國立交通大學

資訊科學與工程研究所

碩 士 論 文

針對異質多資源系統以代幣為基礎之負載平衡演算法

A Token-based Approach to Load Balancing of Heterogeneous, Multi-resource Systems

研 究 生：陳坤定

指導教授：陳俊穎 教授

中 華 民 國 九 十 八 年 八 月

針對異質多資源系統以代幣為基礎之負載平衡演算法

A Token-based Approach to Load Balancing of Heterogeneous, Multi-resource Systems

研 究 生：陳坤定　　Student：Kun-Ting Chen

指導教授：陳俊穎　　Advisor：Jing-Ying Chen

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年八月

針對異質多資源系統以代幣為基礎之負載平衡演算法

研究生: 陳坤定　　　　　指導教授: 陳俊穎 博士

國 立 交 通 大 學
資訊科學與工程研究所
碩 士 論 文

摘要

網際網路由於讓使用者可以使用不同的線上資源像是執行個人化的應用程式，瀏覽與發佈網頁內容，與朋友及同事溝通，逐漸與人們的生活變得密不可分。從伺服器的角度來看，提供這些網際服務，為了同時維持另人滿意的服務品質，需要充份的計算資源與數種其他像記憶體、磁碟儲存空間及網路頻寬等資源。當所提供的服務數目增加，並且使用者偏好程度彼此之間有差異時，如何兼顧其服務品質與避免使用過多的計算機資源於是變得具有挑戰性。現在以網站為主的系統藉由在多台伺服器之間分散使用者的工作來運作負載平衡，然而，大部份這些技術考慮的是擁有相同資源數的同質系統伺服器環境。由於不同的服務對不同的資源的要求不同，且不同工作的比例也隨著時間而有差異，這些平衡負載的技術可能不能最佳地運作，而造成不必要的資源競爭。我們先前研究過在傳統資源排程的環境下，機器有不同的資源分布時負載平衡的問題，並且提出一以市場為主的方法來有效地減少因資源分配不平衡而產生的系統瓶頸。這篇論文中，我們延伸先前的研究，但是集中在工作執行時間短、數量極大的網站為主的系統。為了使

用以市場爲主的方法，我們提出一個以代幣爲主的方法。伺服器不再以工作爲基礎作交換，而是將代表工作量及特定多資源要求的代幣爲基礎作交換，我們的目標是平衡伺服器的負載，以及有效地使用所有可用的系統資源。我們設定許多不同的系統設定，並作了廣泛的模擬實驗，並比較我們的方法與現存負載平衡的方法。結果顯示我們的方法提供了維持在一定水平的效能改進，在相同的工作產出下能使用更少數目的伺服器。

關鍵字：雲端計算、多資源負載平衡、負載分配

A Token-based Approach to Load Balancing of Heterogeneous, Multi-resource Systems

Student: Kun-Ting Chen   Advisor: Dr. Jing-Ying Chen

Institute of Computer Science and Engineering

National Chiao Tung University

1001 Ta Hsueh Road, Hsinchu, Taiwan, ROC

Abstract

The Internet has become an indispensible media where people access various on-line services, such as executing personalized applications, browsing and publishing web contents, communicating with friends and co-workers, and so on. From the server-side perspective, providing an Internet service requires sufficient CPU power as well as other kinds of resources such as memory, disk space, and network bandwidth, in order to maintain satisfactory service quality. When the kinds of services increase and their popularity vary, how to ensure quality of service without acquiring excessive computing resources becomes a challenge. Modern web-based systems employ various load balancing techniques in order to spread user requests among multiple machines. However, most of these techniques assume that servers are homogenous with similar resource capacities. Because different services impose different requirements on different kinds of resources, and the request rates for different services are also different, these load balancing techniques may not operate optimally and result in unnecessary resource contention. We have studied the problem of load balancing among machines with heterogeneous resource capacities in a traditional resource scheduling context, and proposed a market-based approach to effectively reducing unnecessary system bottleneck. In this thesis, we extend the previous study but focus on web-based systems, where requests tend to be small in processing time but large in volume. To apply the same market-based principle, we propose a token-based approach in which the servers do not exchange load on a per-request basis, but rather on a token basis where each token represents a workload with specific multi-resource requirements. Our goal is to balance the server load and to make efficient use of all the available system resources. We have conduct extensive simulation under different system configurations and compare our method with existing load balancing schemes. The results showed that our approach provides substantial performance improvement, capable of delivering equivalent system throughput with fewer machines.

Keywords: cloud computing, multi-resource load balancing, workload distribution

誌謝

首先我要感謝我的指導教授陳俊穎博士兩年來不斷的指導，讓我在伺服器負載平衡技術，技術上得到很多豐富的知識，使我可以在多資源系統環境下考慮不同工作需求，與工作量隨時間變異的問題上找到靈感，以及在作實驗時從多個觀察角度作結果分析、討論實驗數據。非常感謝陳健博士在論文上給予許多指導與鼓勵。另外，也感謝我的畢業口試評審委員江清泉博士、陳健博士以及易志偉博士，提供許多寶貴的意見，補足我論文與實驗不足的地方。

其次，我要感謝實驗室的學長，及共同作論文研究的同學，勝保學長、仁傑、信翔、士維，感謝你們在研究生涯上的指導與照顧，使我在學習上有切磋學習的對象，也有群共患難的夥伴，並且在論文研究的同時一起學得許多做研究的方法和技巧，得以順利的完成論文。

我要感謝我的家人，有你們的支持，是我背後最大的力量。讓我能讀書、作研究到將自己所學授予其他學生，解決困難，在我遇到挫折時總是有你們的關懷與勉勵，有你們一路下來陪著我走過這段研究的歲月。在此衷心地祝福我的家人。

陳坤定 謹誌 2009 年 8 月

交通大學協同合作實驗室

List of Tables

List of Figures

Chapter 1 Introduction

The Internet has experienced rapid growth over the last few years as people rely more and more on the Internet to access their on-line, personalized applications, to browse and publish Web contents, to communicate with friends and co-workers, and so on. Web content providers and e-commerce companies are often inundated by the sheer number of Web page requests, and they need to constantly expand their server capabilities to cope with the increasing demand. Serving dynamic and feature-rich web content, which is crucial to the success of the web site, only intensifies the problem further. When serving dynamic content, the server usually needs to take into account user information such as the location of the request, the user’s permission and other session information. Based on the user information and the types of services requested, the processing flow on the server side may vary a lot, possibly requiring different amount of CPU time, involving different databases or external servers. If the capacities of the various resources on each machine are not configured right, or if the actual user request pattern does not match what is expected when the system is built, performance bottlenecks may emerge from time to time, resulting in low resource utilization for the whole system and unacceptable response time for end users. Moreover, even when the observed request pattern matches what is planned, performance bottlenecks may still emerge when the volume of user requests surge unexpectedly and persist for a certain period of time, which may bring the system into an instable state.

To cope with the heterogeneity in terms of multiple request types, differentiated resource requirements, and diverse request patterns, modern web-based systems often employ some kinds of load balancing techniques so that user requests can be spread among multiple machines evenly. However, these techniques often assume a simpler system model in which the CPU resource is of the primary concern when allocating servers for different applications.

For example, some approaches attempt to adjust the number of servers for a given web application based on statistical information to achieve a specific utilization target. As mentioned previously, however, different types of requests often have different requirements for different resources, and trying to balance the usage of only one resource type may induce inadvertent performance bottleneck, especially when other scarcer resources are under contention.

In previous work [Yang, et al], we have investigated the problem of task scheduling in a more traditional context, where each task is unique, has longer lifespan, and has different requirements for different kinds of resources. We have demonstrated that both inter-server heterogeneity (where different servers have different resource capacities) and intra-server heterogeneity (where the capabilities of different resource types inside each server also vary) can have dramatic impact on the overall system performance. We proposed a market mechanism (MM) to effectively deal with the problem. Basically, the MM approach prices overloaded resources with higher cost, and tasks are swapped among servers dynamically to reduce the overall cost.

In this thesis, we focus on large-scale web-based systems such as clusters or cloud computing environments where user requests tend to have much shorter processing time but large in volume. We adapt the MM model mentioned above and propose a token-based load balancing method in which a token represents a certain amount of workload for a specific request type and is characterized by the requirements of multiple resources. Instead of scheduling individual requests, our method schedules tokens among servers dynamically following the same MM strategy.

We have conducted extensive experiments to investigate the effectiveness of our method under various system configurations, by varying the number of machines, the variation of resource requirements for user requests, the capacity differences among machines, and the heterogeneity of resource capacities inside each machine. Our method outperforms other comparable load balancing methods considerably, especially when the degree of inter-server and intra-server heterogeneity is high. Furthermore, we also study the cases where request rates change over time, i.e. the time-of-day effect studied in [Ranjan, et al], and show that our method can tolerate temporal surge of requests better than the other methods.

The rest of this thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 presents a framework characterizing our token-based load balancing architecture. Chapter 4 discusses our token-based load balancing algorithm based on distributed market mechanism. Chapter 5 presents the adaptive token-based load balancing algorithm that deals with time-varying requests. Chapter 6 discusses the experiments. Chapter 7 gives discussion and future work, and Chapter 8 concludes this thesis.

Chapter 2 Background and Related Work

Modern large-scale web-based systems invariably contain multiple inter-connected machines in order to serve the large volume of income requests in parallel. Depending on the web site traffic, the server-side architecture may range from clusters to larger cloud computing environments that contain multiple clusters serving different Web applications. Within each cluster, the computational resources of the machines may be aggregated into a resource pool to execute potentially diverse requests on behalf of users [Paton, et al]. These clusters are used not only for executing computation-intensive applications, but also for replicating storage and backup servers to provide essential fault tolerance and reliability for critical applications. In such cluster architecture, the system has to distribute user requests among servers properly in order to meet the required quality of service. A simple approach to load distribution is to use a load balancer in front of multiple web servers and dispatches income requests to specific groups of servers based on the request URL pattern.

