

AN ADVANCE RESERVATION-BASED COMPUTATION RESOURCE MANAGER FOR GLOBAL SCHEDULING

HIDEMOTO NAKADA, ATSUKO TAKEFUSA, KATSUHIKO OOKUBO,
TOMOHIRO KUDOH, YOSHIO TANAKA, AND SATOSHI SEKIGUCHI
*National Institute of Advanced Industrial Science and Technology (AIST) Grid
Technology Research Center 1-18-13 Sotokanda, Chiyoda-ku, Tokyo, 1010021,
Japan,*
*{hide-nakada, atsuko.takefusa, ookubo-k, t.kudoh, yoshio.tanaka,
s.sekiguchi}@aist.go.jp*

Advance Reservation is one possible way to enable resource co-allocation on the Grid. This method requires all the resources to have advance reservation capability as well as coordination protocol support. We employed 2-phased commit protocol as a coordination protocol, which is common in the distributed transaction area, and implemented an Advance Reservation Manager called **PluS**. PluS works with existing local queuing managers, such as TORQUE or Grid Engine, and provides users advance reservation capability. To provide the capability, there are two implementation methods; 1) completely replaces the scheduling module of the queuing manger, 2) represents reservation as a queue and controls the queues using external interface. We designed and implemented a reservation manager with both way, and evaluated them. We found that the former has smaller overhead and allows arbitrary scheduling policy, while the latter is much easier to implement with acceptable response time.

1. Introduction

One of the main goals of the Grids research is to co-allocate several resources that spans widely on the network, and perform huge computation on it. And advance reservation is one possible way to enable resource co-allocation on the Grid. All the resources have its own local scheduler with advance reservation capability and the super scheduler co-allocates all the resources by making reservation on all the resources on a specified timeslot.

One important thing here is the protocol between super scheduler and the local resource manager. Co-allocation of several resources is essentially a kind of distributed transaction. To guarantee acceptable behavior on the operation failure, super scheduler and local resource managers have to employ proper protocol between them.

Therefore, we need to have following three things to make the advance reservation based co-allocation happen: 1) A super scheduler that supports advance reservation, 2) Local schedulers that provide advance reservation, 3) Proper bridge protocols to harness 1) and 2)

We already proposed a scheduler that can co-allocate network and computation resources ⁶ as 1), and WSRF (Web Services Resource Framework) based advance reservation protocol ⁵ as 3). In this paper, we describe design and implementation of an advance reservation manager called PluS as 2). PluS supports two-phased commit protocol, which is commonly used distributed transaction area, as a co-allocation protocol. It works with widely used existing local schedulers, namely, TORQUE ⁴ and Grid Engine ² and provides them with advance reservation capability.

There are two methods to ‘add’ advance reservation capability to the existing local queuing system; 1) completely replace scheduling module in the local queuing system with the one does support the advance reservation, 2) keep the scheduling module as is and add another module that control queues to make the advance reservation happen. We designed and implemented our advance reservation manager PluS in both methods and evaluated them.

The result showed that the former has smaller overhead and flexibility to allow implementers for setting up reservation policy, while it requires full re-implementation of the scheduling module putting a huge burden on the implementers. The latter is easy to implement but restricted in setting policy and have acceptable but larger overhead.

2. Coallocation and two-phased commit protocol

Here, we demonstrate the needs for commit protocol for co-allocation, showing an example. Assume that we have two resources (A and B) and already made reservation for specific timeslot on both of them, and want to move the timeslot, say, 1 hour later. In the naive implementation, it will issue modification requests to resource A and B, sequentially. However, this implementation is potentially problematic. Assume that the modification succeeded in resource A and failed in resource B. The expected behavior will be to give up the modification and revert to the original situation, keeping the original reservation time slot. Note that this is not always possible, since the reservation timeslot for resource A is already modified and there is no guarantee that the previous timeslot is still available for reservation. In the worst case, the reservation modification results in failure and the

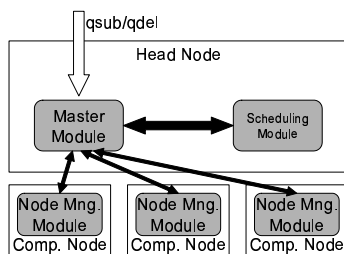


Figure 1. Generic Configuration of Queuing Systems.

timeslot previously reserved is lost.

