A Time-To-Live Based Reservation Algorithm on Fully Decentralized Resource Discovery in Grid Computing

Sanya Tangpongprasit, Takahiro Katagiri, Hiroki Honda, Toshitsugu Yuba
Graduate School of Information Systems
The University of Electro-Communications
jao@yuba.is.uec.ac.jp, {katagiri, honda, yuba}@is.uec.ac.jp

Abstract

We present an alternative algorithm of fully decentralized resource discovery in Grid computing, which enables the sharing, selection, and aggregation of a wide variety of geographically distributed computational resources. Our algorithm is based on a simply unicast request transmission that can be easily implemented. The addition of a reservation algorithm can find more available matching resources in resource discovery mechanism. The deadline for resource discovery time is decided with time-to-live (TTL) value. With our algorithm, the only one resource is automatically decided for any request if multiple available resources are found on forward path of resource discovery, resulting in no need to ask user to manually select the resource from a large list of available matching resources.

We evaluated the performance of our algorithms by comparing with first-found-first-served (FFFS) algorithm. The experiment results show that we can select the algorithm which improves the performance of resource utilization or turn-around time. The algorithm that finds the available matching resource whose attributes are closest to the required attribute can improve the resource utilization, whereas another one that finds the available matching resource which has the highest performance can improve the turn-around time.

Keywords: Grid computing, fully decentralized, resource discovery, TTL, reservation

1. Introduction

Grid computing is a distributed computing model where easy access to large geographical shared computing resources provided to large virtual organization (group of resources which are geographically apart while appearing to others to be a single)[1]. These shared computing resources include computers, storage space, sensors, software application, and data. All are connected through the Internet and a middleware software layer provides basic services for security, monitoring, accessing information about components, etc. The Grid computing applications allow users to harness the idle remote resources. The jobs are sent from users and executed on those resources, then the result of the execution is returned to the users, resulting in the high-performance computing.

A basic service in Grid computing is resource discovery: given a description of resources desired, a resource discovery mechanism returns the information of resources that match the description. Resource discovery is made challenging by a potentially large number of resources and users and considerable heterogeneity in resource types and user requests. Resource discovery is further complicated by the dynamic variation of the number of shared resources in the system, shared resource characteristics such as availability and CPU load which vary with the time.

Today Grid environments rely mainly on centralized architecture [2][3]. This method can do the resource management easily. However, when the number of resources extremely increases, the system will be outgrown and cause a bottleneck problem. Moreover, it is risky to encounter the single point failure problem at the central database and server. Therefore, the centralized resource discovery architecture is not suitable for large-scale networks.

There are alternative resource discovery mechanisms studied by many researchers. These works pay attention to a decentralized architecture instead of a centralized one. In such a network, the central database or server has been removed and all nodes act together to perform the resource management. It has scalability, and has none of the prementioned problems. Since there is no central server where the information of all available resources in the system is located, resource management with a fully decentralized resource discovery mechanism is more challenging. These related works rely on both of flat [4] and hierarchical [5] resource discovery services. A hierarchical architecture is quite complicated, whereas a flat architecture is easier to implement, and there is no doubt to its scalability. Hence, flat decentralized architecture will be studied in this paper.

Until now, on this flat, fully decentralized architecture, most resource discovery mechanisms still let users manually select the resource from the large list of matching resources. Then, users will prefer the available resource with the highest performance as it could be. From the system point of view, this disables other users who should use the resource immediately from accessing the resource. Hence, the mechanisms should not allow users to select the resources manually. The only one matching is performed automatically, then it is convenient for network administrator to control system performance.

In this paper, we introduce and evaluate a new resource discovery mechanism on a flat, decentralized resource discovery architecture. The feature of this algorithm is to find an appropriate matching resource using time-to-live (TTL) value as the deadline of resource discovery time.

In section 2 of the paper, we briefly review related work, then in section 3 we describe decentralized resource discovery mechanisms in detail. In section 4 we present an emulated Grid used in this work, and experiment results are shown in section 5. Finally, we close the paper with our conclusion and future work.