As another example, in cloud computing, there is usually a server migration algorithm which allocates servers on-demand within a cluster by adapting the number of servers according to client demands. It moves servers from a shared server pool into an overloaded cluster and away from under-loaded cluster into the server pool, respectively [Ranjan, et al]. [Ranjan, et al] also proposes a server selection mechanism that enables statistical multiplexing of resources across clusters by redirecting requests away from overloaded clusters. A cluster decision algorithm is proposed to decide between serving a request locally after migrating in additional servers at the local cluster and serving it remotely by redirecting the request to a remote server that can serve the request earliest. They consider dynamic web requests that incur multi-resource requirements of a server, and the resource allocation algorithm considers both network latency effect on user-perceived response time and the CPU utilization of servers. However, they implement the server-on-demand algorithm base only on target CPU utilization, and they allocate servers based on CPU resource utilization only.

In contrast, this thesis explores more complex load balancing strategies that take into account the diverse inter-server and intra-server resource capacities, as well as heterogeneous workload associated with different types of requests. To simplify the load balancing problem, we make several assumptions about the overall system. First, we focus on a cluster system in which there are multiple machines with different resource capacities, and each machine is responsible of processing certain incoming requests, but the types of requests and the implied workload behind these requests may be different for different machines. For simplicity, we also assume that each machine is capable of processing the requests redirected from any other machine if it is asked to. These assumptions simplify the model and make it easier for us to adapt existing load balancing methods for web-based cluster systems. Figure 2.1 depicts the model described above.

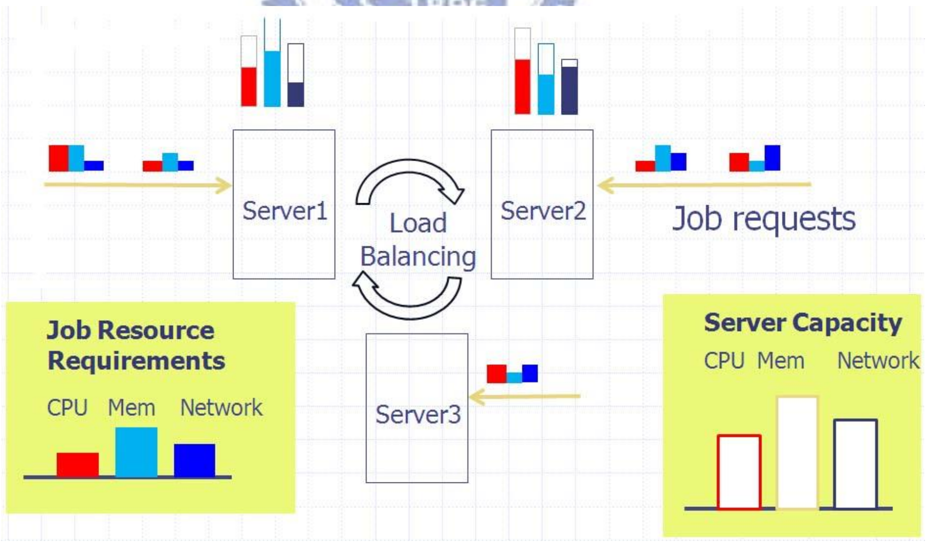


Figure 2.1: Multi-resource requirement and heterogeneous servers

When scheduling tasks with multi-resource requirements concerned, [Leinberger, et al] discusses generalized job selection heuristics to schedule jobs that can best balance the utilization of all the k resources. Specifically, the *backfill lowest* (BL) heuristic searches the job queue looking for the feasible job (i.e. when available resources can still meet the job's multi-resource requirements) which also demands the most for the currently least loaded resource. In contrast, the *backfill balance* (BB) heuristic attempts to pick the feasible job that can result in most balanced resource utilization for the whole system. [Yang, et al] proposes a distributed market mechanism (MM) for multi-resource task scheduling. In addition to the inter-server imbalance degree that is considered in BL and BB, the MM approach also considers the imbalance degree within each server. More specifically, MM not only tries to shift load from higher loaded servers to under loaded ones, but also attempts to exchange load between servers with moderate load to minimize their internal imbalance. This simple heuristic allows more jobs to be packed into each server in general, hence helps increasing the overall system utilization.

The BL, BB, and MM load balancing methods are devised for more conventional task scheduling problem where each task is unique, has longer lifespan, and has different requirements for different kinds of resources. In this context, a task may be allocated to some server initially, executed partially, and be reallocated to another server based on the dynamic state of the system. This model does not fit well with web-based systems, where requests need to be processed timely and responses generated promptly. Although requests may still be redirected among servers, once a request is processed there is no room for rescheduling. To adapt the abovementioned methods in such context, a straightforward approach is to aggregate multiple requests of the same type into a *token*, which serves as the unit of scheduling. Specifically, each token is associated with some multi-resource requirements derived from the type of request and the request rate. *Conceptually*, when a server is assigned a token, it needs to process the designated number of requests before claiming the completion of the token. In practice, however, the server may keep the token for a long time and continuously process the corresponding requests before the token is rescheduled elsewhere. Therefore, the token in fact represents a certain portion of the *workload* associated with the request type.

[Kulkarni, et al] also uses the concept of tokens to facilitate load balancing, but in a distributed system setting. The system is modeled as a network of inter-connected servers where each node has a single load indicator, based on which "local minimum" and "local maximum" can be found by exchanging the load information between neighbors. Two types of tokens, namely *sender* tokens and *receiver* tokens are created accordingly and passed over to neighbors. Servers who receive the tokens may shift its own load or help the other server based on its own load status and the kind of token received. The token may also be propagated further. [Borovska, et al] applies similar token-based load balancing method to a computational model for solving a puzzle problem in parallel computation system. This approach uses token messages to circulate among the parallel processors and to store information about the load distribution throughout the system. The load balance algorithm is initiated and performed by the idle or under-loaded processes by identifying the most heavily loaded processor to make a request for load migration. In addition, the token message also includes additional fields such as terminating conditions and best solution calculated so far, so as to reduce communication overhead since servers need to communicate with each other frequently and send messages to inform system manager of their latest computed solutions to the sub-problems they are designated to solve.

[Lin, et al] proposes a gradient model for load balancing in distributed multi-processor systems. The method first lets each individual processor determine its own load condition (heavy, moderate or light). Secondly, it establishes a system-wide gradient surface represented by aggregate value of all proximities to facilitate task migrations. A proximity value is determined for each processor to denote the minimum distance between the processor and a lightly loaded node in the system. The gradient surface is used as indication of all under-utilized processors, and the load balancing is based on a demand-driven principle which requires the under-utilized processors to dynamically initiate load balancing requests.

Our approach is similar to volunteer computing in many aspects. Briefly speaking, volunteer computing is a distributed computing paradigm in which a large number of computers, volunteered by members of the general public, are aggregated to provide unified computing and storage resources. Not surprisingly, there is usually some entity which is in charge of job scheduling among the participating computers. From load balancing perspective, the main difference between volunteer computing and traditional parallel computing systems is that for the former it is the computers that are idle or under-loaded that ask the scheduler for jobs to execute, while for the latter the scheduler usually takes the active role of maintaining the load status for the member servers and dispatching jobs once they arrive. Notable volunteer computing examples include SETI@Home and BOINC (Berkeley Open Infrastructure for Network Computing) [Anderson, et al]. For example, BOINC is middleware system being used for applications in physics, molecular biology, medicine, chemistry, astronomy, climate dynamics, mathematics, and the study of games. Volunteers participate by running BOINC client software on their computers (hosts). Each client periodically communicates with the task server to report completed work and to get new work. All network communication in BOINC is initiated by the client. The fact that it is the client who triggers job assignment can have importance consequence for the overall system utilization, especially when each computer has its own load beyond the control of the scheduler. Naturally, it is the participating computers who know the best timing to request for jobs. Following the same principle, we also investigate some variations of our token-driven load balancing method, that is, by letting a server to grab additional tokens from other servers only when it is under loaded.

Chapter 3 Token-based Load Balancing Architecture

The overall load balancing architecture is modeled as a set of servers interconnected with high-bandwidth links. Without loss of generality, we assume each server is configured to handle one application; that is, it serves as the entry point for all user requests related to that application. In practice, a server may be in charge of multiple applications, or multiple servers may share the load for the same application. An application distinguishes itself from other applications by the types of requests associated with it; each request type is characterized by its specific arrival rate and multi-resource requirements. In our simulation model that will be described in more details later, for each server, we will assign the mean and variance for its request arrival rate and its requirement for each type of resources. In addition, each server contains multiple types of resources each with different capacity. Again, in the simulation model we will assign the capacity for each resource type in different servers based on certain random distribution.

A request can be executed on a server only when the available resources of the server can satisfy all of the request's resource requirements; otherwise the request is put in a queue waiting for execution. Note that once the request is put in the wait queue, it cannot be rescheduled to another server. When a request is in execution, it claims all the resources from the server and will not return them back until it is finished. Therefore, the load of a server at a given time is the sum of the resources claimed by all the requests in execution. There are two sources of requests to each server: one is generated from outside by the user; the other is from the other servers in the cluster. This implies that a server may also be able to process requests from other applications. For simplicity, we assume each server is capable of executing any kind of request if it is asked to. Note that this assumption is not far from reality because, to be scalable, modern cloud computing environment such as Amazon E3 or Google App Engine indeed try to replicate the same application to different servers based on the application traffic.

Figure 3.1 depicts the system architecture described above. In the figure, Client 1 represents an application with high CPU requirement, but the CPU resource of Server 1 is overloaded, while Server 2 and 3 only have moderate CPU load. From the perspective of Server 1, it is desirable to shift some of the requests from Client 1 to Server 2 or 3 properly to reduce its CPU load, so as to increase its own chance to accept more requests from its wait queue. Of course, since all of the servers have the same goal, the load balancing mechanism needs to ensure that workload is divided and assigned among the servers properly such that the overall resource utilization is kept high.

