Utilizing Jini Features to Implement a Multiagent Framework for Performance-based Resource Allocation in Grid Environment

Sarbani Roy
Department of Computer Science and Engineering
ST. Thomas College of Engineering & Technology
Kolkata 700 023, India
sarbani_roy77@yahoo.co.in

Nandini Mukherjee
Department of Computer Science and Engineering
Jadavpur University
Kolkata 700 032, India
nmukherjee@cse.jdvu.ac.in

Abstract **-** *Jini is able to support applications in heterogeneous, dynamic computing environments and is also being used to develop agent-based applications. This paper presents the design of a multiagent framework for resource brokering and management in a Grid environment and implementation of a part of it utilizing Jini features. The multiagent framework provides services like resource brokering and allocation, analysis of performance-monitoring data, local tuning of applications and also rescheduling in case of any performance problem on a specific resource provider. Part of the framework makes use of mobile agents for scheduling and rescheduling purposes.*

Keywords: Grid environment, Multiagent framework, Resource Brokering, Jini, Mobile agents.

1 Introduction

Grid environment facilitates access to high-performance resources like supercomputers, clusters of workstations, databases and scientific instruments located at geographically distributed sites and connected through high-speed networks. The resource pool is heterogeneous and dynamic and any time a new resource can join the Grid or an existing resource can withdraw. The system loads and status of the resources also change frequently. Thus an application running on a Grid needs to be adaptive to its current execution environment so that it can efficiently utilize the available resources.

The user has no or little knowledge about the current state of a Grid. Consequently, mapping the resource requirements of a particular job onto the actual physical resources cannot be done at the user or at the application level. Thus performance based resource brokering and management at the middleware level is crucial in Grid environment in order to achieve optimum performance of the jobs. The general approach that we undertake is to regularly monitor the performance of the infrastructure and the job and to take appropriate decision on the basis of the specific situation. These decisions include performance tuning of the job by using optimization techniques at local level, adding more resources to its execution environment or simply migration of the job or its components.

Our work focuses on a multi-agent framework [19] for resource brokering and performance monitoring and tuning of computational jobs running in grid environment. The framework aims at providing adaptability to the run-time environment of a compute-intensive job running on a Grid. The framework proposed here is general enough to cope with any distributed system with some extent of dynamism. A part of the framework that we propose is implemented using Jini.

Developed in the Java programming language, Jini aimed at providing protocols for discovering devices as they join the network and making their presence known to other members of the network [5][20]. Jini is able to support applications in heterogeneous, dynamic computing environments and has been used to develop several agent-based applications. Agents discover each other and communicate with each other using the Jini services. In this paper we present the design of the multi-agent framework and implementation of a major part of it using Jini.

The remaining part of the paper is organized as follows. Section 2 presents the overall design of the multiagent framework. Section 3 briefly highlights the features of Jini that are useful for our framework. Section 4 describes a part of the framework, which implements mobile agents for creating an adaptive execution environment for the jobs. Section 5 presents results. Related works are discussed in Section 6 and Section 7 concludes.

2 Multi-agent Framework for Resource Allocation

One important requirement for Grid middleware is to act as a resource broker between the clients and the resource providers. It is also necessary to monitor the performance of the infrastructure and the performance of a

job on regular basis so that the agreements between the clients and the resource providers are maintained. It needs to be seen that the clients get the services that they have been assured for and no overprovisioning is made, as the clients may have to pay for the services. Whenever there is a violation of the agreement or overprovisioning, the middleware must take appropriate actions, which may change the execution environment of the job. In this section we describe the overall design of a multi-agent system that provides services like resource brokering and allocation, monitoring and tuning of application performance and migration of jobs in case of any performance problem.

