Flexible Resource Reservation Using Slack Time for Service Grid

Chunming Hu, Jinpeng Huai, Tianyu Wo
School of Computer Science and Engineering
Beihang University
Beijing, China
{hucm, huaijp, woty}@act.buaa.edu.cn

Abstract

Providing guaranteed QoS for Grid services using resource reservation and allocation is an important feature for today's service Grid. Reservation requests with existing mechanisms are often rejected during the resource utilization peak and lead to resource capacity fragment problem. In this paper, we propose a flexible capacity reservation mechanism, called FIRST, which employs the slack time-enabled request admission control with differentiated selection strategies. We implement the prototype of FIRST in our CROWN node server, the service container of CROWN service grid middleware. The performance of the admission control algorithm for FIRST is evaluated by comprehensive simulations and implementations. Experimental results show that a better resource utilization ratio can be achieved by introducing the slack time into fixed reservation scenario, and a min-min based request selection strategy obtains a better performance compared with existing strategies.

Keyword: service grid, resource reservation, quality of service, admission control, slack time, FIRST

1. Introduction

Grid computing has emerged as a new paradigm of distributed computing technology since mid 1990s [1,2]. It focuses on large-scale resource sharing and coordinated problem solving. Providing non-trivial quality of service (QoS) is one of the primary goals of the Grid approaches. Many applications largely depend on obtaining results within particular QoS requirements. In service grids, resources are often wrapped by Grid services with particular interfaces. User jobs are executed via a set of service invocations. Correct execution of these jobs always requires an end to end provision of high quality of service. This end to end QoS needs to be achieved and guaranteed through proper configuration, reservation and allocation of corresponding resources at each service invocation procedure [4].

Although many efforts have been made on the development of basic grid components such as execution framework and resource brokering system, there are still many issues need to be addressed before we could provide specific QoS guarantees in today's service grids. For example, resource reservation is a popular way to provide guaranteed QoS. Most resource reservation mechanisms require admission test before a client is allowed to use the resource. In each reservation request, a client should provide detailed parameters to identify, when the reservation will start, how long the reservation will last, and how many resource capacities are needed. However, current reservation mechanisms with fixed parameters often lead to rejection due to the peaks of resource utilization and cause resource capacity fragment problem.

In order to solve the problem, previous researchers propose several flexible reservation mechanisms [5][6], in which reservation parameters can be modified based on resource status, such that the gaps of existing allocated reservations are filled. The disadvantage of such approaches is that they need the local resource manager to adjust the allocation of resource capacity during runtime, incurring extra limitations and overhead.

In this paper, we propose a novel mechanism called *FlexIble Reservation using Slack Time* (FIRST), in which reservations are started in particular time spans according to the slack times specified by the reservation clients, so that the local managers can better allocate resource capacity to requests. This design is based on the observation that not all the users require the submitted job being executed immediately. In many cases, users attach a deadline for each job. Such a deadline often means an upper bound to the job execution time. Thus, the local resource manager may have slack time for job executions.

The rest of this paper is organized as follows. In Section 2, we introduce related work. Basic idea of resource capacity management is discussed in Section 3. In Section 4 we present design of FIRST and introduce its prototype implementation in a real service grid environment. We evaluate the performance of FIRST in Section 5, and conclude the work in Section 6.

2. Related Work

Grid[1,2,16-19] and Peer-to-Peer [20,21] have recently gained much attention due to the high potential in sharing resources. In Grid environments QoS is one of the key issues.

Globus Advance Reservation Architecture (GARA) [4] is the most commonly known framework for supporting QoS in the context of the high-end applications based on Globus [1]. It provides advanced reservations with uniform treatment on various types of resources such as network bandwidth, CPU computational capability and storage. GARA guarantees the clients to receive the specific QoS with fixed reservation mechanism, where all parameters, including start time and the demanded amount of the resource capacity, must be provided when clients issue their request.

Grid Quality of Service Management (G-QoSm) [7] supports QoS management in computational grids in the context of the Open Grid Service Architecture (OGSA)[3]. As a basic building block of the G-QoSm, QoS Grid services (QGS) is used to provide capability of resource reservation and allocation. QGS uses a data structure to support the reservations for quantifiable resources, similar with the time slot mechanism discussed in this paper, but this technique does not use slack time to get better resource utilization.