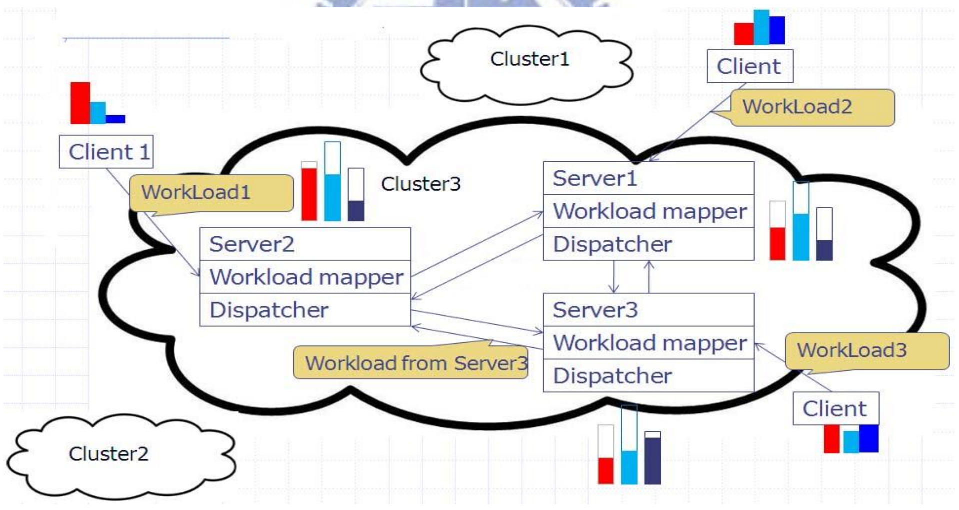


Figure 3.1: Workload distribution model

The load balancing algorithm is implemented in a distributed manner. Each server contains a *workload mapper* which monitors and analyzes incoming workload continuously, and communicates with other servers' mappers to decide how workload is divided among them. In other words, it is the set of workload mappers that together implement the load balancing strategy for the whole system. The *dispatcher* in each server is the one that takes care of actual request dispatching.

In our token-based approach, each workload mapper divides the workload corresponding to each request type into multiple tokens, where the number tokens is determined in a way that is proportional to the size of the workload. Specifically, assume request type i has arrival rate Ri and average CPU time Ci, its workload can be derived as

$$L_i = R_i * C_i$$

We simply pick a system-wise constant Lg as the "unit" of workload all tokens should represent. Therefore, the token size Ni for request type i can be derived using the following formula:

$$N_i = \left\lceil \frac{L_i}{L_g} \right\rceil$$

Note that in this thesis we only study the case where all token represent roughly the same CPU workload. Other variations are also worth further investigation. For example, we can allow tokens to represent different workload sizes and see the impact on the scheduling result.

Initially, each server holds all the tokens created by its mapper. Tokens serve many purposes. Firstly, because tokens can be passed among servers, if a server holds a token, it is responsible for the associated workload, meaning it should accept and process the corresponding requests dispatched from the token originator. Secondly, the workload mapper also uses the tokens to determine how to process each incoming request. For example, if a request type is divided evenly into N tokens, the workload mapper first picks one token from the N tokens randomly (or in a round-robin manner), checks which server currently holds the token, and dispatches the request to that server. Thirdly, because the total workload of a server can be computed by summing up all the tokens the server holds, the load balancing algorithm can use this information to rearrange tokens among servers to increase system utilization.

Figure 3.2 demonstrates how tokens are managed within each server, where T_{mgr} stands for the token manager that controls the creation and management of tokens for the server. In this example, the server's own workload for request type i is divided into 6 tokens. The *buddy set*, which is used to hold tokens from other servers, is empty initially.

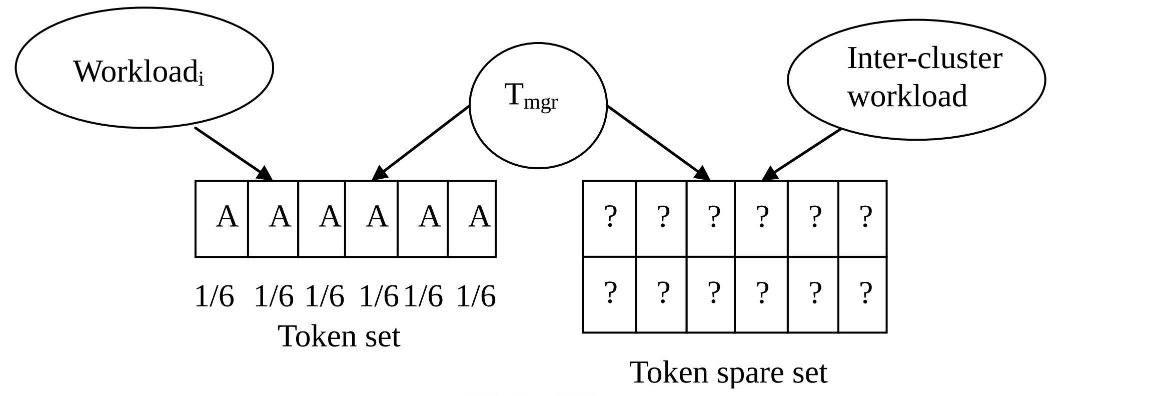


Figure 3.2: Initial token set and buddy set for workload$_i$ on server A

Figure 3.3 shows how tokens migrate between servers. In the figure, server A attempts to shift one of its tokens to server B. The token manager of server A marks the holder of a token to be server B, meaning that future requests of type i will have 1/6 chance to be redirected to server B. The token manager then notifies server B about the token assignment; Server B needs to add a new token in its buddy set to record the newly introduced workload it is responsible of.

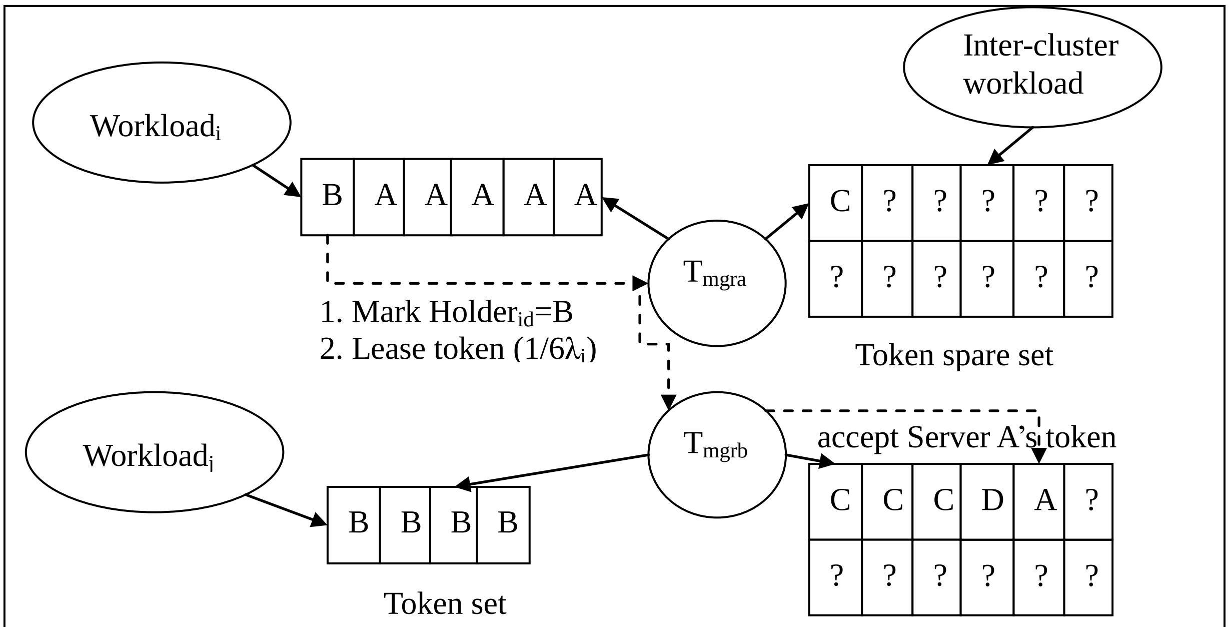


Figure 3.3: Token distribution scheme

Chapter 4 Token-based Load Balancing based on Market Mechanism

In Chapter 3 we outline the token-based load balancing architecture based on which different load-balancing schemes can be conceived. In this chapter, we proceed to describe one specific load-balancing algorithm, namely TBMM (Token-Based Market Mechanism), that dynamically rearrange tokens among servers to improve the overall system utilization. Distributed load balancing methods can be characterized with four policies [Shivaratri, et al][Eager, et al], namely information policy, transfer policy, location policy, and selection policy. We will also use the four policies to describe our load balancing method. Before going into the details, however, we first summarize these four policies:

- **Information policy.** The information policy determines the kinds of state information to be exchanged among servers. Conventional methods usually exchange information about a single resource, such as the CPU load.

- **Transfer policy.** The transfer policy determines the set of servers that need to adjust. The most common approach is threshold-based, which calculates an upper threshold T_s and a lower threshold T_r. The upper threshold T_s may be in the form of $L_{avg}^{avg} + d$, where L_{avg}^{avg} stands for the average resource load of the whole *server cluster* and d a designated constant, or $a\,L_{avg}^{avg}$ where a is a constant value greater than 1. Likewise, T_r may be in the form of $L_{avg}^{avg} - d$ or $a\,L_{avg}^{avg}$ where $a < 1$.

  As shown in Figure 4.1, if the load of a server is greater than T_s, the server is said to be in *sender* state; if less than T_r, the server is in *receiver* state. Otherwise, the server is in *common* state. As the names suggest, a server in sender state tends to send some of its jobs to another server with a lower load, while a server in receiver state tends to receive jobs from another server with higher load. The servers in common state do not have to do anything. In this paper, we also define a new state, call *exchanger* state, to deal with multi-resource load imbalance issue within each server.

