A Balanced Resource Allocation and Overload Control Infrastructure for the Service Grid Environment

Jun Wang, Yi Ren, Di Zheng, and Quan-Yuan Wu

School of Computer Science,
National University of Defence Technology,
Changsha, Hunan, China 410073
junwang@nudt.edu.cn

Abstract. For the service based Grid applications, the applications may be integrated by using the Grid services across Internet, thus we should balance the load for the applications to enhance the resource's utility and increase the throughput. To overcome the problem, one effective way is to make use of load balancing. Kinds of load balancing middleware have already been applied successfully in distributed computing. However, they don't take the services types into consideration and for different services requested by clients the workload would be different out of sight. Furthermore, traditional load balancing middleware uses the fixed and static replica management and uses the load migration to relieve overload. Therefore, we put forward an autonomic replica management infrastructure to support fast response, hot-spot control and balanced resource allocation among services. Corresponding simulation tests are implemented and results indicated that this model and its supplementary mechanisms are suitable to service based Grid applications.

Keywords: Web Service, Service based Grid Applications, Load Balancing, Adaptive Resource Allocation, Middleware.

1 Introduction

The Grid services [1, 2] conform to the specifications of the Web Service and it provides a new direction for constructing the Grid applications. The applications may be integrated across the Internet by using the Grid services and the distributed Grid services and resources must be scheduled automatically, transparently and efficiently. Therefore, we must balance the load of the diverse resources to improve the utilization of the resources and the throughput of the systems. Currently, load balancing mechanisms can be provided in any or all of the following layers in a distributed system:

- **Network-based load balancing:** This type of load balancing is provided by IP routers and domain name servers (DNS). However, it is somewhat limited by the fact that they do not take into account the content of the client requests.
- **OS-based load balancing:** At the lowest level for the hierarchy, OS-based load balancing is done by distributed operating system in the form of lowest system level scheduling among processors [3, 4].

- **Middleware-based load balancing:** This type of load balancing is performed in middleware, often on a per-session or a per-request basis. The key enterprise applications of the moment such as astronavigation, telecommunication, and finance all make use of the middleware based distributed software systems to handle complex distributed applications.

There are different realizations of load balancing middleware. For example, stateless distributed applications usually balance the workload with the help of naming service [6]. But this scheme of load balancing just support static non-adaptive load balancing and can't meet the need of complex distributed applications. For more complex applications, the adaptive load balancing schema [7, 8] is needed to take into account the load condition dynamically and avoid override in some node. However, many of the services are not dependable for loose coupling, high distribution in the Grid environment and traditional load balancing middleware pay no attentions to the resource allocation. Therefore, we put forward an autonomic replica management infrastructure based on middleware to support fast response, hot-spot control and balanced resource allocation among different services.

2 Architecture of the Load Balancing Middleware

Our load balancing service is a system-level service introduced to the application tier by using IDL interfaces. Figure 1 features the core components in our service as follows:

Fig. 1. Components of the Load Balancing Middleware

Service Replica Repository: Instances of services need to register with the Service Group. All the references of the groups are stored in the Repository. A service group may include several replicas and we can add or remove replicas to the groups.

Service Decision Maker: This component assigns the best replica in the group to service the request based on the algorithms configured in our load balancing policy [5]. The service decision maker acts as a proxy between the client and the dynamic services. It enables transparency between them without letting the client knowing about the multiple distributed service replicas.

Load Monitor: Load monitor collects load information (such as CPU utilization) from every load agent within certain time interval. The load information should be refreshed at a suitable interval so that the information provided is not expired.

Load Agent: The purpose of load agent is to provide load information of the Grid hosts it resides when requested by load monitor. As different services might have replicas in the same host, so a general metric is needed to indicate the level of available resources at the machine during particular moment.

Load Prediction: This module use the machine-learning based load prediction method where the system minimizes unpredictable behavior by reacting slowly to changes and waiting for definite trends to minimize over-control decisions.

Resource Allocator: The purpose of this component is to dynamically adjust the resource to achieve a balance load distribution among different services. In fact, we control the resource allocation by managing the replicas of different services.

3 Balanced Resource Allocation and Overload Control

In traditional load balancing middleware, a load balanced distributed application starts out with a given number of replicas and the requests just are distributed among these replicas to balance the workloads. However, different services may need different resources at all and in some occasions such as 911 or world cup the number of some kinds of requests will increase fast while it may be very small in most of the days. When demand increases dramatically for a certain service, the quality of service (QoS) deteriorates as the service provider fails to meet the demand. Therefore, depending on the availability of the resources, such as CPU load and network bandwidth, the number of replicas may need to grow or decrease over time. That is to say, the replicas should be created or destroyed on-demand. So we use an adaptive replica management approach to adjust the number of the replicas on demand to realize the adaptive resource allocation.