In order to improve resource utilization of the reservations with fixed parameters, Xing and Wu propose a flexible advance reservation mechanism [5], in which the reservation parameters can be modified according to resource status in order to fill the gaps of existing allocated reservations. However, this mechanism requires the local resource manager to adjust the allocation of resource capacity during runtime, which may cause extra cost and make extra limitations on the type of resources.

How to make the allocation to different type of resource is not the focus of this paper. A lot of efforts have been put on allocating the resource capacity for CPU (e.g. DSRT[8]) or network bandwidth [6,9,10], forming the foundation of resource reservation in grid QoS management.

3. Resource Capacity Management

3.1 Resource Capacity and Its Metrics

Resource capacity is a kind of abstraction to grid resourced. Different resource has different set of capacities. For example, the computing power of CPU and the amount of available memory are the most important capacities for a computing resource. In a service grid environment, these resource capacities are wrapped by applications and services.

We define two basic types of resource capacities. The first type is the capacities which can be shared by multiple consumers, such as the bandwidth in a broadcast network, or the information carried by some experimental data. The amount of this type of capacity will not be decreased when more uses share it. For the second type capacities, we call them monopolized, once a request is granted, a certain amount of resource capacity is occupied, and the available capacity will be decreased, such as the computing power of CPU, the bandwidth and storage space. Indeed, the second type of resource capacity has larger impact on the performance of grid services. How to manage and allocate this type of resource capacity in an optimized way is the key issue we need to address.

To a certain type of resource, the total amount of monopolized resource capacity is limited and can be identified by several metrics. For example, the available computing power of a CPU can be identified by the percentage of un-allocated computing capacity or by metrics such as *MFlops* or *TFlops*.

When multiple applications are deployed in a single computing resource and all the applications are wrapped by grid services, the total amount of the computing capacity are shared among all the service instances. It is necessary to define the requirement of each capacity consumer and track the resource allocation status, such that the system may get the snapshot of the usage for each monopolized resource capacity at any time and perform the admission test to prevent the system from overload.

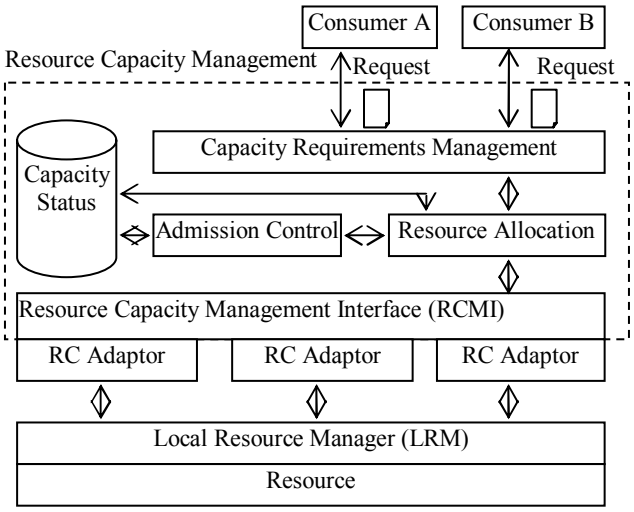

Figure 1. A framework for resource capacity management

We propose a basic framework for resource capacity management in Figure 1. The ultimate goal of the resource capacity management is to allocate resource capacity on demand so as to improve the efficiency of the grid-enabled resource utilization, especially when the total resource capacities are relatively limited.

In this framework, the capacity status is the kernel data structure for tracking current status of the available capacities. The capacity requirement management module receives all requests from the consumers and maintains the request queue. The resource allocation

and admission control module check the capacity status to evaluate the possibility of accepting new requests and try to give an optimized solution. Once the reservation request is accepted, the resource allocation module interacts with related resource capacity adaptors (RC adaptors) to perform the reservation at local resource manager layer. A unified programming interface called resource capacity management interface (RCMI) is employed to improve the extensibility of different RC adaptors.

3.2 Time Slot

Currently, timeslot table is always used to keep track of current allocations and future reservations of the available capacity of a resource if it is an enumerable quantity [4-6,11]. The basic idea of time slot is using an x-y reference frame (x>=0, y>=0) to present the amount of all the available resource capacity, in which x-axis is the time axis, and y-axis shows the available capacity at each time point. The function *Capacity (t)* stands for the maximum available resource capacity at time t where $t \in [t_0, +\infty)$. The area covered by $\int_{t_0}^{+\infty} Capacity(t)$ is the available capacities for the resource. In today's grid environments, the most commonly used *Capacity (t)* is given by

$$Capacity(t) = C_{MAX} \qquad (1a)$$

or

$$Capacity(t) = \begin{cases} C_{MAX1}, t \in [t_0, t_1) \\ C_{MAX2}, t \in [t_1, t_2) \\ ..... \\ C_{MAXn}, t \in [t_{n-1}, t_n) \end{cases} \qquad (1b)$$

Similarly, another function can be used to express a resource reservation request at time *t* is *CapacityDemand*(*t*). If *CapacityDemand*(*t*) is a constant, the request can be denoted as a rectangle in the x-y reference frame, as shown in Figure 2. Based on the above two functions, we can decide whether a new request can be satisfied and accepted at time *t*.