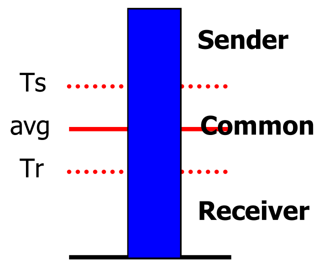
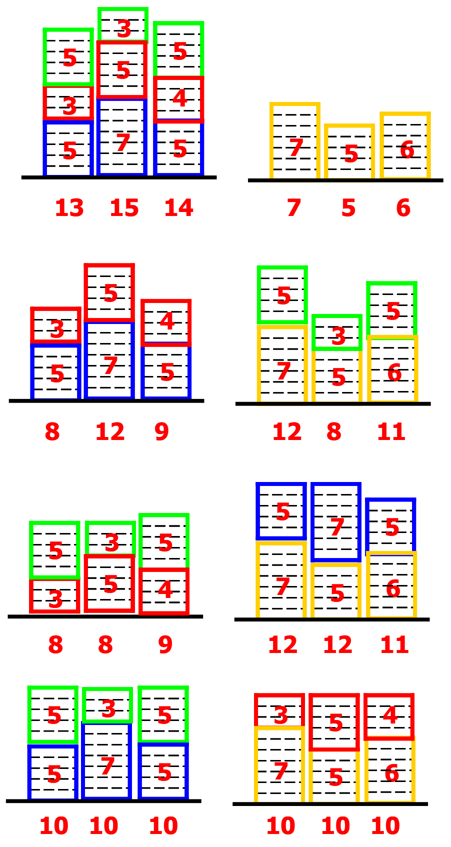


Figure 4.1: Sender, receiver, and common states

- **Location Policy.** The location policy concerns the steps needed to find the target server to which a server in sender state can send jobs to, and to find the source server from which a server in receiver state can receive jobs from. A simple heuristics is for a server in sender state to match the server with lowest load, and similarly for a server in receiver state to match the one with highest load.
- **Selection Policy.** Once the server pair is chosen, the selection policy determines which jobs should be sent to or received from the matching server. We investigate three popular job scheduling approaches, called *latest arrival job, backfill lowest* [Leinberger, et al] and *backfill balance* [Leinberger, et al]:
  - *Latest Arrival Job (LAJ)*: send the latest arriving job from the sending server to the receiving server,
  - *Backfill Lowest (BL)*: find the resource of the receiving server that is most available, and send the job that demands that resource most from the sending server.
  - *Backfill Balance (BB)*: send the job which can minimize the (maximum load / average load) measure for the receiving server.



(a) Before job exchange

(b) Result of LAJ selection policy

(c) Result of BL selection policy

(d) Result of BB selection policy

Figure 4.2: Example of selection policies

As an example, consider Figure 4.2 in which there are two servers with 3 jobs and 1 job, respectively, and the load of their resources are 13, 15, 14 and 7, 5, 6, respectively, before job exchange (Figure 4.2a). Assume the server on the left is in sender state and has to send a job to the server on the right. If the LAJ selection policy is used, the latest arriving job (with resource requirements 5, 3, and 5) will be selected (Figure 4.2b). In case the BL selection policy is used, since the second resource of the receiving server is least busy, the first job (with resource requirements 5, 7, and 5) of the sending server will be chosen because it demands the second resource most (Figure 4.2c). Finally, when the BB selection policy is used, since the measure of executing the three jobs on the receiving server are 1.16, 1.0, and 1.03, respectively, the second job will be selected (Figure 4.2d).

The framework discussed above is for conventional job scheduling problem. In this thesis, only small modification is needed, that is, by replacing the concept of jobs with tokens. Note that in conventional job schedule problem, jobs enter the system continuously, and leave the system when completed. In our model, however, tokens persist in the system since beginning, even though their number may increase or decrease over time. However, such difference has no impact from load balancing perspective, since we can simply think of tokens as long-running jobs that seldom finish.

In TBMM, each token assigned to a server will be associated with a cost, which is directly proportional to the resource load of that server. We define the *cost per resource requirement* of token *w* assigned to server *j* as:

$$C_j(w) = \frac{\sum_{k=1}^{K}\left(J_w^k \times L_j^k\right)}{\sum_{k=1}^{K} J_w^k} \qquad (1)$$

where J_w^k is the requirement of resource *k* for token *w*, L_j^k the load of resource *k* in server *j*, and *K* the number of resource types. The reason we define the cost per requirement rather than the total cost of a token assigned to a server is that the former helps improve the load imbalance degree within a server.

Below we describe our method in terms of the four policies mentioned above:

Information policy. In our method, servers exchange their entire resource load $L_j^1 \sim L_j^K$ with each other, where L_j^k represents the load of resource *k* in server *j*.

Transfer policy. We define the *load imbalance degree* for server *j* as:

$$B_j = \sum_{k=1}^{K}\left(L_j^k - L_{avg}^k\right) \qquad (2)$$

where L_{avg}^k is the average load of resource *k* of the whole server cluster. B_j measures the load imbalance degree between servers. Similarly, the *absolute load imbalance degree* for server *j* is defined as the sum of absolute values of load differences:

$$B_{abs\_j} = \sum_{k=1}^{K} \left| L_j^k - L_{avg}^k \right| \qquad (3)$$

which measures the load imbalance degree between resources inside a server *j*. We then determine the state of a server as follows:

- *Sender*: $T_s < B_j$.
- *Receiver*: $B_j < T_r$.
- *Exchanger*: $T_r \le B_j \le T_s$ and $T < B_{abs\_j}$.
- *Common: the rest.*

T_s and T_r are thresholds that are also commonly used in conventional methods. The difference is that we add an exchanger state with the threshold *T* for servers to further improve load balancing of resources inside of a server.

Location policy. For servers in sender or receiver state, the location policy to find matching server pairs is the same as those in conventional methods. For servers in exchanger state, on the other hand, the policy is different:

Assume server *i* and *j* are both in exchanger state, we define the load imbalance degree for the pair before token exchange as:

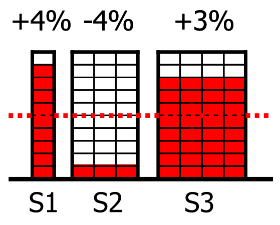
$$B_{abs\_i} + B_{abs\_j} \qquad (4)$$

and the *ideal load imbalance degree* after token exchange as:

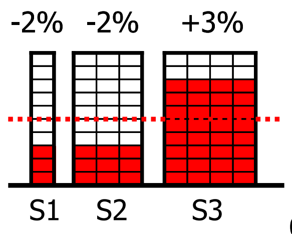
$$2 \times \sum_{k=1}^{K} \left| \frac{\left(L_i^k - L_{avg}^k\right) \times R_i^k + \left(L_j^k - L_{avg}^k\right) \times R_j^k}{R_i^k + R_j^k} \right| \qquad (5)$$

where R_i^k and R_j^k are the capacity ratios of resource *k* for heterogeneous servers *i* and *j*.

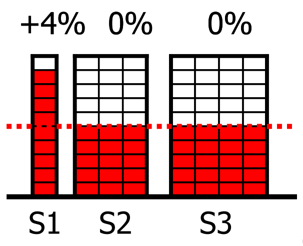
Our goal is to find the pair of servers *i*, *j* such that their imbalance degree before token exchange minus their ideal load imbalance degree afterwards is the largest. For illustration, consider Figure 4.3a in which there are three servers; the ratio of their capacities is 1:3:4 and their current loads compared to the whole system load are +4%, -4%, and +3% respectively. Before token exchange, B_{abs_1}, B_{abs_2}, and B_{abs_3} are 4, 4, and 3, respectively. Without considering server capacity, we may tend to pair server 1 with server 2, because the imbalance degree before token exchange is 4 + 4 = 8, and $2 \times \left| \frac{4 \times 1 + (-4) \times 3}{1+3} \right| = 2 \times 2 = 4$ afterwards (Figure 4.3b), hence the improvement of imbalance degree is 8 – 4 = 4. If we pair server 2 and server 3 instead, the imbalance degree is 4 + 3 = 7 before token exchange, and becomes $2 \times \left| \frac{(-4) \times 3 + (3) \times 4}{3+4} \right| = 2 \times 0 = 0$ afterwards (Figure 4.3c), an improvement by 7.



(a) Before job exchange



(b) Result of pairing server 1 and server 2



(c) Result of pairing server 2 and server 3

Figure 4.3: Example of location policy

Selection policy. Employ Equation (1). For each server pair *i*, *j* selected by the location policy among the servers in sender and receiver states, the token, *w*, to be sent from server *i* to *j* is the one that can result in minimum $C_j(w)$ for server *j*.

For each server pair *i*, *j* where both servers are in exchanger state, we want to pick a token from each server for exchange, and the exchange can benefit both servers overall. We define the benefit of transferring a token *w* from server *i* to server *j* as:

$$C_i(w) - C_j(w) \qquad (6)$$

Accordingly, the total benefit of exchanging token *w1* in server *i* with token *w2* in server *j* becomes

$$\left(C_i(w1) - C_j(w1)\right) + \left(C_j(w2) - C_i(w2)\right) \qquad (7)$$

Furthermore, the benefit should be larger than a threshold *T* for the exchange to take place. Without the threshold *T*, it is possible that two servers in exchanger state may constantly exchange the same pair of token to and from each other without converging.

Chapter 5　Adaptive Token-based Market Mechanism

The algorithm described in previous chapter assumes workload associated with each request type does not change over time. When requests of a certain request type surge and persist for a long time, the tokens for the request types no longer represent the effective workload correctly. This means the estimated total workload for all the servers that hold any of these tokens is not correct, and some adjustment to the tokens is needed. However, it is not desirable to inform all involved servers to update the corresponding tokens because doing so may require many update messages and trigger many new load balancing activities. Instead, the workload mapper that originates the tokens can create additional number of tokens to match the actual workload it observes. Similarly, when the actual workload decreases, the workload mapper can reduce the number of tokens, by removing the tokens hold in the local server or reclaiming some tokens back from remote token holders. Whether the token number is increased or decreased, the individual tokens remain unchanged.