Co-allocation of several resources boils down to distributed transaction. The distributed transaction has been investigated for long time ⁷, and several protocols are proposed to cope with it. The most basic protocol among them is the two-phased commit protocol. The point of the protocol is to postpone the commitment of the operation until the second turn of the communication. When the super scheduler is about to perform some operation, it issues commit-requests to all the concerned local schedulers. If all the local schedulers replied ready-to-commit, the super scheduler performs commit operation. With this protocol, the situation shown above can be avoided. There are several queuing systems that support advance reservation, but none of them does support two-phased commit protocol.

3. Design of PluS

3.1. Generic design of Batch Queuing Systems

In general, a batch queuing system consists of a *head node* and several *compute nodes*. The head node is the submission node through which users submit their jobs. Compute nodes are the worker nodes that actually execute jobs on requests from the head node. Note that these functions are not exclusive. It is possible for one physical node works as both of them.

The head node functionality is realized by two separate modules, typically; *master module* and *scheduling module*. The compute nodes functionality is implemented by *node management module* (figure 1).

Master module: The master module is the central module of the whole queuing system. The roll of the module can be categorized into three as follows: 1) management of job queue, 2) remote management of compute nodes, 3) initiate scheduling cycle and execution of the scheduling assignment. The master module receives job management requests, such as submission, cancellation, monitoring of jobs, from users. At also communicates

with node management modules on compute nodes and keep track of the status of each node. It periodically (or on some events) initiates scheduling for the scheduling module and performs the assignments decided by the scheduling module, by giving orders to the node management modules. This module is called 'pbs_server' in TORQUE and 'sge_qmaster' in Grid Engine, respectively.

Scheduling Module: This module is responsible for the scheduling; i.e., allocation and assignment of compute nodes to jobs. It obtains information on compute nodes and jobs and base on the information, determine the allocation and assignment. This module is called 'pbs_sched' in TORQUE and 'sge_schedd' in Grid Engine, respectively.

Node Management Module: Node Management Module is the module that is responsible for several aspects of managing computation node, such as periodic monitoring of the load average, available memory amount, and available storage amount, and reporting them to the master module, as well as invocation, termination, monitoring of jobs. This module is called 'pbs_mom' in TORQUE, and 'sge_execd' in Grid Engine, respectively.

3.2. Job Queue

Job queue is a basic concept in the queuing systems. Job queue manages jobs submitted by users in (basically) FIFO (First In First Out) fashion, and schedule them one by one. Most queuing systems are capable of managing several queues. Each queue can be assigned dedicated computational nodes, enabling to manage single cluster as separated independent computing facility. Most queuing systems can be set up so that allow specific user group to submit jobs into specific queues.

3.3. Implementation methods for Advance Reservation

To add the advance reservation capability to the queuing systems shown above, there are following strategies: 1) Completely replace the existing scheduling module with specially crafted module with advance reservation, 2) Control job queues and mappings with nodes and users from external module.

Scheduling Module Replace Method:

In this method, scheduling module in the queuing system will be completely replaced by the newly implemented module. The module receives reservation requests from users and returns reservation IDs for each request. The users submit jobs with the reservation IDs. The master module

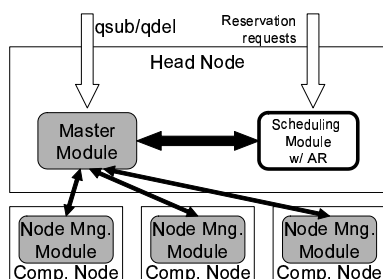


Figure 2. Replacing Scheduling Module Method.

