In: Cyberinfrastructure Technologies and Applications ISBN
Editor: Junwei Cao

Chapter

GRID RESOURCE BROKER FOR SCHEDULING COMPONENT-BASED APPLICATIONS ON DISTRIBUTED RESOURCES

Xingchen Chu, Srikumar Venugopal, Rajkumar Buyya

Grid Computing and Distributed Systems Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia

Abstract

This chapter presents the design and implementation of seamless integration of two complex systems component-based distributed application framework ProActive and Gridbus Resource Broker. The integration solution provides: (i) the potential ability for component-based distributed applications developed using ProActive framework, to leverage the economy-based and data-intensive scheduling algorithms provided by the Gridbus Resource Broker; (ii) the execution runtime environment from ProActive for the Gridbus Resource Broker over component-based distributed applications. It also presents the evaluation of the integration solution based on examples provided by the ProActive distribution and some future directions of the current system.

1. Introduction

Grids have progressed from research subjects to production infrastructure for real world applications. Grids such as TeraGrid[1], LCG[2] and EGEE[3] are being used by scientists to run large-scale and data-intensive simulations thereby allowing them to explore larger parameter spaces than ever before. Current use of Grid technology mostly extends the batch-job paradigm wherein a job encapsulating the application requirements is submitted to a Grid and the results of the execution are then returned. However, the capabilities of Grids are better explored by active applications that can dynamically vary the parameter space based on different scenarios. This requires programming models that are able to go beyond the passive

{xchu, srikumar, raj}@csse.unimelb.edu.au

nature of batch-jobs and consider the Grid as a unified platform rather than a loose aggregation of dispersed heterogeneous resources. In this regard, component-based application development is a promising candidate as it allows Grid applications to be constructed out of loosely-coupled independent components that converse with each other through standard interfaces.

Many component-based software frameworks for Grids have therefore been developed around the world. Examples of such frameworks are ProActive[4], XCAT[6] and SCIRun[7]. Such frameworks have to provide functions for handling the heterogeneity and dynamicity of Grid resources. These functions also have to be abstracted from the programming environment so that developers are able to concentrate on building applications. In this chapter, an integration of ProActive framework and the Gridbus Broker[8] is presented so that the former is able to take advantage of the latter's abilities in resource allocation, job scheduling and management, and support for different middleware. It discusses the challenges involved, and presents the integration process in detail. It also validates the integration by running the existing examples without any modifications. Finally, the chapter concludes and provides directions for future research.

2. Background Knowledge

Before we present the integration solution proposed in this chapter, it would be better to mention about some technical details of the scheduling infrastructure for both systems in order to fully understand the reason why integration is necessary and important.

2.1. ProActive Grid Scheduler

ProActive[4] is a software framework for developing and deploying parallel and distributed applications. The programming model and APIs provided by ProActive greatly simplify the development and execution of those applications. Moreover, it provides a component framework as a reference implementation for GCM (Grid Component Model) for

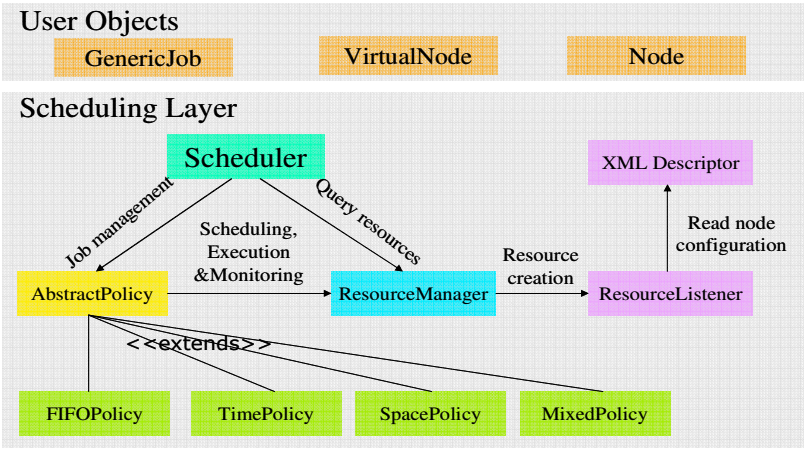

Figure 1 ProActive Grid Scheduler

programming Grid applications as reusable components.