We implement this strategy by extending the original load balancing algorithm described previously. The new algorithm, called adaptive token-based load balancing (ATBMM), adjusts the number of tokens in face of changing workload. Figure 5.1 shows the three cases, namely, when workload surges, drops, or remains within a certain range:

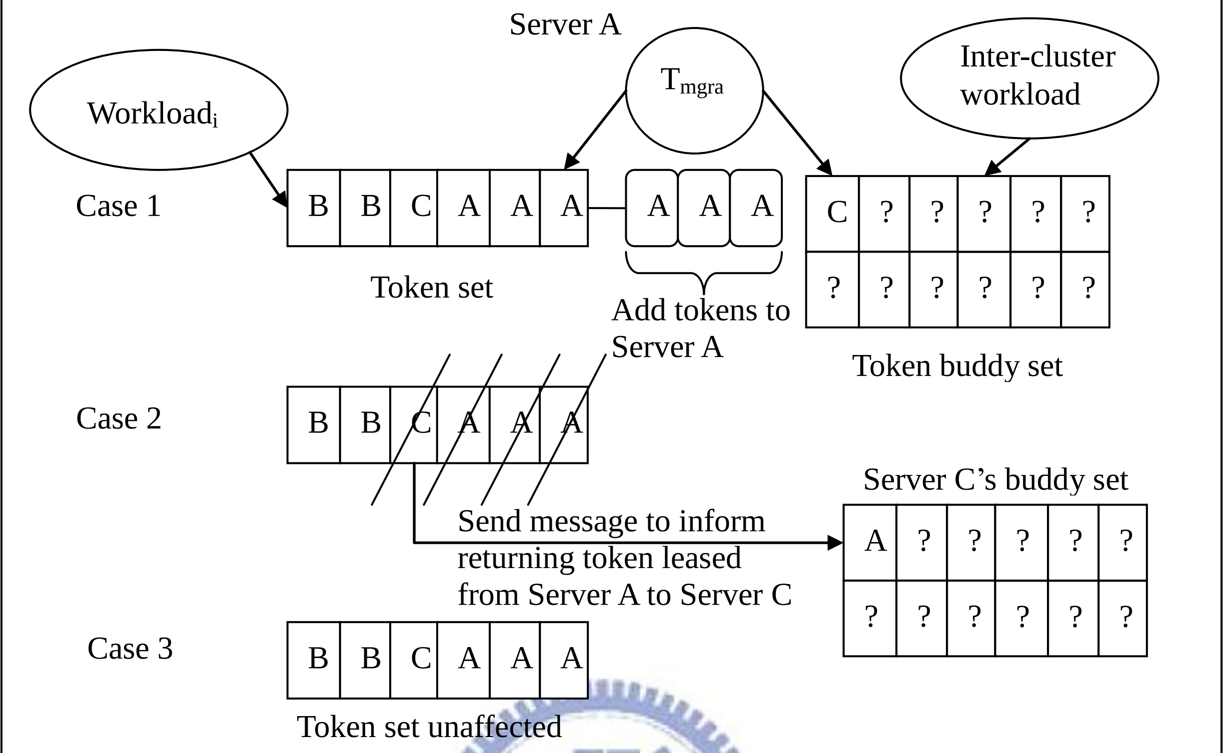


Figure 5.1: Adaptive token-based Market Mechanism

In case 1 where the workload surges, the calculated token number is larger than original number of tokens, and new tokens of the same kind are added to the token set. In case 2, the new token number becomes smaller, so server A informs server C to recall one of the token back; server C also updates its buddy set accordingly. In case 3, the newly calculated token number is equal to original one, and nothing needs to be done.

Chapter 6 Simulation

In this chapter we evaluate the performance of TBMM and ATBMM and compare the results with other methods via simulation. The simulator is built on top of an event-driven simulator PeerSim [PeerSim]. The simulation model is summarized as follows. Each server is pre-configured to serve some of the applications, and the assignment will not change during the entire run. An application consists of several request types, each of which is associated with multi-resource requirements. When a request of a given type is generated, its actual requirement for each of the resources is determined randomly according to the request type's multi-resource requirements. The request can be executed on a server only when the available resources of the server can satisfy all of the request's resource requirements; otherwise the request is put in a queue waiting for execution. Once the request is put in the wait queue, it cannot be rescheduled to another server. To investigate the effectiveness of our load balancing method, we also adapt existing multi-resource load balancing methods BL, BB to fit our model, and compare them with our method.

We conduct the simulation for clusters of 32 servers. Each server has three kinds of resources: CPU (measured in Gflops - giga floating points operations per second), Memory (measured in GBs), and Network Bandwidth (measured in gb/s). There are two independent parameters used to configure the degree of heterogeneity of server resource capacities: average server resource (S_{rm}) and server resource variance (S_{rv}). S_{rv} is used to specify range of capacity for the resources within a server. A resource variance of zero implies the resource capacities of CPU, memory, and network bandwidth is the same for all the servers. We refer to this configuration as homogeneous configuration. Otherwise, the configurations are heterogeneous. A resource variance of $S_{rv} = \pm X$ implies that the capacity of the resource will be assigned the value randomly within the range ($S_{rm} - X$, $S_{rm} + X$). In our simulation we set the value of X to be 0, 20, 40, 60, 80 and 100, respectively.

CPU capacity is specified in terms of Gflops. We assume each machine uses time sharing to allocate the CPU resources among the requests it is processing. In other words, all requests in execution share the CPU resource, and they are executed by the CPU in a round-robin manner.

The workload model is described as follows. All the methods are simulated using static workload, except ATBMM which is simulated with time-varying workload. For each case, a suite of synthesized workload was generated. Two main settings of the simulated workload are the request arrival rate and the generation of multi-resource requirements. For both static workload and changing workload, we have experimented with different combinations of changing workloads. We allocate 1 to 3 workloads out of 6 workloads to each server. A workload can be served by more than one server. When a server is configured to be responsible for a workload, the corresponding request will arrive at the server's local wait queue depending on the specified arrival rate following the Poisson distribution with mean λ_k $(0 \leq k \leq 5)$. The arrival rates for all workloads are adjusted such that resource utilization for the baseline algorithm (without load balancing) is around 80%. The multi-resource requirement of requests is generated as weak correlation among CPU, Memory and Network bandwidth requirements [Leinberger, et al].

The settings of 6 workloads, along with the initial configuration to each server are shown in Table 6.1 and Table 6.2.

Table 6.1: The multi-resource workloads used in the simulation:

Workload (jobs/ time unit)	CPU (gigaflop)	Memory(GB)	Network (gb/s)
W0 (λ = 2)	0.95~1.05	10~20	1~2
W1 (λ = 0.8)	0.35~0.45	10~20	0.5~1.5

W2 (λ = 1.21)	0.55~0.65	5~15	1~2
W3 (λ = 0.29)	0.37~0.47	1~9	2~3
W4 (λ = 0.57)	0.79~0.89	20~30	0.1~0.9
W5 (λ = 1.14)	1.63~1.73	10~20	1~2

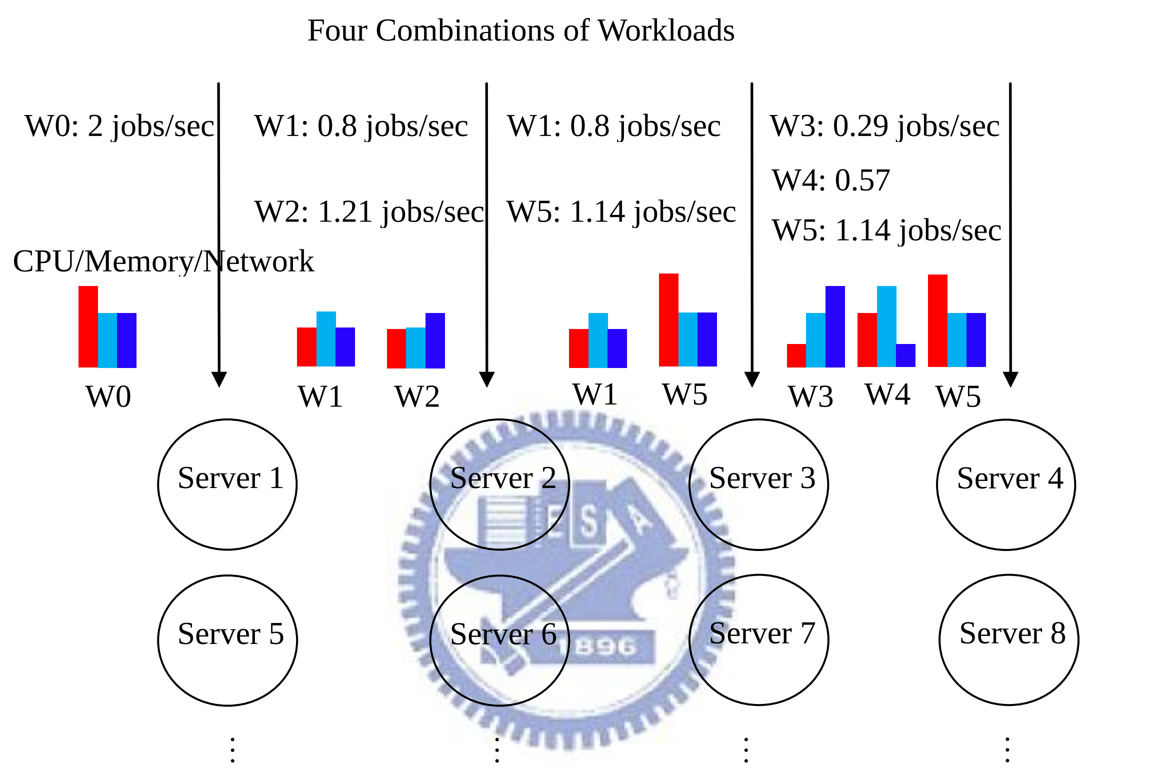


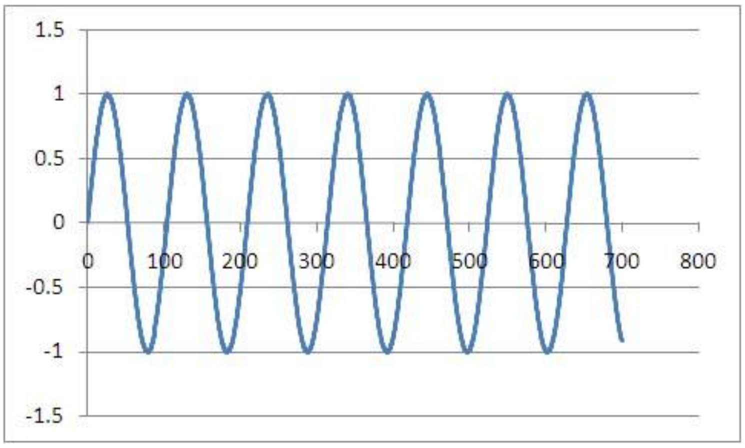
Figure 6.1: Initial configuration of workloads on servers

