突波式流量之網頁伺服器負載平衡架構的研究

A Study of Web Server Load Balancing Architecture for Burst Mode Traffic

研 究 生：陳建伯（Jian-Bo Chen）
指導教授：包蒼龍（Prof. Tsang-Long Pao）

大同大學

資訊工程研究所

博士論文

Ph. D. Dissertation
Department of Computer Science and Engineering
Tatung University

中華民國 九十七 年 七 月
July 2008

大同大學

資訊工程研究所

博士學位論文

突波式流量之網頁伺服器負載平衡架構的研究

陳建伯

經 考 試 合 格 特 此 證 明

博士學位論文考試委員

包蒼龍

許超雲

黃有評

李良德

王金龍

李遠士

指導教授

包蒼龍

所長

鄭福炯

中 華 民 國 97 年 7 月 4 日

ACKNOWLEDGMENTS

Due to the effective help of numerous hearty persons, this dissertation is accomplished and published. Here, I am much obliged to them for their support and suggestions.

I wish to thank Prof. Tsang-Long Pao, my advisor, for having accepted me as a member of his research group at Tatung University. He helped me and encouraged me to get involved in group projects and seminars throughout my doctorate study at Tatung University. He also gave me the opportunity of having different point of view on various subjects.

I would like to express my deepest gratitude to my dissertation committee members, Prof. Hsuan-Shih Lee, Prof. Yo-Ping Huang, Prof. Jin-Long Wang, Prof. Chau-Yun Hsu, and Prof. Liang-Teh Lee for their invaluable guidance, support, encouragement, helpful comments and suggestions. I would also like to acknowledge all the Lab members at Tatung University and colleagues in the Information Network Division at Ming Chuan University for their valuable discussions and constant friendships. Finally, I am grateful to my father, my mother, and my wife. Without their consistent support and patience, this work would not have been possible.

ABSTRACT

The server load balancing architecture is the most efficient way to solve the heavy loading problem of popular server. There are different solutions in implementing the load balancing system. In this dissertation, we adopt a flexible registration protocol that can easily add a new backend server to the load balancing system to share the load. In addition, the registration protocol also reports the real time backend server loading status to the load balancing system. So we can use these information to distribute client requests by any available load balancing algorithms.

The purpose of load balancing algorithm is to improve the load sharing performance of the popular web server. Most of the load balancing architectures are based on supporting homogeneous backend servers. If the hardware specifications of backend servers in the system are different, the load balancing system must have a strategy to fairly dispatch the load to the backend servers. We derive a capacity measurement for heterogeneous backend servers. From the experimental results, the maximum number of connection with certain drop rate can be used as the capacity, but it can not provide fair response time for each client requests. Thus, we must consider the capacity not only depending on drop rate but also on the response time. Using this measurement, the average response time for all client requests will nearly be the same.

In addition to the definition of capacity, we also use the remaining capacity algorithm to reduce the hardware cost when the web site does not have frequent burst requests. We propose the concept of service-on-demand servers, which can bring other servers such as DNS servers or MAIL servers to join the load balancing system during the burst traffic period. For example, the course registration system or ticket reservation systems have burst requests only several times a year. The proposed algorithm can use the remaining capacity of DNS or MAIL server to share the load in the burst request period.

The simulation results show that using the remaining capacity of both DNS server and MAIL server can achieve higher performance compared to using another dedicated backend server.

Due to the different remaining capacities that these servers have, we need an intelligent mechanism to make the load distribution decision. We propose an algorithm using the fuzzy decision algorithm to dispatch the client requests to the appropriate backend server. The CPU idle percentage, the available memory percentage, and the available connection percentage for each backend servers are the input parameters of our fuzzy decision algorithm. The most appropriate backend server can thus be determined. The simulation results show that the fuzzy decision algorithm can achieve higher performance than other load balancing algorithms.

Keywords: load balance, remaining capacity, fuzzy decision, service-on-demand

中文摘要

伺服器負載平衡架構是經常用來解決熱門網站負載分攤的一種方法，而負載平衡系統有許多不同的解決方案。在本論文中，我們採用一種有彈性的註冊協定，使得負載平衡系統能夠很容易增加後端伺服器來分擔負載。除此之外，在這註冊協定的訊息中，同時也將伺服器的即時負載量回報給負載平衡系統，利用這些資訊，便可採用任一可行的負載平衡演算法來分配使用者的需求。

負載平衡演算法的目的是為了改善熱門網站的效能。大部分的負載平衡架構都僅能適用在同質性的後端伺服器上。如果後端伺服器的硬體規格不同時，負載平衡系統必須要有一種策略來將用戶端的連線需求，公平地分配到每一台後端伺服器上。我們推導一個異質性後端伺服器運算能力的度量機制。這些運算能力可以根據在特定丟棄率的條件下，後端伺服器所能提供的最大連線數來決定。然而，根據實驗的結果可知，這種定義方式無法保證每個用戶端均有公平的連線回應時間。因此，在考慮定義運算能力時，我們不僅是要考慮到丟棄率，同時也要考慮到回應時間。採用這樣的定義方式，對於所有用戶端連線的平均回應時間幾乎都是相同的。

除了定義運算能力之外，當網站並不是經常性的有突波連線需求時，我們可以採用剩餘能力負載平衡演算法來降低負載平衡系統中的硬體成本。我們提出一種隨選服務(service-on-demand)伺服器的概念，也就是在有突波連線需求時，將其他的伺服器像是 DNS 伺服器或是 MAIL 伺服器等，加入到負載平衡系統中來分擔流量。舉例來說，學校的選課系統或是訂票系統等，一年之中會有突波需求的次數僅有少數幾次而已。這種剩餘能力演算法可以利用 DNS 伺服器或是 MAIL 伺服器的剩餘能力，當突波連線需求來臨之前，加入到負載平衡系統中來分擔流量。模擬的結果發現，當同時使用 DNS 伺服器以及 MAIL 伺服器的剩餘能力時，所得到的結果，會比

使用另一台專屬後端伺服器的效果來的要好。

由於每一台後端伺服器的剩餘運算能力均不相同，因此需要一種智慧型的機制來做決策。我們提出一種模糊決策的演算法，來將使用者的連線需求派遣給最適合的後端伺服器處理。每一台後端伺服器的 CPU 的閒置百分率、可用的記憶體空間以及可用的連線數，當成是模糊決策演算法的三個輸入參數。根據這些參數來計算最後的明確值以決定最適合的後端伺服器了。模擬的結果發現，當採用模糊決策負載平衡演算法時，可以比其他的演算法，得到更高的效能。

關鍵字:負載平衡、剩餘能力、模糊決策、隨選服務

TABLES OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction

Due to the growing popularity of the World Wide Web, the traffic of popular web sites has grown far beyond the capacity of a single web server. Most popular web sites adopt a distributed or parallel architecture to alleviate the load for the single server [1]. These sites can provide higher performance for a large number of client requests [2,3]. Although the load balancing architecture consists of a number of backend servers, they act as a single unit. User transparency is implemented to allow clients to issue the requests to the central unit without knowing the load balancing architecture the web site implemented. Meanwhile, clients do not need to make any configuration modifications when they connect to the load balancing systems.

There are various methods used to build a load balancing system. These methods include the hardware-based approach [4-11], cluster-based approach [12-17], DNS-based (Domain Name Server) approach [18-29], dispatcher-based approach [30-34], and so on. In hardware-based and DNS-based approaches, the exact workload of each individual backend server in the system may not take into consideration which might lead to an unbalanced load situation. For the DNS-based approach, another major problem is the DNS query result caching in the intermediate DNS server and the client itself [35]. In this case, requests from hosts in the same domain may all be served by the same backend server and may drive that server into overload state.

In the dispatcher-based approach, the central unit is the dispatcher which is responsible for dispatching the client requests to the most appropriate one among backend servers. In the server-state dispatching architecture [36], the dispatcher must collect the status of all the backend servers and make the decision regarding which backend server is the most appropriate one to serve the request. The decision criteria are based on the status of backend servers such as CPU loading, memory usage, current number of connections, and so on [37].

1.2 Motivation

In this dissertation, our focus is on using the dispatcher-based approach to solve the burst traffic load problem. In some web systems, the workload of web servers is light for most of the time, but may incur a heavy traffic load during a specific period of time. For instance, the course registration system and the ticket reservation system will incur burst traffic during a specific period of time several times a year. During most of the time throughout the year, they only have a very light traffic load. Although the load balancing system can efficiently solve the burst traffic problems, the cost of the backend servers is the main issue. If we use a powerful dedicated server as the backend server, the investment in the hardware does not seem to be cost effective. Thus, we proposed to use some service-on-demand servers as the backend servers prior to the anticipated burst traffic period. These service-on-demand servers are not dedicated servers for the load balancing system; in fact they have their routing jobs to do, such as acting as DNS server or MAIL servers. During the heavy web load period, we initiated the web daemon and server registration protocol for the DNS server and the MAIL server to cooperate with the original web server to form the load balancing system. The contents of the web server were already stored inside a separated disk space or SAN, so we only needed to mount the file system. In doing so, the dispatcher knows that these additional backend servers can share the load. In addition to these service-on-demand servers, we can also use the personal computer in the PC classroom booting up with a live-CD to join the load balancing system when all of these service-on-demand servers still cannot handle the traffic load.

1.3 Objective

In our proposed load balancing system, the first issue is the registration process of a new backend server. Before the burst traffic period, the service-on-demand servers must register themselves into the load balancing system. The service-on-demand servers start the process of server registration protocol, which advertise a registration message to inform the dispatcher that a new backend is ready to serve the client requests. In addition, the registration protocol of the backend server also reports the update-to-date loading information to the dispatcher that can be used by the dispatcher to decide which one is the most appropriate to serve the client request.

In the load balancing system, the capacities of each backend servers may be different, such as the CPU speed, memory, and network. In this kind of heterogeneous load balancing system, the dispatcher must have initial weights for each backend servers in order to balance the load [38-44]. The weights can be determined by the static factors such as CPU, memory, and so on. But when considering the fair response time for each client, we proposed a weighted distributed load balancing algorithm that their initial weights are determined by the average response time [45-55]. In this proposed algorithm, the average response time of client requests are nearly the same no matter which backend server serve the requests.

Although each heterogeneous backend servers have different initial weights, the current loading information is another important factor in the load balancing system [56-68]. Some requests need large amount of resource of backend server, but others not. If the load balancing system dispatches the client request to the most appropriate backend server just according to the initial weights, the system load will eventually become unbalance. To avoid this situation, we propose the remaining capacity load balancing algorithm to dispatch the client request to the most appropriate backend according to the current loading information of each backend servers [69]. In this algorithm, we use the CPU idle percentage, available memory, and current connection number to calculate the remaining capacity of each backend server. The backend server with highest remaining capacity is the candidate to serve the request. In this remaining capacity algorithm, the service-on-demand servers can easily join the load balancing system because the dispatcher dispatches the client request according to the remaining capacity. If the routing job for the service-on-demand server is heavy, the remaining capacity of this service-on-demand server is less than others, and the dispatcher will not dispatch the client request to this server. Thus, the routing job will not be influenced by the load balancing system.

In addition to the remaining capacity, we can use an intelligent algorithm to decide which backend server is the most appropriate one. In this dissertation, we adopt a fuzzy decision algorithm [70-73] to determine which backend server should respond to the client request. First, we will collect the status of the backend servers as the input parameters, such as the CPU idle percentage, available memory percentage, and available connection percentage. We quantify these features and define the membership functions for these features. Then, the membership function degree can be used as the parameters for rule evaluation. After the rule evaluation process, we will get the fuzzy decision values and fuzzy degree for these input parameters. The final process, defuzzification, will generate crisp values for these input parameters. The crisp value is used to determine which backend server is the most appropriate one to serve the incoming client request.

1.4 Dissertation Organization

This dissertation is organized as follows. Chapter 2 addresses some related works. Chapter 3 introduces the load balancing algorithms, including the flexible server registration protocol, the weighted distributed load balancing algorithm, the remaining capacity load balancing algorithm, and the fuzzy decision load balancing algorithm. Chapter 4 presents the experimental and simulation results. Chapter 5 states our conclusions.

CHAPTER 2

RELATED WORKS

In this chapter, we will discuss some backgrounds and related researches about load balancing architectures and solutions.