Among a lot of services provided by the ProActive framework such as execution environment, standardized deployment and component model, there is one very important service: Grid scheduler service which promises to schedule applications over heterogenous Grid resources. Let us take a look at how the Grid scheduler has been implemented within ProActive framework, as shown in Figure 1. The upper layer user objects represent the normal terms such as jobs and resources. The Grid scheduler as well as the resource manager and job manager (known as the AbstractPolicy) are all in the scheduling layer. The actual scheduling algorithms are implemented as various policies which extends the AbstractPolicy class. As can be seen from the figure, four kinds of policies are provided by the ProActive framework.

Although the Grid scheduler implementation fulfils the basic requirement for simple scenarios, there is a lack of advanced scheduling algorithms that can be very helpful for scenarios requiring much more complicated resource allocation policies such as economic-based and data-intensive scheduling policies. This is where the Gridbus resource broker can help and the underlying rationale of this integration work.

2.2. Gridbus Broker Scheduling Infrastructure

Gridbus Broker[8] is a user-level middleware that mediates access to distributed resources by discovering available computational resources, and scheduling jobs to best suitable resources. Users can choose various scheduling policies including simple round-robin and more advanced data-intensive or economy-based resource allocation algorithms[9].

As our integration concentrates on the scheduling infrastructure, it will skip other details related to the broker's application creation and job execution. The scheduling infrastructure provided by the Gridbus broker is composed of four main components including the Scheduler, Dispatcher, ServiceMonitor and JobMonitor, as shown in Figure 2. Each object runs as a separate thread that exchanges information via the Broker's storage infrastructure. Various scheduling algorithms have been implemented by subclasess derived from the Scheduler class. The Dispatcher is responsible for dispatching jobs to heterogeneous

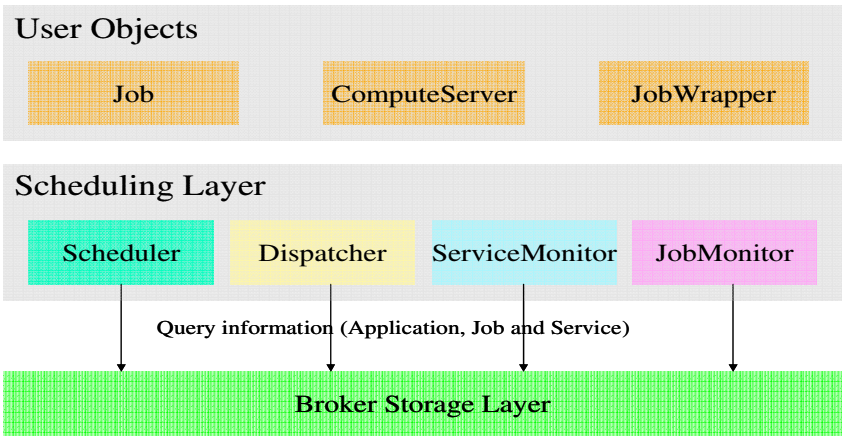

Figure 2 Overview of Gridbus Resource Broker Scheduling Infrastructure

resources. The SeviceMonitor and JobMonitor are responsible for monitoring resources and job status, respectively.

As the broker's point of view, the terms such as job and resources are described using Job and ComputeServer. The JobWrapper provides lifecycle methods that need to be invoked by the broker when the job status is changing. Subclasses of ComputeServer and JobWrapper always come out in pairs which provide a specific runtime environment for different grid middleware such as Globus Toolkit[10] and Alchemi[11]. As we will explain later in the chapter, we have implemented a specific ComputeServer and a JobWrapper for ProActive framework.

3. Integration Challenges

we mentioned, the objective of the integration is to make ProActive use the Gridbus Broker scheduling infrastructure. It in turns has positive effects on both sides. ProActive is able to leverage the economy-based and data-intensive scheduling algorithms provided by the Gridbus Broker. Gridbus Broker is also able to utilise the programming environment especially the component-based programming concept provided by ProActive. However, it is not an easy job as the two systems are both complex and have large codebase. There are several challenges we need to consider in designing a mature and realistic integration solution.