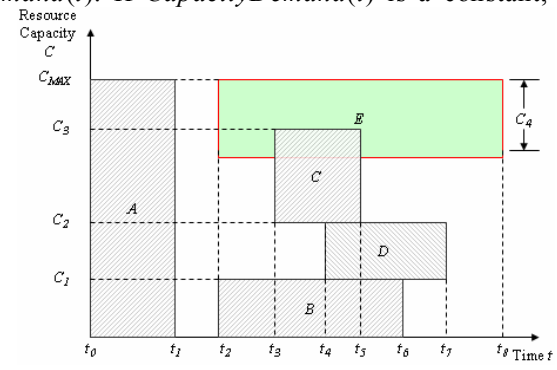

Figure 2: Time-slot based capacity management

Time slot can be represented by data structures. A simple way is to store all the accepted requests into an array. When a new request is received, a new record can be added to the array. That way, every time an admission test is performed, we have to go over all the array members and conduct comparison, which is not efficient. In order to improve efficiency, we introduce another data structure to represent the time slots, as illustrated in Figure 3.

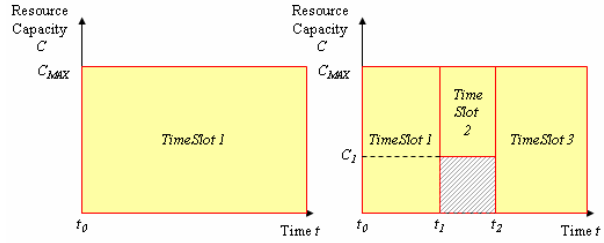

Figure 3: Reservation using 3 TimeSlot objects

When *Capacity*(*t*) is a constant, only one *TimeSlot* object (TimeSlot1) is used, with the start time of the object being t_0, and the end time $+\infty$. Once a request is accepted, requiring C_1 at time $[t_1,t_2)$, TimeSlot1 should be divided into 3 objects, TimeSlot1, TimeSlot2 and TimeSlot3, where the available capacity of TimeSlot2 is C_{MAX}- C_1-, and the available capacity of TimeSlot1 and TimeSlot3 are C_{MAX}. When *Capacity*(*t*) is presented as formula (1b), several *TimeSlot* objects should be used at the beginning to express various maximum available capacities within different time frame. If a request spans the different time frame, all the related *TimeSlot* objects should be modified simultaneously.

The major advantage of such a data structure is that all the *TimeSlot* objects can be sorted by the time frame they represent and shows the available amount of capacity in a distinct way. When a new request arrives, we can quickly find out all the related *TimeSlot* objects and check the available capacity.

3.3 Basic Admission Control Algorithm

Based on the data structure discussed in the previous section, we propose a basic admission control and allocation algorithm for resource capacity management. The major function of this algorithm is finding out the possible position for the new request without affecting allocated requests. Since all the *TimeSlot* objects are sorted by the time frame, we can easily identify the related *TimeSlot* objects and make the admission test.

The *BasicAllocation* algorithm is shown as follows.

```
(1)  Algorithm BasicAllocation()
(2)    Input:
```

```
(3)     ResourceCapacityRequest newReq;
(4)     ResourceCapacity resourceCapacity;
(5)     Output: Boolean;
(6)   {
(7)     n = get number of timeslots in resourceCapacity;
(8)     for (i=1; i<=n; i++) {
(9)     TimeSlot timeSlot = resourceCapacity.timeSlots[i];
(10)        if (((newReq.startTime>=timeSlot.startTime)
(11)        && (newReq.startTime<timeSlot.endTime))
(12)        || ((newReq.startTime<=timeSlot.startTime)
(13)        && (newReq.endTime>=timeSlot.endTime))
(14)        || ((newReq.startTime>=timeSlot.startTime)
(15)        && (newReq.endTime<=timeSlot.endTime))
(16)        || ((newReq.endTime>timeSlot.startTime)
(17)        && (newReq.endTime <=timeSlot.endTime))) {
(18)            if (timeSlot.remainingCapacity<newReq.
(19)                requiredCapacity) return FALSE;
(20)            }
(21)        }
(22)    return TRUE;
(23)  }
```