Within the multi-agent system, a group of interacting, autonomous agents are working together towards a common goal which is *to set up and maintain the service level agreement between a client of computational services and the owners of the computational resources* by regularly monitoring the performance of the jobs and the infrastructure and scheduling and rescheduling jobs onto grid resources depending upon their requirements. Each agent plays a specific role and has a well-defined set of responsibilities or subgoals in the context of the overall system and is responsible for pursuing these autonomously. The major subgoals of the system can be defined as resource brokering, controlling the execution of a single job or more than one concurrent jobs, performance analysis of the infrastructure and individual jobs, performance tuning of a specific job at local level Altogether six agents are used in the multi-agent system. These are:

- Broker Agent,
- ResourceProvider Agent,
- JobController Agent,
- JobExecutionManager Agent,
- Analysis Agent, and
- Tuning Agent.

An agent class diagram is shown in Figure 1.

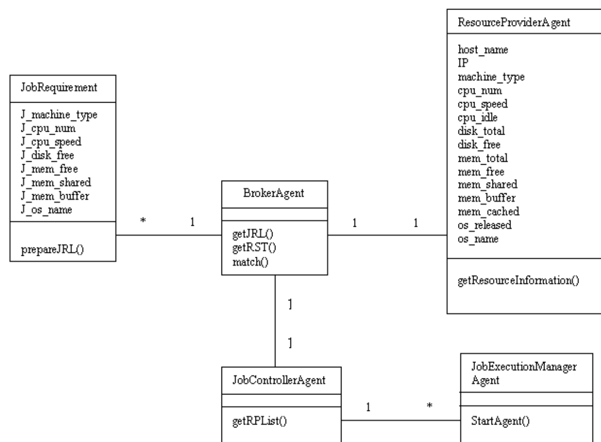


Figure 1 Agent Class Diagram

In our system, a *Broker Agent* is responsible for finding suitable resources for multiple concurrent jobs submitted to the system by a client. The Broker Agent prepares a Job Requirement List (JRL) from the description of each job. The Resource Provider (RP) at a particular Grid Site prepares a ResourceSpecificationMemo specifying the resources that are available for the clients. In the current implementation of our multi-agent system, the information about resources and their quality is collected with the help of Ganglia Information provider [7]. The ResourceSpecificationMemo is stored in a ResourceSpecificationTable (RST) until the RP withdraws all of its services. The RST is later used by the Broker Agent for resource brokering. The JRL is matched with the entries in RST and the match response along with a ResourceProviderList is sent to the JobController Agent. A ResourceProviderList is a collection of all the resource providers who can meet the requirements of a particular job. The collection of all ResourceProviderLists for all the concurrent jobs forms a JobResourceMatrix. A ResourceProviderList for job J_a becomes a row in the JobResourceMatrix and the i-th cell of this row is 1 if RP_i can satisfy the requirements of J_a, it is 0 otherwise. Figure 2 depicts a sample JobResourceMatrix.

	RP_1	….	RP_i	….	RP_j	….	RP_n
….							
J_a	0	….	1	….	1	….	0
…..							

1: selected
0: not selected

Figure 2 Example of a JobResourceMatrix

The *JobController Agent* is responsible for establishing *Service Level Agreements (SLA)* between the client and the resource owners [4]. It decides the optimal mapping (description of this algorithm is not within the scope of this paper) of all the concurrent jobs onto the Grid sites and accordingly creates a JobMap. While JobController Agent maintains the JobMap and keeps track of the SLAs for each of the concurrent jobs submitted by a client, one *JobExecutionManager Agent* becomes associated with each job and controls and keeps track of its execution. Thus a JobExecutionManager (JEM) Agent liaises with the *Analysis Agent* and as soon as it receives warning from the Analysis Agent regarding an SLA violation (or overprovisioning) for which rescheduling is necessary, it gets back to the JobController Agent and obtains an SLA which is an agreement with another resource owner (selected from the JobResourceMatrix) and reschedules the job there. Sometimes the Analysis Agent may indicate that the performance of the application may be improved by applying some local optimization techniques. The WarningMessage generated by the Analysis Agent holds this indication and on the basis of this indication the

JobExecutionManager Agent either reschedules or invokes a Tuning Agent sitting on the particular Grid site. The JobController Agent supervises all the JobExecutionManager Agents, which are associated with multiple concurrent jobs (one JEM for each job). Thus, the JobExecutionManager Agents become subordinates to the JobController Agent and work in close collaboration with the latter. Figure 3 shows the sequence diagram of the resource brokering scheme.

