An SNAP-based Resource Management System for Grid Environments

Zhonghua Yang and Rajbabu Gunasheelan

Information Communication Institute of Singapore (ICIS)
School of Electrical and Electronics Engineering
Nanyang Technological University, Singapore 639798

ezhyang@ntu.edu.sg

Abstract

Resource management is a fundamental issue in grid computing environments. In a typical grid environment involving several virtual organizations (VOs), the resource management requires negotiation among application and resources to discover, reserve, acquire, configure, and monitor resources. The existing resource management approaches tend to specialize for specific resource classes, and address coordination across resources only in a limited fashion. SNAP is a resource management model that provides Service Level Agreements (SLA)s, formalizing agreements to deliver capability, perform activities, and bind activities to capabilities respectively. This paper presents a working resource management system based on SNAP. The system is implemented using Java RMI. The design and construction of the system is described and the effectiveness of the system is evaluated.

Keyword: Grid, Resource management, SNAP, Agreement-based, Service-level agreement

I. Introduction

A Grid is an infrastructure for coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations (VOs) [5]. A resource is defined as any capability that may be shared and exploited in a networked environment, including computational resources (CPU or memory), storage, network bandwidth, even applications (virtualized as “services” for sharing). Resource management is concerned with how, when and in what way the resource is to be utilized rather than its core capability (what it does for client). Resource management is much more complicated in Grid environments due to the fact that the resources are under different administrative domains which enforce difference policies including security measures. Very often, Grid applications require the concurrent allocation of multiple resources which further complicated the matter [4, 3]. The issues addressed by the resource management in the Grid environment include: the mechanisms for job submission which describe the resource requirements, workload management (scheduling) for effective utilization of resources, advance reservations to make resource capability available at a specified point in time or for a specified duration, and co-scheduling for making a set of resources available simultaneously.

A resource management system acts as a broker between the client and resources. The resource management in a Grid must therefore resolve the conflict requirements between the resource clients

and owners. The client often requires some degree of guarantee on quality of service being provided by the resource which is however under the owner's local control based on usage policy. The owner tends to maintain discretion over how the resource can be used and how much service information is exposed to clients. The common approach to reconciling this is to negotiate and reach agreement. SNAP is a well-known protocol for agreement-based resource management in distributed environments and is being adopted by the Grid community as a base for resource management [2]. In this paper, we present a resource management system based on SNAP protocol and its design. The proposed SNAP-based architecture is a 3-tier resource management model consisting of an interactive Java based client, RMI based resource broker and local resource managers or the actual resource. The resource broker implements the SNAP protocol in order to mediate access to the resources. The resources are virtualized as heterogeneous RMI services running in the various hosting platform thereby simulating a distributed environment.

In this resource management system, a security framework has also been proposed for this SNAP protocol (DES /AES) algorithm using the J2SE 1.4 Java Authentication and Authorization Service (JAAS), Java Cryptography Extension (JCE), Java Secure Socket Extension (JSSE). This security framework has been incorporated in order to provide a secure resource negotiation between the client and the broker. The implementation is based on Globus Toolkit (GT3) which supports resource discovery by providing hierarchical resource information using the Lightweight Directory Access Protocol (LDAP). However, we do not use the Globus Security Infrastructure (GSI) service of GT3.

This paper is organized as follows. The related work on resource management in Grid environments is discussed in Section2, followed by our proposed system based on SNAP approach, the Java-based implementation model is presented. The APIs for client to access the resource management system is presented in Section 4. The experimental evaluation is described in Section 5, and Section 6 concludes the paper.

II. Related Work

The resource management is fundamental and crucial for the Grid environments, and it has attract extensive research efforts in the last decade. There appears a huge volume that provides an extensive coverage of the state of the art in Grid resource management [6]. In a nutshell, the key to the resource management is to develop a range of management abstractions and corresponding interfaces to the different classes of resources that need to be managed.

Much of the early efforts were focused on managing/scheduling computational resources, for example, Condor and its Preemptive resume scheduling [10], Grid resource allocation manager (GRAM) specialized for computational resources [1], and storage resource manager (SRM) unctions specifically for storage [11]. In Legion, resource management and scheduling are considered as placing objects on processors, and objects can be any resources required by the scheduled job (e.g., files, directories,or applications), thus the resource management becomes resource/object placement and multiple placement algorithms are supported in Legion [7]. The early resource management also addresses the heterogeneity in the way that similar resources are configured and administered through the definition of standard resource management protocols [3, 1] and standard mechanisms for expressing resource and task requirements [9].