The input of the algorithm is the new request and a data structure discussed in the previous section. *ResourceCapacity* object is used as a container of all the *TimeSlot* objects. A *boolean* value is returned as the result of the admission test. In line 7, variable *n* denotes the total number of *TimeSlot* objects in the *ResourceCapacity* object. In the algorithm, each TimeSlot that overlaps with the new request (line 10-17) is examined to check whether there is enough resource capacity left for the new request (line 18).

Here we give a brief analysis for the proposed algorithm above. If a new request is accepted, 2 TimeSlots (that overlaps with start and end time of the new request separately) will be split into 4 or, in another case, one TimeSlot (that overlaps with both the start and end time of the new request) will be split into 3. So at most the number of time slots in *resourceCapacity* will be increased by 2. When n requests are accepted, in the worst case, 2*n TimeSlots will be generated. Thus the complexity of the algorithm should be *O(n)*.

4. Slack Reservation

Figure 2 shows the problem of resource capacity fragment using an example. In the scenario, the maximum available capacity is a constant function, i.e., $Capacity(t)=C_{MAX}$, where t_0 is the current time. A, B, C and D are the reservation of four accepted requests. Now consider a new request E, in which specific capacity amount (C_4) is required during time span [t_2, t_8]. Although there are enough capacity for E at time from t_2 to t_4 and from t_5 to t_7, we cannot accept it because no enough resource capacity at time t_4 to t_5 (here we assume $C_4 < C_{MAX} - C_3$). In this situation, E fails to pass the admission test and will be rejected by the system.

In the design of FIRST, we re-define *request* by adding a property called *SlackTime*. Reservation with slack time means that the demand of capacity may be started at any time during [*StartTime*, *StartTime*+*SlackTime*], so that the resource capacity manager is able to better place the accepted requests to improve the utilization of resources. The data structure of extended reservation request is listed as follows.

```
structure ResourceCapacityRequestWithSlack {
        ResourceCapacity capacity;      // type of the capacity
        DateTime startTime;             // start time
        TimeSpan duration;              // duration of the request
        TimeSpan slackTime;             // slack time of the request
        double requiredCapacity;
                                // the required amount of resource capacity
}
```

FIRST can be treated as a special case of the traditional resource allocation problem [12]. We add two constrains, i.e., the job start time and the job processing time, denoted by T_{start} and $T_{start}+T_{duration,}$ respectively. Reservation with slack time requires a job to be started between T_{start} and $T_{start} + T_{slack}$. The traditional resource allocation problem do not have constrains, and it is a special case of the reservation with slack time when $T_{slack}=+\infty$. Hence, the admission control algorithm may be designed based on existing static resource allocation algorithms such as Min-min[14], Min-max[14], and Suffrage[15].

4.1 Admission Control with Slack Time

Three sets are used to record all the received resource capacity requests. Set U comprises all the unallocated requests, set A comprises the requests which have been allocated but the reservation does not start, and set L comprises all the locked requests which have been allocated and the reservation has already started. Suppose a new resource capacity request *newReq* is received, it can be admitted only if an new allocation schema can be found without any impact on requests in set L, which also means the requirements of *newReq* and all the requests in set A and U can be satisfied without changing the resource capacity reservation for the requests in set L. Based on such observation, we list the new admission control algorithm as follows.

```
(1)   Algorithm AdmissionControlWithSlack
(2)   Input:
(3)     ResourceCapacityRequest newReq;
(4)     ResourceCapacityRequestSet U, A, L;
(5)     ResourceCapacity resourceCapacity;
(6)   Output:
(7)     Boolean admissionResult;
(8)     ResourceCapacity scheduleSchema;
(9)   {
(10)    scheduleSchema = new ResourceCapacity();
(11)    Rebuild scheduleSchema by L;
(12)    U = A □ {newReq}; A = {}
```

```
(13)    While (U <> {}) {
(14)      ResourceCapacityRequest currentReq = pickOut(U);
(15)      If (BasicAllocation(scheduleSchema,currentReq))== FALSE) {
(16)        scheduleSchema = NULL;
(17)        admissionResult = FALSE;   return;
(18)      } else {
(19)        U = U – {currentReq};
(20)        A = A □ {currentReq};
(21)      }
(22)    }
(23)    admissionResult = TRUE; return;
(24)  }
```