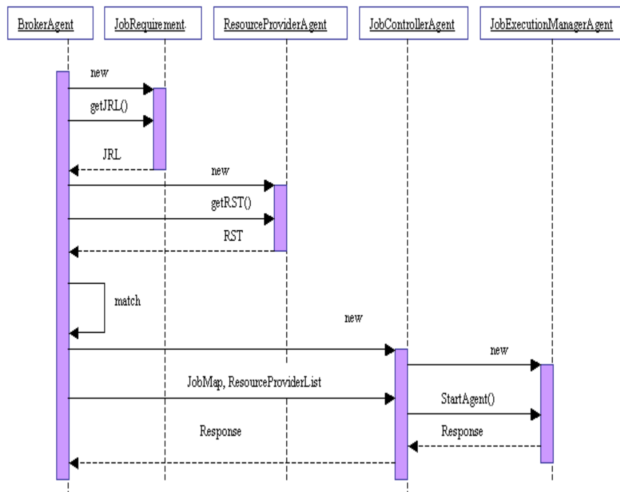


Figure 3 Sequence diagram of Resource Brokering

The remaining part of the paper focuses on the Jini implementation of the JobExecutionManager agent. The JobExecutionManager agent creates a mobile agent which carries the job to the selected resource provider and gets executed using the available resources on that RP. Before getting into the detail of the implementation of the JEM agent, the following section first briefs some salient features of Jini that have been used in our agent system implementation.

3 Implementing Agent-based Systems using Jini

Jini technology attempts to simplify interactions on a network [13][20]. The Jini architecture is built upon a fully object-oriented approach to network programming. Jini can be effectively used for building distributed applications including agent based systems. In general, Jini offers a wide range of supporting services for communications. With Jini Transactions and Distributed Events a wide range of communication protocols, especially in relation to negotiations between agents can be supported.

Jini uses Java Remote Method Invocation (RMI) for this purpose. Below we describe some of the capabilities of Jini which have been used to implement a part of our multi-agent Framework.

Jini technology operates around three protocols that enable discovering and registering services and joining them in *federations* of Jini services [13]. A Jini-enabled service uses the *discovery* protocol to announce its presence on a network. This is done by *multicasting* a message, i.e. sending information to a number of other machines that belong to the same sub-network simultaneously. The message contains basic information about the service and details about how it can be contacted. Dedicated *lookup services* listen to such announcements, and when they receive one, they contact the announcing service and retrieve detailed information about the service as well as the specific software code that provides an implementation of the service interface (the service proxy). This process is defined in the *join* protocol. Finally, other services use the *lookup* protocol to discover the lookup services and query them about the registered services. At the time of querying the lookup services about the registered services, clients specify the Java types of the service proxy interface implementations that have been uploaded to the lookup services or the service attributes or both. Service attributes are defined using Jini *Entrie*s, where entries are typed sets of objects that can be tested for exact match with a template. Once a required service is discovered, the interface implementation is downloaded to the client and the client can then access the service using Java RMI. The use of a service in general can be controlled via the Jini *leasing mechanism* that allows *leases* to be defined on the use of services. The leases must be renewed as and when required, otherwise the services may become unavailable. This is especially useful in case of lookup services, which require that all other services registering with them renew their leases with the lookup services (or define indefinite leases). Through this mechanism Jini ensures that if other services are not available for some reason they will eventually be removed from the registry of the lookup services.

Other significant services available through the Jini technology toolkit include Jini *Distributed Event*s, which allow events that take place on one Jini service be propagated to listeners across the network and Jini *Transaction*s, which allow groups of operations be considered as unique atomic transactions [20].