The recent efforts is to make broadly applicable basic management functions that can be applied to a range of resources and services in a uniform fashion. Every thing that can be scheduled is considered a resource, potentially for sharing. It is now widely acknowledged that Grid-based resource

management systems generally require cooperation from the resource being managed, particularly when a resource is not dedicated to a specific user community, or virtual organization (VO), but rather is shared across VOs or, as is often the case, between Grid and non-Grid users [4]. The negotiation and agreement /contracting are the essential elements in Grid resource management. The agreement-based resource management is the state of the art approach to Grid resource management. In the future, the resource management will take the form of provisioning as a fundamental capability of the Grid infrastructure, as being similar to the case in networks [8].

The work presented in this paper takes an agreement-based approach based on SNAP protocol [2].

III. Proposed Model

The proposed resource management model is a broker-based architecture that runs an SNAP protocol daemon to negotiate the resource requirement of the clients. The requirements of the client are given as an RSL (Resource Specification Language) specification that is validated and refined by the resource broker. Many resources have parameterized attributes, i.e. a metric describing a particular property of the resource such as bandwidth, latency, or space [2]. Client desires access to a resource with the specified qualities. The resource brokers handles the mapping of the high level application requests to the resources through the local resource managers (e.g., Sun's Grid Engine or Platform's Load Sharing Facility) or Grid resource allocation manager (GRAM). The direct interaction between the broker and resources is also possible (Figure 1).

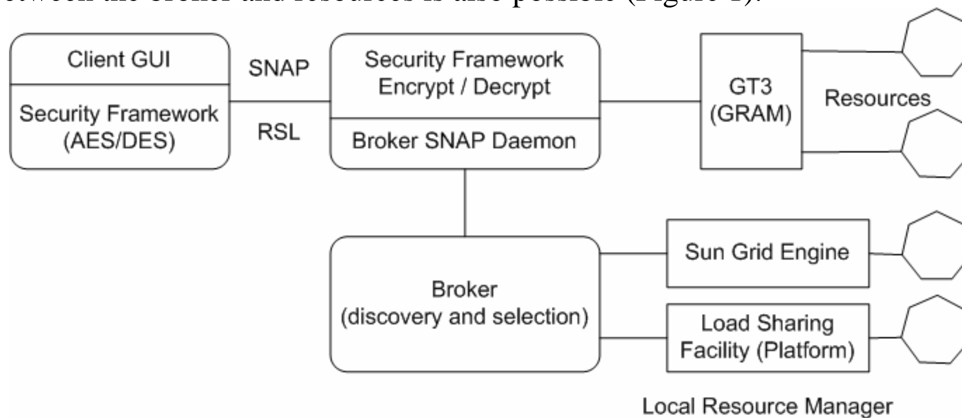

Figure 1: Proposed Model

An example of an RSL specification used by the proposed SNAP is as follows:

```
& (executable = exec.bat)
              (directory = ./ )
              (count = 1)
```

where executable refers to the remote executable file that has to be executed and the directory refers to the location of the executable and count refers to the number of processors required for execution. Since the testing environment is a single processor environment the value of count is specified as 1 [2].

A security framework to be integrated along with the protocol to provide a secure negotiation between the client and the broker. The choice of the encryption algorithm to be followed is either the Data Encryption Standard (DES) or the Advanced Encryption Standard (AES). A similar testbeds shown in Figure 2 was used to deploy and test the SNAP protocol.

A. *Assumptions in SNAP implementation*

The following are the assumptions in the implementation of the SNAP protocol.

- The resource broker runs the SNAP protocol to manage the access to the RMI based services running in the globus environment.
- The various entities considered in the current resource management architecture are the client (that request the resource), the resource broker that manages the access to the resource and the resource itself.
- The SNAP protocol implementation has been done using the Java Remote Method Invocation (RMI) technique. The SNAP is provided with various interfaces for accessing the services available in GT3, especially for accessing the GRAM service.
- The client must enter into an SLA agreement with the resource broker before gaining access to the resource.
    - If the agreement is successful then the client can access the resource.
    - If the agreement is not successful then the client request is made to wait in the queue.
- The SLA states, the information pertaining the SLA agreements are stored in a Java based Hashtable described later.