The first challenge we need to identify is on the potential impact of the integration on both systems. There should be no or minimum impact on both systems: (i) each complex system should have no or as little knowledge of each other as possible, and (ii) each complex system only need to concentrate on its own terms and conditions. It means that we should avoid direct dependencies between each other as much as possible, and the data or object representation of each system should remain the same.

The second challenge for the integration is the reuse of existing infrastructure and codebase provided by both systems. The goal is to maximise reusability and avoid modifying the existing source code. This can be divided into two aspects: (i) the scheduling infrastructure provided by Gridbus Broker should be reused without changing existing source codebase within the Broker, and (ii) the deployment and runtime execution infrastructure provided by ProActive should be reused to deploy and execute the applications.

In order to validate that the integration solution we have provided can meet all the challenges presented here, all the legacy applications that used to run with existing ProActive scheduling policy should work with the Broker's scheduling policy without recompiling and redeveloping the source code.

4. System Implementation

As shown in Figure 3, the integration solution we have proposed is purely based on basic object-oriented design principles such as inheritance and delegation. The top layer is the existing ProActive scheduling infrastructure which contains a scheduler, a job manager (the policy class) and a resource manager along with the resource listener that is used to acquire resources. The bottom layer is our proposed implementation for the integration. The

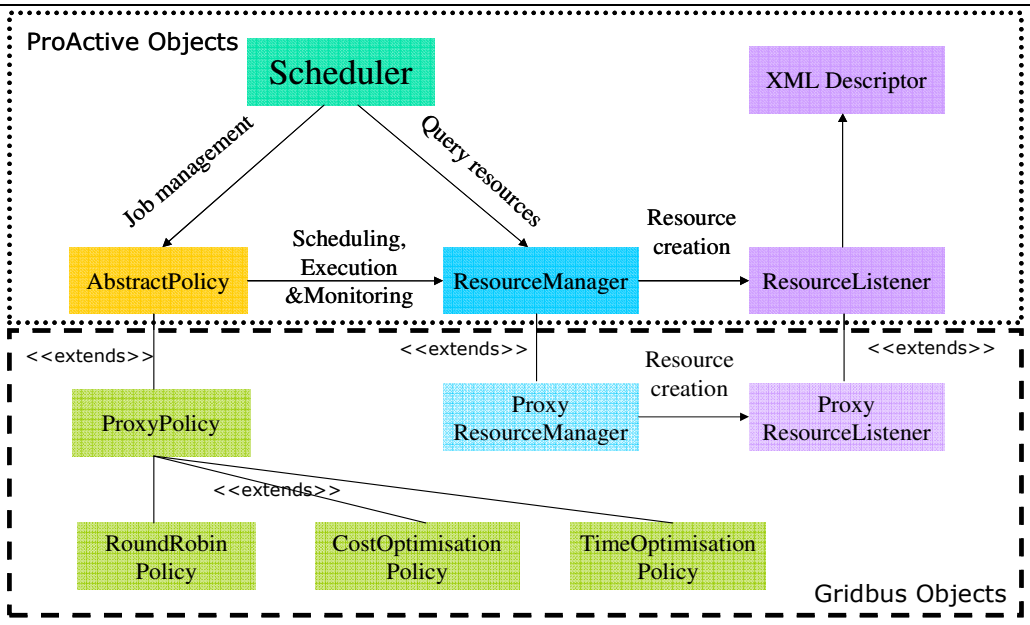

Figure 3. ProActive and Gridbus Broker Integration

ProxyPolicy, ProxyResourceManager and ProxyResourceListener extend the ProActive classes respectively. They are proxies and the links between ProActive and Gridbus Broker. Instead of using the ProActive objects, the proxy objects are responsible for initializing the relative components that exist in the Broker codebase. Various proxy policies can be implemented independently in order to enforce certain scheduling algorithms such as Round-Robin, and Cost/Time optimization that have already been implemented in the Broker.

