Model for Grid Service Instance Migration

Dhaval Shah and Sanjay Chaudhary

Dhirubhai Ambani Institute of Information and
Communication Technology, Gandhinagar, 382009, India

Abstract. Grid computing, emerging as a new paradigm for next-generation computing, enables the sharing, selection, and aggregation of distributed resources for solving large-scale problems in science, engineering, and commerce. The resources in the Grid are heterogeneous and geographically distributed. The resources in the Grid are dynamic in nature. Resource owners are at discretion to submit/donate the resources in to the Grid environment. The term Web services describes a standardized way of integrating Web-based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone [8]. Grid Services is an extension of Web Services in a Grid environment having statefulness as a key feature. State of any Grid Services is exposed with the help of Service Data Elements. Grid Services may fail during its life cycle due to failure of a resource or a withdrawal of a resource by the resource owner. Thus, there is a need to provide a reliable solution in the form of Grid Service instance migration to protect the work of the users, which was carried out. This paper proposes a model that supports Grid Services instance migration. Migration of an instance can take place based on the failure of resource, increase in load at the resource, change in the policy of the domain in which resource resides, user specified migration, or migration due to withdrawal of a resource by the resource owner. It enables the users to specify the migration if the user does not trust the domain in which instance is running. The model includes an incremental checkpointing mechanism to facilitate migration.

1 Need for Grid Service Instance Migration

An important requirement for computational Grids is the provision of non-trivial qualities of service to be provided to the applications. Since the Grid resources belong to different administrative domains and are geographically distributed, their availability may be very dynamic. Additionally, reliability of such resources is also very difficult to guarantee [1][2]. The above-mentioned problems are magnified if the applications are long running as well as distributed. Since resource availability may change over the execution period of long running applications, an application should be able to store its state onto stable storage, and migrate to more suitable resources if need arises. We have identified few issues that are likely to arise in Grid environment.

- Grid is a collection of Virtual Organizations.[1] Each and every VO has its own policy. Resource owner, i.e. service provider has got complete control over the re-

Shah D. and Chaudhary S. (2007).
Model for Grid Service Instance Migration.
In *Proceedings of the 1st International Joint Workshop on Wireless Ubiquitous Computing*, pages 96-105
DOI: 10.5220/0002405700960105

sources. If there is a change in policy of the domain where resource resides and if that policy is not acceptable to the requestor, then requestor should be able to select another resource in another domain where migration can take place.

- There is a need to provide a mechanism that can enable retrieval of computations carried out.
- Resources may get some task having higher priority than the task submitted by the client. So, there is a need to facilitate a rescheduling/migration of a task.
- Migration based on economical factors.
- Avoid loss of work in case of withdrawal of a resource by migration.

All the above-mentioned scenarios require having Grid Service instance migration to provide better service to the requestor. This paper discusses the model for Grid service instance migration, incremental checkpoint approach, RAID approach, model for parallel job support and algorithms for total and partial migration.

2 Scenario to Make a Request for a Grid Service

Requestor searches the registry to find a service providing particular functionality. Registry returns the list of all the factories capable of creating an instance of a Grid Service. Requestors have to specify the requirements in the standard form such as RSL. We propose to use XML file that is very similar to RSL [3][4].

2.1 XML like File Resembling RSL

Each node would have registered with the domain registry by supplying characteristics file. Client contacts the central registry and obtains the handle of the factory where it submits the job. The client provides information about the name of the service, required resource specifications, and its policy. The requestor searches the central registry for a specified functionality. A sample file for specifying details about requirements is shown in Listing 1. Requestor specifies the required properties such as processing power required, hard disk space required etc., time period for which resource/service is required, resource type required, service required etc.

```
<Required_properties>
<ProcessingSpeed>1.8</ProcessingSpeed>
<Deadline> 1200</Deadline>
<HardDiskSpace>40</HardDiskSpace>
</Required_properties>
```

Listing 1. Sample XML like file for specifying the requirements.