2.1 Hardware-based Load Balancing Solutions

2.1.1 High Performance Server

A single server is generally not able to process the burst requests because of the limitation of hardware performance. The simplest solution is to upgrade its hardware. If the CPU is too busy, we can add extra CPU(s). If the memory utilization is too high, we can add more memory. If the network is too busy, we can upgrade with a high speed network interface card. The advantages of this kind of systems are easy to setup and maintain. And the web server administrator only needs to manage one single machine. The disadvantages are the cost and performance limitation of the hardware. Furthermore, the server is the single point of failure which may not be desired for reliable operation.

2.1.2 Cluster Approach

The second solution is to adopt the cluster approach. The cluster server system consists of many independent servers that work together. The workload is evenly dispatched to independent servers by using a proper dispatching algorithm. The advantage of this approach is that it can add more servers into the cluster easily, and from the client point of view, the server is still a single unit. The disadvantages are hard to setup and maintain. It must use some specific hardware and software to group servers as a single unit.

2.1.2.1 Microsoft Windows Cluster

While Windows 2000 represents a dramatic improvement over its predecessors in terms of the total uptime (availability), reduced system failure (reliability) and ability to add resources and computers to improve performance (scalability), Windows Server 2003 takes the availability, reliability and scalability of the Windows operating system to the next level by enhancing existing features and providing new options.

Microsoft clustering technologies are the key to improve availability, reliability and scalability. With Windows 2000 and Windows Server 2003, Microsoft uses a three-part clustering strategy.

Network Load Balancing provides failover support for IP-based applications and services that require high scalability and availability. With Network Load Balancing (NLB), organizations can build groups of clustered computers to support load balancing of TCP or UDP traffic requests.

Component Load Balancing provides dynamic load balancing of middle-tier application components that use COM+. With Component Load Balancing (CLB), COM+ components can load balanced over multiple nodes to dramatically enhance the availability and scalability of software applications.

Server Cluster provides failover support for applications and services that require high availability, scalability and reliability. With clustering, organizations can make applications and data available on multiple servers linked together in a cluster configuration. Backend applications and services are ideal candidates for server cluster.

2.1.2.2 Linux Cluster

Linux Virtual Server (LVS) is a highly scalable and highly available server built on a cluster of real servers. The architecture of cluster is transparent to end users, and the users interact with the system as if it were only a single high performance virtual server.

The real servers may be interconnected by high-speed LAN or by geographically dispersed WAN. The front-end box in front of the real servers is a load balancer, which schedules requests to the different servers and make parallel services of the cluster to appear as a virtual service on a single IP address. Scalability is achieved by transparently adding or removing a node in the cluster. High availability is provided by detecting node or daemon failures and reconfiguring the system appropriately. The three-tie architecture consists of :

Load Balancer, the front-end machine of the whole cluster systems, balances requests from clients among a set of servers. The clients will consider that all the services is from a single IP address.

Server Array, which is a set of servers running actual network services, such as WEB, MAIL, FTP, DNS or Media service.

Shared Storage, which provides a shared storage space for the servers so that it is easy for the servers to have the same contents and provide the same services.

2.1.3 Server Switch

During the last few years, an active commercial market for server switching products has emerged [5]. Many of these products are Ethernet switches supplemented with build-in processing power to examine the incoming packet and manage service traffic intelligently, assign requests to servers based on request content, client session, and/or server status. Server switches and their request routing (server selection) policies play a key role in managing content and server resources for scalable Internet services [4,6].

Server switching is a technique to virtualize services at the IP level. An ensemble of servers cooperates to serve the request loading. Clients interact with the service through a client/server protocol such as HTTP, addressing their request to a virtual IP address representing the service. The server switch intercepts the incoming traffic and redirects each request to a specific server according to predefined policies. The set of functioning servers to choose from may grow and shrink dynamically, allowing a site to manage server resources locally to adapt to load changes. The switch isolates clients from internal details of the service structure, so that the ensemble appears to clients as a single virtual server that is powerful and reliable. Commercial server switches are available from Notel (Alteon) [7], Cisco (Arrowpoint) [8], Extreme [9], F5 Networks [10], and other companies.

The Alteon server switch [6,7] recognizes when a client is requesting a new TCP session by identifying the TCP SYN packet. The request is forwarded to the best available server, based on the configured load balancing policy. Once the switch determines the best server, it binds the session to that server's real IP address. The server switch maintains a binding table that associates each active session with the real server to which is assigned. After the server switch binds a connection request to a real server, it performs address substitution, so the real server will transparently receive packets for that session. The switch replaces the virtual IP address in the IP destination address with the server's real IP address and replaces the switch's MAC address in the destination address field with the server's MAC address. Figure 2.1 illustrates how IP addressing substitution takes place as traffic flows inbound from the client to the real server [11].

After performing the necessary address substitution, the server switch forwards the connection request to the chosen server. All subsequent packets belonging to that session undergo the same address substitution process and are forwarded to the same real server until the switch sees a session termination packet (that is, a TCP FIN packet). Likewise, the server switch intercepts packets traveling form the real server to the client and performs the reverse address substitution. It replaces the real server's actual IP address in the Network Layer source address field with the virtual IP address and forwards each modified frame to the client. The process is described in Fig. 2.2.

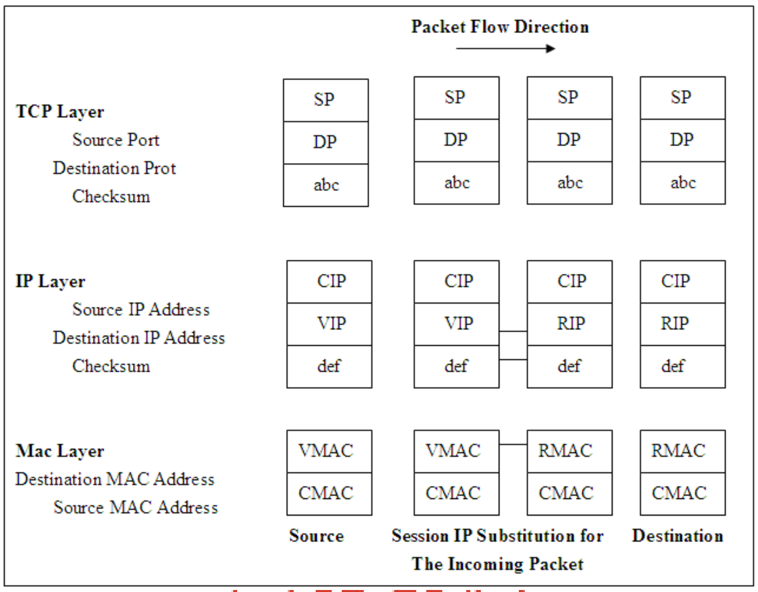


Figure 2.1: Incoming session ID substitution.

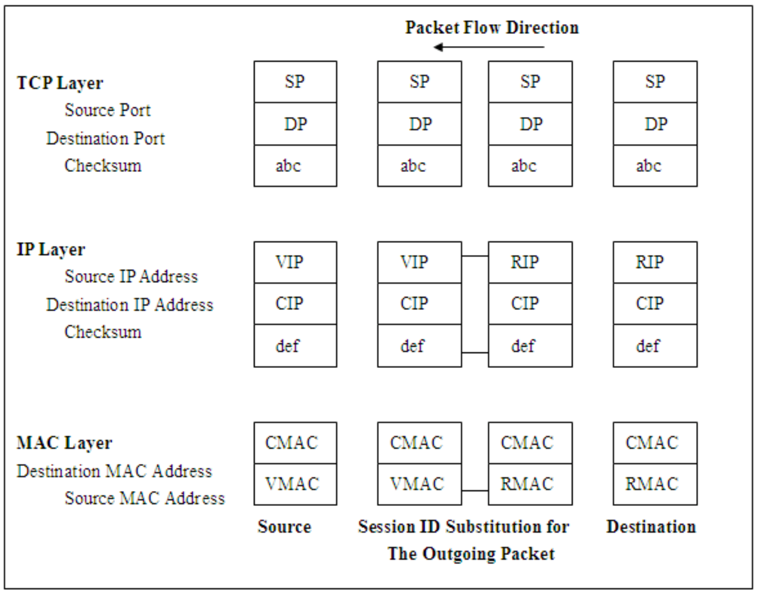


Figure 2.2: Outgoing session ID substitution.

On the receiving of a TCP FIN packet, the server switch performs the necessary address substitution and forwards the FIN packet to the appropriate real server, causing the server to teardown the connection. Then it removes the session-server binding from its binding table.

The advantages are that the load balancing policies can be varied by the devices, and the implementations are hardware-based, so they have the highest performance. The disadvantages are that they were designed by the manufactures, so we are not able to modify the source codes of devices. Furthermore, these devices are always quite expensive.

2.2 Software-based Load Balancing Solutions

2.2.1 DNS-based Approach

In the distributed web server architectures that use request routing mechanisms on the cluster side, there is no additional action to take in the client side. Architecture transparency is typically obtained through a single virtual interface to the outside world, at least at the URL level. The cluster DNS — the authoritative DNS server for the distributed web nodes — translates the symbolic site name (URL) to the IP address of one server. This process allows the cluster DNS to implement many policies to select the appropriate server and spread client requests. The DNS, however, has a limited control on the request reaching the web cluster. Between the client and the cluster DNS, many intermediate name servers may cache the logical-name-to-IP-address mapping to reduce network traffic. Moreover, the client will also cache the result of address resolution.

In addition to provide the IP address of a node, the DNS also specifies a validity period (Time-To-Live, or *TTL*) for caching the result of the logical name resolution [22,23]. When the *TTL* expires, the address-mapping request is forwarded to the cluster

DNS to obtain the IP address map again; otherwise, an intermediate name server will handle the request. Figure 2.3 shows the resolution.

If an intermediate name server holds a valid mapping for the cluster URL, it resolves the address-mapping request without forwarding it to upper level name server. Otherwise, the address request reaches the cluster DNS, which selects the IP address of a web server and the *TTL*. The URL-to-IP-address mapping and the *TTL* value are forwarded to all intermediate name servers along the path and to the client.

We distinguish the DNS-based architectures by the scheduling algorithm that the cluster DNS uses. These algorithms are classified by the system state information that the DNS uses to select a web server node.

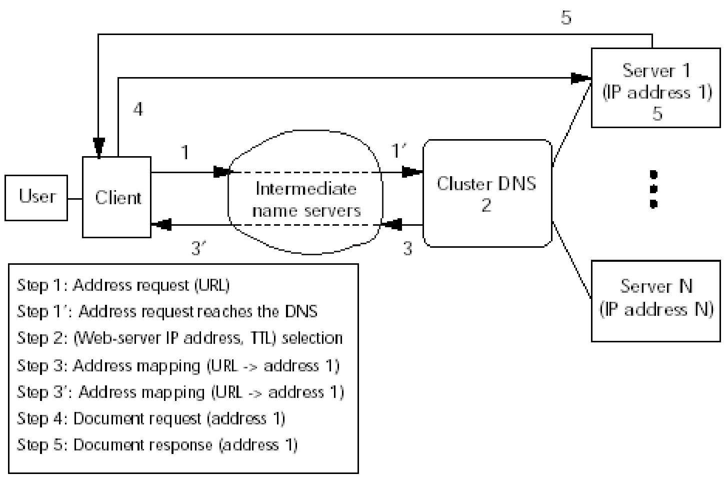

5
Server 1
(IP address 1)
5
4
1
1′
User
Client
Intermediate name servers
Cluster DNS
2
3′
3
Server N
(IP address N)
Step 1: Address request (URL)
Step 1′: Address request reaches the DNS
Step 2: (Web-server IP address, TTL) selection
Step 3: Address mapping (URL -> address 1)
Step 3′: Address mapping (URL -> address 1)
Step 4: Document request (address 1)
Step 5: Document response (address 1)

Figure 2.3: DNS-based approach to load balancing.

2.2.1.1 System-stateless Algorithms

The Round-Robin DNS (RR-DNS) approach, first implemented by the National Center for Supercomputing Applications (NCSA) to handle increased traffic at its site, is for distributed homogeneous web server architecture [3]. NCSA developed a web cluster comprising the following entities: a group of loosely coupled web servers to respond to HTTP requests; a distributed file system that manages the entire WWW document tree; and one primary DNS for the entire web server system.