2. Related Work

There have been relatively few papers published on the problem of large scale resource discovery in Grid. To our knowledge, work by Iamnichi and Foster on decentralized resource discovery [4] comes closest to our research. In their paper, they proposed a flat, decentralized, self-configuring architecture, where resources are located on network nodes. A user connects to a local node and it either responds with the matching resource or forwards the request to another node. The request is forwarded until a resource is found or the initial time-to-live (TTL) value in the request message is decreased to zero. A node can forward a request using one of four request forwarding algorithms which consist of “random”, “experience-based + random”, “best-neighbor”, and “experience-based + best-neighbor”. For the result, “experience-based + random” algorithm gives the best performance among four algorithms. All algorithms are based on first-found-first-served (FFFS) algorithm. In FFFS, the request is sent back immediately if it finds any resource whose attributes match to the type described in the request.

In the real world of Grid computing, requests are often described as a set of desired attributes, not only the type of required resource (e.g., “a Linux machine with speed more than 500MHz and 128MB of available memory”) [6]. We think that FFFS is insufficient for the resource discovery mechanism.

In this paper, we present a new algorithm differing from FFFS, which is based on “experience-based + random”. With the addition of the reservation algorithm, more available matching resources can be found by using TTL value in the user’s request message. Our mechanism automatically decides which matching resource should be informed back to the user.

Our framework relies on unicast request transmission. Although it is quite complicate to decide algorithm, multicast request transmission from users can perform resource discovery. The transmission of a unicast request, however, can implement easily. Besides, it does not spread out the packets of requests on the network like multicast transmission which is often implemented in small networks.

In the next section we show how this resource discovery mechanism works with the reservation algorithm.

3. Resource Discovery Algorithm

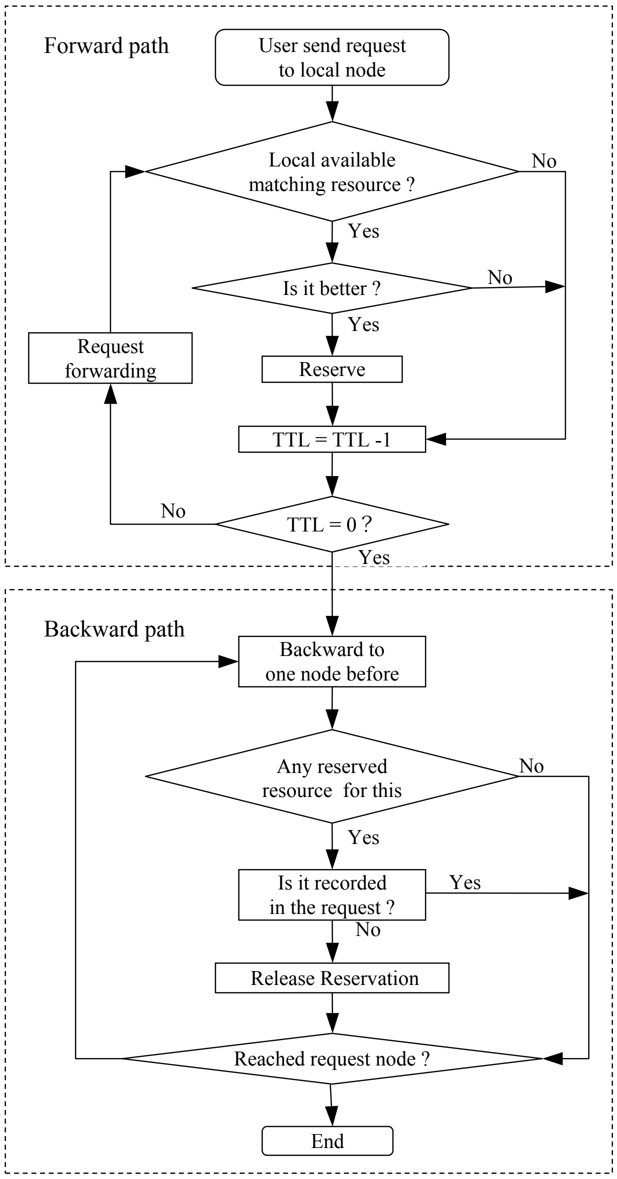
In Grid computing networks there are a great deal of resources shared by all members in virtual organization. Resource discovery service acts as an intermediary between users and resource providers. Its function is to look for the resources whose attributes match to the required description in the user request in large networked environments.

3.1 Model

We assume that all members in the virtual organization have at least one server in order to store and provide access to local resource information. We call these servers “nodes”. Each node may provide information about one or multiple resources. We also assume that all resources are not able to execute more than one job simultaneously.