B. *Description of SNAP implementation*

In this section, we describe a prototyping system of resource management based on the model above and the SNAP protocol. The system allows for managing the process of negotiating access to, and use of, resources in a distributed systems. In contrast to other architectures that focus on managing particular types of resources (e.g. CPUs or network bandwidth), the SNAP defines a general framework within which reservation, acquisition, task submission, and binding of tasks to resources can be expressed for any resource in a uniform fashion. The SNAP protocol maintains a set of manager-side SLAs using the client-initiated messages. All SLAs contain an SLA identifier I, the client c with whom the SLA is made, and an expiration time tdead until which the SLA is alive as well as a specific SLA description d [2].

$$SLA \Rightarrow < I, c, tdead, d >$$

The identifier, I, is valid and live for time duration defined by tdead . There are four stages in the operation of the SNAP protocol viz:

- Allocate identifier operation
- Agreement operation
- Set termination operation
- SLA change operation

and they are described below. Reader is also referred to [2] for a more formal discussion on the
SNAP protocol design.

Allocate identifier Operation

The initial step in the negotiation of resources is to get allocated with an identifier that is valid by the time defined by tdead. The client can either explicitly specify the value of tdead or else a default value is assigned by the resource broker. The client sends:

getIdent(tdead)

asking the resource broker to allocate a new identifier that will be valid until time tdead . On success, the broker will respond with:

useIdent(I, tdead)

and the client can use this identifier to negotiate, reserve or acquire access to the resource [2].

Agreement Operation

After obtaining a valid identifier, the client can negotiate for an SLA using the valid identifier obtained. The client issues an SLA protocol message with the arguments expressed in the agreement language

request(I, c, tdead, a)

The SLA description, a, captures all the requirements of the client for the resources. On success, the resource broker will respond with the message of the form:

agree(I, c, tdead, a')

where $a' \subseteq a$. In other words, the resource broker agrees to the SLA description a’, and this SLA as said earlier will terminate at time tdead unless or until the client performs a setdeath(I, t) operation to change the scheduled lifetime.

Also due to the existence of a unique identifier for each SLA agreement between the client and the resource broker, the client is free to re-issue requests after a successful agreement. Under such circumstances the broker is supposed to treat these requests for acknowledgment on the existing agreement [2].

Set Termination Operation

As said earlier, each SLA has a termination time defined by tdead. With this operation, a client can now define its new termination time for the specified identifier. The client changes the lifetime by sending a message of the form:

setdeath(I, tnewdead)

here tnewdead is the new termination time of the SLA denoted by I. On success the broker will respond with the new termination time:

willdie(I, tnewdead)

Obviously, it is a negotiation process and the broker can reject the request. The client may reissue the setdeath ( ) message with the new expiration time if there is a response failure from the broker. Agreements can be abandoned with the simple request of the setdeath (I, 0) that forces the expiration of the agreement [2].

SLA Change Operation

The protocol also includes a common prototype of atomic change by allowing client to resend the request with the same SLA identifier, but with modified requirement content. The service will respond as for an initial request, or with an error if the given change is not possible from the existing SLA state [2]. When the response indicates a successful SLA, the client knows that any preceding agreement named by I has been replaced by the new one depicted in the response. When the response indicates failure, the client knows that the state is unchanged from before the request. Currently the only change operation supported by the protocol is the tdead value of the SLA.

The purpose of the change SLA operation is to preserve the state in the underlying resource behavior, e.g. a change in the bandwidth requirement of a job may provide a better QoS guarantees. Whether such a change in the resource specification is possible depends on the resource type, implementation, and local policy [2]. The SNAP protocol handshake messages are shown in Figure 2. Messages SLAstate and SLAstatus are newly included to get the current status of the SLA. The possible states of the SLA are S0 (initial state) , S3 (Active) and finally dead (when t > tdead).