In order to seamlessly integrate the two systems together, we have to divide the implementation into two different aspects: (i) new classes that should belong to the ProActive codebase are responsible for replacing the existing scheduling infrastructure provided by ProActive dynamically, and (ii) new classes that should be part of the Broker codebase are responsible for wrapping the ProActive terms into the Broker's terms that can be scheduled and managed by the Broker's scheduling infrastructure.

4.1. Proxies for ProActive scheduling layer

Let us first examine the objects created belonging to the ProActive codebase.

- ***Proxy Objects*** : they bridge ProActive classes with Gridbus Broker classes, and internally delegate the responsibility to the corresponding behavior in the Gridbus Broker.
- ***ProxyPolicy*** : it is a subclass of ProActive AbstractPolicy class which is responsible for initialising the Broker runtime including JobMonitor, ServiceMonitor, Scheduler and Dispatcher. Besides, it also overrides the ProActive's job management functions by delegating to the proper broker functionality. Its subclasses provide the information required by the Broker to match the scheduling algorithms and create an appropriate scheduler for a certain policy. For example, the RoundRobinPolicy is used to guide the Broker to utilise the basic round-robin scheduler that has already been implemented in the Broker scheduling infrastructure.

- ***ProxyResourceManager*** : this class extends the ProActive ResourceManager class. It overrides the ProActive's resource management functions using Broker's functionality.
- ***ProxyResourceL*istener**: as ProActive utilises the ResourceListener to acquire the resources from the XML deployment descriptor, the ProxyResourceListener simply overrides those implementation and add those resources into the Broker system by converting the terms from ProActive (such as virtual node and node) to the terms recognized by the Broker.

The proxies we have implemented that are fully compliant to ProActive scheduling infrastructure provide an alternative approach for the ProActive community that is promised to utilise Gridbus Resource Broker's advanced scheduling algorithms and infrastructure.

4.2. ProActive Wrappers for Gridbus Broker

We have already mentioned about the proxies for the ProActive's scheduling infrastructure that delegate ProActive's responsibilities to the Gridbus Broker. However, it is just one side of the coin, we also need to look at another side of the problem. As both systems use their own terms representing the same set of information such as resources and jobs. We have to also provide wrapper classes that translate different terms from one system to another system. As our goal of the integration is to utilise the Broker's advanced scheduling infrastructure for ProActive, it is very important to translate the terms from ProActive into the terms that the Broker can understand and manage.

As we have already seen the terms ProActive defined for the resources (virtual node, node) and jobs (GenericJob), an entity mapping has been implemented by wrapping proper objects into the Broker's terms as follows:

- ***ProActiveJob*** : this class extends the Broker's Job class and maintain a ProActive's GenericJob object internally. Whenever a GenericJob is created by the ProActive's job submission module, a ProActiveJob object will be created and queued for schedule by the Broker system.
- ***ProActiveComputeServer*** : it is the subclass of the ComputeServer class which is a representation of the compute resource for Broker. It internally keeps track of one virtual node and a list of nodes with that virtual node. Once the resource is created by ProActive deployment component, the ProActive compute server will be created by giving the virtual node and a list of nodes associated with the virtual node.
- ***ProActiveJobWrapper*** : the job wrapper is a unique class for the Broker that is used to execute a task using a given way. It provides lifecycle methods for a task that will be managed and executed by the Broker's scheduling infrastructure. This particular implementation for ProActive is responsible for setting up the ProActive runtime environment and execute the job using the API provided by ProActive runtime.

4.3. Assembling the System

We have provided proxies for the ProActive system and also wrappers for the Broker system, the remaining work is to assemble the two independent parts into an integrated environment. Figure 4 demonstrates how the whole system works together.