There are many resource attributes in Grid computing, e.g., OS, speed, memory size, local load, and etc. “Matching resources” means the resources whose all attributes are equal to or more than all required attributes specified in the request message.

Our resource discovery algorithm is presented in the following section.


Forward path
User send request to local node
Local available matching resource ?
No
Yes
Is it better ?
No
Yes
Request forwarding
Reserve
TTL = TTL -1
No
TTL = 0 ?
Yes
Backward path
Backward to one node before
Any reserved resource for this
No
Yes
Is it recorded in the request ?
Yes
No
Release Reservation
Reached request node ?
End
Figure 1. Flow chart of resource discovery algorithm

3.2 Framework

The framework is divided into two main paths: the forward path and the backward path as shown in Figure 1. In the forward path, when users need to use the computing resource, they send their requests to their local node. Then, the node checks whether there is any local available matching resource whose attributes are better than one reserved before which is recorded in the request. If so, that resource is automatically reserved. In the matching resource check and reservation process, if there are more than one available matching resources in the same node, the node decides to reserve only one. After reserving, the information of the last reserved resource is added in the request. Then, the node forwards the request to one of its neighbors. The node decides which node to forward the request to with the "experienced-based + random" algorithm; i.e., nodes learn from experience by recording answers by other nodes. The record at any node has size (number of neighbors)*(type of resource). Then, the request is forwarded to the node that answered the same type of request previously. If there is no any record for that type of request, the request is forwarded to a randomly chosen node. This process is continued until TTL decreases to zero.

In the backward path, if the number of reserved resource is more than one, these resources, except the chosen one, have to be released from the reservation. The user reply message is sent back from the destination node along the forward path to release all unnecessary reserved resources until it reaches the node that generated that request. Then it will be sent to its local user. The node records the experience by determining this reply message on the path between target node, where the decided available resource is located, and the node that generated that request. After the resource discovery mechanism finishes (the reply message reaches the node initiating the request), the details of the job are sent directly from the user to the target node. Then, the target resource starts the job execution. The execution time of each job can be approximately computed by the following equation.

$$\text{Execution time} = \frac{\text{Job size (CPU Clock)}}{\text{Resource performance (speed, Hz)}}$$

We assume that when the resource is occupied by the execution of any job, its status becomes unavailable. That means it cannot be reserved or executed by another job until the job execution finishes.

For the clearer view of this algorithm, now the case that the user requires resource whose OS = Linux, performance (speed) $\geq$ 1GHz, and memory size $\geq$ 256MB is shown. (Assume that "closest attribute" consideration is used). First, if at the first node, two matching resources (resource A: Linux, 1.2GHz, 256MB and resource B: Linux, 1.5GHz, 256MB), resource A is reserved since its performance is closer to the required description than resource B and the information of resource A is recorded in request packet. The request is kept forwarding and reserves more resource if it finds the idle resource C whose OS = Linux, 1 GHz $\leq$ performance (speed) $\leq$ 1.2 GHz, and memory size $\geq$ 256MB. Now the information of resource C is recorded in the request packet instead of that of resource A. In the backward path, the reservation at resource A is released. After the user receives the reply packet back, the job is immediately sent to the node where resource C is located.

To evaluate this algorithm, we also compared the performance with the FFFS algorithm.

3.3 Resource Discovery Algorithms

Instead of letting users select the resource manually from a large list of resources, we present four alternative resource discovery algorithms which automatically try to find only one resource for each user request.

ALG1. TTL + closest attribute: The request is forwarded until TTL value decreases to zero. The resource reservation algorithm is used to find the resources whose attributes are closest to the description in the request.

ALG2. TTL + highest performance: the same as ALG1, but it looks for the resource which has the highest performance.

ALG3. FFFS + closest attribute: The request is sent backward immediately if it finds the first available matching resource. In the case that there are multiple available matching resources found in the same node, the user is answered with the resource whose attribute is closest to the description in the request.

ALG4. FFFS + highest performance: The same as ALG3, but in the case that there are multiple available matching resources found in the same node, the user is answered with the resource which has the highest performance.

The following section describes the simulated Grid used to evaluate these algorithms.

4. A Simulated Grid for Resource Discovery

In order to study large scale network environments, we decided to evaluate our resource discovery algorithms with a simulation which can be designed as a fully decentralized, flat large-scale architecture.