NCSA modified the primary DNS for its domain to map addresses by a Round-Robin algorithm. The load distribution under the RR-DNS is unbalanced because the address-caching mechanism lets the DNS control only a small fraction of requests. An uneven distribution of client requests from different domains further adds to the imbalance such that many clients from a single domain can be assigned to the same web server, which overloads server nodes [24,25].

Additional drawbacks result because the algorithm ignores both server capacity and availability. With an overloaded or non-operational server, no mechanism can stop the clients from trying to access the web site by its cached address continuously. The RR-DNS policy’s poor performance needs further study for alternative DNS routing schemes that require additional system information. Example of RR-DNS is shown in Fig. 2.4.

```
www    IN    A    192.168.1.1
       IN    A    192.168.1.2
       IN    A    192.168.1.3
       IN    A    192.168.1.4
       IN    A    192.168.1.5
       IN    A    192.168.1.6
       IN    A    192.168.1.7
       IN    A    192.168.1.8
```

Figure 2.4: Configuration of RR-DNS.

2.2.1.2 Server-state-based Algorithms

Knowledge of server state conditions is essential for a high available web server system to exclude servers that are unreachable because of fault or congestion. DNS policies, combined with a simple feedback alarm mechanism from highly utilized servers, effectively avoid web server system overload [24]. The Sun-SCALR framework implements a similar approach combined with the RR-DNS policy [26].

R. J. Schemers proposed and developed the *lbnamed* algorithm which make scheduling decision based on the web servers current loading [27,28]. The DNS, after receiving an address query, selects the least-loaded server. To inhibit address caching at name servers, the *lbnamed* algorithm requires that the DNS sets the *TTL* value to zero. This requirement limits the applicability.

The *lbnamed* is a load balancing name server written in Perl. Of course it was meant to be a proof of concept that would get added back into BIND [29]. *lbnamed* allows the creation of dynamic groups of hosts that have one name in the DNS name space. A host may be in multiple groups at the same time [28].

The load balancer consists of two *perl* programs, *lbnamed* and *poller*. These programs run in parallel and communicate using signals and *configuration files*. The *poller* program contacts the daemon running on the backend servers being polled. It reads a *configuration file* that tells the *poller* which backend servers to poll. The *poller* periodically sends out requests and receives the responses asynchronously. After it has received all the responses, it dumps the information into a *configuration file* and sends a signal to *lbnamed* which then reloads its *configuration file*. If the *poller* does not receive a response from one of the backend servers being polled, it simply removes it from the *configuration file* it feeds to *lbnamed*.

The *lbnamed* reads the configuration file generated by the *poller* and stores the configuration into its memory. Each group of backend servers is stored in an array, while the weights of all the backend servers are stored in one hash table. When a request for a particular group comes in the array for that group is sorted based on the weight of each backend server in that group. The backend server with lowest weight is then returned as the best server. The weight in the corresponding entry is increased.

To other name servers, *lbnamed* looks like a standard DNS server, with the exception that it does not answer recursive queries. It only handles requests for the dynamic groups it maintains. *lbnamed* gets a normal DNS query and based on the name in the query, it selects the best host to return. *lbnamed* then constructs a standard DNS response and sends it back to the client. The *TTL* value in the response is set to 0 to ensure the response from being cached by other name servers which would defeat the whole mechanism.

Backend servers that are going to be polled by the *poller* need to run a special daemon. That daemon responds to *poller* requests (over UDP) using a simple protocol. The protocol format is described in Fig. 2.5.

```
#define PROTO_PORTNUM 4330
#define PROTO_MAXMESG 2048              /* max udp message to receive */
#define PROTO_VERSION 2

typedef enum P_OPS {
    op_lb_info_req              =1,      /* load balance info, request and reply */
} p_ops_t;

typedef enum P_STATUS {
    status_request              =0,     /* a request packet */
    status_ok                   =1,     /* ok */
    status_error                =2,     /* generic error */
    status_proto_version        =3,     /* protocol version error */
    status_proto_error          =4,     /* any other protocol error */
    status_unknown_op           =5,     /* unknown operation requested */
} p_status_t;

typedef struct {
  u_short   version;                     /* protocol version */
  u_short   id;                         /* requestor's uniq request id */
  u_short   op;                         /* operation requested */
  u_short   status;                     /* set on reply */
} P_HEADER,*P_HEADER_PTR;

typedef struct {
  P_HEADER h;
  u_int boot_time;
  u_int current_time;
  u_int user_mtime;                     /* time user information last changed */
  u_short l1;                           /* (int) (load*100) */
  u_short l5;
  u_short l15;
  u_short tot_users;                    /* total number of users logged in */
  u_short uniq_users;                   /* total number of uniq users */
  u_char  on_console;                   /* true if somone on console */
  u_char  reserved;                     /* future use, padding... */
} P_LB_RESPONSE, *P_LB_RESPONSE_PTR;
```

Figure 2.5: *lbnamed* protocol format.

2.2.2 Dispatcher-based Approach

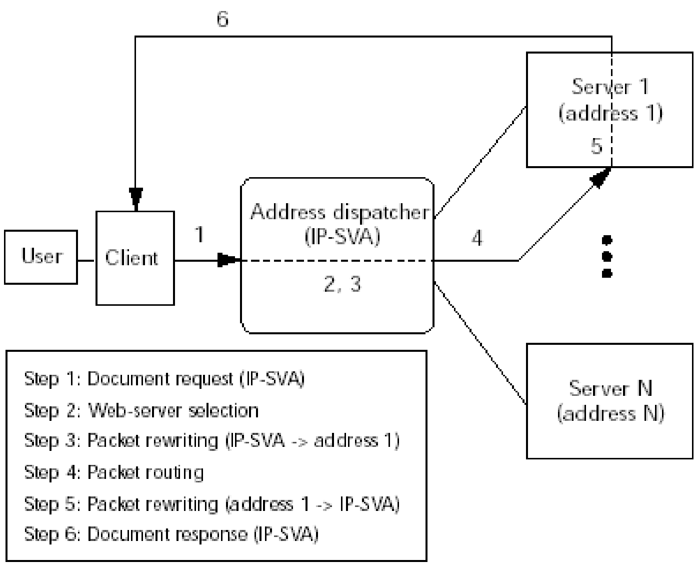
To centralize request scheduling and completely control client-request routing, a network component of the web server system acts as a dispatcher. Request routing among servers is transparent. Unlike DNS-based architectures, which deal with addresses at the URL level, the dispatcher has a single, virtual IP address (IP-SVA).

The dispatcher uniquely identifies each backend server in the system through a private address that can be at different protocol levels, depending on the architecture. We differentiate dispatcher-based architectures by routing mechanism — packet single-rewriting, packet double-rewriting, HTTP redirection, or server-based HTTP redirection [22].

Dispatcher-based architectures typically use simple algorithms to select the web server (for example, Round-Robin, server loading) to handle incoming requests. Simple algorithms help minimize request processing.

2.2.2.1 Packet Single-Rewriting

In some architectures, the dispatcher reroutes client-to-server packets by rewriting their IP address, such as in the basic TCP router mechanism. The web server cluster consists of a group of backend servers and a load balancer that acts as an IP address dispatcher. Figure 2.6 outlines the mechanism, in which address *i* is the IP address of the *i-th* web server.



Step 1: Document request (IP-SVA)
Step 2: Web-server selection
Step 3: Packet rewriting (IP-SVA -> address 1)
Step 4: Packet routing
Step 5: Packet rewriting (address 1 -> IP-SVA)
Step 6: Document response (IP-SVA)

Figure 2.6: Packet single-rewriting by the dispatcher.

All HTTP client requests reach the dispatcher because the IP-SVA is the only public address. The dispatcher selects a backend server for each HTTP request through a Round-Robin algorithm and forwards the packet by rewriting the destination IP address of each incoming packet. The dispatcher replaces its IP-SVA with the IP address of the selected server. Because a request consists of several IP packets, the dispatcher tracks the source IP address for every established TCP connection in an address table. The dispatcher can thereby route packets regarding the same connection to the same web server.

Furthermore, the web server must replace its IP address with the dispatcher's IP-SVA before sending the response packets to the client. Therefore, the client is not aware that its requests are handled by a hidden web server.

This approach provides high system availability because, when a backend server fails, its address can be removed from the dispatcher to prevent further request routing. Moreover, the dispatcher architecture can be combined with a DNS-based solution to scale from a LAN- to a WAN-distributed web system.

2.2.2.2 Packet Double-Rewriting

This mechanism also relies on a centralized dispatcher to schedule and control client requests but differs from packet single-rewriting in the source address modification of all packets between server and client. Packet double-rewriting is based on Network Address Translation mechanism published by the Internet Engineering Task Force, as shown in Fig. 2.7. The dispatcher receives a client request, selects the web server and modifies the IP header of each incoming packet, and also modifies the outgoing packets that compose the requested document.

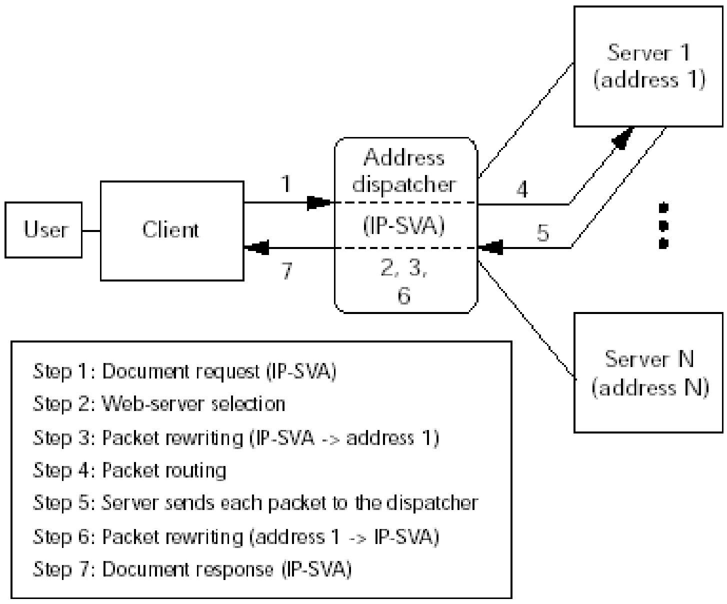

Server 1
(address 1)
Address
dispatcher
(IP-SVA)
2, 3,
6
User
Client
1
7
4
5
Server N
(address N)
Step 1: Document request (IP-SVA)
Step 2: Web-server selection
Step 3: Packet rewriting (IP-SVA -> address 1)
Step 4: Packet routing
Step 5: Server sends each packet to the dispatcher
Step 6: Packet rewriting (address 1 -> IP-SVA)
Step 7: Document response (IP-SVA)

Figure 2.7: Packet double-rewriting by the dispatcher.

2.2.2.3 HTTP Redirection

A centralized dispatcher receives all incoming requests and distributes them among the web server nodes through the HTTP redirection mechanism. The dispatcher redirects a request by specifying the appropriate status code [32] in the response, indicating in its header the server address where the client can get the desired document. Such redirection is largely transparent; at most, users might notice an increased response time. Unlike most dispatcher-based solutions, HTTP redirection does not require IP address modification of packets reaching or leaving the web server system. HTTP redirection can be implemented with two techniques.

Server-state-based dispatching: Used by the Distributed Server Groups architecture [33]. It adds new methods to HTTP protocol to administer the web system and exchange messages between the dispatcher and the servers. Since the dispatcher must be aware of the server loading, each server periodically reports the number of processes in its run queue and the number of received requests per second. The dispatcher then selects the least-loaded server, as shown in Fig. 2.8.

Location-based dispatching: Used by Cisco Systems' Distributed Director [8] appliance. It provides two dispatching modes. The first applies the DNS-based approach with client and server state information. The second use the HTTP redirection. The Distributed Director estimates a client's server proximity and the node availability with algorithms that apply to the DNS-based solution. Client requests are redirected to the server that is evaluated as most suitable for each request at a certain time.