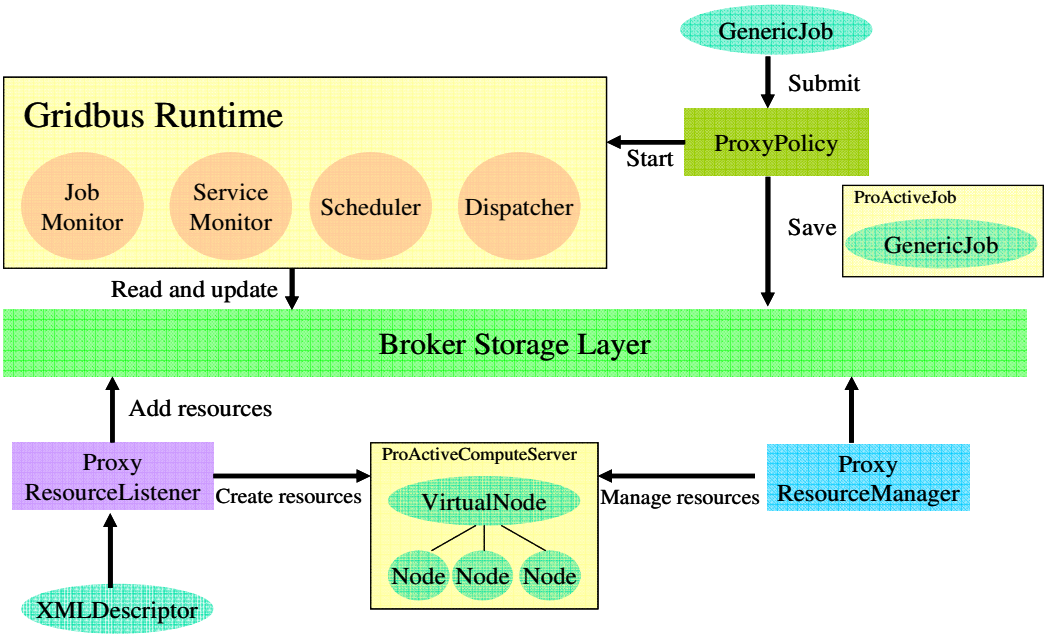

Figure 4. How the integration

As can be seen from the figure, the center part of the system is the Broker storage base, all the information related to the jobs and resources are pushed into the storage. It is the core part for decoupling ProActive and the Broker. The proxies at the ProActive side are totally unaware of the Broker runtime environment because they do not need the direct communication with those components such as scheduler, dispatcher and job/service monitor. The Broker on the other hand only need to retrieve information such as jobs and resources in the same manner without even knowing who has given those information.

The entire workflow can be described into three independent parts:

- ***Resource Acquisition and Management***: the proxy resource manager is activated at the very beginning when the ProActive scheduler starts and initialises the proxy resource listener to acquire the resources from the XML deployment descriptor. The listener will create the ProActiveComputeServer objects based on the virtual node and nodes deployed by the XML deployment descriptor, and save those compute server objects into the Broker storage system. The proxy resource manager simply manages those resources via the Broker storage system.
- ***Job Submission*** : the ProActive user is able to submit their applications in terms of GenericJob objects to the ProActive scheduler just as before. Once the GenericJob objects have been submitted to the system, the proxy policy class will create the ProActiveJob objects that wrap each GenericJob object. The ProActiveJob objects are saved into the Broker storage system for later scheduling.
- ***Job Scheduling, Monitoring and Execution*** : this part is fully controlled by the Broker scheduling infrastructure and totally transparent to the ProActive system. The proxy policy class will intialise the Broker runtime before any job submissions occur. The four Broker components including the Scheduler, Dispatcher, Job and Service Monitor will run as separate threads in the system. They periodically poll the storage system and retrieve or update required information. For example, the scheduler will try to find out all the newly submitted jobs and all the available resources, and

schedule jobs over best resources based on the scheduling algorithms. If no job or no available resource has been found, it just waits for a certain time and polls it again in the next round. Similarly, the other three components work in the same manner.

The three independent parts work simultaneously without knowing each other. They obtain required data by querying the storage system, and update certain information accordingly. This separation enables decoupling for both systems, and they just concentrate on what they need to deal with.

