5). The spawn() method is an asynchronous call that returns a special proxy object. The proxy is created dynamically in the Compute Service using Java reflection. The details of this process are described in Section 4.4.

The proxy implements several interfaces and thus needs to be casted to the type that defines the required functionality (lines 6-9). In our example, the client casts the proxy to the `Renderer` interface, consequently it can invoke the `render()` method on the proxy, and execute

```
1. public class ResultListener
2.   implements RemoteEventListener {
3.     public void notify(RemoteEvent ev) {
4.       try {
5.         Double result = (Double)ev.
6.           getRegistrationObject().get();
7.         System.out.println(result);
8.       } catch (Exception ex) {
9.         /* handle exceptions */
10.       }
11.     }
12. }
```

Figure 6. Listener class handling result collection based on remote events.

```
1.  Task process = new SceneRenderer();
2.  try {
3.    Object processProxy =
4.      computeService.spawn(process,
5.                          60*1000);
6.    TaskControl controlProxy =
7.      (TaskControl)processProxy;
8.    Renderer rendererProxy =
9.      (Renderer)processProxy;
10.   Scene scene1 =
11.     rendererProxy.render(0, 0, 10, 12);
12.   Scene scene2 =
13.    rendererProxy.render(10,10,100,100);
14. } catch(Exception ex) {
15.     /* handle exceptions */
16. }
```

Figure 7. Programming the task spawning process.

this functionality on the service. This provides the illusion of invoking the method locally on the task object, but in fact the method invocation and the parameters are delegated to the remote service that, in turn, invokes the `render()` method on the spawned task (lines 10-13).

4.4. Dynamic Proxy Creation

In our dynamic and interactive grid environment, we envisage programmatic clients that send executable objects to Compute Services discovered in run time. This cannot be achieved with existing distributed computing techniques.

Java RMI mandates the use of the `Remote` interface and relies on the developer and the `rmic` compiler to generate stubs (proxy). The use of the `Remote` interface differentiates remote code from local one in development time, and the remote object must be deployed on the server before starting the server. Objects implementing arbitrary interfaces cannot be deployed and exported as server objects in run-time.

Our approach builds on Java reflection and Jini proxy objects. We describe our solution with the help of the sequence diagram in Fig. 8. When the client spawns a task (1) on the compute service (the service proxy object is not shown for simplicity) than the URL classloader of the service downloads all the necessary class files from the client class server (2) making them available for execution and dynamic proxy generation. The task object is than wrapped in an adapter object (3) that is placed into the thread-pool and waits for incoming calls (not shown). Then, the Compute Service generates the dynamic proxy (4) that implements the `TaskControl` and all other task interfaces, and returns it to the client. This dynamic proxy will contain a special invocation handler and a reference

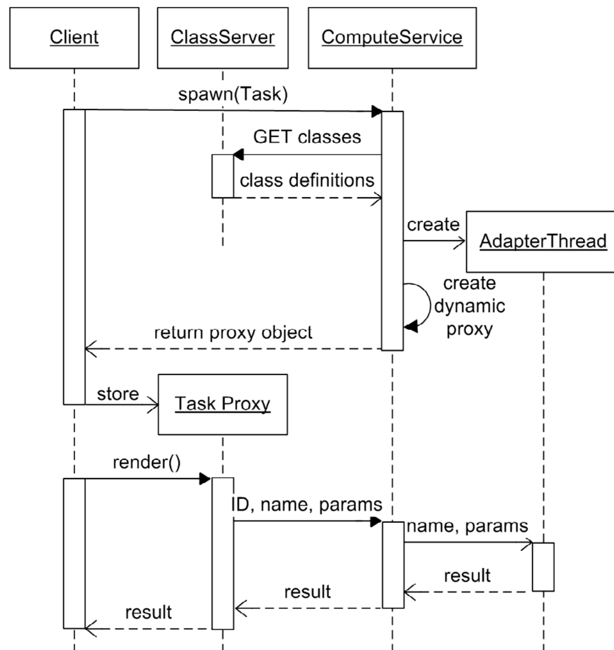

Figure 8. Simplified sequence diagram of the task spawning process.

to the remote service. The client then casts the returned proxy to the required interface and stores it (5) for later use (see lines 8-9 of Fig 7).

Since the user interfaces that the task implements are not remote interfaces and part of the Compute Service interface, their methods (e.g. `render()`, Fig. 8) cannot be invoked directly using traditional remote method invocation. The solution to this problem used in the Compute Service implementation is as follows.