IV. SNAP implementation semantics

The core of the SNAP architecture is a client-broker interaction used to negotiate SLAs. Each interaction is a unidirectional message sent from client to broker or broker to client. For a negotiation to be succssful and both the client and the broker enter the agreement, the broker needs to consult with the local resource manager who actual control the resource or Grid resource allocation manager. The interactions between the broker and local managers are not covered in this paper.

All of these operations follow a client-server remote method invocation technique. The underlying transport protocol is the TCP. One way of interpreting the protocol handshake described in Figure 2 is that the client to service message corresponds to the remote procedure call, and the return messages represent the possible result values of the call [2].

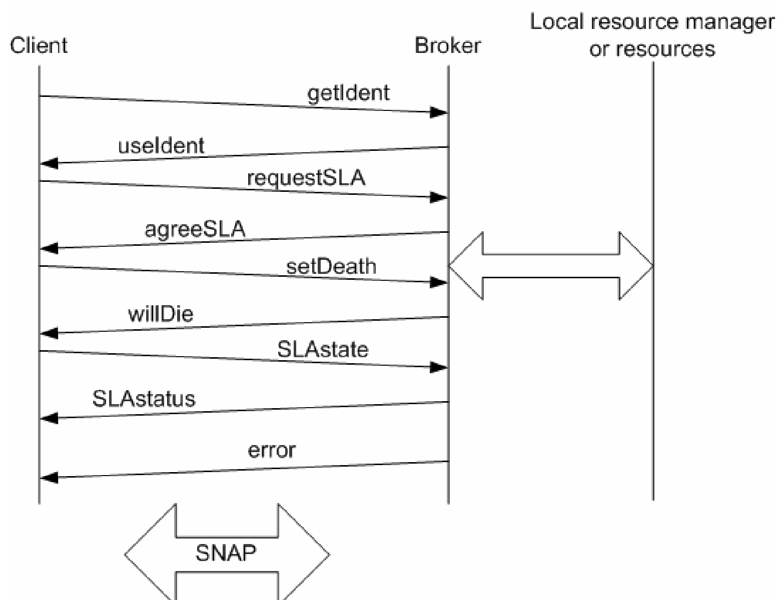

Figure 2: SNAP resource management Protocol messages

The SNAP protocol works as follows:

1. The client initiates the handshake for getting a unique job-identifier from the broker and the resource broker issues a unique identifier (I) that is active for a time duration (tdead).
2. The client I uses the identifier (I) issued by the broker to make SLA requests.
3. If the broker approves the SLA request by responding with an agree message then the client can submit the job and acquire the access to the requested resource.
4. The broker keeps track of all the current SLAs and their tdead values and decrements it every 5 seconds. If tdead value of any SLA reaches 0, then the SLA is considered to be dead and inactive.
5. The broker maintains a set of hash tables with the identifier I as the hash-key to store the following information viz (Table 1):
   - The first hash table maintains the identifier and the client information that access the resources whereas the third hash table maintains the identifier and the job description information.
   - The second hash table that contains the tdead which is accessed every 5 seconds and decremented by 5. If any of the value reaches zero then the identifier I is considered inactive and removed from the hash table and cannot be used by the clients to access the resource.
   - The final hash table 4 is used to keep track of all the SLAs and their current state, whether “Active” or “Dead”. Once the tdead value of any identifier I reach 0, the state of the identifier in the hash table 4 is updated to “Dead”.

	Hash Table	Description
1	Hash table 1	SLA identifier (I), client c information
2	Hash table 2	SLA identifier (I), SLA duration (tdead)
3	Hash table 3	SLA identifier (I), Job description (a)
4	Hash table 3	SLA identifier (I), state

Table 1: Java Hash-tables maintained by the broker

The resource broker follows the SNAP protocol by executing the following algorithm:

A. *Algorithm for Resource broker*

1. Initialize the hash tables 1, 2, 3 and 4 to null
2. When there is an incoming request for a new job identifier, respond with a unique identifier I (the duration of the SLA is specified along with the request).
3. Insert the details into the hash tables 2 and 4 as follows
   a. Hash table 2: Job Identifier, SLA duration (tdead)
   b. Hash table 4: Job Identifier, SLA state (S0)
4. If the client initiates an SLA request using the issued identifier I, the broker checks whether the requested resource specification using RSL, r, can be serviced.
5. If the request can be serviced issue a “agree” message.
6. If the request cannot be serviced issue a “disagree” message
   a. The client can re-negotiate with the resource broker with a relatively less resource value r’, i.e. r┌ < r.
   b. If r’ can be serviced by the broker then perform step 5 else perform step 6 and update the corresponding job identifier state to S3 in the hash table 4.
7. Insert the details into the hash tables 1 and 3 as follows
   a. Hash table 1: Job Identifier, client

   b. Hash table 3: Job Identifier, Job description
8. Update the hash table every 5 seconds. If any of the identifier tdead value reaches zero, remove the entry from the hash table 1,2 and update the SLA state in the hash table 4 to "dead".
9. Schedule the remote job requested by the client and returns the results.
10. If the client requests for the change of SLA duration tnewdead, the new SLA duration is updated in the hash table 2 and an acknowledge message is send back to the client.

The client follows the SNAP protocol by executing the following algorithm:

B. Algorithm for Client

1. Obtain a unique job identifier for negotiating SLAs with the broker.
2. Request an SLA for the resource request with the job identifier, SLA duration and the resource specification r, using the RSL
   a. If the response is "agree" then the client can proceed to submit the job request to the broker.
   b. If the response is "disagree" then the client has to re-negotiate the resource request, i.e. r┌ < r.
3. The client has to loop in step 2 until the broker responds with the "agree" message.
4. Once the "agree" message is obtained the client can now submit the job request to the resource broker.

V. SNAP Application Programming Interface

The client should initially obtain a remote object handle from the SNAP Server using the lookup service of RMI to negotiate the resource requirements with the broker. The client uses the "snap.ini" file to identify the host machine where the SNAP service is running. The entry specified in the snap.ini file specifies the IP address of the host in which the service is running. The end user is supposed to configure the snap.ini file initially.

```
snapServer = remoteAddress
ObjectHandle objH = (snap)Naming.lookup(‘‘rmi://’’ + host + ‘‘/snapService’’)
```

Once the client obtains the remote object handle , the object handle can be used to invoke the API's on the server to initiate the SLA negotiations with the resource broker. As per the SNAP handshake model, obtain a unique identifier from the broker using

```
JOBID = objH.getIdent(tdead)
```

After obtaining the identifier using the above API , initiate the requestSLA using the obtained OBID. The response from the broker will be either "agree" or "disagree" message. If the response s "agree" the client can proceed to submit the job. The API errMsg is used to return the error essages that occur in the process of negotiations. The error message pertaining to the JOBID s returned to the client. The setDeath API can be used to update the tdead value of the existing OBID in hash table 2 (from Table 4). If the tnewdead value is 0 then the corresponding JOBID s removed from the hash table 2 and the status is updated in hash table 4 as "dead". The code nippet is as follows:

```
objH.requestSLA(JOBID, hostname, tdead , desc);
response = objH.responseSLA(JOBID, hostname , tdead ,desc);
```

```
errresponse = objH.errMsg(JOBID, errmsg);
objH.setDeath(JOBID, tnewdead);
response = objH.submitJob(JOBID , no of threads);
```

On the other hand, the type of resource managed by the broker is a heterogeneous collection of RMI services scattered across the various nodes in the network. The broker with the help of an ini file namely broker.ini accomplishes the resource discovery and mapping. The number of entries in the file is proportional to the number of services managed by the SNAP . The entries specified in the file are, for example,

```
Start
GRAMService = remoteAddress1
TimeService = remoteAddress2
End
```

SNAP also provides API's for accessing the Globus services.

VI. Evaluation

The SNAP protocol was deployed on a Windows machine. The protocol has interfaces that can communicate with the GT3 services that are running on a Linux platform. A Java-based client is used to connect to the SNAP host machine. Once connected, the time elapsed for the series of handshakes between the client and the broker are measured and tabulated below for various number of runs (Table 2). The jobs that are triggered by the remote clients are computation intensive multi-threaded Java programs and hence the time taken to completion is in the range (66.418 , 68.252) sec. The time taken for completion of these jobs varies proportionally with the number of concurrently running jobs.