In line 11 of the algorithm, all the *TimeSlot* objects are recalculated according to the locked requests to form the new function of *Capacity*(*t*). Then *newReq* with all the requests in set *A* will be moved to *U* to perform the next iteration of the request selection. Once a new request is picked out from *U* (line 14), the algorithm will try to find out an allocation solution for it with the earliest finished time. If no solution is found, the algorithm exits with a return value FALSE and the new request will be rejected. The algorithm continues until all the requests in set *U* are processed.

According to existing static scheduling algorithm, we identified five different request selection strategies, shown in line 14 as the *AdmissionControlWithSlack* algorithm, which can be implemented in function *pickOut*(*U*), including FIFO, Min-Slack, Min-min, Min-max and Suffrage-based.

FIFO. This is the simplest request selection strategy. Only the request with earliest arrival time will be selected as the next target request. No other property is considered.

Min Slack. In this strategy, the request with the minimum slack time is selected as the next target request.

Min-min based. The strategy tries to find out the next target request using the min-min algorithm [14]. Only the request with the minimum earliest finished time is selected as the result.

Min-max based. The strategy tries to find out the next target request using the min-max algorithm [14]. Only the request with the minimum latest finished time will be selected as the result.

Suffrage based. The basic idea is to find out the next target request using the suffrage algorithm [15]. First, select a request r_i from the set U, find the best allocation solution for this request and record the earliest finished time as t_i. Then try to allocate resource capacity for another request r_j first ($i \neq j$), then try to find out the best solution for r_i again based on it and calculate the earliest finished time as t_{ji}(Generally $t_{ji} \geq t_i$.). We define the suffrage value of this request by $Suffrage_i = t_{ji} - t_i$. Only the request with the maximum suffrage value will be selected as the next target.

4.2 Implementation

CROWN is a middleware suit for service-oriented Grid environment [16-18]. There are 11 software components in CROWN including the Node Server, the Resource Location and Description Service (RLDS), CROWN Designer, scheduling service, web portal and rich client framework. Node Server is the basic container and runtime environment for Grid service.

CROWN Node Server is an extension of GT 4.0 core with value added components. Pluggable handler chain structure is commonly used in these containers such as AXIS, GT 4 and lots of EJB containers. Under such a structure, the request will be processed by a set of pluggable handlers in a sequential way.

In order to support service reservation and SLA management, we implemented two handlers called *ServiceReservationHandler* and *SLAManagementHandler*. Service Reservation Handler is used to intercept the service request, issue reservation according to the QoS requirement of user request and conduct admission test. The test result will be put into a data structure as the service context. We can add the handler to a specific Grid service if the service reservation mechanism is needed. A service without the handler will act as the traditional best-effort service.

5. Performance Evaluation

5.1 Simulation Methodology

To better evaluate the performance of FIRST, we develop a simulation toolkit, which includes a workload generator and a scheduler. The generator creates a resource capacity request list as the workload and serializes it into an XML file. The scheduler uses different pre-configured request selection strategies to generate the allocation solution.

During the simulation, we set $Capacity(t)=C_{MAX}$, which is a constant function. The characters of the generated workload may be controlled by a set of parameters, which are listed as follows.

Average Request Interval ($\overline{T_{interval}}$) and **Request Rate** (μ): In general, the request arrivals follow the Poisson distribution. $\overline{T_{interval}}$ gives the mean value of the interval between the arrival of any two requests.

Based on the average request interval, we define the request rate by

$$\mu = \frac{1}{\overline{T_{interval}}} \qquad (2)$$

By adjusting the $\overline{T_{interval}}$ and μ, the overall system load can be controlled.