2.2 Registry

Registry does the matchmaking after parsing the XML file submitted by requestor and then it returns URI(s) of all the factories providing specified functionality [4]. We propose to modify the registry to provide some more features to users such as, though

not limited to, such as reliability, economic details i.e. payment required to be made to
a particular service, location details for a factory, bandwidth required to access the service etc., to help requestors in the selection of the appropriate factory from the list of many factories.
We propose that registry should have a matchmaking module that will do the matching process based on some criteria. The requestor can specify search criteria.
Reliability for a factory can be determined from the past experience with that factory.
Once a requestor obtains URI(s) of factories, along with some of the criteria information to enable user to take some decision regarding the selection of a factory, requestor
can select an appropriate factory. The one most important criterion for the selection of an appropriate factory can be economic policies adopted by factories. Users would like to select a factory providing the economical solution without compromising the reliability or quality of service.

2.3 Selection of a Location for Creation of an Instance

Requestor sends a createService request after selecting an appropriate factory. We prefer the use of following approach for the location of instance creation as in dynamic environment such as Grid acceptable policies to the user could not be obtained in advance and availability of resources could not be predicted.

Factory returns the addresses of the hosting environment capable of creating an instance to the requestor and also information regarding the policy relationship to the requestor. Having received information, requestor can select a location where he would like to create an instance.

2.4 Need for Negotiator Module

To justify the selection of the second approach, we propose to have a module, named negotiator, which will be working in conjunction with the factory module. It is a module that stores policy documents of different domains. It will do the matchmaking to decide the compatibility between two domains. Upon receiving a request for the creation of an instance, negotiator module does the matchmaking and returns the addresses of only those domains whose policy is acceptable to the requestor. Policies of all the domains are available with the negotiator module.

Policy File for the Domain. The policy file that is to be submitted to the negotiator module by each domain will be an XML file. Listing 2 discusses the sample XML like policy file (actual file may require some more parameters) specifying the requirements, preferences etc.

```
<policyfile>
<security>
All the messages have to be encrypted
```

```
</security>
<AllowedUser>
All the users from da-iict domain should be allowed.
</AllowedUser>
</policyfile>
```

Listing 2. A sample policy file for the domain.

A negotiator module converts all the policy files into a standard normal file, so that merging of two policy files or intersection (matchmaking) can be done effectively. If there is a change in a policy of the domain then that domain sends a modified policy file to the negotiator and negotiator does the matchmaking with the changed policy file. It checks for the violation of agreement and informs the requestor regarding it. The negotiator module works in conjunction with the factory to find out a list of all active instances, do matchmaking, and inform users regarding policy violation if any.

3 Schemes for Grid Service Instance Migration

There are two schemes for Grid Service instance migration.

1. User initiated migration
2. System initiated migration.

User initiated migration is intended to give requestors more control over their data/task. Requestor or system decides to migrate an instance depending on the need.

3.1 Methods for Finding the Location for Migration

After taking a decision regarding the migration, there is a need to find out an appropriate location for migrating an instance. The policy of the domain where migration can take place has to match the policy of the requestor domain.
There are two approaches in which the location for the migrated instance can be found out.

- Requestor submits the list of factories where migration can take place at the time of sending createService request. When factory decides to migrate an instance; factorycan find out the locations where migration can take place from the list supplied by the user.
- In this approach, factory multicasts a request to all the nearby factories. Factories will receive a request and will send back a reply if they are capable of serving the request or willing to serve the factory's request. Here, nearby factories meaning that a request will traverse to specified number of hops and not more than that. This constraint is needed to reduce the network traffic. All the nearby factories are as summed to be reachable by a particular factory and multicast messages by a particular factory would be received by the listeners existing at factories.

3.2 Checkpointing Mechanism

In a failure prone scenario, an ability to checkpoint applications and restart them from a particular checkpoint upon failure of Grid resources is desirable. Additionally, checkpointing can also be used for dynamic resource scheduling, e.g. if a high priority task needs to preempt a long running lower priority one, the lower priority task can be checkpointed and restarted later when the higher priority application has finished executing [5]. In our model we have proposed to use a special case of a distributed checkpointing mechanism. We have avoided the use of centralized checkpointing mechanism as it may lead to single point of failure. A service provider providing a service; is responsible for checkpointing the job so that in the case of a migration or failure, task can be rescheduled at some other location. There is another approach for deciding about the timing intervals for uploading a checkpoint image. Notification approach specified by OGSI can be used here. The format for the checkpoint should be standardized. A checkpoint file may contain name of a service, location of a service, node identifier for the service, SDE values and jump levels.