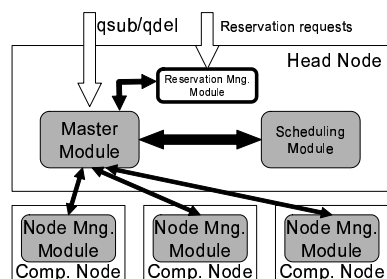


Figure 3. External Queue Control Method.

asks the scheduling decision for the replaced scheduling module, passing job information including the reservation IDs. The scheduling module will allocate node for the reserved jobs only when the timeslots are reserved for the jobs (figure 2).

This method has advantage over the other in the aspect of the policy setting latitude. This method will completely replace the 'heart' of the scheduling system, giving implementers freedom to setting arbitrary policy.

On the other hand, this method has several disadvantages especially in its implementation. Firstly, the implementers have to know the communication protocol between master module and scheduling module. Adding to it, there is no guarantee that the protocol will stay the same when the queuing system upgraded. It means that the implementers have to keep fixing the module to keep up the upgrade.

Secondly, the method requires re-implementation of the existing scheduling module functionality. Assume that we want to modify a working queuing system so that it will accept advance reservation requests. We have to re-implement all the functionality used on the site at least to guarantee the behavior of the system unchanged except for the reservations. This requires a lot of works in general.

External Queue Control Method: In this method, advance reservation capability will be implemented in the independent reservation management module, and the reserved timeslots will be represented as queues. The reservation management module will respond to the requests from users (figure 3).

The reservation management module dynamically creates queues on reservation requests from the users, and returns the name of the queue. It utilizes command line interface provided by the queuing system for creating queues. The queues will be created in *inactive* status; the queue can

store jobs user submitted, but does not actually run them. It will be setup so that only the specific users can submit to it. When the reserved time arrives, the reservation management module activate the queue so that the jobs in the queue can run, as well as control the other queues so that they do not use the nodes allocated to the reservation queue.

The largest advantage of the method is that, it can easily guarantee that the behavior of the queuing system to stay the same except for the reservation. The reservation management module is completely external and do nothing without reservation request. The implementation itself is also easier. The disadvantage is that it requires queuing system to support several queue related functionalities; the queuing system have to tie a queue to specific compute node set and user set. It is not so demanding but there are several queuing systems that do not support this, including TORQUE.

The other potential disadvantage is the extra cost to control the queue using command line interface. This method requires several times queue control command invocation when processing reservation requests as well as when it makes the reservations happen.

4. Implementation of PluS

4.1. Overview of the PluS Reservation Manager

We implemented PluS based on two methods shown in previous section. PluS works with TORQUE and Grid Engine and it is implemented thoroughly in Java for portability and high productivity, except for few communication modules written in C for compatibility. We implemented a version for TORQUE in the scheduling module replace method and two versions for Grid Engine with both of the methods. Note that both of the two implementations with the scheduling module replace method are somewhat ‘subset’; we implemented only the essential portions of the scheduling module and a few functionalities are left unimplemented. We could not implement a version for TORQUE with the queue control method, since TORQUE lacks required capability to implement the method.

PluS provides command line interface to operate with PluS. Table 1 shows a list of the command. Note that some of them have `-T` flag that denote ‘two phased operation’. With this option the operation made by the command will not complete immediately. Instead, the operation will remain in the ‘wait for commit’ status. Successive `plus_commit` (or `plus_abort`) will commit (or abort) the operation. Each command is written in small shell script that wraps around a Java written client program. The program

Table 1. Commands for Reservation Management

name	function	inputs	outputs
plus_reserve	Request Reservation	Requirements	RSV_ID
plus_cancel	Cancel Reservation	RSV_ID	
plus_modify	Modify Reservation	RSV_ID, Requirements	
plus_status	Show Reservations	RSV_ID	Reservation Status
plus_commit	Commit Reservation Operations	RSV_ID	
plus_abort	Abort Reservation Operations	RSV_ID	

communicates with the PluS reservation management module with RMI.