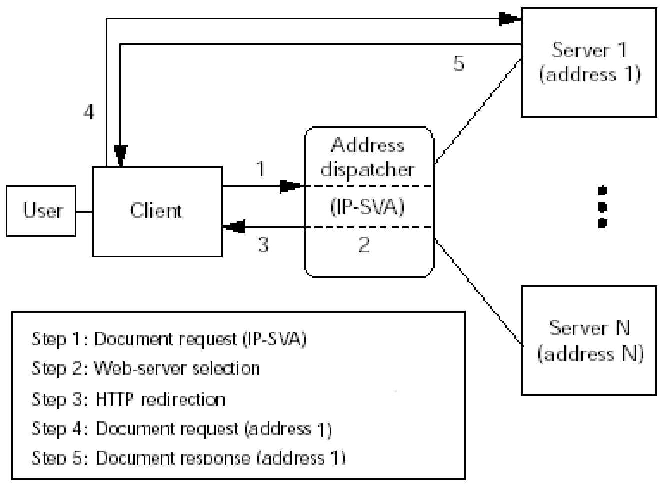

Server 1
(address 1)
5
4
Address
dispatcher
(IP-SVA)
1
3
2
User
Client
Server N
(address N)
Step 1: Document request (IP-SVA)
Step 2: Web-server selection
Step 3: HTTP redirection
Step 4: Document request (address 1)
Step 5: Document response (address 1)

Figure 2.8: HTTP redirection.

2.2.2.4 Server-based HTTP Redirection

The scalable server World Wide Web (SWEB) system and similar architectures [34] use a two-level distributed scheduler, as shown in Fig. 2.9. Client requests, initially assigned by the DNS to a web server, can be reassigned to another server via HTTP redirection. Figure 2.9 shows server 1 receiving the client request, then redirecting the request to server 2. The first level web server selected by the DNS can be prevented by the caching mechanism of the intermediate name servers.

Redirecting individual client connections is crucial to better load balancing at a fine granularity level. In most instances, however, it is preferable to combine client redirection with domain redirection [34].

The SWEB architecture uses a Round-Robin DNS policy as a first-level scheduler. In second-level scheduler, each web server redirects requests according to server selection that minimizes the client request's response time, a value estimated on the basis of server processing capabilities and Internet bandwidth/delay.

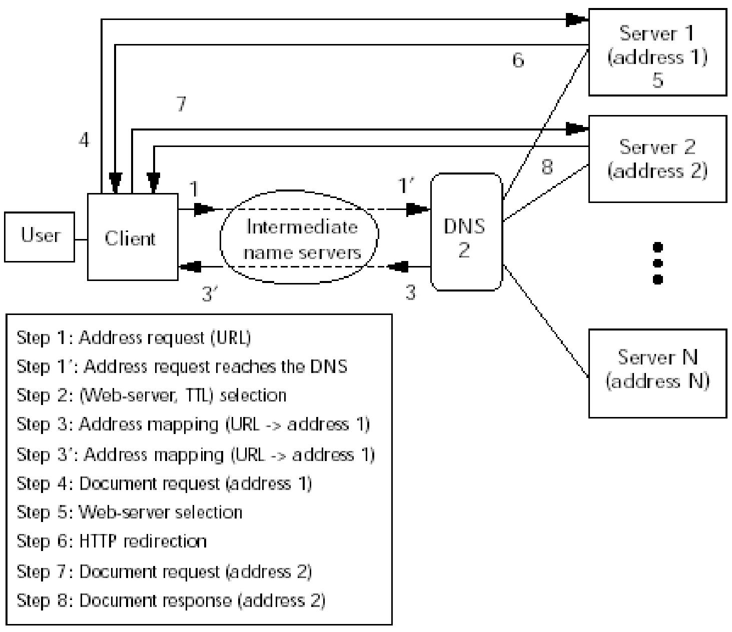

Server 1 (address 1) 5
Server 2 (address 2)
Server N (address N)
User
Client
Intermediate name servers
DNS 2
1
1′
3
3′
4
6
7
8
Step 1: Address request (URL)
Step 1′: Address request reaches the DNS
Step 2: (Web-server, TTL) selection
Step 3: Address mapping (URL -> address 1)
Step 3′: Address mapping (URL -> address 1)
Step 4: Document request (address 1)
Step 5: Web-server selection
Step 6: HTTP redirection
Step 7: Document request (address 2)
Step 8: Document response (address 2)

Figure 2.9: HTTP redirection by the server.

These mechanisms imply an overhead of intra-cluster communications, as every server must periodically transmit status information to the cluster DNS or other servers. But such cost is usually negligible as compared to the client-request-generated network traffic. To users, the main drawback of HTTP redirection is the increased response time. This is because each redirected request requires a new client-server connection.

2.3 Load Balancing Algorithms

The load balancing system scheduler makes decisions regarding which backend server to be assigned a new connection based on the load balancing algorithms [4,6,7]. Various algorithms available are addressed as follows.

2.3.1 Least Connection

With the least connections algorithm, the number of connections currently open on each backend server is measured in real-time. The backend server with the fewest current connections is considered to be the best choice for the next client connection request. This algorithm is the most self-regulating, with the fastest servers typically getting the most connections over time.

2.3.2 Round-Robin

With the Round-Robin algorithm, new connections are issued to each backend server in turn. That is, the first backend server in the group gets the first connection, the second backend server gets the next connection, followed by the third backend server, and so on. When all the backend servers in this group have received at least one connection, the process starts over with the first backend server.

2.3.3 Minimum Misses

The minimum misses algorithm is optimized for WAN-link load balancing. It uses IP address information in the client request to select a server. The specific IP address information used depends on the application.

For WAN-link load balancing, the client destination IP address is used. All requests for a specific IP destination address is sent to the same server. This algorithm is particularly useful in caching applications, helping to maximize successful cache hits. Best statistical load balancing is achieved when the client IP addresses are spread across a broad range of IP subnets.

For server load balancing, the client source IP address and backend server IP address are used. All requests from a specific client are sent to the same backend server. This algorithm is useful for applications where client information must be retained on the server between sessions. With this algorithm, backend server loading becomes most evenly balanced as the number of active clients with different source or destination addresses increases.

2.3.4 Hash

The hash algorithm uses IP address information in the client request to select a backend server. The specific IP address information used depends on the application. For WAN-link load balancing, the client destination IP address is used. All requests for a specific IP destination address will be sent to the same server. This is particularly useful for maximizing successful cache hits.

For server load balancing, the client IP address is used. All requests from a specific client will be sent to the same backend server. This option is useful for applications where client information must be retained between sessions.

When selecting a backend server, a mathematical hash of the relevant IP address information is used as an index into the list of currently available servers. Any given IP address information will always have the same hash result, providing natural persistence, as long as the backend server list is stable. However, if a server is added to or leaves the system, then a different backend server might be assigned to a subsequent session with the same IP address information even though the original server is still available. Open connections are not cleared.

The hash algorithm provides more distributed load balancing than minimum misses at any given instant. It should be used if the statistical load balancing achieved using minimum misses is not as optimal as desired. If the load balancing statistics with minimum misses indicate that one backend server is processing significantly more requests over time than other servers, consider using the hash algorithm.

2.3.5 Response Time

The response time algorithm uses backend server response time to assign sessions to servers. The response time between the servers and the load balancer is used as the weighting factor. The load balancer monitors and records the amount of time it takes for each backend server to reply to a health check to adjust the backend server weights. The weights are adjusted so they are inversely proportion to a moving average of response time. In such a scenario, a server with half the response time as another server will receive a weight twice as large.

2.3.6 Bandwidth

The bandwidth algorithm uses backend server octet counts to assign sessions to a server. The load balancer monitors the number of octets sent between the server and itself. Then, the backend server weights are adjusted so they are inversely proportion to the number of octets that the backend server processes during the last interval.

Backend servers that process more octets are considered to have less available bandwidth than those that have processed fewer octets. For example, the backend server that processes half the amount of octets over the last interval receives twice the weight of other backend servers. The higher the bandwidth used, the smaller the weight assigned to the server. Based on this weighting, the subsequent requests go to the backend server with the highest amount of free bandwidth. These weights are automatically assigned.

CHAPTER 3

LOAD BALANCING ALGORITHMS

In this chapter, we will discuss our proposed load balancing algorithms. Section 3.1 will describe the flexible server registration protocol which enables the backend servers to join and leave the load balancing system easily. Section 3.2 will describe the weighted distributed load balancing algorithm and how to define the initial weights for heterogeneous backend servers. Section 3.3 will describe the remaining capacity load balancing algorithm which can be used when the service-on-demand servers join into the load balancing system. Section 3.4 will describe the fuzzy decision load balancing algorithm which dispatches the client request based on the fuzzy decision mechanism.

3.1 Flexible Server Registration Protocol

In our proposed server load balancing system, it consists of a dispatcher and a number of backend servers that serve the same services. The backend servers must register their services to the load balancing system. The dispatcher will maintain the validity of the information of those servers. When a request comes from a client, the dispatcher redirects the client request to the appropriate backend server by the pre-defined algorithm.

3.1.1 Server Registration Protocol

William V. Wollman et al. [74] proposed a plug and play server load balancing architecture. Their proposed method is described as below. First, the backend server registers itself with the dispatcher. The dispatcher will acknowledge the registration of the backend server. Next, the server registers its services with the dispatcher. Once the backend server issues the service registration request, the dispatcher performs a health check on the service. If the health check success, the dispatcher responses the registration acknowledge to the backend server.

After the initial registration protocol, the backend servers will delivery service health status messages to the dispatcher by the heartbeat messages. When the dispatcher receives this heartbeat, it will send back an acknowledge message to the backend server. In addition, the dispatcher will also perform its own independent health check on the registration service. If a service health failure occurs, the dispatcher will automatically remove the backend server from the load balancing system. The message flows between the backend server and the dispatcher are shown in Fig. 3.1.

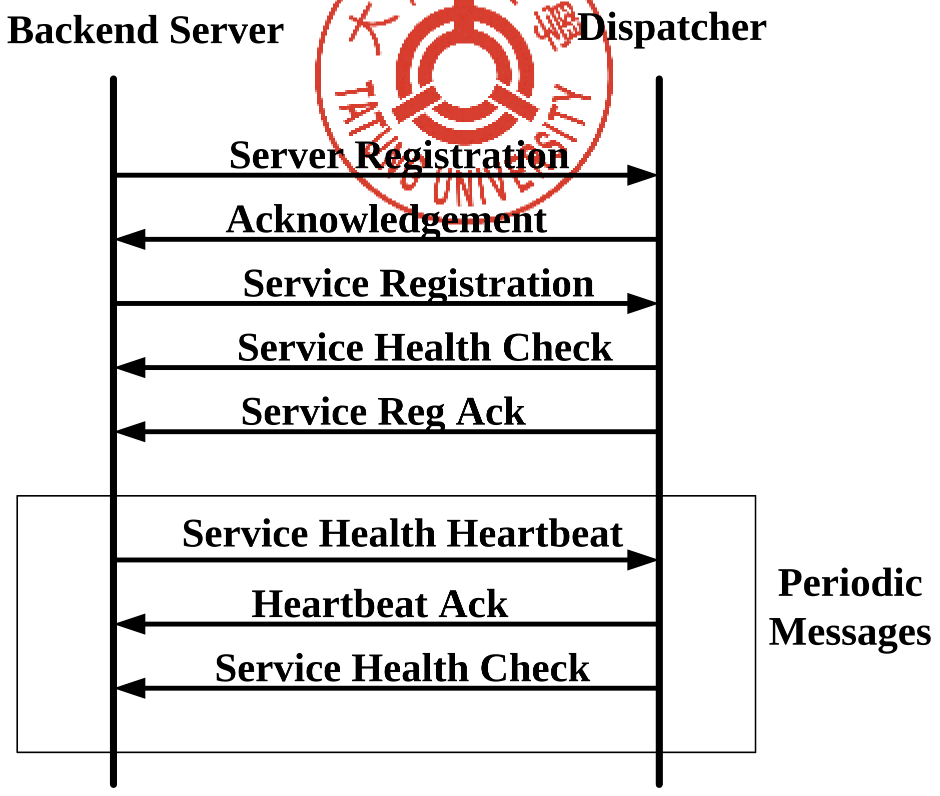


Figure 3.1: The message flows between dispatcher and backend server.