Handshake messages	Elapsed Time for Handshake		
	Run 1 (sec.)	Run 2 (sec.)	Run 3 (sec.)
getIdent	0.047	0.047	0.087
RequestSLA	0.375	0.079	0.153
SubmitJob	66.418	66.729	68.252
hline SetDeath	0.042	0.047	0.042

Table 2: Elapsed Time for Handshake messages

N=4	Job 1 (Kb)	Job 2 (Kb)	Job 3 (Kb)	Job 4 (Kb)
1	854.7422	645.52344	655.9766	749.5703
2	703.1797	559.5078	606.2188	663.2344
3	653.0547	746.5703	793.3281	886.9219
4	840.08594	970.2578	1017.016	627.8672
5	581.03125	674.625	757.9609	851.5547
6	804.7969	898.3125	981.6484	697.3438
7	1033.1797	632.53906	693.9375	838.6719
8	791.83594	885.4297	946.75	1040.336
9	1030.0781	603.64844	583.9609	687.7422
10	667.375	734.5	791.5	791.5
11	677.5625	744.66406	801.6641	822.1172

Table 3: Memory used by Jobs when N=4

N=5	Job 1 (Kb)	Job 2 (Kb)	Job 3 (Kb)	Job 4 (Kb)	Job 5 (Kb)
1	666.10156	824.0469	915.4453	611.6484	667.8047
2	555.5703	751.9453	825.2734	891.4922	947.375
3	845.46875	1004.0547	582.6875	648.75	704.6172
4	602.8047	760.5	862.0469	928.125	965.8594
5	872.2422	1044.5391	632.7188	734.6953	773.0703
6	679.33594	861.02344	953.7344	1002	1039
7	973.65625	669.3594	725.2422	687.4453	735.375
8	724.4531	938.125	994.0078	1059.828	1116.117
9	1105.8594	761.91406	562.3047	618.1875	694.4141
10	674.0625	796.0781	816.5313	882.6719	903.0391
11	684.25	806.27344	826.7969	892.8594	913.3125

Table 4: Memory used by Jobs when N=5

The computation intensive Java programs that are triggered by the client are locally managed Jobs by the resource broker. Since these jobs are multi-thread Java programs they share the CPU execution time and the memory space provided by the Java Virtual Machine (JVM) in a time sliced pre-emptive manner. The SNAP protocol is capable of keeping track of the memory used by these jobs at any instant of time. The memory used by 4 and 5 concurrently executing threads (N) are determined separately (Table 4 and 5 and Figure 3 and 4).