Comparison between RAID and Incremental Checkpoint Approach. There are two approaches for uploading checkpoint images to the server. Table 1 shows the comparison between both the approaches: RAID checkpoint approach and Incremental Checkpoint Approach.

Table 1. Comparison between RAID and Incremental checkpoint approach.

Sr. No	RAID checkpoint approach	Incremental checkpoint approach
1	Application provider knows about checkpoint server address and uploads an image to that server only. That server will take back ups.	Application knows addresses of all the checkpoint servers. It will upload first checkpoint to 1st server, 2nd to second server and so on.
2	The checkpoint server should take regular backups. It is not always preferable to modify server functionality to take backup.	This approach does not require modifying server but applications have to be modified to achieve incremental checkpoint facility.
3.	The latest copy of the checkpoint is always retrieved in case of a failure.	The work carried out between two checkpoints is lost. If 2nd server fails than retrieval must be made from 1st server.
4.	It requires a much more storage space, to store the previous checkpoint images for some time.	This approach saves storage space, if there is a need to store previous checkpoint images for some time.

3.3 Pseudo Code for the Instance Migration Scenario

Listing 3 mentions the pseudo code for the instance migration scenario to show that all the domains must supply a policy file to the negotiator module. If a node fails then checkpoint can be used to restart the task. The migration can be carried out if nodes are heavily loaded or policy of the domain is not acceptable to the requestor.

```
For all domains
      Send policy files to the negotiator module
For all nodes
      If node fails then
        Select another appropriate node.
        Take data from checkpoint server
        Restart the task
For all nodes
      If node is heavily loaded
      Find the less heavily loaded node where migration can take place
      Initiate migration of the task.
For all domains
      If there is a change in policy then
      Inform the negotiator module
      Negotiator informs the requestor regarding the same when needed.
```

Listing 3. Pseudo code for instance migration scenario.

4 Model for Instance Migration for Parallel Independent Tasks

An application may be divided into multiple subtasks to reduce the execution time [6]. It is assumed that application has inherent parallelism and it is possible to divide the task into multiple independent and parallel subtasks. In that case, there is a need for synchronization between different subtasks of the main task. A coordinator module can be used which does the job of dividing a task into number of sub jobs independent of each other and it is responsible for synchronizing the different sub tasks.

When the application is submitted, the first process that starts is the coordinator. Requestor can supply a file quite similar to deployment descriptor to inform coordinator details regarding how to divide task into subtasks. According to the file details, it divides the task into number of subtasks. It then spawns them on the nodes and instructs them to start execution.

During the task runtime, migration mechanism is inactive, i.e. coordinator need not stay alive throughout the execution of subtasks, since subtasks themselves do not need any of the functionality provided by the coordinator. Migration is inactive meaning that coordinator is passivated by storing all the information to a stable storage. Migration is activated when a subtask is about to be killed or failure has happened. Coordinator takes the checkpoint on receiving a signal or it may be a periodic checkpointing employed by the application. After the actual migration, the migrated tasks first execute post-checkpoint instructions before resuming the real user code. When all the checkpointed and migrated tasks are ready to run, coordinator allows them to continue their execution. On the event of a failure of a resource running the subtask, a need arises to find out the resource having the similar capability.

4.1 Application Level Migration by a Global Coordinator

It may happen that the resources may not be available in the same domain to replace failed/overloaded nodes or resources. In this scenario, there are two approaches:

1. Migrate only partial part to some other domain. This case has the disadvantage of making application migration aware. A negotiator module can be consulted to find out the domain where migration of a partial job can be done. There has to be synchronization between the migrated partial task and the coordinator who is the actual controller for the job.
2. In this approach, migration of the whole application can be done on the resources in the other domain. This approach is required if user does not want to indulge in the complexities of managing parts of the job in more than one domain. This type of migration is called a total migration.