Table 6.2: The 4 combination of 6 workload in the simulation:				
Workload (jobs/ time unit)	Comb. 1	Comb. 2	Comb. 3	Comb. 4
W0 (λ = 2)	○	N/A	N/A	N/A
W1 (λ = 0.8)	N/A	○	○	N/A
W2 (λ = 1.21)	N/A	○	N/A	N/A
W3 (λ = 0.29)	N/A	N/A	N/A	○

W4 (λ = 0.57)	N/A	N/A	N/A	○
W5 (λ = 1.14)	N/A	N/A	○	○

To simulate demands surge with different duration and magnitude, we first referenced a 24-hour e-commerce trace from a large scale e-commerce site [Ranjan, et al]. In the study, the workload consists of 18 percent static requests, 57 percent dynamic request, 8 percent that are un-cacheable, and 17 percent request of other types. Static request usually incurs resource bottleneck of single resource, such as network transmission latency, while dynamic page request incurs a combination of multiple resources. The un-cacheable request incurs 500 and 100 times more CPU demand than a cache hit request. To parameterize and characterize the workload changing phenomena, we synthesize our changing workload using sine waves. The baseline of the sine wave is chosen to be workload without changing, that is, the baseline workload without surging user demands. We use two parameters, $W_{amplitude}$, W_{cycle_time} to control the "shapes" of the changing workload.

The amplitude of the sine wave ($W_{amplitude}$) is taken from 0.3 to 1, and the cycle time is 14 to 128. Figure 6.3 and Table 6.3 shows our synthesized changing workload pattern and the configuration for each parameter. All simulation is conducted on a time line of 700 units. We set $W_{amplitude}$ to be $\pm$0.3 to $\pm$1.0 request per time unit. The baseline ($\pm$0 request per time unit) incurs no change to the workload. In Figure 6.3, for example, the cycle-time for the sine wave is 106 time units, with amplitude $\pm$1.0 request per time unit. For the entire simulation run, there are about 6.6 cycles. As the cycle time increases, the number of cycles decreases (6.5 to 120 cycles in our experiments). Finally, the average token size is configured such that number of tokens each server has initially ranges 12 to 50.



Cycle inter-arrival unit: 106; Amplitude=±1.0request (per time unit)

Figure 6.2: Our synthesized workload

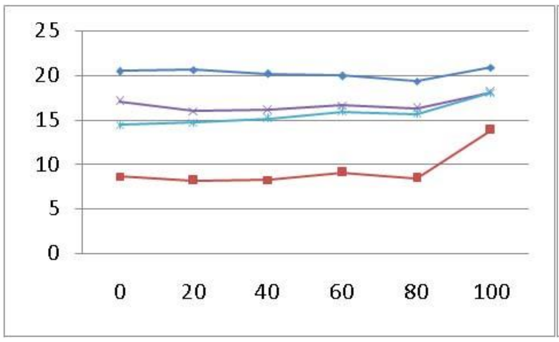
Table 6.3: The changing workload configurations of simulation:

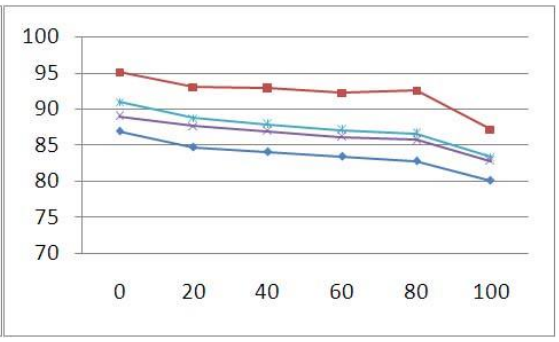
Amplitude	±0.3~1.0 request per time unit
Cycle time (Number of cycles per run)	14(50), 26(27), 66(10.6), 80(8.75), 106(6.6), 128(5.5) (unit: time unit (number of cycles per run)
Token size (Use token size and number of requests per 10 time units to calculate number of token)	1.68 (unit: gigaflop): used in all experiment except for token size effect comparison
Token size (Use token size and number of requests per 100 time units to calculate number of token)	4, 6, 8, 10, 12, 14, 16, 16.80 (unit: gigaflop): used in token size effect comparison

The major performance metrics measured in our simulation are response time, server utilization, queue length, and standard deviation of queue length. For server utilization, we measure CPU utilization as percentage of CPU in busy state, and use it as the server utilization. For memory and network utilization, it is measured as percentages of occupied capacities. The utilization of both can be affected by the job's CPU requirement. In our model, CPU is busy as long as there are jobs in execution, and jobs have different execution time depending on CPU capacity of each server. Even if there is only one job, the CPU is busy but the other resources may be under-utilized. Therefore we separately observe 3 resources utilization and show only the CPU utilization as server utilization. We will discuss memory and network utilization shortly.

The following sections provide performance evaluation of TBMM and ATBMM, which are compared with other algorithms BL and BB as well as the baseline algorithm where no load balancing is performed.

First, we consider the case of static workload with varying server capacities. The results are shown in Figure 6.3 - 6.7. In Figure 6.3 and 6.4, as the heterogeneity of server capacity grows, all the methods except our TBMM have high server variance and low server utilization, which suggests that the inter-server imbalance degree is high, and there exist bottlenecks for some resources while the other resources remain idle. This causes more jobs to wait in the wait queue, and the overall average utilization becomes low. TBMM achieves the lowest server variance and the highest server utilization in all the cases, suggesting that a request has more chance to be redirected to servers with under-utilized resources, and thus more jobs can get executed instead of waiting in the queue. This is also confirmed in Figure 6.5 and 6.6, in which TBMM achieves the lowest response time and queue length. Figure 6.7 further shows that TBMM has lowest variance for queue length, showing that the quality of service seen from each request is more uniform.

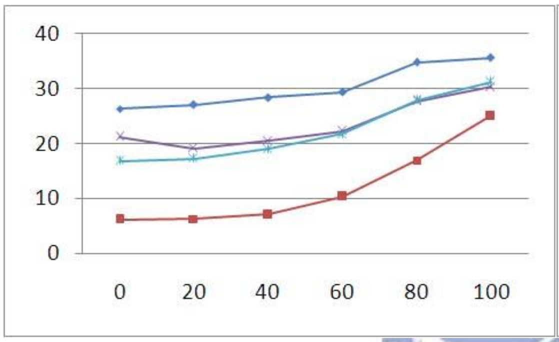


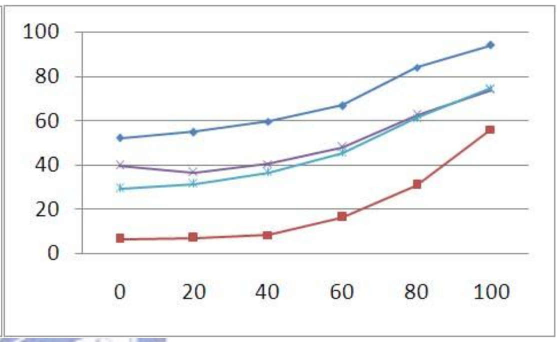


Y: Average Standard Deviation of Server Utilization

Y: Average Server Utilization (100%)

Figure 6.3, 6.4: Average standard deviation of server utilization and average server utilization

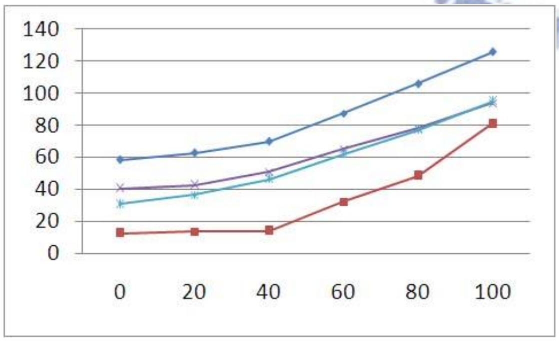




Y: Average Response Time (unit)

Y: Queue Length

Figure 6.5, 6.6: Average response time and average queue length

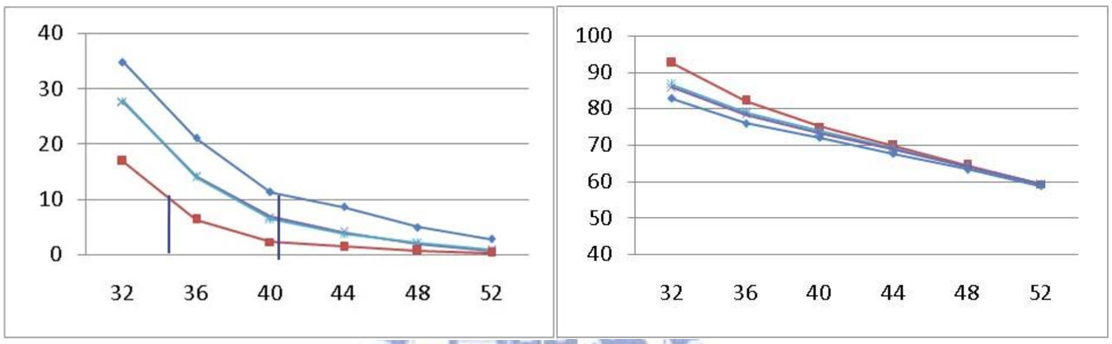


X-axis: Server Heterogeneity Degree

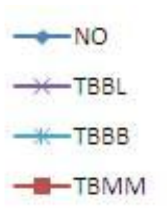
Y: Average Standard Deviation of Queue length

Figure 6.7: Average standard deviation of queue length

Figures 6.8 - 6.11 show the average number of servers required against response times under a fix load. The load for each of the 32 servers is fixed load in this experiment. For a targeted response time such as 10 sec, TBMM uses 34 servers, while TBBL and TBBB uses about 38 servers, and the baseline method uses about 41 servers. Moreover, for a server number of 32, TBMM outperforms all of the other resource balancing algorithms Note that as the number of servers grows, all methods have lower server utilization. This means allocating more servers can improve performance, but utilization will drop, wasting resources. Increasing the number of servers also lowers average queue length and variance, which reflects the fact that with more servers, jobs has more opportunities to be executed on any server, therefore fewer jobs will be queued and the performance improves.