The information at each node contains information about its local shared resources and information about experience which it records about neighbor nodes. We assume there is adequate memory at every node.

In real world environments, the user requests have various patterns. Some require single resource, some require multiple resource. In this work, we simplify and assume that all users require only single resources, and matching resources are the resources whose attributes are equal or more than the required description in the request.

The following demonstrates more details in our simulation model.

4.1 Network Topology

We generated the starting topology using Tiers network generator [7]. We assume that all the nodes are the members of virtual organization and connected through time. In the topology generation process, we also pay attention to the point that the network should be similar to the real network, avoiding unrealistically configurations, such as a star topology that is found only in small networks. In this study, we do the experiments with an assumption similar to Iamnichi and Foster's with a 1000-node network.

Links connected in the network are all duplex links with queue and have no link failure. The transmission can be done two ways simultaneously. If any packet tries to transmit via the busy link, it is added in a link queue, and initiates transmission as soon as the link becomes idle.

4.2 Resource Distribution

The resource distribution in our experiment is related to real environments. The nodes that share a large number of resources are fewer than the nodes that share only one or two resources. Hence, the distribution of resources on node is decided by geometric distribution, where the average number of resources is the constant set to five.

In order to evaluate all algorithms, we neglect some parameters. In our experiment, resource attributes consist of only one type of resource (OS) and performance (speed). We assume that each resource has adequate memory for the required memory size described in every request.

4.3 User Requests Generation

Every node generates its own request through simulation time. The random time of request generation is the Poisson process, the most common distribution of spike generation. Information in the request consists of source address, description of required resource, traveling path of message, and description of the most appropriate resource.

All of the distribution functions used in this work is shown in Table 1.

Table 1. Distribution functions of all parameters

Parameter	Distribution Function
Number of resources on node	Geometric
Resource type (OS)	Uniform
Resource performance (speed)	Poisson
Required resource type (OS)	Uniform
Required performance (speed)	Poisson
Job size	Negative exponential
Request generated time	Poisson process

5. Experiment Result

We studied the performance of all algorithms within three parameters: the number of requests which found matching resources, turn-around time, and resource utilization. The number of requests which found matching resources is considered by dividing by the number of total generated requests. A turn-around time means the period counted from when the user's node initiates the request transmission until the result of execution is sent back to the user. Resource utilization is the percentage of time that all resources are occupied by the job execution. The value of all parameters used for the following results is shown in Table 2.

Table 2. Average value of all parameters

Parameter	Average value
Number of resources on node	5
Resource type (OS)	10
Resource performance (speed)	10 (*10^8 Hz)
Required performance (speed)	10 (*10^8 Hz)
Job size	10 (*10^{11} CPU Clock)

5.1 The relationship to average request generation time

In this section, we show the relationship between average request generation time and the two parameters: number of requests which found matching resources and resource utilization which are shown in Figure 2 and 3, respectively.

When the request generation rate decreases (average request generation time increases), the number of requests which found matching resources increases and the resource utilization value decreases. This is because the increment of request generation time leads to decrease a number of requests traveling in the network. This means less resource access competition. Hence, there is more probability for each request to find available matching