3.1.2 Flexible Registration Protocol

The goal of our proposed registration protocol is to simplify the registration flows. Our registration flow also checks the health status at the same time. The process is described as follows. First, if a new backend server is initialized, it will issue an HTTP request to itself, that is, *localhost*. In this step, we not only check the status of backend server, but also check the health of the service, ie, HTTP. In addition, the HTTP request also gets system information of the backend server, so the response messages will include the real time server loading. If the health check of the service failed, nothing will be done. If it can get an HTTP response, that means the HTTP service is available, the server will insert an entry into the central database. The entry includes the server's IP address, registration time, CPU idle percentage, available memory, current number of connection, etc.

The HTTP request to itself of each backend server is issued periodically. So the database always have the current server status information. The dispatcher will periodically query the database to get the most updated messages about the backend servers. In this protocol, we also check the status of database implicitly to avoid the database failure. Figure 3.2 shows the message flows for our proposed registration protocol.

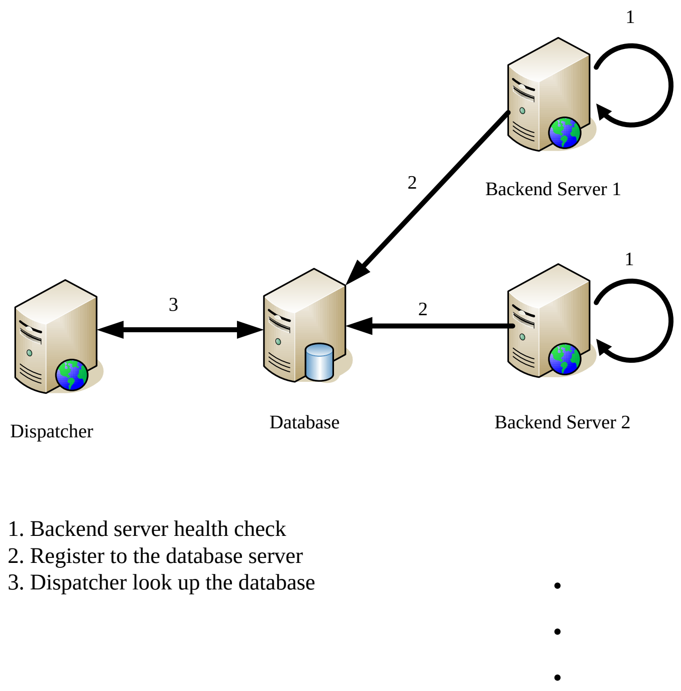

1
Backend Server 1
2
1
3
2
Dispatcher
Database
Backend Server 2
1. Backend server health check
2. Register to the database server
3. Dispatcher look up the database

Figure 3.2: Flexible registration protocol flows.

If one of the backend servers does not issue registration operation within a specific time, the dispatcher will remove that backend server from the load balancing system. There are three cases that the backend server will not register to the load balancing system. The first case is that the backend server is going to quit from the HTTP service. In this case, the backend server will stop registering to the database server. The old entry will be removed by the dispatcher in the next maintainance check. In the second case, the service daemon or the operating system of the backend server is crashed. The third case happens when the network traffic is jammed or the service daemon reaches to its maximum capacity. In the later two cases, the backend server can no longer provide the service to any new connection. Therefore, we need to remove it from the list of available servers to ensure that no new connection will be redirected to it.

Consider a load balancing system with N backend servers and one dispatcher as shown in Fig. 3.3. When a client wants to access a web page, it will issue a DNS query to find the IP address of the web server. The DNS server will reply with the IP address of the dispatcher instead of the address of the web server that provides the service. The client then issues an HTTP request to the dispatcher, marked as step 1. When the dispatcher receives the request, a redirection page will be sent back to the client, marked as step 2. The redirection page contains the IP address or domain name of the web server which is the most appropriate one to serve this request. The client then issues the HTTP request again to the real backend server, marked as step 3. The backend server will then serve the client request and transfer document directly to the client in the final step [20].

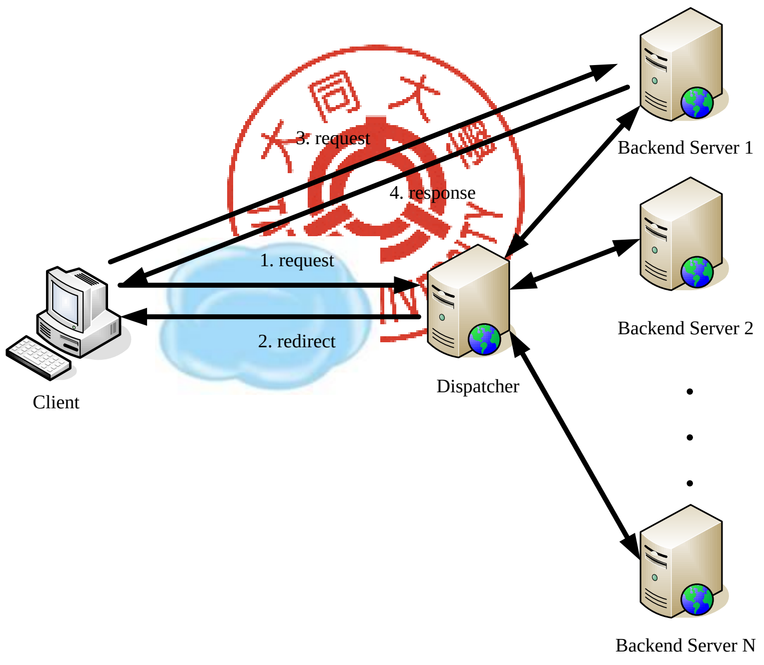


Figure 3.3: Intelligent forwarding mechanism.

3.1.3 Implementation of Flexible Registration Protocol

The backend server in our load balancing architecture can be added or removed at any time. We can register a new backend server and report the real time loading in just one step. That is, we can execute only one program on the backend server to register and report. In our proposed registration protocol, we can use some servers whose primary function is MAIL server, FTP server, or DNS server, etc. to act as the backend server prior the burst period and release them after the burst period. For example, during the course registration period, almost all the students are focus on the course registration, the other servers will have light loading at that time. When we activate the HTTP daemon and execute the registration program on those servers, they will join the load balancing architecture, and provide the HTTP services to the students. In this case, we can efficiently use the resource of all available servers.

If all the other servers already join the load balancing system, and the system is still experienced heavy loading, we can use the PCs in the PC room as the backend servers. In this situation, we may use the Konppix Linux distribution [75]. Knoppix is a great Linux tool for all skill levels. We can modify the Knoppix Live-CD, install the registration program into the Live-CD. When we insert the Live-CD and reboot that computer, the computer automatically becomes one of the backend servers. If we have a PC room with about 100 personal computers, we can boot them with Knoppix Live-CD, and we will have 100 more backend servers.

The registration program must check the network status, HTTP service, and the database server. To simplify the registration protocol, we use a script to do all the function. Each backend server use the *crond* to execute the registration program, that will ensure the server is active. The *crond* will issue a *wget* to execute a *PHP script*, which makes sure the HTTP service is alive. The entry in the *crond* is shown in Fig. 3.4.

In the *registration.php* script, it calculates the real time loading such as CPU, memory, number of connections, etc, and it inserts these information into the database server. This process will guarantee that the database server is running and the connection between HTTP server and database server is operational.

We retrieve the speed of CPU from the command *"cpuinfo"* which is under */proc* directory. To retrieve the CPU usage and memory utilization, the *"vmstat"* is used. We also use the *"netstat"* to get the number of connection for the backend server. These information will be sent to the database server during the registration process. The detail of *registration.php* is shown in Fig. 3.5.

```
wget -q --delete-after http://localhost/registration.php
```

Figure 3.4: Cron table for registration.

```
$hostaddr = exec('ifconfig eth0 | sed -n 2p | cut -b 21-35 ');
$cpu_idle = exec('vmstat | sed -n 3p | cut -b 74-76 ');
$mem =  exec('vmstat | sed -n 3p | cut -b 13-19 ');
$connect_num = exec('netstat -tn | wc -l | cut -b 1-5');
```

Figure 3.5: Part of the *registration.php* script to retrieve system information.

3.2 Weighted Distributed Load Balancing Algorithm

In this Section, we consider the heterogeneous backend servers in a load balancing system. Due to the different capacities of each backend server, we must define the weights for each backend server. The definition factors for each backend server can be considered as the maximum number of connections, drop rate, or response time. We will introduce some definitions for the weights, and compare the results of each definition. The weighted distributed load balancing algorithm is also introduced.

3.2.1 Definition of Capacity

Assume that there are three heterogeneous backend servers in the load balancing system, say S_1, S_2, and S_3. We define the capacities of these servers be C_1, C_2, and C_3, respectively. The capacities are the weights of these backend servers. Using the weights in our proposed weighted distributed load balancing algorithm, we can accomplish higher performance and fair response time for the clients.

In the definition of capacity, we first need to know the maximum number of connections for a single backend server. To define this, we also need to define a threshold that the drop rate is below acceptable value. Assume that the clients issue M requests to the backend server, the response time of the server will vary in length randomly. The response time R for each request is defined as R_i. We partition the possible values into n disjoint intervals, $\{[T_0=0,T_1), [T_1,T_2),\ldots, [T_{n-1},T_n=\infty)\}$, where $T_0<T_1<\ldots<T_n$, and R_i fall in the ith interval $[T_{i-1},T_i)$. The number of connection for $[T_{i-1},T_i)$ is m_i, where $m_1+m_2+\ldots+m_n=M$. For user perceived latency, we define the maximum response time R_{max} as the threshold which fall in the jth interval $[T_{j-1},T_j)$. For each request, if the response time is higher than the R_{max}, we then consider that the request was dropped. We define the drop rate when the R_{max} fall in the jth interval as in Eq. 3.1. If the drop rate is

below certain percentage, we can obtain the maximum number of connections or the capacity for that server by Eq. 3.2.

$$Drop\ \ rate\ \ D = \frac{\sum_{i=j}^{n} m_i}{M}, \text{where the max. threshold fall in the } j\text{th interval} \tag{3.1}$$

$$Capacity\ \ C = \sum_{i=1}^{j-1} m_i, \text{where the max. threshold fall in the } j\text{th interval} \tag{3.2}$$

For example, if the total number of requests is 500 (M=500). The response time have 10 disjoint intervals {[0,0.25), [0.25,0.5), [0.5,0.75), [0.75,1), [1,1.25), [1.25,1.5), [1.5,1.75), [1.75,2), [2,2.25), [2.25,∞)}, and the number of responses for each interval is {120, 95, 75, 65, 45, 30, 25, 22, 15, 8}. If we define the threshold as 2 seconds, which means that R_{max}=2, then the drop rate D is (15+8)/500 or 4.6% and the capacity C is (120+95+75+65+45+30+25+22) or 477.

3.2.2 Capacity Enhancement for Fair Response Time

In Section 3.2.1, we define the capacities C_1, C_2, and C_3 for the three servers S_1, S_2, and S_3. In the experimental results shown in Section 4.1, we find that the more powerful server S_3 will serve more client requests because we define higher capacity for that server. But in the figure of response time analysis, the S_3, which serves more client requests, still response faster than others. It is not fair in the user perceive latency. We need to consider both the drop rate and response time when defined the capacity. Using the same environment as above, the drop rate definition is still the same, but we want to modify the definition of the capacity. Rather than only consider the drop rate, we use the response time as another factor to calculate the capacity. Assume that we define the acceptable response time R_{min} as the threshold, which means the capacity definition must under the minimum threshold rather than the maximum threshold R_{max}. If the R_{min} falls in the kth interval $[T_{k-1},T_k)$, where $k < j$. We can re-define the capacity as in Eq. 3.3.