The invocation handler of the task proxy will forward the name and parameter list of the task method and the task ID to the Compute Service (`invoke(ID, name, params)`, Fig. 8). Using these parameters, the Compute Service identifies the Task Adapter in the thread-pool that controls the execution of the client task. The adapter then is responsible for invoking the given method on the task using Java reflection and returning the result. Note that if the Compute Service runs in secure mode, then all method calls via the task proxy will also be secure, consequently, the Compute Service will ensure that clients can only invoke methods on tasks they own.

5. Service-Oriented Communication Support

Tasks spawned at runtime that act as user-generated application-specific services along with the dynamic proxy connection are ideal for creating coarse-grain parallel grid applications. Depending on the tasks used, developers can easily create various application topologies (farming, SPMD, workflow, etc.), as well as integrate non-computational services into their applications. (The Compute Service also supports explicit MPI-style message passing but due to space limitations, this is not discussed in this paper.)

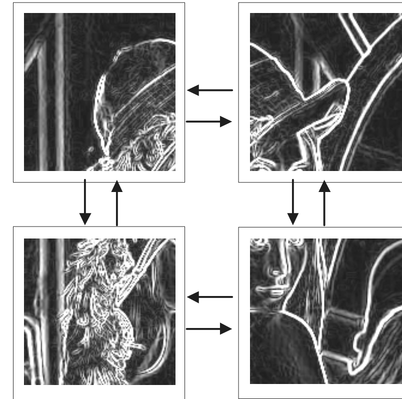

Figure 9. Logical view of the parallel image processing example.

We use a simple SPMD-style parallel image processing problem to illustrate the programming aspects of our service-oriented communication model and its execution support. The example performs simple image processing operations, e.g. edge detection, in parallel during which nearest-neighbour communication is required.

After the tasks are spawned, the client will configure the topology by exchanging task proxies between neighbouring tasks. Assuming four compute services creating a 2 x 2 processor array, this will result in the structure illustrated in Fig. 10. Once the configuration is set, the client distributes the image segments and starts the computation.

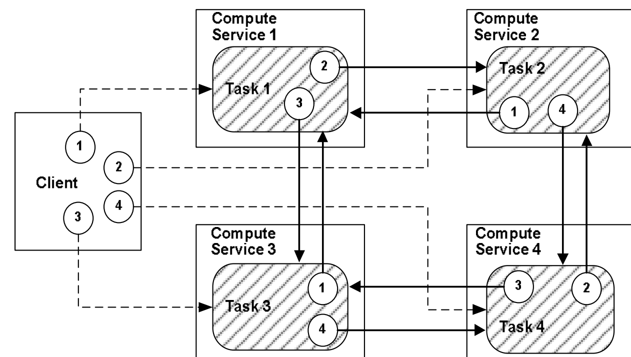

Figure 10. The structure of the example image processing application after spawning and configuration.

```
interface ImageProcessor {
  public void setNeighbour(
        ImageProcessor n, int position);
  public void setInput(byte[] imageData);
  public void process();
  public void setBorder(int borderWidth,
        byte[] imageData, int position);
}
```

Figure 11. Interface of the image processing task.

The fundamental step in creating this parallel program is the definition of the image processing task, which is done via a Java interface (Fig. 11). The interface in our example contains processing and communication methods. Method `setNeighbour()` is used to configure the topology by passing a neighbouring task proxy and its location to the task. The `setInput()` method passes the input data segment to the task before the call to `process()` starts the computation. The method `setBorder()` will be used by the tasks to send border information to each other.

The client program discovers compute services as before, creates the service tasks (lines 1-5, Fig. 12), spawns the tasks (lines 6-7), then starts configuring the parallel service architecture (lines 9-16). Once the configuration is done, it distributes the image parts (line 18) and starts the computation (line 19). The client collects the results asynchronously via the listener object passed to the task objects (not shown in code).

The outline of the processing taking place in the image processing task is shown in Fig. 13. The execution starts upon the call to `process()`. The task first exchanges the border values, then performs the imaging operation. The communication of border values consists of two steps: (*i*) in `sendBorders()` each task invokes the `setBorder()` method of its neighbour tasks to send its border values; (*ii*) in `recvBorders()` the task collects the border values sent by its neighbours via `setBorder()`. On completion the task creates the result object and returns it to the client through the listener proxy.

Although task spawning requires complex support from the underlying service infrastructure, it provides an intuitive programming model for parallel applications. The programmer can use the familiar Java models used in sequential programs and hence concentrate more on the application logic. The support for discovery enables clients to react to changes in the computing environment and add newly arrived services to its resource pool.