The PluS reservation management module maintains the reservation table in it. The table has to be persistent to guarantee the table to survive the head node reboot or crush. We employed Java native object database *db4objects*¹. Its interface was quite simple and easy to use and contributed to make our implementation time shorter.

4.2. Advance Reservation Policy in PluS

Current implementation of PluS prioritize the advance reservation over the ordinary queued jobs. The reservation is only restricted by the existence of the other reservations and is not affected by existence of the queued jobs.

I.e., jobs with advance reservation always have higher priority than the non-reserved jobs and kick out them when needed. For example, assume that a user wants to reserve a node, from 10 min. later for 1 hour. Even though all the compute nodes are occupied by non-reserved jobs and there will not be vacant nodes 10 min. later, the reservation request will succeed. 10 min. later, the PluS reservation management module kicks out a running job from a compute node so that the reserved job can use the node.

This policy is effective when the site prioritizes coordination with other resources and treats the local jobs as the backfill jobs.

4.3. PluS for TORQUE

TORQUE is a descendant of the OpenPBS which is an open source queuing system has been not maintained for years. Since the queues in TORQUE is not adequate for the queue control method implementation as mentioned above, we implemented the scheduling module replacement method for TORQUE

In TORQUE the protocol used between the master module (*pbs_server*)

and scheduling module (`pbs_sched`) is relatively simple, text-based protocol. we reverse-engineered the protocol and developed a scheduling module that can communicate with this protocol.

4.4. *PluS for Grid Engine*

Grid Engine is a queuing scheduler developed by Sun Microsystems, which is widely used for many projects in the world. We implemented PluS with both of the two methods. The scheduling module replacement method for Grid Engine is done in just the same way with TORQUE, except for the fact that the protocol used there is a binary and we have to implement a module to translate plain XML notation. Implementation with the queue control method works as shown in figure 4.

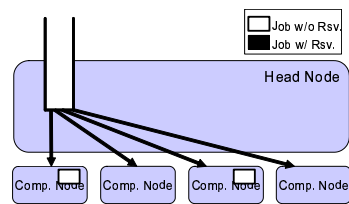
5. Evaluation

In this section we show comparisons between the scheduling module replacement method and the queue control method. We evaluate ease of implementation based on number of lines of code and command execution speed for reserve/cancel operation.

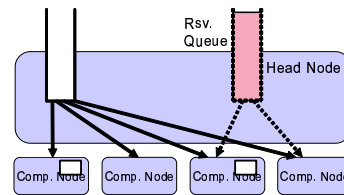
5.1. *Evaluation based on lines of codes*

Here, we show the lines of codes required for each implementations. Currently, we have three PluS implementations; for TORQUE, for SGE with replacement method, and for SGE with queue control method. We counted the number of lines of codes for each implementation. While the number of lines might not accurately reflect easiness of implementation, especially the implementation languages span from C to Java and sh, it still makes fair index for easiness. The three implementations share some portions of codes, such as for reservation and allocation management, command line interface, as well as dedicated codes specific for each implementation. Figure 5 shows the number of lines for each implementation. The shared 8000 lines are shown as the underlying portion.

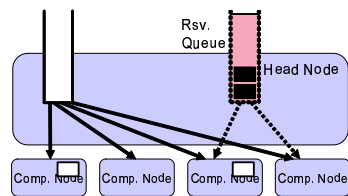
The dedicated portion is about 3000 lines for TORQUE, about 5200 lines for scheduling module replacement version for Grid Engine, and about 1800 lines for queue control version for Grid Engine. We can see that queue control version requires smallest dedicated code, proving that easiness of implementation of the method. Please note that the two implementations for scheduling module replacement method are not complete, i.e., they do



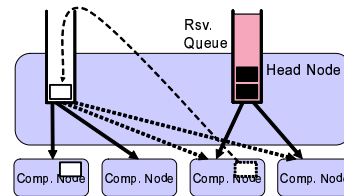
(A) Initial Status. The queue shown left is the default queue that is tied up to all the compute nodes.