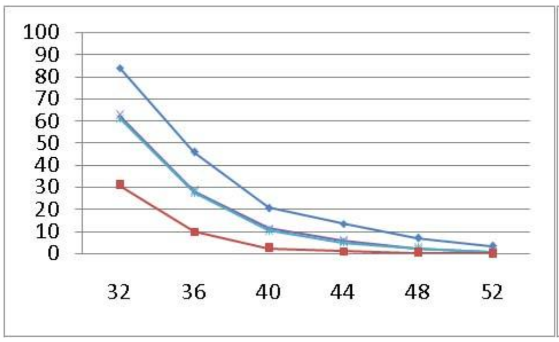


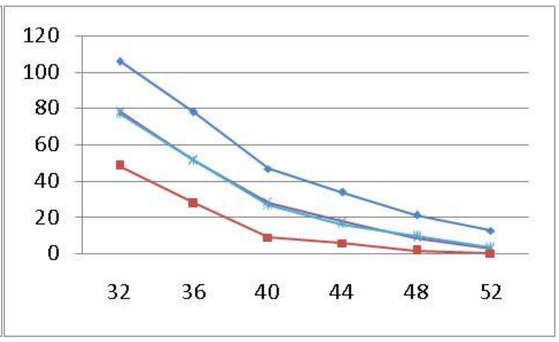
Y: Average Response Time (unit) Y: Average Server Utilization (100%)



X-axis: Number of servers

Figure 6.8, 6.9: Average response time and average server utilization





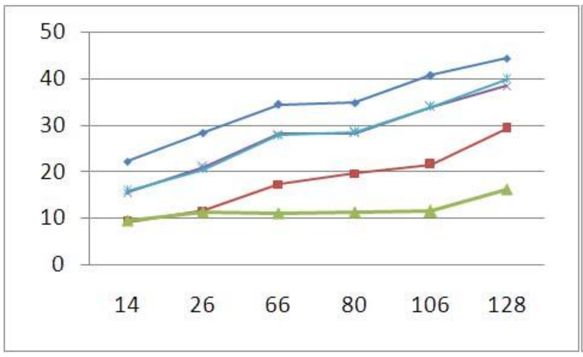
Y: Queue Length

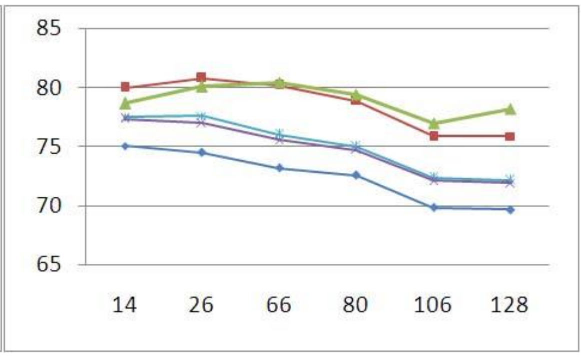
Y: Average Standard Deviation of Queue length

X-axis: Number of servers

Figure 6.10, 6.11: Average queue length and standard deviation

We examine the performance of ATBMM with changing workload as follows. Figures 6.12 - 6.15 show the results when increasing the cycle times. As cycle time increases, the average queue length increases. For shorter cycle time, although the job arrival rate surges during short period, it drops down quickly, too. The chances of forming a system bottleneck due to unhandled jobs increase as cycle time increases. We observe from the response time that ATBMM achieves lowest wait queue length. Moreover, it achieves lowest queue variance as cycle time increases. This shows that our method outperforms other multi-resource token-based algorithms.

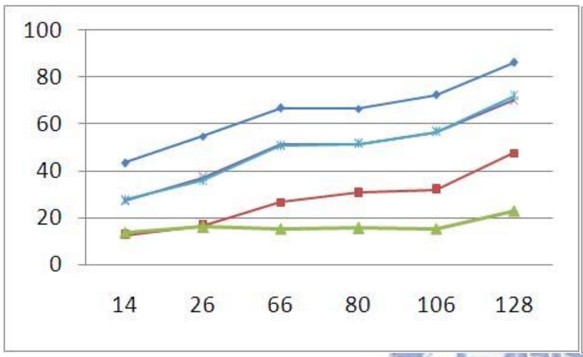


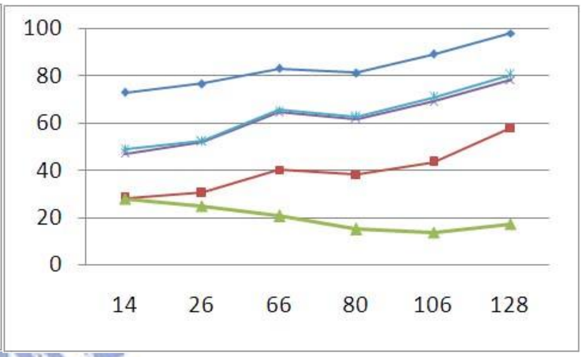


Y: Average Response Time (unit)

Y: Average Server Utilization (100%)

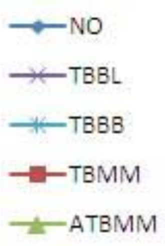
Figure 6.12, 6.13: Average response time and average server utilization





Y: Queue Length

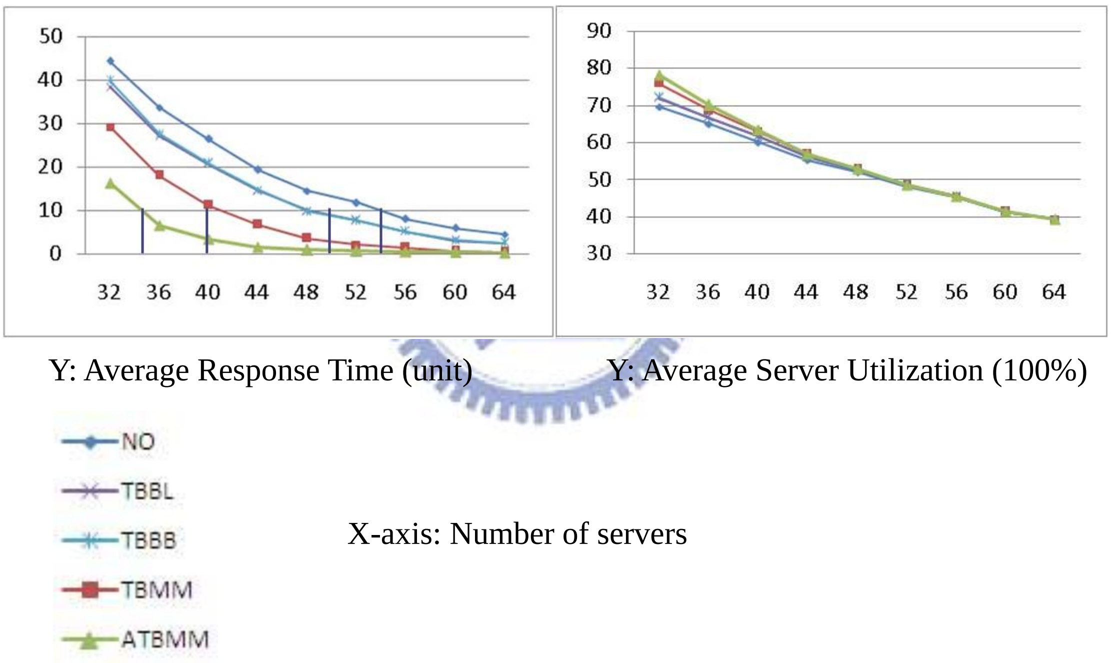
Y: Average Standard Deviation of Queue length



X-axis: Cycle-inter-arrival time (unit)

Figure 6.14, 6.15: Average queue length and standard deviation

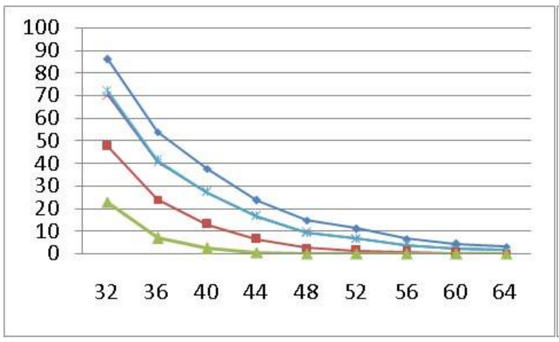
We also evaluate the number-of-servers effect on the response time for changing workload. Figures 6.16 - 6.19 show the results for high magnitude (±1.0 request per time unit) and long cycle time. Again, the load of each server is fixed. For a fixed response time constraint such as 10 sec, ATBMM needs 34 servers, while TBMM needs 40 servers. For TBBL and TBBB, it’s about 50 servers. For baseline method, a number of about 54 servers is required. The queue length results have similar trend. Note that in the case of average standard deviation of queue length, ATBMM achieves a relatively low standard deviation, and remains low even if number of servers decreases. This shows that the queue lengths are balanced for each of the servers, while the other methods have high imbalance degree for the queue lengths. Another noticeable phenomenon in the simulation is the dramatic improvement due to increasing number of servers. However, this significant improvement does not imply that the larger number of servers, the better the performance, because the server utilization will be lowered. In addition, we configure the system with a fix workload that is about 70% to 80% server utilization and average queue length about 90 jobs for baseline method. This already saturated workload may cause a high response time, while increasing more servers help distribute the workload to more servers, and average queue length decreases, and thus lower response time.