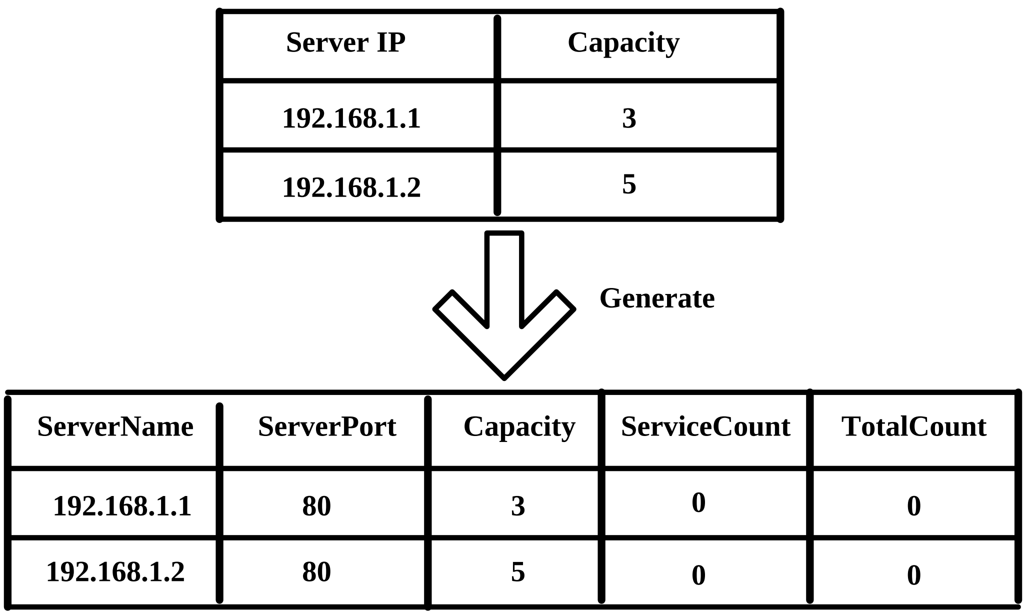
$$Capacity \;\; C = \sum_{i=1}^{k-1} m_i \text{, where the min. threshold fall in the } k\text{th interval} \qquad (3.3)$$

For the above example, we define the threshold as 1 second, which means that R_{min}=1, then the capacity C is (120+95+75+65) or 355. Under this definition, the experimental results in Section 4.1 show that the average response time for all the servers is almost the same. The algorithm for the drop rate and capacity are given as

```
for each time intervals $\{[T_0=0,T_1), [T_1,T_2),\ldots, [T_{n-1},T_n=\infty)\}$
    where the number of connection for $[T_{i-1},T_i)$ is $m_i$
    and $m_1+m_2+\ldots+m_n=M$
the maximum threshold $R_{max}$ is fall in the $j$th interval $[T_{j-1},T_j)$
    Drop rate $D=(m_j+m_{j+1}+\ldots+m_n)/M$
the minimum threshold $R_{min}$ is fall in the $k$th interval $[T_{k-1},T_k)$, where $k < j$
    Capacity $C= m_1+m_2+\ldots+m_{k-1}$
```

3.2.3 Weighted Distributing Load Balancing Algorithm

In this section, we will show the weighted distributing algorithm used in the load balancing system. In our load balancing system, each backend server has different capacity. The capacity is the weights of the server. After obtaining the capacity for each server, a *serverlist* table is generated. Figure 3.6 shows an example.



Server IP	Capacity
192.168.1.1	3
192.168.1.2	5

ServerName	ServerPort	Capacity	ServiceCount	TotalCount
192.168.1.1	80	3	0	0
192.168.1.2	80	5	0	0

Figure 3.6: Example of *serverlist* table.

The IP address of individual backend server is stored in *ServerName* field. The *ServerPort* field specifies the TCP port number of individual backend server. The *ServiceCount* field stores the current number of connection which was dispatched to that backend server, and will be reset to 0 when all entries with *ServiceCount* equals to *capacity*. The *TotalCount* field stores the total dispatched number of each backend server.

When a client issues a request to the dispatcher, the dispatcher retrieves the entry from the *serverlist* table from the database server. We use a simple but elegant method to choose the most appropriate entry from the *serverlist* table. The method is to choose the entry with the largest remaining capacity, that is, the entry with largest difference between *capacity* and *ServiceCount*. If all the remaining capacities are same, the dispatcher will return the entry which registered first in the table. Then the *ServiceCount* of the chosen entry will be incremented by one. When all the values of *ServiceCount* are equal to *capacity*, the dispatcher will add *ServiceCount* to *TotalCount* and reset *ServiceCount* to zero.

3.3 Remaining Capacity Load Balancing Algorithm

Although load balancing system can efficiently distribute the client requests, the cost of these backend servers are the main issues. In some cases, the backend servers are not always experienced heavy loading. They have heavy request traffic only in a certain time period. Some web servers, such as course registration or ticket reservation systems, have burst requests only several times a year. In these cases, we can predict the start time of burst requests. Thus we can prepare some service-on-demand servers as the backend servers before that time. The flexible server registration protocol, described in Section 3.1, allows those service-on-demand servers to register and coordinate their services with the load balancing system. This architecture uses the registration messages to automatically configure a backend server to support the services. In addition, the registration protocol also checks the status of the backend server, so that it can report the loading information of the backend servers. In our architecture, a new server can register to the load balancing system while needed and withdraw themselves from the system after the burst period.

In this proposed algorithm, there are three heterogeneous servers S_1, S_2, and S_3 in the system. Each backend server periodically reports its current load information to the dispatcher. The load balancing algorithm calculates the remaining capacities as R_1, R_2, and R_3. And then selects the best server i, where $R_i = max(R_1,R_2,R_3)$. If a request T comes to the dispatcher, the IP address of backend server S_i with the maximum remaining capacity R_i will be sent back to the client. We assume that the maximum capacity needed by the request T is T_k. If the maximum remaining capacity R_i is less than the request T_k, it means that all the backend servers do not have enough remaining capacity to serve the request task T_k. The dispatcher drops this request T_k and return a server busy page to the client. When the backend servers finish their previously assigned requests, they will then have capacity to serve future requests.

Assume that the CPU clock-rate of the servers are C_1, C_2, and C_3. The backend server i reports the current load information parameters as $\{c_i, m_i, n_i\}$, where c_i is CPU idle percentage, m_i is available memory, and n_i is current number of connections. We define the three membership functions as in Figures 3.7 to 3.9. Then the remaining capacity R_i of server S_i can be obtained by the following equation.

$$R_i = \alpha * f_1(C_i) * c_i + \beta * f_2(m_i) + \gamma * f_3(n_i) \qquad (3.4)$$

where α, β, γ are weights and $\alpha + \beta + \gamma = 1$. In our simulations, we find that the CPU has the significant impact for server loading. In the simulation presented in Section 4.2, we use $\alpha = 0.4$, $\beta = 0.3$, $\gamma = 0.3$ as weights.

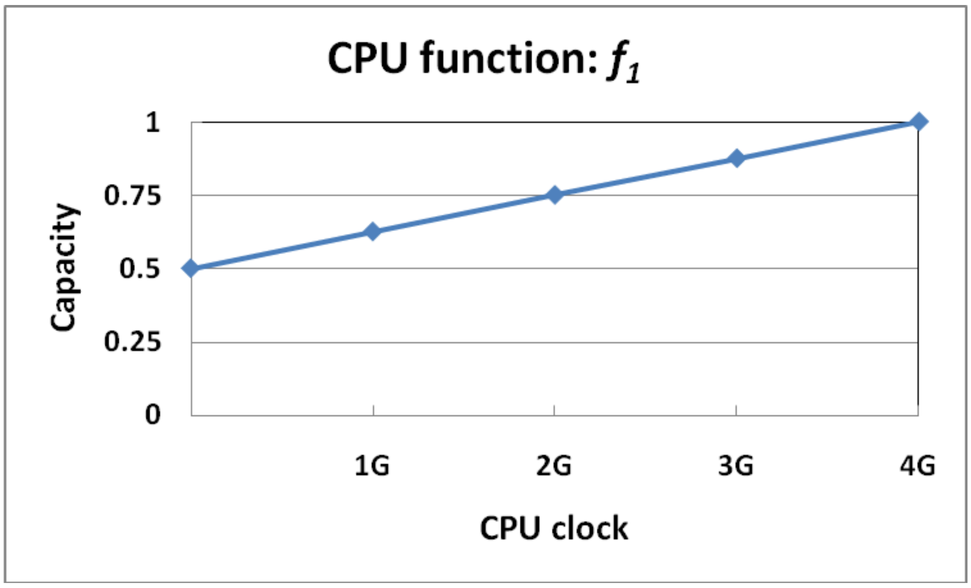


Figure 3.7: Membership function for CPU.

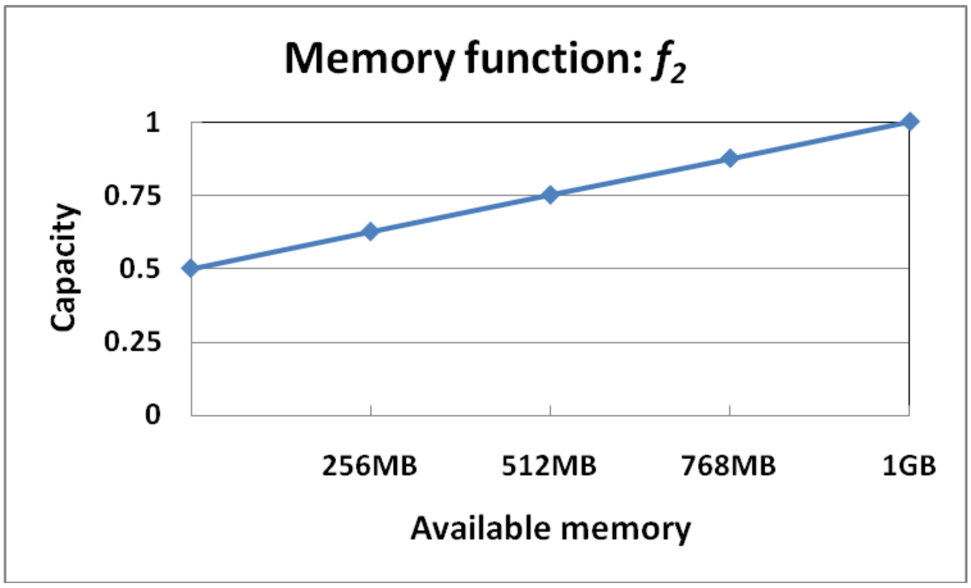


Figure 3.8: Membership function for Memory.

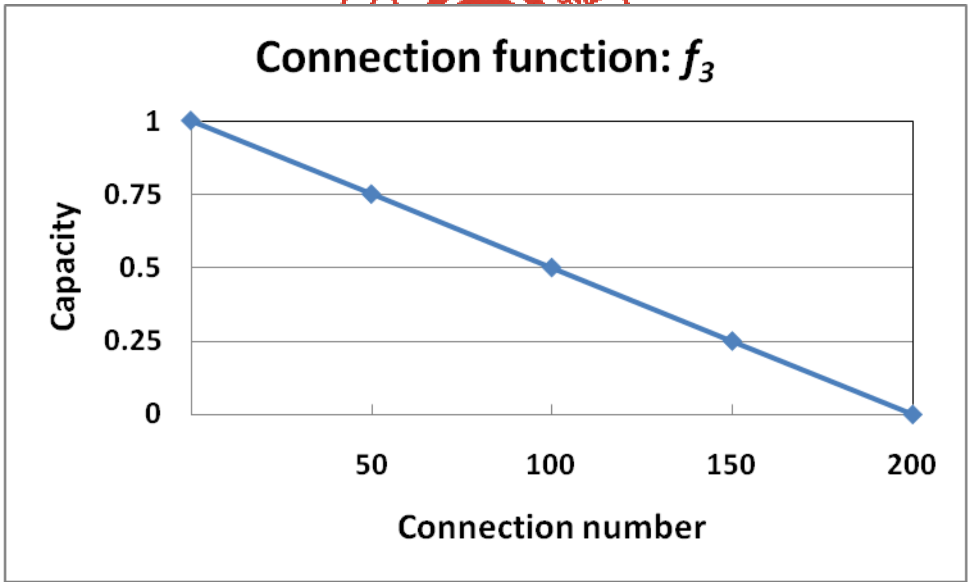


Figure 3.9: Membership function for Connection.

The pseudo codes for the proposed algorithm are as follows:

```
Report Phase:
1.  calculate the three parameters $\{c_i,m_i,n_i\}$ periodically;
2.  report $(c_i,m_i,n_i)$ for each backend server periodically;
```

```
Decision Phase:
1.  $R_i = \alpha * f_1(C_i) * c_i + \beta * f_2(m_i) + \gamma * f_3(n_i)$, for each backend server;
2.  $R_i = max(R_1,R_2,R_3)$;
3.  if (Capacity($R_i$)) > Require($T_k$)
        send($T_k$,$S_i$);
    else
        drop($T_k$);
```

3.4 Fuzzy Decision Load Balancing Algorithm

3.4.1 Feature Selection

The proposed approach of this section is based on the features of backend servers. So, a set of the effective features of the backend server should be selected before the fuzzy decision. Some proposed features were selected to create the feature set in our proposed approach.