In order to migrate the whole application, there is a need for a global coordinator that can facilitate migration of whole application in some other domain. Global coordinator will be responsible for taking decision regarding the kind of migration, i.e. partial migration or total migration. The other requirement is that the checkpoint server should be able to checkpoint itself in case of a total migration to facilitate the rescheduling of a whole application at some other location. [7]

4.2 Proposed Algorithm for Instance Migration for Parallel Independent Tasks

The algorithms for instance migration for parallel tasks are discussed here.

4.2.1 Checkpointing Algorithm for Partial Migration

1. A requestor initiates the checkpointing process by calling Init_checkpoint on the factory/coordinator, if coordinator is the receiving part.
2. The Coordinator obtains the taskId of the subtask to be checkpointed.
3. The Coordinator sends a take_checkpoint request to the appropriate taskId.
4. On receipt of the take_checkpoint request. Task stores the checkpoint and it sends checkpoint taken message to the coordinator.
5. Checkpoint server stores the checkpoint on the stable storage.
6. Checkpoint method for partial migration is over and control is returned back.

4.2.2 Checkpointing Algorithm for Total Migration

1. A requestor initiates the checkpointing process by calling Initi_checkpoint on the factory/coordinator, if coordinator is the receiving part.
2. The coordinator retrieves the taskIds for all subtasks part of the application.
3. For every subtask that is part of the application, the coordinator sends a take_checkpoint request to the appropriate taskId.
4. On receipt of the take_checkpoint request. Task stores the checkpoint and sends chceckpoint taken message to the coordinator.
5. On receiving messages from every subtask that is part of an application, the coordinator can infer that all individual checkpoints have been taken.

6. Checkpoint server stores the checkpoint on the stable storage and checkpoints itself for that task.
7. It should be ensured that this step either completes successfully in its entirety, or not at all. This is required to avoid a situation where set of checkpoints becomes a combination of old ones and newer ones.
8. Checkpoint method is now complete and control is returned to the user.

4.2.3 Migration Algorithm for Total Migration

1. A requestor initiated migration of an application by invoking the migrate_application method on coordinator.
2. Coordinator can work in conjunction with the negotiator module and information service to find out the location for the migration.
3. Requestor sends an approval to migrate an application to a specified location.
4. The coordinator is now ready to migrate the individual tasks. However, before migration, the said task has to be checkpointed.
5. On receipt of the take_checkpoint request, it takes the checkpoint and sends checkpoint taken message to the coordinator.
6. On receiving messages from every subtask, coordinator can infer that all individual checkpoints have been taken.
7. After storing the task checkpoint, the coordinator destroys the executing instance on the current resource and re-instantiates it on the target Grid resource using deployment descriptor. It finds the appropriate domain to reschedule the task.
8. Task can be resumed on the target resource.
9. Now, migration method is complete and control is returned to the user.

4.2.4 Migration Algorithm for Partial Migration

1. A requestor initiates migration of an application by invoking the migrate_application method on the coordinator.
2. Coordinator can work in conjunction with the negotiator module and information-service to find out the location for the migration.
3. Requestor sends an approval to migrate an application to a specified location.
4. The coordinator is now ready to migrate the individual task. However, before migration, the said task has to be checkpointed.
5. On receipt of the take_checkpoint request, it takes the checkpoint and sends checkpoint taken message to the coordinator.
6. After storing the task checkpoints, the coordinator destroys the current executing instance and re-instantiates it on the target Grid resource.
7. Task can be resumed on the target resource.
8. Now, migration method is complete and control is returned to the user.

5 Related Work

Many distributed systems exists that support check pointing and migration mechanism in a distributed environment (e.g. Condor [5] and Migol [10]). Condor system

provides a feature for check pointing the job that has been submitted. By default, a checkpoint is written to a file on the local disk of the machine where the job was submitted. A Condor pool can be configured with a checkpoint server that could be used as a repository for checkpoints. Condor also supports periodic checkpoint feature to prevent the loss of computation carried out in the event of shutdown or crash. Though condor system provides check pointing as well as periodic check pointing feature, incremental checkpoint feature would be useful reducing the space required for jobs. Condor supports migration of jobs to different condor pool, but the policy based pool finding is a limitation of a condor system. Migol is also a fault tolerant service framework for MPI applications but it does not discuss incremental checkpoint as well as policy based migration in Grid.