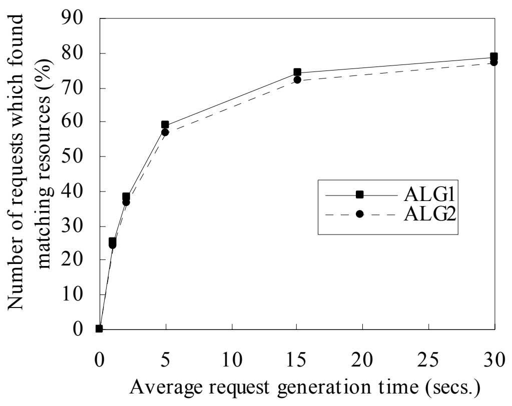


Figure 2. Percentage of requests which found matching resources when TTL = 10

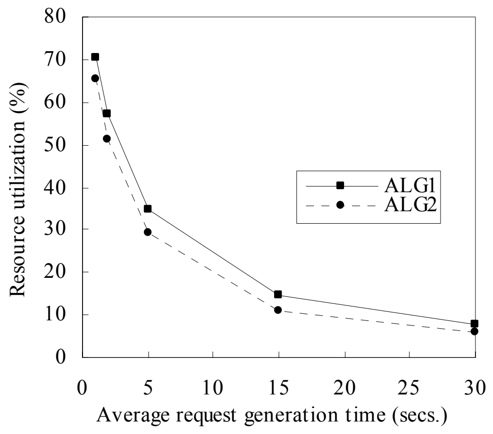


Figure 3. Percentage of resource utilization when TTL = 10

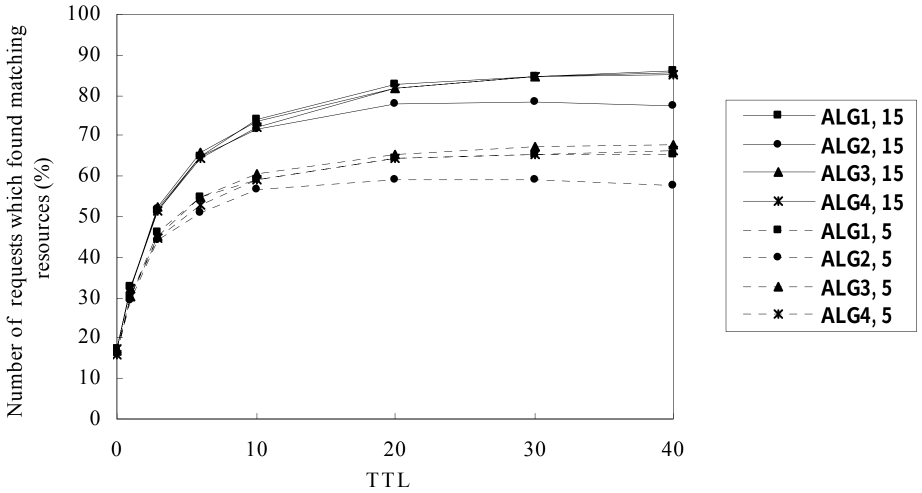


Figure 4. Percentage of requests which found matching resources when average request generation time = 15 and 5 mins.

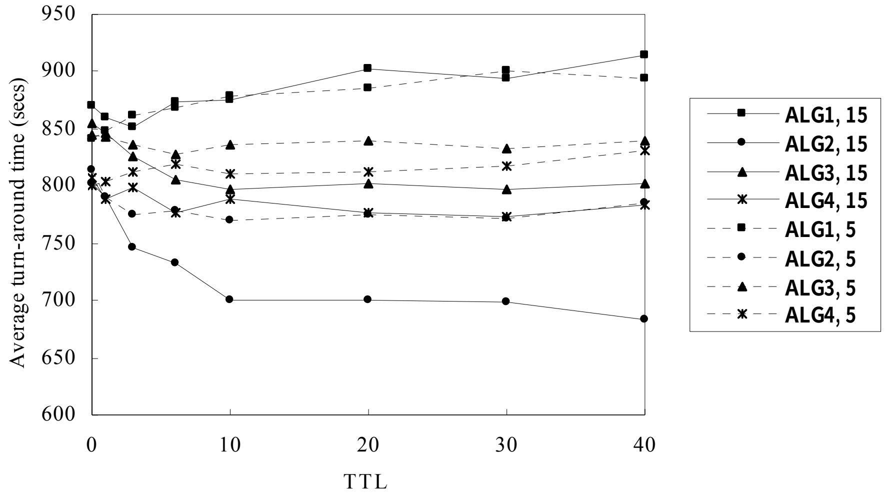


Figure 5. Average turn-around time when request generation rate = 15 and 5 mins.