5. Validation

In this section, we look at how to validate our integration that meets the objective and solves the challenges, by running an application developed using the ProActive framework and being scheduled via the Gridbus broker's scheduling infrastructure. The integration solution is valid unless the following two conditions are satisfied:

- Legacy applications developed using ProActive should work without changing and recompiling the source code. The only acceptable modification will be the configuration file that needs to be used to connect to the Grid scheduler.
- The dependencies between each system should be minimized. Users from ProActive should be able to dynamically choose which scheduler to use, either the existing one or the Gridbus broker's scheduler. And the Gridbus broker should not be aware of any ProActive runtime information.
- Reuse the existing infrastructure at both sides without adding new features. The job scheduling should be delegated to the Gridbus broker, and the ProActive's application deployment and job execution need to be reused.

C3D Application

The application we are using is the C3D which is a Java benchmark application measuring the performance of a 3D raytracer renderer distributed over several Java virtual machines. Image rendering is the process of generating an image from a model by means of computer programs. The model is described as a three dimensional object that can be understood by computer programs. Information such as geometry, texture, lighting, viewpoint and shading may be carried by the objects. Ray tracing is just one of the algorithms that can be used to render the image, which is a brute-force method calculating the value of each pixel [12]. In general, the ray tracing rendering approach is too slow to consider for the realtime image rendering. The C3D application makes use of the distributed rendering engines working in parallel which is able to accelerate the speed of the normal ray tracing rendering approach. Users can interact through messaging and voting facilities so that they can choose a scene that is rendered using a set of distributed rendering engines.

Launch the Application

```
public class StartScheduler{
   public static void main(String [] args){
     String policyName =
           "org.objectweb.proactive.scheduler.gridbus.policy.RoundRobinPolicy";
     String resourceManager =
           "org.objectweb.proactive.scheduler.gridbus.ProxyResourceManager";
     Scheduler.start(policyName,resourceManager);
   }
}
```

This application has been provided by the ProActive distribution. We have configured a scheduler node with 5 nodes (pure JVM) which uses the default round-robin scheduler and asks the C3D application to connect to the scheduler node. We also monitor this application via the IC2D monitor GUI, as shown in Figure 5. From the user's point of view, there is nearly no difference between this run and the one without the Gridbus broker except that the RoundRobinPolicy and ProxyResourceManager have been presented in the scheduler virtual node show on the IC2D panel.