An agent-based system may be implemented using Jini by allowing all communications to be done through RMI and one needs to download a proxy (the relevant interface implementation) to communicate with other agents. Byassee in his article “Unleash mobile agents using Jini” [3] discusses in detail how mobile agents can be implemented on a Jini platform. A similar approach has been adopted in the work presented in this paper. We are using Jini for implementing the JobExecutionManager agent of our framework. The JEM initiates a mobile agent to support the adaptive execution of a job on a Grid

resource. The downside of using Java RMI for this purpose is that it complicates communication between agents that are not located within the same subnet.

4 Jini Mobile Agent-based Adaptive Execution

This section describes the implementation of JobExecutionManager agents using Jini and demonstrates how job execution is adapted to dynamic resource conditions and application demands. Adaptation is achieved by supporting automatic migration of a mobile agent carrying the job following performance degradation or remote resource failure. The JobExecutionManager agent is an initiator of mobile agents which carry the concurrent jobs to the Grid resources following the SLAs set up by the JobController Agent.

Building a mobile agent framework on top of Jini architecture has already been presented elsewhere [3]. In our work, implementation of a part of the multi-agent framework (described in Section 2), in particular the JobExecutionManager agent has been carried out with similar idea. Mobile agents are employed here for executing the client's job at remote nodes.

The mobile agent component consists of two main entities: Mobile Agent and Mobile Agent Station. The former is the mobile agent itself, that is, an entity with some job to do. The latter is the agent station, a service that provides the mobile agents' execution platform. To be an active agent platform, a given node in the system must have at least one active agent station. These two entities can be easily mapped onto the Jini model described in the previous section. Jini, at the highest level, provides the infrastructure that enables clients to discover and use various services. In the context of the mobile agent component, the agent station may be offered as a Jini service [3] and the mobile agent becomes the Jini client.

In our implementation MobileAgentStation is a class that implements a platform for mobile agents on top of a Java virtual machine. It executes mobile agents and makes host resources available to them. Resource providers that are willing to offer computational services start Mobile Agent Stations (one mobile agent station per resource provider) following some direction from the JobControllerAgent. The agent stations are run autonomously and cooperatively to form an agent space. Mobile agents move within this space from one station to another to perform their tasks on behalf of their creators. Each mobile agent has a home station in the space. Mobile agent stations register with one or more Jini lookup services by providing a service proxy. In turn, clients, i.e. mobile agents query the lookup services for mobile agent stations that might be of interest.

JobExecutionManagerAgent (Figure 4) initiates a mobile agent to perform a job on the selected resource provider. The mobile agent performs its job on that mobile agent station on behalf of its creator. During the execution of the job, if AnalysisAgent detects any performance problem, a warning is sent to the JobControllerAgent and at the same time the value of a performance variable *Perf_v* is set. *Perf_v* indicates whether rescheduling is necessary or not (the variable may also be set with other values to indicate other types of actions, such as local tuning of the job). The mobile agent monitors *Perf_v* and, if required, it will stop execution on that mobile agent station and will move to another mobile agent station.

In order to construct a mobile agent station, a remote interface, which is essentially a service template, is required. MobileAgentStationRemoteInterface has been created for this purpose. The mobile agents actually look for this service template via the Jini lookup service. The MobileAgentStationRemoteInterface provides a method called monitor(), which monitors the performance variable *Perf_v* (set by analysis agent). An implementation of this remote interface is the actual Jini service. We use the MobileAgentStation class that implements the MobileAgentStationRemoteInterface. The MobileAgentStation constructor first creates a LookupDiscoveryManager to locate the Jini lookup service, and then it creates a JoinManager. The JoinManager simplifies service management by encapsulating the functionality required for both registering with and withdrawing from a dynamic lookup service pool. The MobileAgentStation binds an incoming agent to a thread and subsequently executes. It also initiates a thread that checks the performance variable *Perf_v* in order to determine whether rescheduling is required or not. If rescheduling is required the execution of the agent will stop. JobExecutionManagerAgent will then select a new mobile agent station and move the mobile agent to that station. The method newMobileAgentStation() (discussed later) selects one mobile agent station.