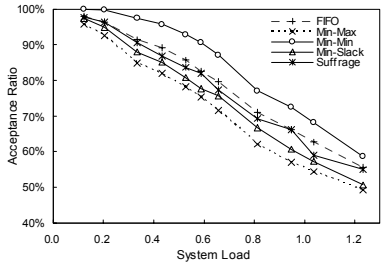

Figure 4: System load vs R_A

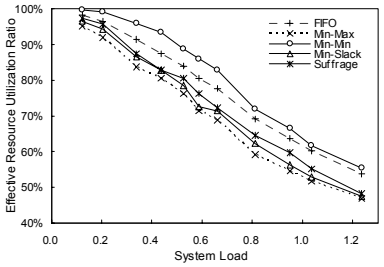

Figure 5: System load vs U_E

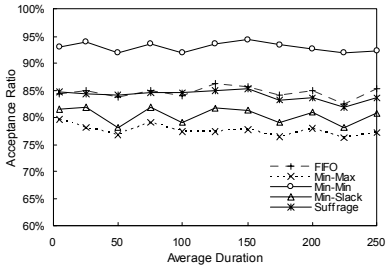

Figure 6: Average duration vs R_A

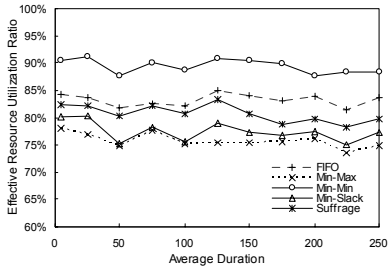

Figure 7: Average duration vs U_E

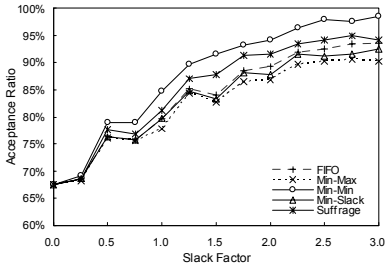

Figure 8: Slack ratio vs R_A

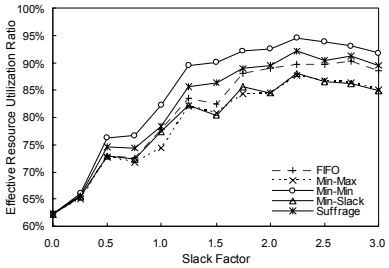

Figure 9: Slack ratio vs U_E

System Load (δ) is the ratio of required capacity and total amount of available capacity during a specific period. δ is given by

$$\delta = \frac{\sum_{i=1}^{n} T_{duration,i} \times C_i}{\int_{t_0}^{t_0+\Delta t} Capacity(t)} \tag{3}$$

where C_i is the required amount of capacity in ith request, and δ is ranging at [0.0,1.0].

Average Duration ($\overline{T_{duration}}$) is the mean value of the duration time of all the requests.

Average Capacity Demand ($\overline{C}$) is the mean value of the required amount of capacity of all the requests.

Slack Factor (λ) is defined as the ratio of slack time and average duration time.

$$\lambda = \frac{\overline{T_{slack}}}{\overline{T_{duration}}} \tag{4}$$

If the average duration time is pre-defined, the slack time can be controlled by the slack factor directly.

5.2 Metrics

We mainly examine two performance metrics in this simulation: **acceptance ratio** (R_A) and **effective resource utilization ratio** (U_E).

The *acceptance ratio* R_A is given by

$$R_A = \frac{N_{accept}}{n} \tag{5}$$

where N_{accept} denotes the number of accepted resource reservation requests and n denotes the total number of received requests during the simulation. The *effective resource utilization ratio* is given by

$$U_E = \frac{\sum_{i=1}^{n} AdmissionTest(i) \times C_i \times T_{duration,i}}{\sum_{i=1}^{n} C_i \times T_{duration,i}} \tag{6}$$

It is the ratio of total amount of allocated resource capacity and the sum of the required resource capacity in each received request. *AdmissionTest*(i) is a switch function, given by

$$AdmissionTest(i) = \begin{cases} 1, (1 \le i \le n, if\_the\_request\_is\_accepted) \\ 0, (1 \le i \le n, if\_the\_request\_is\_rejected) \end{cases} \tag{7}$$

In relevant to U_E, there is another metric called *absolute resource utilization ratio* (U).

$$U = \frac{\sum_{i=1}^{n} AdmissionTest(i) \times C_i \times T_{duration,i}}{\int_{t_0}^{t_0+\Delta t} Capacity(t)} \tag{8}$$

Obviously, $U \le \delta$, which means U will be influenced by the system load (δ) heavily when the system load is below 100%. So we use U_E instead of U to evaluate the performance of the algorithm.