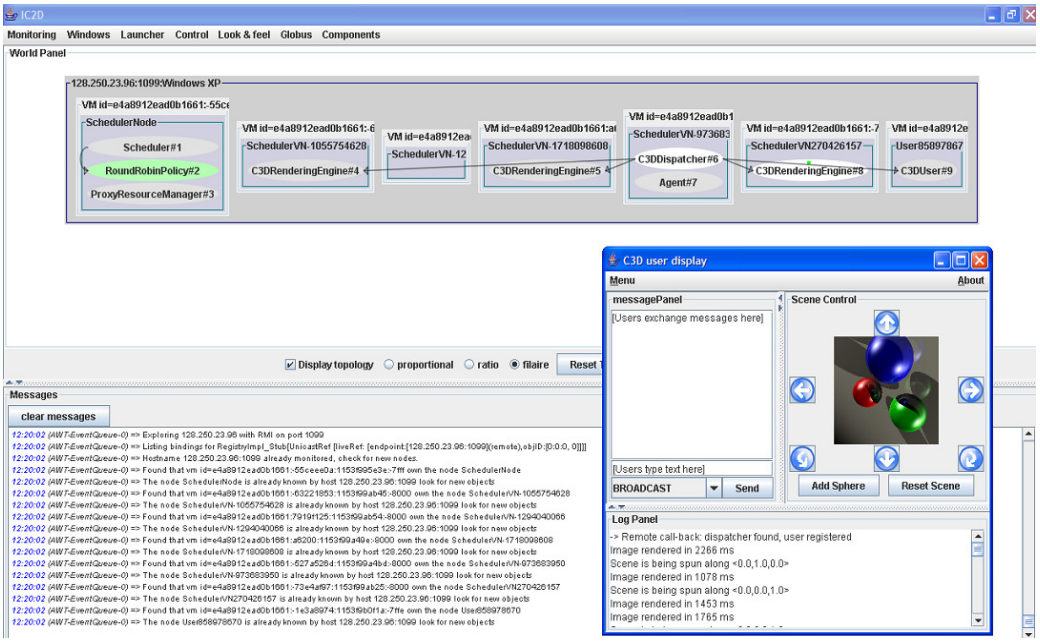

Figure 5 C3D application using Gridbus Broker in IC2D monitor

As the application can be run without modifying any source code, the only thing we need is to connect the application to the scheduler node via an XML configuration file provided by the ProActive's deployment infrastructure. It satisfies the first condition we have set for the validation.

The dependencies between each system have been minimized as well. The following code shows how to start the grid scheduler with the broker's proxy classes.

The ProActive still uses it grid scheduler and it does not know about the existence of the broker. The implementation classes related to the broker have been dynamically loaded via Java reflection by giving the name of those classes. Different policies can be easily applied by giving a different class name that delegates to the appropriate policy implementation. So we could say that the integration also fulfils the second condition.

The last condition has also been satisfied by our implementation. All the implementations we have done for the integration are simply providing proxies and wrappers for both system. No new features are introduced into either of them.

Since all the conditions we have set for the validation have been achieved, we could conclude that our proposed integration not only works with the two complex systems, but is also a valid solution that does not comprise the user's prospective.

6. Remarks of the Integration

To evaluate a successful integration solution, only considering whether it is working or not is not sufficient. It is very important and necessary to look at what impacts the integration has made to each of the independent systems. In this section of the chapter, we will examine the impacts that our work brings to each system.

6.1. Impacts on ProActive

At the ProActive side, the integration solution provided in this chapter has the least impacts on the original ProActive system both at the system level and the application level. First of all, it is necessary to check the impacts on the ProActive's system level functions. In terms of the changes of the source code for the ProActive existing codebase, it only requires tiny changes in the scheduler class that has been modified to add a new overload start method supporting dynamic configuration of proxy resource manager and proxy policy. Other than that, the most relevant and important infrastructures such as the deployment service, the runtime execution environment and the scheduler services have been reused. In other words, the reusability of the system has been greatly achieved with this integration solution which has minimised the impacts on the system level.

Another aspect to look at is the application level, or the necessary changes that might be required by the client applications developed using the ProActive framework. As the previous section has shown the C3D application is totally are unaware of the existence of the Gridbus broker at all. This is achieved by decoupling the process of creating the resource manager and policy from the Scheduler class via the new start method. It is very important to understand that the implementation of the broker's scheduling service including the proxy resource manager and proxy policy are plugged at runtime via the modified scheduler class not at compile time which eliminates the recompilation process of existing legacy applications requiring the original scheduling service. This decoupling helps to reduce the unnecessary dependencies between the client and the scheduling service, which means that the client applications can safely ignore the existence of the underlining scheduling service.

The last notable aspect is to check how many extra dependencies (extra jar files) that are required by the ProActive system to perform the scheduling via the Gridbus broker. this is so important to mention is because the integration approach adopted can largely affect the size of the jar files required by certain configuration. With inappropriate approach, the runtime system will require a huge amount of jar files on the classpath and it will cause more problems such as class conflicts if the two systems use the same open source products with different versions. The integration solution in this chapter considers this problem. Only one Java jar file which is the broker runtime library is needed for ProActive to work with the Gridbus broker.

6.2. Impacts on Gridbus Broker

The impact of this integration on the broker is even lesser. The architecture of the Gridbus broker is very flexible in which the scheduler plays a core role, and various types of

runtime Grid middleware such as Globus, Alchemi, SGE, Condor and PBS can be plugged into its runtime environment at runtime via the configuration. The broker itself acts as a middle-man who is responsible for matching jobs to heterogeneous resources. In order to implement a middleware that can be supported by the Gridbus broker, only few classes are required to be extended including the ComputeServer class representing the grid resource and the JobWrapper class which contains the job execution logic for that particular middleware.