An interface for agents is required for implementation of the mobile agent. We create a MobileAgentInterface that extends the Serializable interface. The Serializable interface makes the implementer of the MobileAgentInterface a serializable entity. The MobileAgentInterface consists of two methods - is_finished() to check whether the mobile agent finishes its task and setInterrupt() to stop the execution of the mobile agent as and when required. After being shifted to another mobile agent station the mobile agent can restart its execution there by invoking its own run() method.

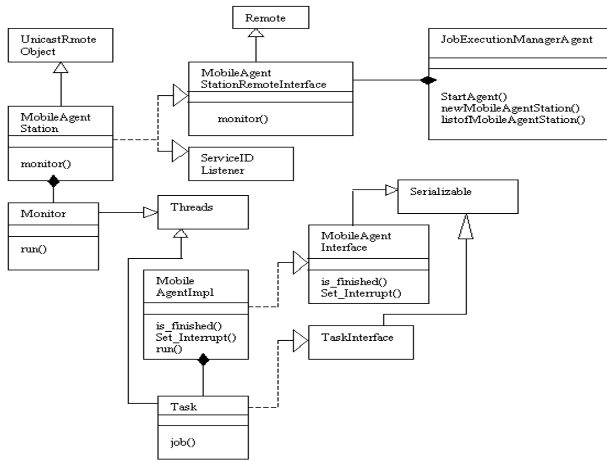


Figure 4 Implementation of JobExecutionManagerAgent

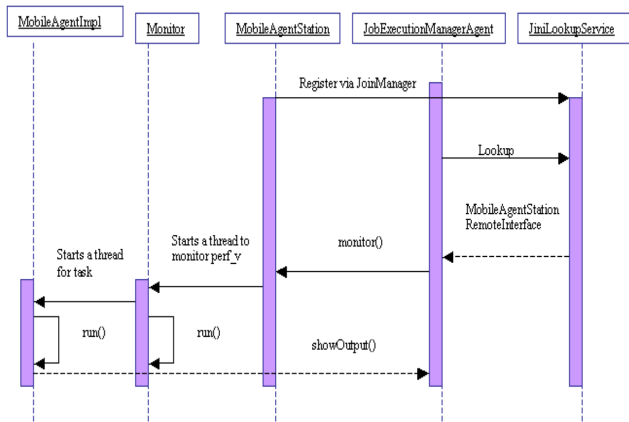
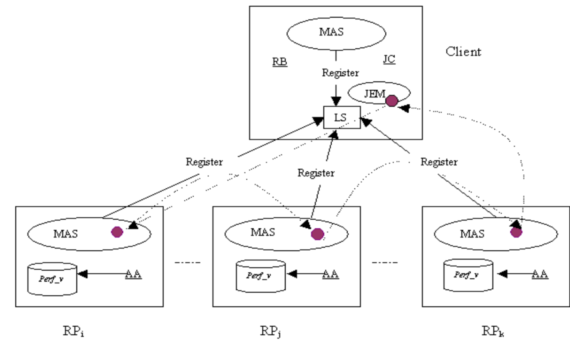


Figure 5 Sequence diagram for implementation of JobExecutionManagerAgent

MobileAgentImpl class provides an implementation for the MobileAgentInterface class. When the agent needs to move, JobExecutionManagerAgent requires to find an available mobile agent station and invokes newMobileAgentStation() method for this purpose. The method newMobileAgentStation() checks the list of available mobile agent stations and selects one mobile agent station from that list. To obtain the list of currently available mobile agent station it calls listofMobileAgentStation() method. Resource Providers that can fulfill the requirements of the job and are selected by the resource broker (collected in the form of ResourceProviderList) register themselves as mobile agent stations. The method listofMobileAgentStation() retrieves a list of all current mobile agent stations registered with lookup service and selects one of them as the new execution platform. When the agent arrives at the new mobile agent station, the job() method in the MobileAgentImpl class is invoked and the execution of the job starts. After completion of the execution, the mobile agent returns back to the initial instance of the JobExecutionManagerAgent. Figure 5 shows the sequence diagram for implementation of the JobExecutionManagerAgent.