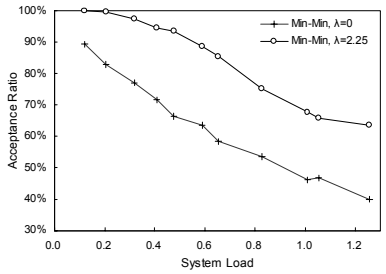

Figure 10: Benefit of slack time: R_A

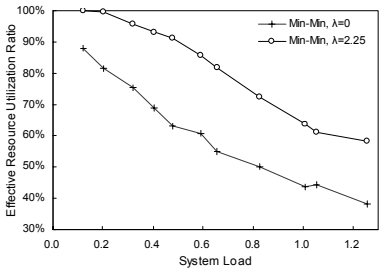

Figure 11: Benefit of slack time: U_E

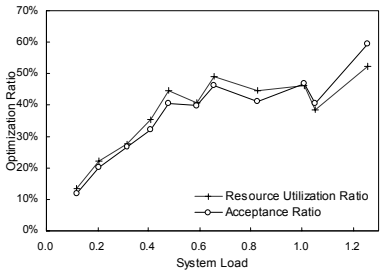

Figure 12: The optimization ratio of R_A and U_E

5.3 Performance Evaluation

In our first set of simulation, we study the performance of FIRST admission control algorithm when the system loads are changing. During the simulation, the slack factor λ follows a unified distribution within the range of [2.0, 2.5], and the average duration time is set to be 25. The average capacity demand follows a unified distribution within the range of [0.2,0.8]. The system load changes from 0.1 to 1.5 by adjusting the average request interval $\overline{T_{interval}}$. In Figure 4 and Figure 5, we see that with the increment of system load, the acceptance ratio (R_A) and effective resource utilization ratio (U_E) are decreased. When the system is busy (δ=1.0), only 55%-70% requests can be accepted. Min-min based request selection strategy gives the best performance.

In the second simulation, we study the relationship between the performance of the admission control algorithm and the average duration time ($\overline{T_{duration}}$). The slack factor λ and the average capacity demand are similarly defined as in last simulation. The system load is set to be 50%. Let the average duration change within the range of [5,250]. Figure 6 and Figure 7 plot the results. We see that the performance of algorithm is not affected by the average duration. On the other hand, Min-min based strategy again shows the best performance, both in acceptance ratio and the effective resource utilization ratio.

In the third simulation, we study impact of slack factor (λ). During the simulation, average duration $\overline{T_{duration}}$ is set to 25, and average capacity demand follows a unified distribution in the range of [0.2,0.8]. The system load is set to 50%. Let the slack factor change in the range of [0.0,3.0]. The result shows that the slack factor makes significant influence on the performance of the algorithm, as shown in Figure 8 and Figure 9. When λ>2.0, less benefit can be achieved with the increment of λ. Min-min based strategy improves the R_A by 4-7% and improves the U_E by 4-11% compared with other request selection strategies.

The last simulation is used to show the benefit of the reservation with slack time. We compare the performance of admission control algorithms for fixed reservation (λ=0) and the reservation with slack time (λ=2.25). Since the Min-min based strategy yields the best performance among all the other strategies, FIRST selects the min-min based strategy. All the other parameters are the same with previous experiments.

The simulation results in Figure 10 and Figure 11 show that the reservation with slack time improves the acceptance ratio and resource utilization ratio significantly. Figure 12 calculates the optimization ratio of R_A and U_E, which is defined as follows,

$$OptRatio_{R_A} = \frac{R_{A,\lambda=2.25} - R_{A,\lambda=0}}{R_{A,\lambda=0}} \quad (9)$$

$$OptRatio_{U_E} = \frac{U_{E,\lambda=2.25} - U_{E,\lambda=0}}{U_{E,\lambda=0}} \quad (10)$$

The result shows that when the system load increases, better optimization ratio can be achieved. For example, when system load δ reaches 1.25, the performance of admission control algorithm for reservation with slack time improves the R_A by 59.5%, and improves the U_E by 52.4%.

6. Conclusion

Providing guaranteed QoS for Grid services using resource reservation and allocation is an important feature for today's service Grid. However, the reservation request with fixed parameters leads to unnecessary rejection and low resource utilization.

In this paper, we propose a flexible capacity reservation mechanism with slack time, as well as a slack time-enabled admission control mechanism with different selection strategies. We employ this design in a real grid system, CROWN [16-18].

The performance of the reservation mechanism using slack time is evaluated by comprehensive simulations. Simulation results show that (1) the better resource utilization ratio can be achieved by using the slack time, and (2) the min-min based request selection strategy has the best performance compared with other

strategies such as the min-max, min slack time, FIFO and suffrage based strategy.