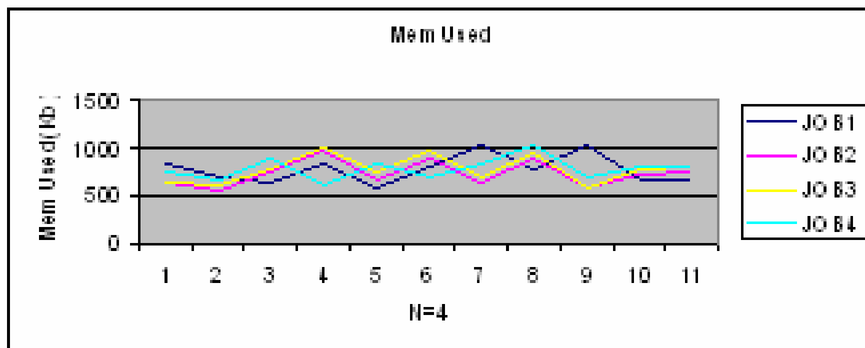

Figure 3: Memory used by 4 concurrent jobs

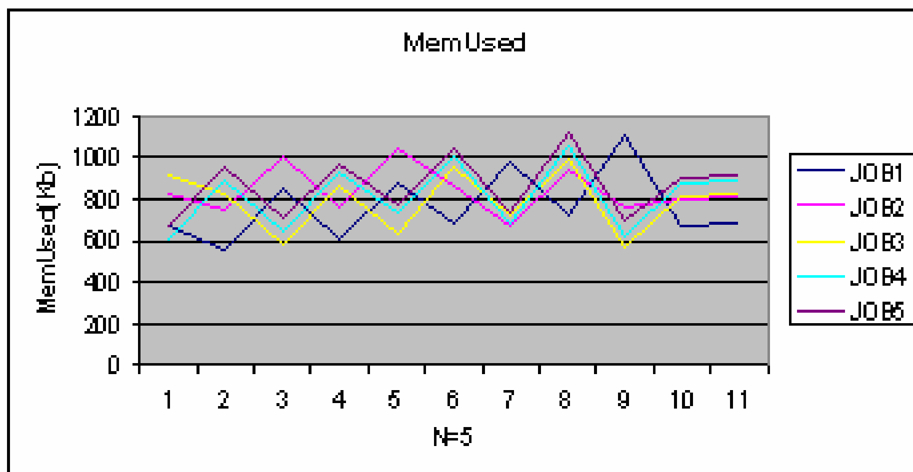

Figure 4: Memory used by 5 concurrent jobs

VII. Conclusion

The Grid resource management is advancing towards agreement-based approach. In this paper, we presented an implementation of a Grid resource management system based on SNAP's design in a GT3 environment. A layered implementation model is proposed to provide a service for a virtualized resource. The resource is then accessed a service via well-defined interface. The Java RMI based SNAP protocol developed thus provides a mechanism for negotiating the resource requirements with the broker. In our prototyping, the SNAP is capable of supporting any number of services provided they have an entry specified in the "broker.ini" file, in this way they will be able to perform an efficient resource discovery. The memory tracking for the locally managed jobs is an added feature of the prototyping of SNAP. The proposed security framework for SNAP using the Java Cryptography Extension (JCE) is an efficient substitute for the GSI that provides service authentication and authorization. Once the SNAP protocol is integrated with the GSI feature of GT3 the proposed security framework may become obsolete.

References

[1] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pages 62–82, 1998.

[2] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating ResourceManagement in Distributed Systems. In JSSPP-2002, Lecture Notes in Computer Science 2537, pages 153–183. Springer-Verlag, 2002.

[3] Karl Czajkowski, Ian Foster, and Carl Kesselman. Co-allocation services for computational grids. In 8th Proc. IEEE In. Symp. High Performance Distributed Computing, 1999.

[4] Karl Czajkowski, Ian Foster, and Carl Kesselman. Agreement-Based Resource Management. PROCEEDINGS OF THE IEEE, 93(3):631–643, March 2005.

[5] S. Tuecke I. Foster, C. Kesselman. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications, 15(3), 2001.

[6] Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors. Grid Resource Management. Kluwer Academic Publishers, 2003.

[7] Anand Natrajan, Marty A. Humphrey, and Andrew S. Grimshaw. Grid Resource Management in Legion. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, Grid Resource Management, chapter 9, pages 145–160. Kluwer Academic Publishers, 2004.

[8] K. Nichols, V. Jacobson, and L. Zhang. A Two Bit Differentiated Services Architecture for the Internet. IETF RFC 2638, July 1999.

[9] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing . In Proc. 7th IEEE Int. Symp. High Performance Distributed Computing,, 1998.

[10] Alain Roy and Miron Livny. Condor and Preemptive Resume Scheduling. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, Grid Resource Management, chapter 9, pages 135–144. Kluwer Academic Publishers, 2004.

[11] Arie Shoshani, Alexander Sim, and Junmin Gu. Storage Resource Managers. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, Grid Resource Management, chapter 20, pages 321–340. Kluwer Academic Publishers, 2004.

Dr. Yang, Zhonghua is currently an associate professor at Information Communication Institute of Singapore, School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. He has research interests in various aspects of Grid computing, Semantic Web and Semantic Grid, Autonomic and service-oriented computing, and software agents. He is a Programme Director (Grid Computing) in the School. Dr. Yang is involved primarily in teaching Web Services, Data Structures and Algorithms, Software Engineering, Software Systems and their development, and Distributed Computing.

Prior to the current appointment, Dr Yang had an extensive university and industry career which includes at Griffith University (Australia), University of Alberta (Canada), and Imperial College (London, UK), Distributed Systems Technology Center (DSTC), University of Queensland, Australia, Singapore Institute of Manufacturing Technology (SIMTech). Dr. Yang spent a significant part of his professional life with the Ministry of Aerospace of China, serving in various senior positions.

Rajbabu Gunasheelan’s photo and his bio are not available at the time of the publication