Selection of the next mobile agent station should be accomplished in consultation with the JobControllerAgent and new SLAs should be set up accordingly after updating the JobMap. However, currently we are dealing with a single job and the JobControllerAgent is not fully implemented. This extension will be included in our future work.



MAS: Mobile Agent Station; JEM: JobExecutionManagerAgent; RB: Broker Agent; JC: Job Controller agent; AA: Analysis Agent; LS: Lookup Service; ● Mobile Agent

Figure 6 Movement of mobile agent from one mobile agent station to another

Figure 6 depicts the movement of mobile agent from one mobile agent station to another. Initially the job is submitted to the JobControllerAgent and then to the JobExecutionManagerAgent (according to the figure both are sitting on the client, although they may be located on different Grid resources as well).

5 Result

The preliminary implementation of the mobile agent-based adaptive job execution in a Grid environment has been carried out on a local Grid test-bed comprising a node having an Intel Pentium-4 processor of 2.8 GHz speed and 512 MB physical memory as server1 and another node having Intel Xeon processor of 3.0 GHz speed and 1024 MB physical memory as server2.

Scenario1: The mobile agent is initially scheduled to server1 and completes its execution on it.
Scenario2: The mobile agent is initially scheduled to server1, and at a certain instance of time it is rescheduled to server2 (because the server1 withdraws its services) and executes the remaining part of the job on server2.

The two test codes considered for this work are a simple sorting program (Figure 7) and a matrix multiplication program (Figure 8). In both cases overheads are involved while shifting the mobile agents from Server 1 to Server 2 (Scenario 2) due to the additional time required for discovering the services and rescheduling the jobs on a second machine. Figure 7 and Figure 8 depict the results and compare the total time taken by the mobile agents in Scenario 1 and Scenario 2. Execution times are measured with varying data sizes.

It is apparent in both cases that rescheduling from server1 to server2 does not achieve significant improvement in running time (possibly due to little difference in the server configurations); however the results demonstrate that rescheduling can be done successfully using mobile agents and thus making the job execution adaptive to the changing environment. We could obtain improved results if we have chosen an enhanced system as a server2, such as a multi-processor system.

6 Related Works

Several mobile agent systems have been developed and used in various environments for carrying out different jobs. Some of these are the Naplet system [25], the Aglet system by IBM [2], and Jini-based mobile agent implementation as described in [3] . The Naplet system is an experimental framework in support of Java-compliant mobile agent for network-centric distributed applications. An excellent review of other mobile agent systems is described in [22].

We present a multi-agent framework in which some agents are mobile. The mobile agents within our framework enable adaptive execution of jobs in Grid environment. Huedo et al presents a framework for adaptive executions in grids [12]. This is a Globus based framework that allows execution of jobs in a 'submit and forget' fashion. While our objective is also similar to the above, we focus on the multi-agent system for performance-based resource brokering and execution in Grid environment. Moreover our framework supports concurrent execution of multiple independent jobs on different Grid resources.

Maintaining performance QoS for individual applications is one of the major objectives of our research. A similar objective is pursued by the ICENI project which emphasizes a component framework for generality and simplicity of use [6]. On the other hand GrADS project focuses on building a framework for both preparing and executing applications in Grid environment [14]. Each application has an application manager, which monitors the performance of that application for QoS achievement. Failure to achieve QoS contract causes a rescheduling or redistribution of resources. GrADS monitor resources using NWS and use Autopilot for performance prediction [24][18]. A performance steering system developed within the RealityGrid project [17] has been described in [15]. The framework proposed in this paper works with similar concept. But unlike the previous systems, it employs autonomous agents for carrying out the performance tuning jobs.