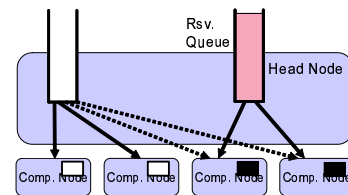
(B) Reservation made. When the reservation management module receives a reservation request from user, it creates a queue (on the right) in suspended status and returns the name of the queue as the reservation ID. The queue is tied to specific compute nodes, but is not allowed to assign jobs to the nodes.



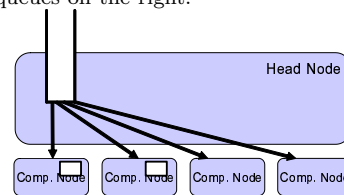
(C) The users submit jobs to the newly created queue. The jobs will not be assigned until the reservation start time, since the queue is created as suspended.



(D) Reservation period starts. The queue is activated. If the compute nodes already have running jobs, Plus kills the jobs and resubmits them to the original queue, with *qresub* command. The jobs will start over on the other compute nodes. The default queue has lost control over the two queues on the right.



(E) During the reservation period. The preempted job is re-assigned to the other compute node.



(F) Reservation period is over. The reservation management module deactivates and removes the reservation queue. If some jobs for the queue are still running, reservation management module will kill them. It also re-configure other queues so that they can use the compute nodes

Figure 4. Implementation with the queue control method.

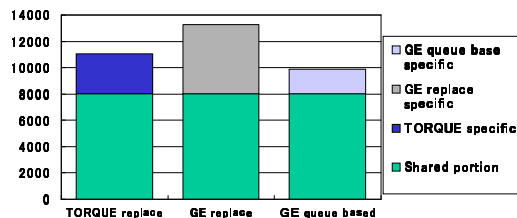


Figure 5. Evaluation based on Num. of lines.

not support whole functionality the original scheduling module has. It means that implementation of the whole functionality will require much more lines of codes.

5.2. Time to make/cancel Reservations

There is anticipation on the queue control method; it might have extra overhead on reservation operations due to queue control commands issue. We evaluated the overhead by comparing the execution time with Plus implementation for Grid Engine with both methods.

The evaluation was performed on a small cluster with one head node and four compute nodes with Pentium III 1.4 GHz Dual CPUs, with 2 GBytes memory, Redhat 8 Linux installed. We measured time spent for making and canceling reservation using the *time* command.

We performed 10 experiments for each and got average number; 1.02[s] for reservation and 0.92[s] for cancellation with the Scheduling Module Replacement Method, and 1.95[s] for reservation and 1.02[s] for cancellation with the Queue Control Method. We can see that while both of them are acceptably fast with the scheduling module replacement method is slightly faster than the queue control method. The difference comes from *qconf* command invocation which is required only in the queue control method. We can also see that the difference between two methods is substantial for making reservation, while it is not for cancellation. This is because four *qstat* invocations are required for making reservation, while just one is required for cancellation.

6. Related Work

Several commercial batch queuing systems, such as PBS Professional or LSF, have advance reservation capability. There are also plug-in scheduling modules for existing batch queuing system, that support advance reservation, such as Maui and Catalina, They all does not provide two-phased

commit protocol. The reason why they do not support two-phased commit is that they are not meant for fully automatic co-allocation but for reservation only for the single site.

Maui: Maui Scheduler ³ is a plug-in scheduler for TORQUE, which is developed by the Cluster Resources Inc., the maintainer of the TORQUE.