3.1 Load Metrics

In traditional distributed systems, the load can be measured by the CPU utilization, the memory utilization, the I/O utilization, the network bandwidth and so on. At the same time, the load may be different for different applications. We take multiple resources and mixed workloads into consideration. The load index of each node is composed of composite usage of different resources including CPU, memory, and I/O which can be calculated with:

$$L_j = \sqrt{\sum_{i=1}^3 a_i k_i^2} \quad (0 \leq a_i \leq 1, \sum_{i=1}^3 a_i = 1) \quad (1)$$

L_j denotes the load of host j and k_i denotes the percentage according resource has been exhausted. Furthermore, a_i denotes the weighted value of the certain load metric and the value can be configured differently for diverse applications.

3.2 On-Demand Creation and Destruction of the Replicas

At the beginning of the discussion let us give some definitions firstly. Let $H = \{h_1, h_2, \dots, h_i\}$ where h_j represents the j^{th} node of the system and

let $S = \{s_1, s_2, \dots, s_l\}$ where s_k represents the k^{th} service of the system. Furthermore, let N_k represents the number of the replicas of the k^{th} service of the system. So the set of the replicas of the k^{th} service can be denoted by $R(S_k) = \{S_{k1}, \dots, S_{kN_k}\}$. At the same time, the host the m^{th} replica of the l^{th} service is residing in can be denoted by $H(S_{lm})$ whose load at time t is denoted by $L_{H(S_{lm})}(t)$.

The first problem is when to create the new replica. In normal conditions, new requests will be dispatched to the fittest replicas. But all the hosts the initial replicas residing in may be in high load and the new requests will cause them to be overload. So we should create new replicas to share the workload. We set the **replica_addition** threshold to help to make the decisions. For example, to the i^{th} service of the system, if the equation (2) can be true, then new replica will be created.

$$\forall x(L_{H(S_{ix})}(t) \geq replica_addition) \quad x \in R(S_i) \quad (2)$$

The second problem is where to create the new replica. As the load metrics we have discussed before we can compute the workloads of the hosts. Furthermore, the hosts may be heterogeneous and the workload of each host is different. In fact, we set the **replica_deployment** threshold for every host. According to the equation (3), if the workloads of some hosts don't exceed the threshold the new replica can be created on them. Otherwise no replica will be created because of the host will be overloaded soon and all the system will become unstable for the creation. Therefore the incoming requests should be rejected to prevent failures.

$$\exists x(L_{H_i}(t) < replica_deployment) \quad x \in \{1, \dots, i\} \quad (3)$$

The third problem is to create what kind of replicas. Because the applications may be composed of different services and the services may all need create new replicas. However, the services may have different importance and we should divide them with different priorities. Therefore, we classify the services as high priority services, medium priority services and low priority services according to the importance of them. The services having different priorities may have different maximum replicas. For example, supposing the number of the hosts is n , then the maximum number of the high priority can be n , the maximum number of the medium priority service can be $\lfloor \frac{n}{2} \rfloor$ and the maximum number of the low priority service can be $\lfloor \sqrt{n} \rfloor$. These

configurations can be revised according to practical needs. Secondly, the Resource Allocator module maintains three different queues having different priorities. Each queue is managed by the FIFO strategy and the replicas of the services having higher priority will be created preferentially.

The last problem is the elimination of the replicas. For the coming of the requests may fluctuate. If the number of the requests is small, then monitoring multiple replicas and dispatching the requests among them is not necessary and wasteful. We should eliminate some replicas to make the system to be efficient and bring down the extra overhead. So we set the **replica_elimination** threshold to control the elimination of the replicas. For example, to the i^{th} service of the system, if the equation (4) can be true, then some certain replica will be eliminated.

$$\exists x(L_{H(S_{ix})}(t) < replica_elimination) \quad x \in R(S_i) \quad (4)$$

The elimination will be performed continuously until the equation (4) becomes false or the number of the replicas becomes one.

Furthermore, in some unusual occasions such as 911 and world cup some certain simple services will become hot spot. Therefore, we should adjust the priority of the services according to the number of the incoming requests to avoid overload. The low priority services may have higher priority with the increasing client request and more replicas will be created to response the requests. At the same time, once the number of the requests decreases, the priority of these services will be brought down and the exceeding replicas will be eliminated.

4 Performance Results

As depicted in figure 2, our middleware StarLB is running on RedHat Linux 9/Pentium4/256/1.7G. The clients are running on Windows2000/Pentium4/512M/2G. Furthermore, to compare the results easier the Grid hosts are as same as each other and all of the hosts are running on Windows2000/Pentium4/512M/2G.