The integration solution extends this idea of the broker and makes ProActive as another type of Grid middleware. The benefit of this decision is that all the classes related to ProActive can be an independent project that does not need to change the original broker's source. At runtime, the broker is totally unaware of the existence of ProActive and just treats it as a type of normal Grid resource. The broker schedules and dispatches jobs as before without worrying about the terms defined in ProActive such as GenericJob, Virtual Node and Node.

7. Conclusion and Future Work

This chapter presented the design, implementation and evaluation of an integration solution for enabling schedule component-based applications on global Grids by utilising the Gridbus resource broker. The integration solution provided in this chapter strictly follows the object-oriented design principles, which seamlessly makes the two complex systems collaborate without knowing each other. Any legacy applications running under the original runtime can still work without modifying and recompiling the source code. The glue between the two systems has been provided via the configuration file and the execution environment loads it dynamically at runtime through the existing deployment service. To validate whether the integration meets the objective, a legacy 3D raytracer renderer application along with the ProActive monitor has been tested via the integration.

From the validation, this chapter concludes that the integration solution has minimum impacts on both systems. They all can work independently and the community users of the ProActive can freely choose either to use the Gridbus broker scheduling service or not to worry about it at all. Although this integration solution has demonstrated the feasibility and potential benefits of scheduling component-based applications via Gridbus broker, a much closer integration is required in the future to facilitate the economy-based scheduling services. The job model of the ProActive has to be extended to support QoS parameters such as budget and deadline via a configurable way. Moreover, the provided implementation only focuses on the RMI runtime provided by ProActive, a much more comprehensive testing needs to be done with other types of runtime environments provided by ProActive.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. This work is fully supported by the international linkage research grant from the Australian Department of Innovation, Industry, Science and Research (DIISR).

References

[1] R. Pennington, Terascale Clusters and the TeraGrid , Invited talk, *Proceedings for HPC Asia*, Dec 16-19, 2002, pp. 407-413.

[2] I. Bird, L. Robertson and J. Shiers, Local to Global Data Interoperability - Challenges and Technologies, 20-24 June 2005, CERN, Geneva, Switzerland.

[3] EGEE, Enabling Grids for EScience, http://www..eu-egee.org.

[4] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, A Component Based Services Architecture for Building Distributed Applications. In *Proceedings of the 9th IEEE international Symposium on High Performance Distributed Computing,* Pittsburgh, PA, USA, August 2000, IEEE Computer Society, Washington, DC, 2000.

[5] F. Baude, L. Baduel, D. Caromel, A. Contes, F. Huet, M. Morel and R. Quilici, Programming, Composing, Deploying for the Grid, *GRID COMPUTING: Software Environments and Tools*, Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.

[6] D. Gannon, et al., Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. *Cluster Computing* 5(3):325-336, Springer-Verlag, Berlin, Germany, 2002.

[7] S. G. Parker, and C. R. Johnson, SCIRun: a scientific programming environment for computational steering. *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC '95)*, San Diego, California, United States, December 04 - 08, 1995. ACM Press, New York, NY, 1995.

[8] S. Venugopal, R. Buyya and L. Winton, A Grid Service Broker for Scheduling e-Science Applications on Global Data Grids, *Concurrency and Computation: Practice and Experience*, 18(6): 685-699, Wiley Press, New York, USA, May 2006.

[9] S. Venugopal and R. Buyya, A Deadline and Budget Constrained Scheduling Algorithm for eScience Applications on Data Grids, *6th International Conference on Algorithms and Architectures for Parallel Processing*, Oct. 2-5, 2005, Melbourne, Australia.

[10] I. Foster and C. Kesselman, The Globus Project: A Status Report, *Proceedings of IPPS/SPDP'98 Heterogeneous Computing Workshop*, 1998, pp. 4-18.

[11] A. Luther, R. Buyya, R. Ranjan, S. Venugopal, Alchemi: A .NET-Based Enterprise Grid Computing System, *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*, June 27-30, 2005, Las Vegas, USA.

[12] G. H. Spencer and M. V. R.K. Murty (1962). General Ray-Tracing Procedure . *J. Opt. Soc. Am.* **52** (6): 672–678.