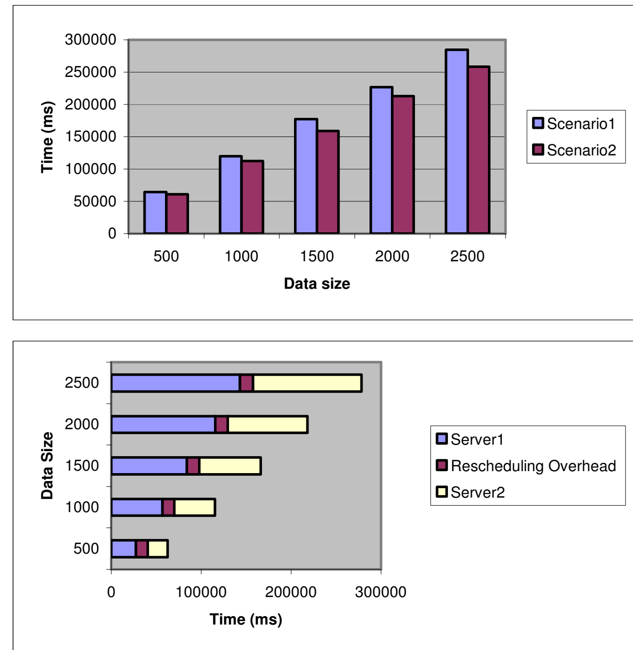


Figure 7 Execution Times of the Sorting Test Code - (a) Comparison of the performances in Scenario 1 and Scenario 2. (b) Results of rescheduling from Server1 to Server2.

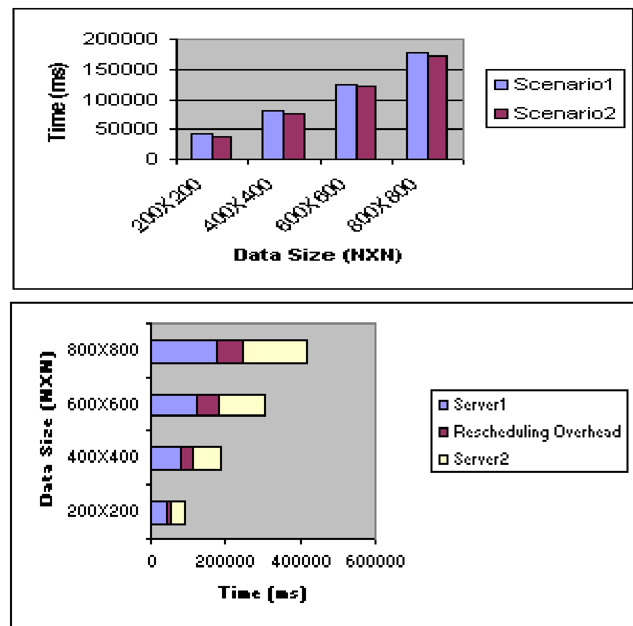


Figure 8 Execution Times of the Matrix Multiplication Test Code - (a) Comparison of the performances in Scenario 1 and Scenario 2. (b) Results of Rescheduling from Server1 to Server2

Agent-based framework has also been proposed by other researchers. The AgentScape project [23][16] provides a multi-agent infrastructure that can be employed to integrate and coordinate distributed resources in a computational grid environment. The objective of the AgentScape system is to provide a "minimal but sufficient" environment for agent applications. The A4 [1][11] project at Warwick has likewise developed a framework for agent based resource management on grids. As observed in [9], these multi-agent systems are used to trade for grid resources at the higher "services" level and not at the base "resource" level. Raw computational resources are normally scheduled using a more classical scheduler, using performance prediction or time based techniques. This paper, in contrast to the above mentioned multi-agent systems, focuses on a multi-agent framework for enhancing the performance of an application at runtime.

7 Conclusion

The paper presents a multiagent framework for resource brokering and resource allocating to applications running in Grid environment. Our multiagent framework works on top of the available Grid middleware (such as Globus Toolkit [8]), and uses the services available with it. A brief discussion about the design of the framework is presented in this paper. The discussion elaborates how the agents interact within the framework to improve the performance of an application during run-time. So far we have successfully implemented the Broker Agent, the ResourceProvider Agent, the JobController Agent and the JobExecutionManager Agent. The objective of this paper is also to present the implementation of a part of the multiagent framework. Thus the second part of this paper discusses the implementation of the JobExecutionManager Agent which facilitates adaptive execution of a job in spite of changing resource availability and resource requirement. Further work is being carried out for implementing the other agents.