3.4.1.1 CPU Idle Percentage

The CPU loading of the backend server is one factor that influences the performance of the load balancing system. The more loads the backend server has, the less CPU idle percentage it has. Thereafter, the CPU idle percentage is our first feature in the fuzzy decision mechanism. We define the CPU idle percentage f_{cpu} as

$$f_{cpu} = the\ \ idle\ \ percetage\ \ of\ \ t\,h\,e\,C\,P\,U \tag{3.5}$$

A larger f_{cpu} value means the higher possibility of available CPU resources that the backend server has. Thus, we can define the membership function for f_{cpu} in Fig. 3.10. Here, the High CPU Idle means the membership function for a high CPU idle percentage, the Medium CPU Idle means the membership function for a medium CPU idle percentage, and the Low CPU Idle means the membership function for a low CPU idle percentage.

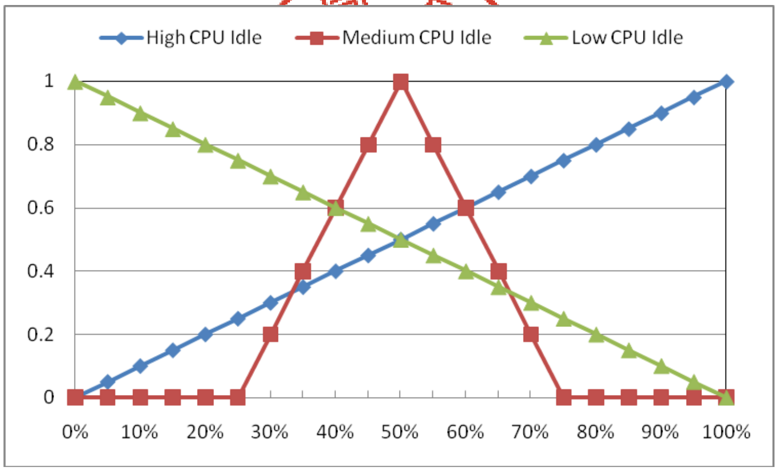


Figure 3.10: Membership function for CPU idle percentage.

3.4.1.2 Available Memory Percentage

The memory usage is another important factor that determines the loads of the backend server. The more memory the backend server uses, the less capacity the backend server has. However, the backend servers may have different memory installed. Thus, we use the percentage of available memory as the feature in our fuzzy decision mechanism. Assuming that the total memory of the backend server is M_{total}, and the available memory of the backend server is M_{avail}, then we define the available memory percentage f_{mem} as

$$f_{mem} = \frac{M_{ava}}{M_{tot}} \tag{3.6}$$

A larger f_{mem} value means the higher possibility of available memory that the backend server has. Thus, we can define the membership function for f_{mem} as shown inFig. 3.11. Here, the High Available Memory means the membership function for a high available memory percentage, the Medium Available Memory means the membership function for a medium available memory percentage, and the Low Available Memory means the membership function for a low available memory percentage.

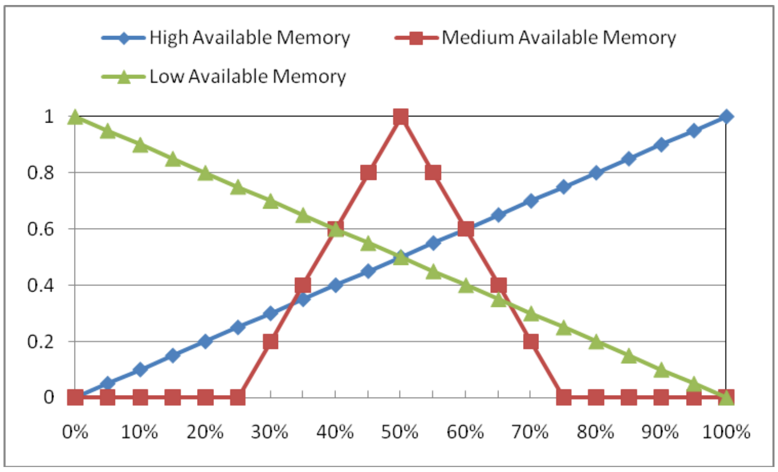


Figure 3.11: Membership function for available memory percentage.

3.4.1.3 Available Connection Percentage

When the current number of connections increases, the performance of the backend server tends to decrease. Thus, the current number of connections is the third factor of the load balancing system. Assuming that the maximum number of connections for the backend server is C_{max}, and the current number of connections is C_{cur}, then we define the percentage of available connection f_{con} as

$$f_{con} = \frac{C_{max} - C_{cur}}{C_{max}} \tag{3.7}$$

A larger f_{con} value means the higher possibility of available connections that the backend server has. Thus, we can define the membership function for f_{con} as shown inFig. 3.12. Here, the High Available Connection means the membership function for a high available connection percentage, the Medium Available Connection means the membership function for a medium available connection percentage, and the Low Available Connection means the membership function for a low available connection percentage.

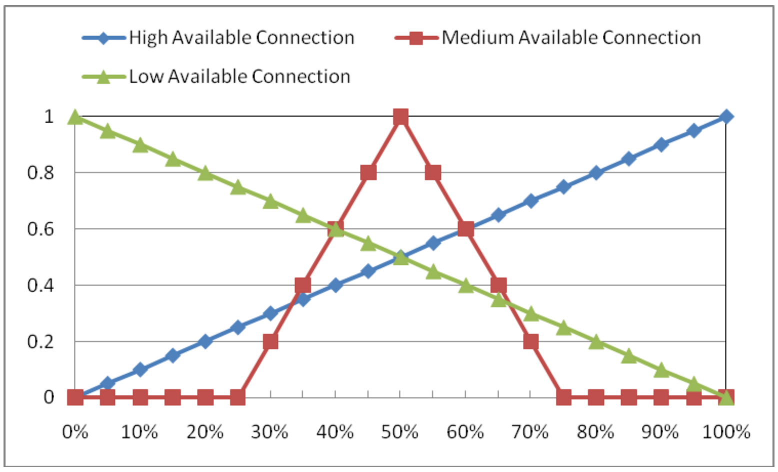


Figure 3.12: Membership function for available connection percentage.

3.4.2 Fuzzy Decision Load Balancing Algorithm

In this section, the proposed load balancing mechanism is introduced, which employs the fuzzy decision to decide the appropriate backend server. A fuzzy based approach consists of three steps, including the fuzzification, the rule evaluation, and the defuzzification, as shown in Fig. 3.13 [76].

3.4.2.1 Fuzzification

Based on the membership functions presented in Section 3.4.1, the fuzzy degree can be obtained according to the crisp values of the selected features. In order to describe the fuzzy decision mechanism, we use the following example. The given CPU idle percentage f_{cpu}, the available memory percentage f_{mem}, and the available connection percentage f_{con} are as following.

$$
\begin{aligned}
f_{cpu} &= 0.72 \\
f_{mem} &= 0.65 \\
f_{con} &= 0.55
\end{aligned}
$$

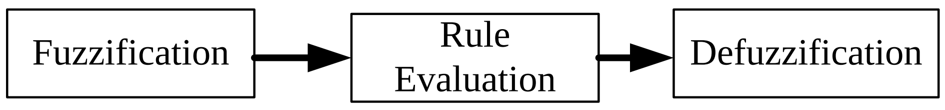


Figure 3.13: Fuzzy decision steps.

The degree of membership function can be determined from Figures 3.10 to 3.12. Hence, we can obtain the degree of High CPU idle percentage (HCPU), Medium CPU idle percentage (MCPU), Low CPU idle percentage (LCPU), High available memory percentage (HMEM), Medium available memory percentage (MMEM), Low available memory percentage (LMEM), High available connection percentage (HCON), Medium available connection percentage (MCON), and Low available connection percentage (LCON). These values are shown as below.

$$HCPU=0.72,\ \ MCPU=0.12,\ \ LCPU=0.28$$

$$HMEM=0.65,\ MMEM=0.4,\ \ LMEM=0.35$$

$$HCON=0.55,\ \ MCON=0.8,\ \ LCON=0.45$$

3.4.2.2 Rule Evaluation

The main purpose of rule evaluation is to apply the fuzzy values to the rule base for obtaining the fuzzy decision values. Table 3.1 shows the rule base, in which the influenced rules are illustrated. *Y*, *PY*, *PN*, and *N* indicate the *Yes*, *Probably Yes*, *Probably No*, and *No*, respectively, corresponding to the load balancing decision. The degree of membership can be assigned the minimum, maximum, or average of the degree of membership of the rules.

Table 3.1: Load balancing decision rule base.

CPU idle	*Memory available*	*Connection available*	*Decision*
HCPU	HMEM	HCON	Y
HCPU	HMEM	MCON	Y
HCPU	HMEM	LCON	PY
HCPU	MMEM	HCON	Y
HCPU	MMEM	MCON	Y
HCPU	MMEM	LCON	PY
HCPU	LMEM	HCON	PY
HCPU	LMEM	MCON	PY
HCPU	LMEM	LCON	PN
MCPU	HMEM	HCON	PY
MCPU	HMEM	MCON	PY
MCPU	HMEM	LCON	PY
MCPU	MMEM	HCON	PY
MCPU	MMEM	MCON	PN
MCPU	MMEM	LCON	PN
MCPU	LMEM	HCON	PY
MCPU	LMEM	MCON	PN
MCPU	LMEM	LCON	PN
LCPU	HMEM	HCON	PY
LCPU	HMEM	MCON	PN
LCPU	HMEM	LCON	PN
LCPU	MMEM	HCON	PN
LCPU	MMEM	MCON	N
LCPU	MMEM	LCON	N
LCPU	LMEM	HCON	PN
LCPU	LMEM	MCON	N
LCPU	LMEM	LCON	N

Since each factor has three membership functions, the combinations have 27 cases in total. Each load balancing decision in Tale 3.1 has a decision among *Y*, *PY*, *PN*, and *N*. The fuzzy values are used to evaluate rules for obtaining *Fuzzy Decision Values (FDV)* by assigning the minimum of the degree of membership of the rules. In this way, the decision and the corresponding *FDV* can be precisely determined, as shown in Table 3.2.

Table 3.2: Fuzzy decision values.