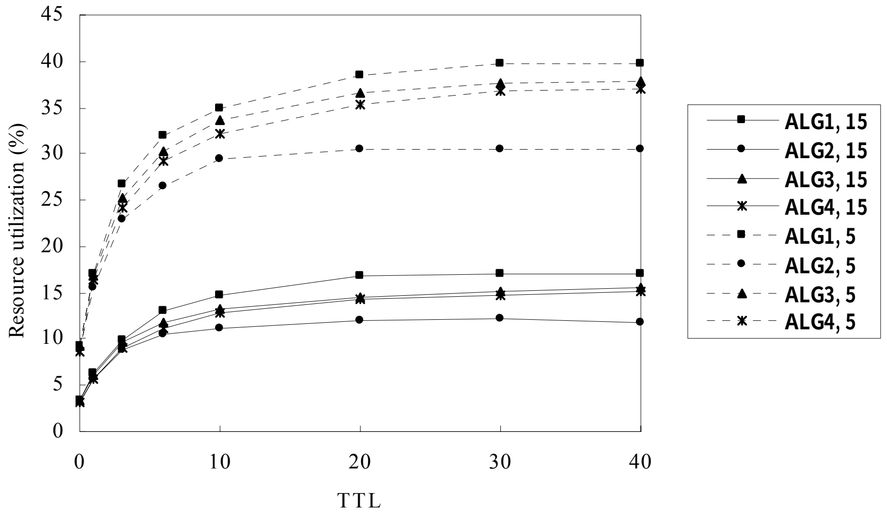


Figure 6. Resource utilization when average request generation time = 15 and 5 mins.

resources. With few requests traveling in the network, the occupied time of all resources becomes decreased and results in less resource utilization. The change of these two values is like the exponential curve. When the average request generation time is large enough, the resource utilization value gets close to zero, whereas the number of requests that found matching resources gets close to one value less than 100% because area which requests can reach is limited by TTL value.

5.2 The performance of four algorithms

Figure 4, 5, and 6 show the performance of all four algorithms when TTL value varies. We divided the results into two parts: high request traffic condition and low request traffic condition (the average request generation times are, respectively, 5 and 15).

When TTL value is increased, the number of requests that found matching resources of all the algorithms increase but do not get close to 100% because of a uniform request generation in all network areas, the resources in the area far from the user node also occupied by other users' requests. In Figure 4, throughout TTL value, ALG1, ALG3, and ALG4 have similar number of requests that found matching resources in low request traffic conditions (around 85%), whereas ALG3 seems to have a best performance (67.5%) in high request traffic condition, but it is not that different from ALG1 and ALG4 (65.6 % for ALG1 and 66.4 % for ALG4). In high request traffic conditions, the effect of reservation on the forward path makes other requests not able to access that resource, which results in lower number of requests that finds matching resources in ALG1 than that in ALG3. Not surprisingly, ALG4 gives the worst performance value (77.4% and 57.8% at low and high request traffic condition, respectively). To find the resource with the highest performance, it makes whoever has more need unable to use that resource, whereas ALG1 tried to keep the high performance resource idle in order to support other requests.

According to Figure 5, if comparing the same average request generation time, ALG2 can finish resource discovery and user's job execution in the shortest time, and the next is ALG4, ALG3, and ALG1 as can be predicted from the function of each algorithm. ALG1 executes the user job at the resource whose attributes are closest to the description in the request, resulting in the longest execution time, whereas ALG2 has the shortest execution time. When TTL increases, the turn-around time of ALG1 tends to be longer, but that of ALG2 tends to be shorter because it is possible to find the resource with higher performance. The higher the chosen resource performance, the shorter the turn-around time.

In both request traffic conditions, the turn-around times of ALG1 are not so different because the attributes of the chosen resource is limited by the required attributes described in the request, whereas the resource chosen by ALG2 is the highest performance resource that is found in the forward path. As can be seen, the gap between the longest and shortest execution time in high request traffic conditions (109 secs., TTL = 40) is much smaller than that in low request traffic conditions (230 secs., TTL = 40).

Considering resource utilization in the system in Figure 6), the sequence of algorithms is the same as when considering average turn-around time; that is ALG1, ALG3, ALG4 and ALG2, respectively, where ALG1 results in the maximum resource utilization and ALG2 results in the minimum resource utilization. This parameter has a direct relationship with average turn-around time that includes resource discovery time, job transmission and execution time. The resource discovery time and job transmission time is very small compared with the execution time in this study, so change in the execution time has a direct effect on turn-around time. Avoiding the use of high performance resources in ALG1 makes chosen resources occupied for a long job execution time and causes the highest resource utilization. On the other hand, ALG2 has the shortest execution time, resulting in the smallest resource utilization. The other algorithms, ALG3 and ALG4, have resource utilization performance between that of ALG1 and ALG2, where ALG3 has slightly higher resource utilization than ALG4.

It can be obvious from Figure 4, 5, and 6 that increasing of TTL value that is more than 10 can improve the performances of all algorithms only a little. Considering at TTL = 10 and average request generation time = 15, the number of requests that found matching resources of all algorithms are not different, but the turn-around time and the resource utilization differs in each algorithm. ALG2 has the shortest execution time (875, 700, 796, and 789 secs. in ALG1, 2, 3, and 4, respectively), but ALG1 has the best performance on resource utilization (34.9%, 29.3%, 33.6%, and 32.2% in ALG1, 2, 3, and 4, respectively).