^a Maui scheduler is implemented in the scheduling module replacement method, i.e., it completely replaces the scheduling module of TORQUE.

Catalina: Catalina ⁸ is a scheduling module that can work with TORQUE and Load Leveler. It is used in the Tera Grid project in US. Catalina provides advance reservation capability, which is called 'User-Settable Reservation', in Catalina terminology. Catalina is implemented in the scheduling module replacement method. Another significant feature of Catalina is that it is totally written in Python, allowing administrators to modify parameters embedded in the source code.

Catalina prioritize jobs from ordinary queues over the reservation, while PluS prioritize the reservation. Reservations are possible only when there are no jobs scheduled for the time period, from the ordinary queue,

7. Conclusion

We proposed a reservation management system PluS that supports two-phased commit protocol, to allow safe and efficient resource co-allocation on the Grids. PluS works with existing queuing system such as TORQUE or Grid Engine and provides them with advance reservation capability. We proposed two implementation methods; scheduling module replacement method and queue control method, and actually implemented PluS in both way, and evaluated them.

We found that both of them have advantage and disadvantage. The former has smaller overhead and flexibility to allow implementers for setting up reservation policy, while it requires full re-implementation of the scheduling module putting a huge burden on the implementers. The latter is easy to implement but restricted in setting policy and have acceptable but larger overhead.

For future work, we will address the following issues:

Resource specification improvements: Current implementation assumes that the compute nodes are homogeneous and does not allow specifying the resource characteristics, such as architecture, amount of memory or disk space size. We will address this.

^aMaui did work with Grid Engine previously, but current version does not.

Sophisticated reservation policy: As described in 4.2, current implementation always gives highest priority on the reservation and the reservations are made first-comes-first-served base. Obviously, these will not be acceptable for the production cluster administrators. We are designing a mechanism to allow administrators to setup their own policy in a simple script language, so that each administrator can describe a policy suitable for his/her system.

Application to the other queuing systems: The queue control method implementation will be easily applicable to the other queuing systems, if the system supports queue control mechanism required by PluS. We will confirm the easiness through application of the method to the other queuing systems, such as Load Leveler or Condor.

Acknowledgement

This work is partly funded by the Science and Technology Promotion Program's "Optical Paths Network Provisioning based on Grid Technologies" of MEXT, Japan.

References

1. db4objects. <http://www.db4o.com/>.
2. Grid Engine. <http://gridengine.sunsource.net>.
3. Maui cluster scheduler. <http://www.clusterresources.com/pages/products/-maui-cluster-scheduler.php>.
4. TORQUE Resource Manager. <http://www.clusterresources.com/pages/products/torque-resource-manager.php>.
5. Hidemoto Nakada, Atsuko Takefusa, Katsuhiko Ookubo, Makoto Kishimoto, Tomohiro Kudoh, Yoshio Tanaka, and Satoshi Sekiguchi. Design and implementation of a local scheduling system with advance reservation for co-allocation on the grid. In *Proceedings of CIT2006*, 2006.
6. Atsuko Takefusa, Michiaki Hayashi, Naohide Nagatsu, Hidemoto Nakada, Tomohiro Kudoh, Takahiro Miyamoto, Tomohiro Otani, Hideaki Tanaka, Masatoshi Suzuki, Yasunori Sameshima, Wataru Imajuku, Masahiko Jinno, Yoshihiro Takigawa, Shuichi Okamoto, Yoshio Tanaka, and Satoshi Sekiguchi. G-lambda: Coordination of a grid scheduler and lambda path service over gmpls. *Future Generation Computing Systems*, 22(2006):868–875, 2006.
7. Andrew S. Tannenbaum. *Distributed Operating Systems*. Prentice Hall, 1994.
8. Kenneth Yoshimoto, Patricia Kovatch, and Phil Andrews. Co-scheduling with user-settable reservations. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 146–156. Springer Verlag, 2005.