6. Applications of the Compute Service

The Compute Service of the JGrid project has been used in various tests and projects to experiment with its

```
1.  ImageProcessor[] tasks = new
2.      ImageProcessorImpl[4];
3.  for(int i=0;i<4;i++){
4.    tasks[i] = new
5.      ImageProcessorImpl(i,listenerProxy);
6.    tasks[i] =
7.     services[i].spawn(tasks[i],60*1000);
8.  }
9.  tasks[0].setNeighbour(tasks[1],RIGHT);
10. tasks[0].setNeighbour(tasks[2],BOTTOM);
11. tasks[1].setNeighbour(tasks[0],LEFT);
12. tasks[1].setNeighbour(tasks[3],BOTTOM);
13. tasks[2].setNeighbour(tasks[0],TOP);
14. tasks[2].setNeighbour(tasks[3],RIGHT);
15. tasks[3].setNeighbour(tasks[1],TOP);
16. tasks[3].setNeighbour(tasks[2],LEFT);
17. for(int i=0;i<4;i++){
18.   tasks[i].setInput(getImagePart(i));
19.   tasks[i].process();
20. }
```

Figure 12. The client program of the image processing examples.

```
1.  sendBorders();
2.  recvBorders();
3.  doWork(i);
4.  Object result = new
5.    MarshalledObject(imageData);
6.  listenerProxy.notify(new
7.  RemoteEvent(null,taskNumber,0,result));
```

Figure 13. The execution steps of the image processing task.

programming model, study its performance, and evaluate the developer support for creating high-level grid applications.

A financial Monte Carlo simulation project demonstrated that developers with little experience in grid computing could develop grid programs and use our system for its execution. The execution runs also demonstrated the reliability of the Compute Service implementation.

Based on the example programs of the ProActive project [12] we have investigated how difficult it is to port a third-party Java program onto the JGrid Compute Service. We have ported the ray tracing and N-body simulation sample programs and found that except adding additional interfaces for exported objects and changes to the deployment system, the application program did not require modifications. In contrast to ProActive, in JGrid we did not need static resource descriptors to execute the program; they could execute dynamically on available services discovered at run-time.

We have also examined how a sequential but compute-intensive, long-running applications can execute on the Compute Service. After minor modifications the system

could execute BioJava [18] based bioinformatics programs with execution time of several hours per run.

Due to its service-oriented design, the Compute Service can work together with other, non-computational services as well. As a demonstration, we have created a media streaming service that used could use a dynamically changing number of Compute Services hosting streaming server tasks providing a scalable internet media distribution mechanism.

7. Conclusion

This paper described the Compute Service of the JGrid project that is a fundamental building block for creating computational grid applications. We showed that it allows the construction of dynamic and interactive grid applications. A range of execution modes is supported by the service that is presented to the developers in a compact Java programming API. The service also supports explicit message passing and service-oriented communication for parallel programs. We showed in detail how these execution modes are mapped to and supported by the internal architecture of the service.

We argue that future grid systems need new styles of execution support that complements traditional batch execution and allows users to integrate non-computation services with computational components using a high-level programming abstraction. The JGrid project and its rather universal Compute Service is a possible step into this direction. Future work on the Compute Service include further improvements of the service in functionality and performance, and creating further demonstration applications to illustrate the use and benefit of its execution modes.

Acknowledgment

The authors thank Krisztian Kuntner and Mark Magyarodi for their contribution to the discovery and security architecture of the JGrid environment.

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

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

[3] 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

[4] M. Baker, B. Carpenter, G. Fox, S. H. Ko, "mpiJava: An Object-Oriented Java interface to MPI," in *Proc. Intl. Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*, 1999, LNCS, Springer Verlag, Heidelberg, Germany

[5] G. Judd, M. Clement, Q. Snell, V. Getov, "Design issues for efficient implementation of mpi in Java," in *Proc. ACM Java Grande Conference*, 1999, pp. 58-56

[6] S. Mitchev, V. Getov, "Towards Portable Message Passing in Java: Binding MPI," in *Recent Advances in PVM and MPI*, Lecture Notes in Computer Science, 1997, Vol. 1332, pp. 135-142.

[7] M. Baker. B. Carpenter, A. Shafi, "MPJ: Enabling Parallel Simulations in Java," DSG Technical Report DSGTR19062005, June 2005

[8] M. Bornemann, R. V.v. Nieuwpoort and T. Kielmann, "MPJ/Ibis: a Flexible and Efficient Message Passing Platform for Java," in *Proc. EuroPVM/MPI 2005*, Eds. B. DiMartinoet al., vol. 3666, pp. 217-224.

[9] 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.

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

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

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

[13] 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).

[14] 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.

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

[16] 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.

[17] B. Carpenter, V. Getov, G. Judd, A. Skjellum and G. Fox, "MPJ: MPI-like message passing for Java," *Concurrency: Practice and Experience*, Nov. 2000, vol. 12, No. 11, pp. 1019-1038.

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