5.3 The performance to support the special requests

The previous results show the common characteristics of four algorithms when the requests are generated uniformly throughout the network. In this part, we study one more case when generating the special requests on 20 random nodes in the high request traffic condition and TTL = 10. The value of parameters used in the part and the result are respectively shown in Table 3 and Table 4.

Table 3. The parameters of common and special request

	Average Required performance (speed)	**Average Generation time**
Common request	10 (*10^8 Hz)	5 mins.
Special request	15 (*10^8 Hz)	60 mins.

Table 4. Number of special requests which found matching resource when average common request regeneration time = 5 mins.

Algorithm	**Number of special requests which found matching resource (%)**
ALG1	29.0
ALG2	21.5
ALG3	24.7
ALG4	22.2

From Table 3, the special requests require the resources with higher performance than the common request, but the number of special requests is lower. According to Table 4, it is obvious that ALG1 has the best performance on the number of special requests which found matching resource (29.0%), where ALG2 has the worst performance (21.5%). This is because ALG1 tries to keep the high performance resource idle, thus the special requests can make use of those resources more than those of ALG2.

6. Conclusion and Future Work

In this paper we have introduced resource discovery algorithms with TTL-based reservation and unicast request forwarding algorithms on a flat, fully decentralized architecture in Grid computing. In order to evaluate our algorithms, we created a simple large-scale network of Grid computing and decided all the necessary parameters and their distribution patterns by considering real environments. Our reservation algorithm is implemented on the forward path of request to find more available matching resources, then on the backward path it releases all reserved resources except one that will be used to execute the job specified by the user.

Our results show that the reservation algorithm improves the performance of resource discovery from first-found-first-served (FFFS) algorithm which is implemented in Iamnichi and Forster's work [6]. ALG1 attempts to make use of resources in the system as much as possible and keep high performance resource idle and result in higher resource utilization, whereas ALG2 attempts to find the resource with the highest performance and can improve turn-around time.

However, there are tradeoffs between the number of requests which can be supported and the turn-around time. When we decide which algorithm to operate, it is important to think whether we would like to support the need of the network administrator – to maximize resource utilization (ALG1) or the user's – to minimize the turn-around time (ALG2).

We can apply this algorithm to real Grid computing by deciding the appropriate TTL value; i.e., not too small and too large value. Too small TTL value results in bad performance, but too large one can improve the performance only a little, but takes much more resource discovery time. The appropriate TTL value depends on all the network environments; e.g., network size, number of all resources, resource distribution, and etc. In this study, the appropriate value is around TTL = 10.

We have done the simulation on a simple assumption and with few conditions. In the future, we plan to extend our Grid environments, implement in large network and more complex condition, and design more efficient rules in reservation algorithms.

References

[1] I. Foster, C. Desselman, and S. Teucke, "The Anatomy of the Grid: Enabling acalable virtual organizations," *International Journal of High Performance Computing Applications*, pp. 200-222, 2001

[2] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, "Grid Information Services for Distributed Resource Sharing," in *Proceeding of the Tenth IEEE International Symposium on High-Performance Distributed Computing* (HPDC-10), IEEE Press, pp. 181-184, 2001

[3] M. Livny and R. Raman, "High-throughput resource management," In I. Foster and C. Kesselman, editors, *The Grid : Blueprint for a New Computing Infrastructure*, chapter 13. Morgan Kaufmann Publishers, Inc., 1998.

[4] A. Iamnitchi and I. Foster, "On Fully Decentralized Resource Discovery in Grid Environments," in *International Workshop on Grid Computing*, Denver, Colorado, 2001

[5] Z. Juhasz, A. Andics, S. Pota, "Towards a Robust and Fault-tolerant multicast discovery architecture for global computing grids," 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2002), Linz, Austria, pp. 74-81, 2002

[6] R. Raman, M. Livny, and M. Solomon, "Matchmaking: An extensible framework for distributed resource management," *Cluster Computing: The Journal of Networks, Software Tools and Applications, 2*: pp.129-138, 1999.

[7] M. Doar, "A Better Model for Generating Test Networks," *IEEE Global Telecommunications Conference/ GLOBECOM'96*, London, UK, pp. 83-96, 1996