Acknowledgements Part of this work is supported by grants from the China National Science Foundation (No.91412011), China 863 High-tech Programme (Project No. 2005AA119010), China 973 Fundamental R&D Program (No. 2005CB321803) and National Natural Science Funds for Distinguished Young Scholar (Project No. 60525209). We would also like to thank members in CROWN team in Institute of Advanced Computing Technology, Beihang University for their hard work on CROWN grid middleware.

References

[1] Foster I., Kesselman C. Globus: A Metacomputing Infrastructure Toolkit. Int'l Journal of Supercomputer Applications, 1997, 11(2): 115-129.

[2] Foster I., Kesselman C., Tuecke S. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *International Journal of High Performance Computing Applications*, 2001, 15(3): 200-222.

[3] Foster I. The Physiology of the Grid – An Open Grid Service Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum*. 2002.

[4] Foster I., Kesselman C., Lee C., et al. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-allocation. *Proceedings of the International Workshop on Quality of Service (IWQoS'99)*. 1999: 27-36.

[5] Xing J.B., Wu C., Tao M.L., et al. Flexible Advance Reservation for Grid Computing. H.Jin, Y.Pan, N.Xiao, and J.Sun (eds). *Proceedings of the 2nd International Workshop on Grid and Cooperative Computing (GCC 2003)* , Shanghai, China. 2003.

[6] Burchard L.O. On the Performance of Computer Networks with Advanced Reservation Mechanisms. *Proceedings of 11th IEEE International Conference on Network (ICON'03)*. 2003: 449-454.

[7] Al-Ali R.J., Rana O., Walker D., et al. G-QoSM: Grid Service Discovery using QoS Properties. *Computing and Informatics Journal. Special Issue on Grid Computing*, 2002, 21(4): 363-382.

[8] Nahrstedt K., Chu H., Narayan S. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking, IOS* Press, 1998.

[9] Hoo G., Johnston W., Foster I., et al. QoS as Middleware: Bandwidth Reservation System Design. *Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing*. 1999:345-346.

[10] Nanda P., Simmonds A. Providing End-to-end Guaranteed Quality of Service over the Internet: A Survey on Bandwidth Broker Architecture for Differentiated Service Network. *Proceedings of 4th International Conference on IT (CIT'01)*. Berhampur, India. 2001: 211-216.

[11] Ferrari D., Gupta A., Ventre G. Distributed Advanced Reservation of Real-time Connections. *ACM/Spinger Verlag Journal on Multimedia Systems*, 1997, 5(3).

[12] Katoh N., Ibaraki T. Resource Allocation Problems: Handbook of Combinatorial Optimization (Vol.2). Du Z., Pardalos P.M. editors. *Kluwer Academic Publishers*. 1998: 159-260.

[13] Wolf L.C., Steinmetz R. Concepts for Reservation in Advance. *Kluwer Journal on Multimedia Tools and Applications*, 1997, 4(3).

[14] Braun T.D., Siegel H.J., Beck N., et al. A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems . *Proceedings of 8th IEEE Heterogeneous Computing Workshop (HCW'99)*, 1999: 15-29.

[15] Casanova H., Legrand A., Zagorodnov D., et al. Heuristics for Scheduling Parameter Sweep Applications in Grid Environment . *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'2000)*, Cancun, Mexico. 2000: 349-363

[16] The CROWN Project, http://www.crown.org.cn/

[17] Hu C.M., Huai J.P., Zhu Y.M., et al. Efficient Information Service Management Using Service Club in CROWN Grid. In *proceedings of 2005 IEEE International Conference on Service Computing (SCC 2005)*, in conjunction with 2005 *IEEE International Conference on Web Services (ICWS 2005)*. Orlando, FL, USA. 2005:5-12, Vol. 2.

[18] Sun H.L., Zhu Y.M., Hu C.M., Huai J.P., et al. Early Experience of Remote and Hot Service Deployment with Trustworthiness in CROWN Grid. In proceedings of 6th International Workshop on Advanced Parallel Processing Technologies (APPT 2005), Hong Kong, China. 2005: 301-312.

[19] R. Chu et al, "A Distributed Paging RAM Grid System for Wide-area Memory Sharing", in Proceedings of IEEE IPDPS, 2006

[20] D. Qiu and R. Srikant, Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks, in Proceedings of ACM SIGCOMM, 2004.

[21] Y. Liu et. al, "Location-Aware Topology Matching in P2P Systems," in Proceedings of IEEE INFOCOM, 2004.