6 Conclusions

Resources in Grid are dynamic as well as probability of a failure increases with the addition of more number of resources. There is a need to enable requestor to retrieve the work carried out at some resource in case of a failure of a resource. Service may be deployed on a resource in some domain. There is a need to give more control over the data to the user in case of a change in the policy of a domain in which resource resides. Overloaded resources will generate the need to migrate some of the tasks. We suggest that checkpoint/restart approach is preferable in an environment where probability of failure is more so that the work carried out on the resources can be retrieved later on. To provide more control to the requestor over the data/task we propose to have a negotiator module. The negotiator module is responsible for matching the policies of the domains. In a scenario where changed domain policy is not acceptable to the requestor, negotiator can find out the domains whose policy is acceptable to the requestor. We suggest the use of incremental checkpoint/restart approach instead of RAID approach to save storage space in the cases where previous checkpoint images are to be stored.

7 Future Direction

- The Globus toolkit is an open source toolkit. The Globus toolkit can be modified to incorporate the model proposed. Negotiator, checkpoint server and queue modules can be incorporated in to the existing Globus toolkit.
- A model for Grid Service instance migration for tasks having parallel but dependent jobs can be one of the most challenging tasks required to be done in the future.
- The negotiator module proposed in a model can be automated. Negotiator module should maintain the relationship between different domains instead of finding the relationship at the time of request for service.

References

1. Foster I, Kesselman C, Tuecke S., "The anatomy of the Grid: Enabling scalable virtual organizations,". *International Journal of Supercomputer Applications* 2001
2. Foster I, Kesselman C, Nick J, Tuecke S., "The physiology of the Grid: An open Grid services architecture for distributed systems integration", TTThttp://www.globus.org/research/papers.html#OGSA
3. S. Tuecke, I. Foster, et al.,"*Open Grid Services Infrastructure (OGSI) Version 1.0*."2003; Open Grid Services Infrastructure: Draft http://www.ggf.org/ogsi-wg
4. Tim Banks et al., "Open Grid Service Infrastructure Primer," GWD-I (draft-ggf-ogsi-gridserviceprimer-3), OGSI, http://www.ggf.org/ogsi-wg
5. Condor Team, "Checkpointing in Condor", Condor Manual, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI http://www.cs.wisc.edu/condor/manual/v6.6/4_2Condor_s_Checkpoint.html
6. Milojicic D., Douglis F., et al., "Process Migration," *ACM Computing Surveys*, Vol. 32, No. 3, September 2000, pp. 241-299.
7. József Kovács, Péter Kacsuk, "A migration framework for executing parallel programs in the Grid", MTA SZTAKI, Parallel and Distributed Systems Laboratory, Budapest, Hungary
8. http://www.webopedia.com/TERM/W/Web_services.html
9. Migol: A fault-tolerant service framework for MPI applications in Grid, http://www.cs.uni-potsdam.de/bs/research/grid/index.html
10. C. Lia, X. Yang, N. Xiao, "*A Novel Checkpoint Mechanism Based on Job Progress Description for Computational Grid*" , Lecture Notes in Computer Science : Parallel and Distributed Processing and Applications, 2004. Volume 3358/2004, pp. 594-603
11. S. Rodríquez, A. Pérez, R. Méndez, "*A new checkpoint mechanism for real time operating"* ACM SIGOPS Operating Systems Review, Volume 31 , Issue 4 (October 1997), Pages: 55 - 62
12. L.M. Silva, J.G. Silva, S. Chapple, L. Clarke, "*Portable checkpointing and recovery*," *hpdc*, p. 188, Fourth IEEE International Symposium on High Performance Distributed Computing (HPDC-4 '95), 1995
13. Check pointing in CATALINA, http://www.ece.arizona.edu/~hpdc/projects/CATALINA/research.html
14. Y. Zhang, H. Franke et. Al. "*The impact of migration on parallel job scheduling for distributed systems*" http://www.ece.rutgers.edu/~yyzhang/research/papers/europar00.pdf
15. K.M. Al-Tawil, M. Bozyigit, S. K. Naseer, "*A Process Migration Subsystem for a Workstation-Based Distributed Systems*," *hpdc*, p. 511, Fifh IEEE International Symposium on High Performance Distributed Computing (HPDC-5 '96), 1996