Fig. 2. Load Balancing Experiment Testbed

Fig. 3. Initial Replicas of different Services

At the beginning of this test, we used the services without the help of the autonomic replica management. Supposing there are six hosts and there are six different services. (This test just be used to analyze and present the mechanisms and more complex tests using many more hosts and services are ignored here.) Among

these services there are two high priority services, two medium priority services and two low priority services. Among the six services the service1 and the service2 are high-priority services, the service3 and the service4 are medium-priority services and the remaining two are low-priority services. All the services have only one initial replica. Each replica resides in a host respectively and response to the client requests. The distribution of the replicas is as depicted in figure 3.

Furthermore, we set the low priority service can have at most two replicas, the medium priority service can have three replicas and the high priority service can have six replicas. As depicted in figure 4(a) and figure 4(b), according to our setting when the load index arrived at 85% new replica was created. From the two above broken lines in figure 4(a) we can see with the requests coming new replicas was created in the hosts the low priority services residing in. At the same time, because of the creation of new replicas the response time of the high priority services could be brought down and the throughput of these services was increased efficiently. Furthermore, the workload of all the hosts was balanced efficiently. All the creations are depicted in figure 5 and the services with higher priority may create new replicas more preferentially and have larger maximum replica number.

Fig. 4. (a) Response Time with Replica Management. (b)Load Index with Replica Management.

Fig. 5. Creation and Elimination of the Replicas

Then there is still a question to be discussed. That is the elimination of the extra replicas and the elevation of the priority. We deployed all the replicas as the initial state depicted in the figure 3 and made the service6 become hot spot. As the figure 6(a) and the figure 6(b) depicted, adding the number of the requests of the services6 gradually. Then the CPU utilization of the Host6 and the response time of the service6 increased too. According to our setting of the *replica_addition* threshold, when the Load index arrived at 85% new replica was created. For the load index of the Host5 was lowest, a new replica of the service6 was created in the Host5. By the creation of the new replica, the Load index of the Host6 decreased as well as the response time. At the same time, the response time of the service5 was just affected a little.

Fig. 6. (a) Response Time with Priority Elevation. (b)Load index with Priority Elevation.

Fig. 7. Creation and Elimination of the Replicas with Priority Elevation

However, because the requests for the service 6 kept increasing and the Load index arrived at 85% again. As we have discussed before the service6 is low priority service and the maximum number of the replicas is two. So the priority of the service6 should be elevated to allow the creation of the new replicas. Then the new replica was

created in the Host3 and the requests could be distributed with the help of the new replica. At last, when we decreased the requests of the service6. The Load index decreased and too many replicas were not necessary. So the priority of the service6 should be decreased and unnecessary replicas should be eliminated. As depicted in the figure 6(b), when the Load index was below the *replica_elimination* threshold the replica in the Host3 was eliminated for the highest Load index among the three replicas. The remaining two replicas were keep dealing with the requests until the Load index shall become higher or lower. All the creation and elimination of the replicas with priority elevation are depicted in the figure 7.

5 Conclusions

Kinds of load balancing middleware have already been applied successfully in distributed computing. However, they don't take the services types into consideration and for different services requested by clients the workload would be different out of sight. Furthermore, traditional load balancing middleware uses the fixed and static replica management and uses the load migration to relieve overload. Therefore, we put forward an autonomic replica management infrastructure based on middleware to support fast response, hot-spot control and balanced resource allocation among different services. Corresponding simulation tests are implemented and their result s indicated that this model and its supplementary mechanisms are suitable to service based Grid applications.

Acknowledgments

This work was funded by the National Grand Fundamental Research 973 Program of China under Grant No.2005cb321804 and the National Natural Science Foundation of China under Grant No.60603063.

References

1. FOSTER, I., KESSELMAN, C., NICK, J.: Grid services for distributed system integration [J]. Computer(2002)37 -46
2. http://www.gridforum.org/ogsi-wg/drafts/GS_Spec_draft03_2002-07-17.pdf
3. Chow, R., Johnson, T.: Distributed Operating Systems and Algorithms, Addison Wesley Long, Inc.(1997)
4. Rajkumar, B.: High Performance Cluster Computing Architecture and Systems, ISBN7.5053-6770-6.2001.
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Reading: Addison-Wesley(2002)223-325.
6. IONA Technologies, "Orbix 2000." www.iona-iportal.com/suite/orbix2000.htm.
7. Othman, C., O'Ryan, Schmidt, D. C.: The Design of an Adaptive CORBA Load Balancing Service. IEEE Distributed Systems Online, vol. 2, (2001)
8. Othman, O., Schmidt, D.C.: Issues in the design of adaptive middleware load balancing. In: ACM SIGPLAN, ed. Proceedings of the ACM SIGPLAN workshop on Languages, Compilers and Tools for Embedded Systems. New York: ACM Press(2001)205-213.