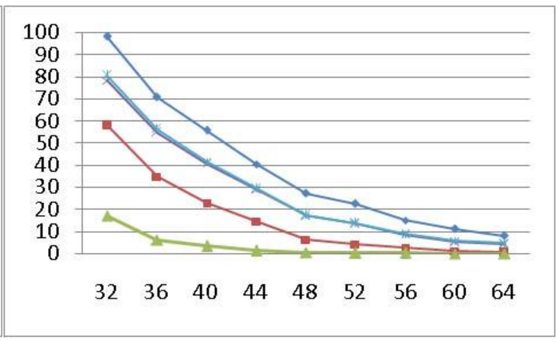


Y: Average Response Time (unit) Y: Average Server Utilization (100%)

X-axis: Number of servers

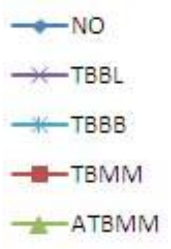
Figure 6.16, 6.17: Average response time and average server utilization





Y: Queue Length

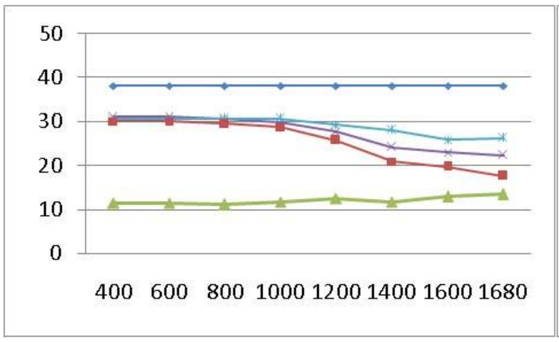
Y: Average Standard Deviation of Queue length

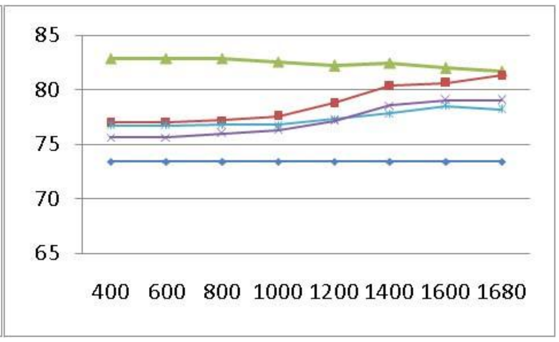


X-axis: Number of servers

Figure 6.18, 6.19: Average queue length and standard deviation

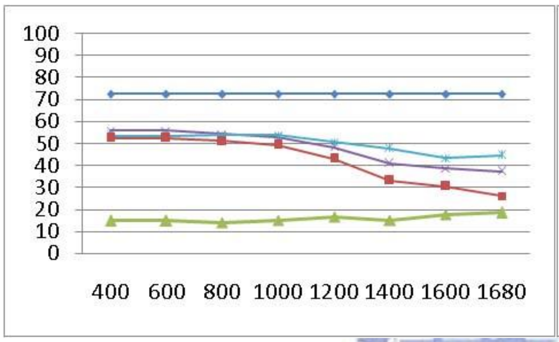
We also investigate the effect of increasing token size. Figures 6.20 - 6.23 show the simulation results. As size of token increases, the number of tokens for each server decrease. All the load balancing methods except ATBMM have long response times when the token size is small. The reason is that, when the token size is small, it becomes more difficult for other methods to balance the surging workload. In this case, our algorithm can adapt to changing workload, adding more tokens when workload surges, and can balance more system imbalance. Although not shown here, we have also observed that using small token size does not incur too much overhead because it does not induce excessive token exchanges when compared to the cases where tokens are larger.

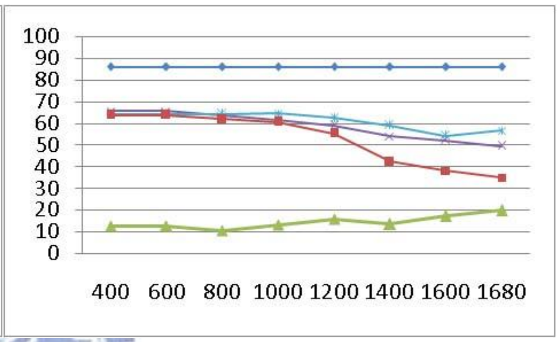




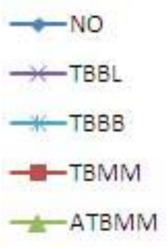
Y: Average Response Time (unit) Y: Average Server Utilization (100%)

Figure 6.20, 6.21: Average response time and average server utilization





Y: Queue Length Y: Average Standard Deviation of Queue length



X-axis: Token size (unit: 100 gigaflop)

Figure 6.22, 6.23: Average queue length and standard deviation

Chapter 7 Discussion and Future work

It is interesting to compare the effect of changing workload on the number of servers required to meet a given response time requirement. Under static workload, performance of all load balancing methods improves quickly as the number of servers increase. For a target response time such as 8 time units, the differences of TBMM compared with existing load balancing methods (including the baseline method) is 4 and 10 servers, respectively. Under changing workload, the differences between TBMM and the other methods grow to 8 and 12 servers, respectively, with the same response time constraint (8 time units). ATBMM reduces an additional 6 to 8 servers compared to TBMM. In fact, as workload changes, the surging workload above baseline causes a large number of jobs queued at each server. If there is not a good adaptive way to adjust the system to tackle the surging workload, it becomes a source of system bottleneck. On the contrary, as workload declines, the number of jobs queued at server also declines, and all methods can handle the requests quickly, hence the differences of all methods are not significant.

Token size is also an issue. As the token size increases, the number of token transfers used to balance workload of each server also increases for all the load balancing methods except ATBMM. Higher number of token exchanges implies higher system overhead (although not further investigated in this thesis). In the future we will also study mechanisms that can dynamically adjust token size based on workload change. The goal is to reduce token transfers while maintaining high system performance.

Chapter 8 Conclusion

In this thesis, we have defined the load balancing problem that takes into account multi-resource requirements in the context of web-based systems, and investigated an token-based approach to load balancing for such systems. We investigated how well existing multi-resource load balancing algorithms perform when request patterns are heterogeneous and time-varying. Our proposed token-based load balancing algorithm adapts well to the changing workload by maintaining a dynamically changing token set that reflects the workload change for each server, so that temporal surge in workload for a server can be shifted to other servers via the market mechanism. Simulation shows that TBMM outperforms other multi-resource load balancing methods under various system configurations, especially when inter-server and intra-server resource heterogeneity are high. We also showed that when workload changes over time, the ATBMM method also outperforms the other methods, including TBMM.

References

[Anderson, et al] David P. Anderson, Eric Korpela, “High-Performance Task Distribution for Volunteer Computing”, IEEE International Conference on e-Science and Grid Technologies, December 2005, pp. 196 - 203

[Anderson, et al] David P. Anderson, John McLeod, “Local Scheduling for Volunteer Computing”, IEEE International Conference on e-Science and Grid Technologies. December 2005,

[Borovska, et al] Plamenka Borovska, Milena Lazarova, “Token-Based Adaptive Load Balancing for Dynamically Parallel Computations on Multicomputer Platforms,”International conference on Computer systems and technologies, vol. 285, 2007, Article No. 10.

[Bryhni, et al] H. Bryhni, E. Klovning, and O Kure, “A Comparison of Load Balancing Techniques for Scalable Web Servers”, IEEE Network, Vol. 14(4), July/August 2000, pp. 58 - 64.

[Cardellini, et al] V. Cardellini, M. Colajanni, and P. S. Yu, “Dynamic Load Balancing on Web-Server Systems”, IEEE Internet Computing, Vol. 3May, 1999, pp. 28 - 39.

[Eager, et al] D.L. Eager, E.D. Lazowska, and J. Zahorjan, “A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing,” Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems, Oct. 1985, pp. 1 - 3.

[Kulkarni, et al] Parag Kulkarni, Indranil SenGupta, “Dual and multiple token based approaches for load balancing,”Journal of Systems Architecture Vol. 51 (2005) pp. 95–110.

[Leinberger, et al] W. Leinberger, G. Karypis, and V. Kumar, “Load Balancing Across Near-Homogeneous Multi-Resource Server,” Proceedings of Heterogeneous Computing Workshop, Aug. 2000, pp. 60 - 74.

[Leinberger, et al] W. Leinberger, G. Karypis, and V. Kumar, “Job Scheduling in the presence of Multiple Resource Requirements,” Proceedings of the 1999 ACM/IEEE conference on Supercomputing, Article No. 47.

[Lin, et al] Frank C. H. Lin and Robert M. Keller,, “The Gradient Model Load Balancing Method,” IEEE Transactions on Software Engineering, vol. 13(1), January 1987, pp. 32 - 38.

[Paton, et al] Norman W. Paton, Marcelo A. T. de Arag˜ao, Kevin Lee, Alvaro A. A. Fernandes, Rizos Sakellariou, “Optimizing Utility in Cloud Computing through Autonomic Workload Execution,” IEEE Data Engineering Bulletin, vol. 32, 2009, pp. 51 - 58

[PeerSim] http://peersim.sourceforge.net/

[Ranjan, et al] Supranamaya Ranjan and Edward Knightly, “High-Performance Resource Allocation and Request Redirection Algorithms for Web Clusters,” IEEE Transactions on parallel and distributed systems, vol. 19, NO. 9, September 2008, pp. 1186 - 1200.

[Shin, et al] Chee Shin Yeo and Rajkumar Buyya, “Pricing for Utility-driven Resource Management and Allocation in Clusters,” International Journal of High Performance Computing Applications,vol. 21(4), November. 2007, pp. 405 - 418.

[Shivaratri, et al] N.G. Shivaratri, P. Krueger, and M. Singhal, “Load Distributing for Locally Distributed Systems,” Computer, Vol. 25, December, 1992, pp. 33 – 44.

[Ting] Nyik San Ting, “A Generic Peer-to-Peer Network Simulator,” Department of Computer Science, University of Saskatchewan, 2003

[Yang, et al] Chih-Chiang Yang, Kun-Ting Chen, Chien Chen and Jing-Ying Chen, “Market-based Load Balancing for Distributed Heterogeneous Multi-Resource Servers”, (proposed, accepted on) International conference on parallel and distributed systems, ICPADS, December, 2009)

[Zhang, et al] Z. Zhang, and W. Fan, “Web server load balancing: A queueing analysis,” European Journal of Operational Research, Vol. 186 (2007), pp. 681-693.