8 References

[1] A4 Project http://www.ccrl-nece.de/˜cao/A4/.
[2] Aglet http://aglets.sourceforge.net/.
[3] Byassee J. "Unleash mobile agents using Jini, Leverage Jini to mitigate the complexity of mobile agent applications " Available from http://www.javaworld.com/javaworld/jw-06-2002/jw-0628-jini.html
[4] Czajkowski K., I. Foster and C. Kesselman, Resource and Service Management, The Grid 2: Blueprint for a New Computing Infrastructure (Chapter 18) by Ian Foster and Carl Kesselman, Morgan Kaufmann; 2 edition (November 18, 2003)
[5] Franklin S. and A. Graesser "Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents", Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996. Available from http://www.msci.memphis.edu/~franklin/AgentProg.html
[6] Furmento N., A. Mayer, S. McGough, S. Newhouse, T. Field and J. Darlington, ICENI: Optimisation of Component Applications within a Grid Environment, Proceedings of Supercomputing 2001. http://www-icpc.doc.ic.ac.uk/components.
[7] Ganglia http://ganglia.sourceforge.net/
[8] Globus http://www.globus.org.
[9] GrADS: Grid Application Development Software Project, http://www.hipersoft.rice.edu/grads/.
[10] Gradwell P., Overview of Grid Scheduling Systems.
[11] HPSG, http://www.dcs.warwick.ac.uk/research/hpsg/.
[12] Huedo E, Ruben S. Montero, Ignacio M. Llorente, A framework for adaptive execution in grids, Software: Practice and Experience, 34[7], Pages 631-651, March 2004.
[13] Jini http://www.jini.org
[14] Kennedy K., et al, Toward a Framework for Preparing and Executing Adaptive Grid Programs, Proceedings of the International Parallel and Distributed Processing Symposium Workshop (IPDPS NGS), IEEE Computer Society Press, April 2002.
[15] Mayes K., G.D. Riley, R.W. Ford, M. Lujan, T.L. Freeman, The Design of a Performance Steering System for Component-based Grid Applications.
[16] Overeinder B. J., N. J. E. Wijngaards, M. van Steen, and F. M. T. Brazier Multi-Agent Support for Internet-Scale Grid Management.
[17] RealityGrid Project http://www.realitygrid.org.
[18] Ribler R.L., H. Simitci, D. A. Reed, The Autopilot Performance-Directed Adaptive Control System, Future Generation Computer Systems, 18[1], Pages 175-187, September 2001.
[19] Roy S, N. Mukherjee, Towards an Agent-based Framework for Monitoring and Tuning Application Performance in Grid Environment, ICDCIT, December 2005.
[20] Sun's Jini Web Site: http://www.sun.com/jini/
[21] Tomarchio o, L. Vita, and A. Puliafito. Active monitoring in GRID environments using mobile agent technology. In 2nd Workshop on Active Middleware Services (AMS'00) in HPDC-9, Pittsburgh (Pennsylvania (USA)), August 2000.
[22] Wang D, N. Paciorek, and D. Moore, Java-based mobile agents, CACM, 42[3], Pages 92-102, March 1999.
[23] Wijngaards N. J. E., B. J. Overeinder, M. van Steen, and F. M. T. Brazier, Supporting internet-scale multi-agent systems, Data Knowledge Engineering, 41(2-3): 229–245, 2002. Available from http://www.iids.org/publications/bnaic2002_dke.pdf.
[24] Wolski, R., N.T. Spring and J. Hayes, The Network Weather Service: A distributed performance forecasting service for metacomputing, Future Generation Computer Systems 15(5), 757-768.
[25] Xu C, Naplet: A Flexible Mobile Agent Framework for Network-Centric Applications, Proceedings of the 16th International Parallel and Distributed Processing Symposium, 2002. Available from www.ece.eng.wayne.edu/~czxu/paper/icec02-naplet.pdf.