Rules	***FDV***
Rule(HCPU,HMEM,HCON)=Y	minimum(HCPU=0.72,HMEM=0.65,HCON=0.55)=0.55
Rule(HCPU,HMEM,MCON)=Y	minimum(HCPU=0.72,HMEM=0.65,MCON=0.8)=0.65
Rule(HCPU,HMEM,LCON)=PY	minimum(HCPU=0.72,HMEM=0.65,LCON=0.45)=0.45
Rule(HCPU,MMEM,HCON)=Y	minimum(HCPU=0.72,MMEM=0.4,HCON=0.55)=0.4
Rule(HCPU,MMEM,MCON)=PY	minimum(HCPU=0.72,MMEM=0.4,MCON=0.8)=0.4
Rule(HCPU,MMEM,LCON)=PY	minimum(HCPU=0.72,MMEM=0.4,LCON=0.45)=0.4
Rule(HCPU,LMEM,HCON)=PY	minimum(HCPU=0.72,LMEM=0.35,HCON=0.55)=0.35
Rule(HCPU,LMEM,MCON)=PY	minimum(HCPU=0.72,LMEM=0.35,MCON=0.8)=0.35
Rule(HCPU,LMEM,LCON)=PN	minimum(HCPU=0.72,LMEM=0.35,LCON=0.45)=0.35
Rule(MCPU,HMEM,HCON)=Y	minimum(MCPU=0.12,HMEM=0.65,HCON=0.55)=0.12
Rule(MCPU,HMEM,MCON)=PY	minimum(MCPU=0.12,HMEM=0.65,MCON=0.8)=0.12
Rule(MCPU,HMEM,LCON)=PY	minimum(MCPU=0.12,HMEM=0.65,LCON=0.45)=0.12
Rule(MCPU,MMEM,HCON)=PY	minimum(MCPU=0.12,MMEM=0.4,HCON=0.55)=0.12
Rule(MCPU,MMEM,MCON)=PN	minimum(MCPU=0.12,MMEM=0.4,MCON=0.8)=0.12
Rule(MCPU,MMEM,LCON)=PN	minimum(MCPU=0.12,MMEM=0.4,LCON=0.45)=0.12
Rule(MCPU,LMEM,HCON)=PY	minimum(MCPU=0.12,LMEM=0.35,HCON=0.55)=0.12
Rule(MCPU,LMEM,MCON)=PN	minimum(MCPU=0.12,LMEM=0.35,MCON=0.8)=0.12
Rule(MCPU,LMEM,LCON)=PN	minimum(MCPU=0.12,LMEM=0.35,LCON=0.45)=0.12
Rule(LCPU,HMEM,HCON)=PY	minimum(LCPU=0.28,HMEM=0.65,HCON=0.55)=0.28
Rule(LCPU,HMEM,MCON)=PN	minimum(LCPU=0.28,HMEM=0.65,MCON=0.8)=0.28
Rule(LCPU,HMEM,LCON)=PN	minimum(LCPU=0.28,HMEM=0.65,LCON=0.45)=0.28
Rule(LCPU,MMEM,HCON)=PN	minimum(LCPU=0.28,MMEM=0.4,HCON=0.55)=0.28
Rule(LCPU,MMEM,MCON)=N	minimum(LCPU=0.28,MMEM=0.4,MCON=0.8)=0.28
Rule(LCPU,MMEM,LCON)=N	minimum(LCPU=0.28,MMEM=0.4,LCON=0.45)=0.28
Rule(LCPU,LMEM,HCON)=PN	minimum(LCPU=0.28,LMEM=0.35,HCON=0.55)=0.28
Rule(LCPU,LMEM,MCON)=N	minimum(LCPU=0.28,LMEM=0.35,MCON=0.8)=0.28
Rule(LCPU,LMEM,LCON)=N	minimum(LCPU=0.28,LMEM=0.35,LCON=0.45)=0.28

From Table 3.2, it is shown that the fuzzy decisions have more than one value for the degree of membership. Generally, the minimum, maximum, or average of the membership degree can be used to obtain the final fuzzy degree. Here, the minimum rule evaluation is adopted to obtain the four *Fuzzy Degrees (FD)* of *Y*, *PY*, *PN*, and *N*.

$FD(Y) = \min(0.55,0.65,0.4,0.4)=0.4$

$FD(PY) = \min(0.45,0.4,0.35,0.35,0.12,0.12,0.12,0.12,0.12,0.28)=0.12$

$FD(PN) = \min(0.35,0.12,0.12,0.12,0.12,0.28,0.28,0.28,0.28)=0.12$

$FD(N) = \min(0.28,0.28,0.28,0.28)=0.28$

3.4.2.3 Defuzzification

In the defuzzification step, a set of weightings are assigned to the four truth values (*Y*, *PY*, *PN*, *N*). For instance, the four weightings may be assigned with the weight of *0.4*, *0.3*, *0.2*, and *0.1*, respectively. Therefore, the *Crisp Value (CV)* can be determined based on the *FD* weightings and the degree of the membership. The CV is calculated by Eq. 3.5.

$$CV = \frac{\sum_i FD(i) * w(i)}{\sum_i w(i)} \tag{3.8}$$

where $FD(i)$ and $w(i)$ represent the degree of membership and the weights respectively, in which $FD(i)$ belongs to *(Y, PY, PN, N)*. In the example considered above, the *CV* is obtained as

$$CV = \frac{0.4*0.4+0.12*0.3+0.12*0.2+0.28*0.1}{0.4+0.3+0.2+0.1} = 0.248$$

After the *CV* has been obtained, the backend server with the highest *CV* is the most appropriate one to serve the client request.

CHAPTER 4

EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS

4.1 Weighted Distributed Load Balancing Algorithm

In our experimental environment, the load balancing system consists of a dispatcher, a central database server, and a number of heterogeneous backend servers. A log server is installed to keep track of the transaction log between servers and clients. The log server is used to check whether the requests are distributed evenly, and we can decide to add new backend server if needed. In other words, we must make an effort to achieve the maximum performance with existing servers by more sophisticated load balancing algorithm before adding extra backend servers.

The operations of the load balancing system are described as follows. First, backend servers register to the database. The database server will generate a *serverlist* table on the database. When client issues request to dispatcher, the dispatcher looks up the *serverlist* table to obtain the IP address of the most appropriate backend server. Then the dispatcher returns an HTML document with HTTP header redirection to notify the IP address of backend server to the client. After client receives this document, it issues the request to the appropriate backend server. The assigned server serves the client requests from now on until the transaction finished.

To verify our proposed system can be applied in the heterogeneous system, we use three servers with different computational power as the backend servers. The servers are named S_1, S_2, and S_3. Table 4.1 shows the CPU speed of the three backend servers and Fig. 4.1 shows the maximum numbers of connection per second for each backend server.

Table 4.1: CPU speed for the three backend servers.

Server name	**CPU Speed**
S_1	***1.0G Hz***
S_2	***1.6G Hz***
S_3	***2.6G Hz***

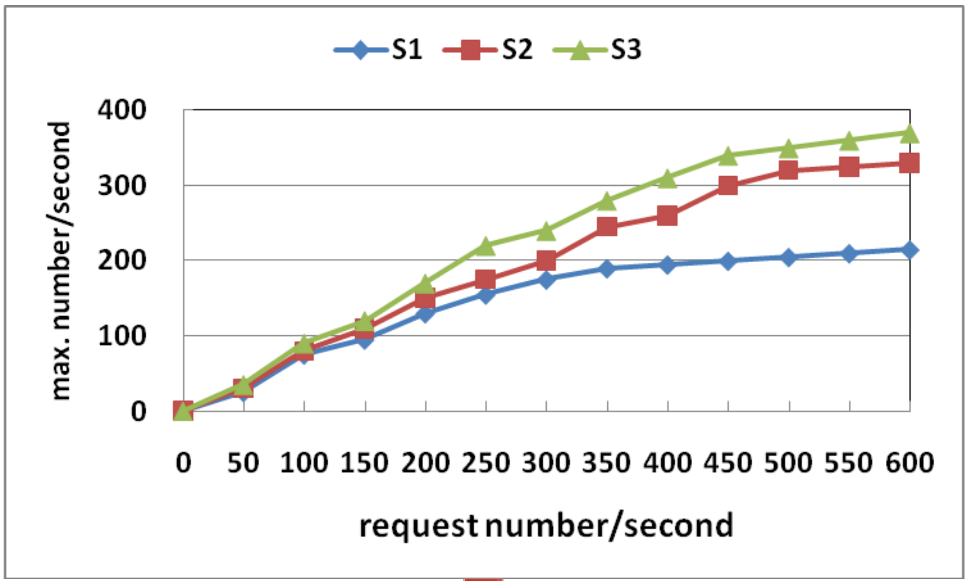


Figure 4.1: Maximum number of connections per second.

Figures 4.2 and 4.3 show the drop rate and the average response time of these three backend servers with different abilities. On Figures 4.1 to 4.3, we can observe that the high-end backend server, say S_3, can handle more client requests at the same time. Furthermore, the high-end backend server takes little time to response the same number of requests. Based on the experimental results above, we got some conclusions in Table 4.2.

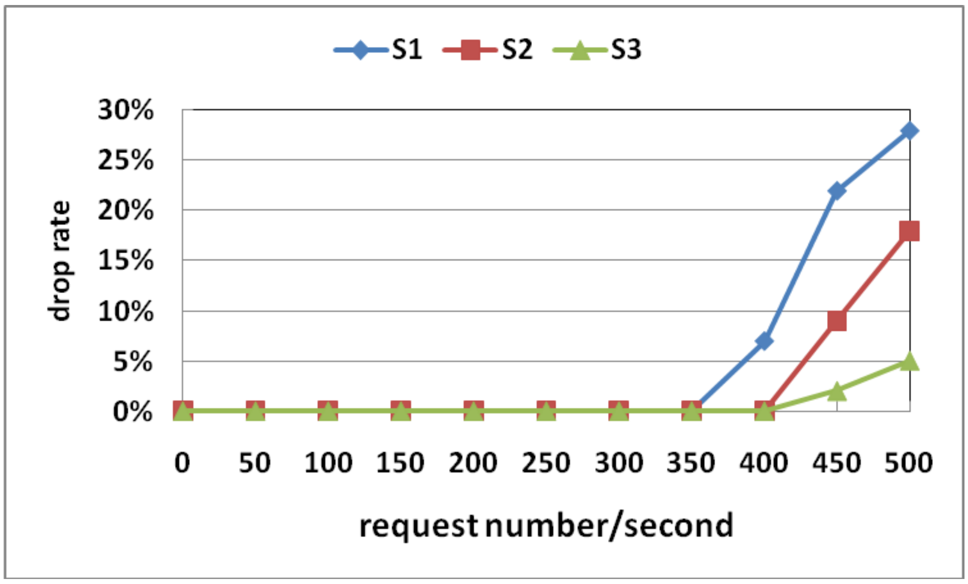


Figure 4.2: Comparison of drop rate for each backend servers.

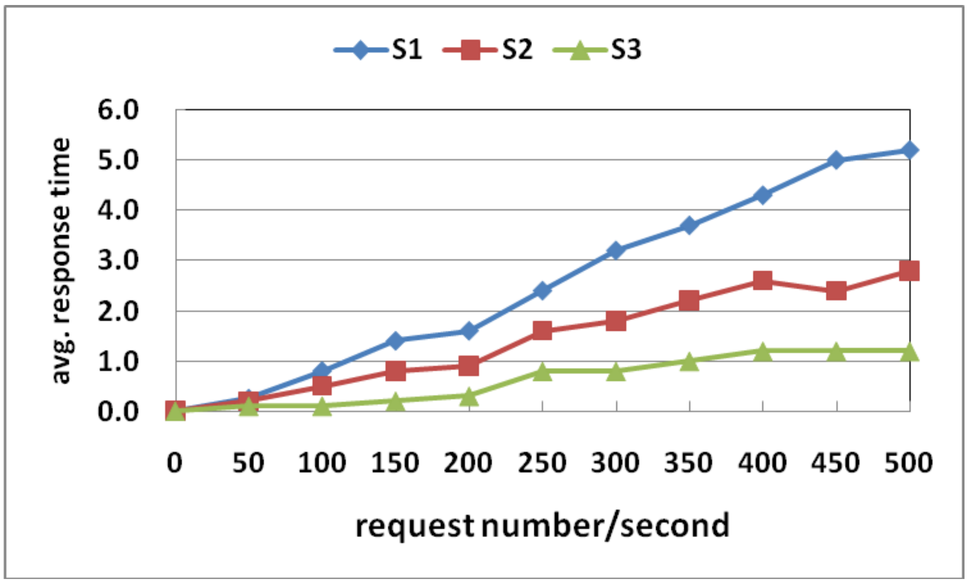


Figure 4.3: Comparison of average response time for each backend servers.

Table 4.2: The max. number of connections for each backend servers.

Server	**CPU**	**Max. connection number** **- Drop rate below 5%**	**Max. connection number** **- Drop rate below 5%** **- Avg. response time below 1 second**
S_1	***1 GHz***	**360**	**120**
S_2	***1.6 GHz***	**440**	**200**
S_3	***2.6 GHz***	**520**	**340**

The next experiment needs to set the capacities for the heterogeneous backend servers. The capacities derived from maximum request numbers with less than 5% drop rate in Table 4.2 is shown in Table 4.3.

Figures 4.4 and 4.5 show the experimental results when the capacity ratio of S_1:S_2:S_3 is *9:11:13*. The highest capacity server S_3 can serve more request than the others, as shown in Fig. 4.4. Although S_3 can server more requests, the average response time is still shorter than S_1 and S_2. In fact, it is unfair for users that connect to our load balancing system in first come first serve sequence. The reason is that we define the capacity values only considering the drop rate.

Table 4.3: Capacities of each backend server with drop rate below 5%.

Server	**Max. request numbers** **- Drop rate below 5%**	**Capacity**
S_1	**360**	**9**
S_2	**440**	**11**
S_3	**520**	**13**

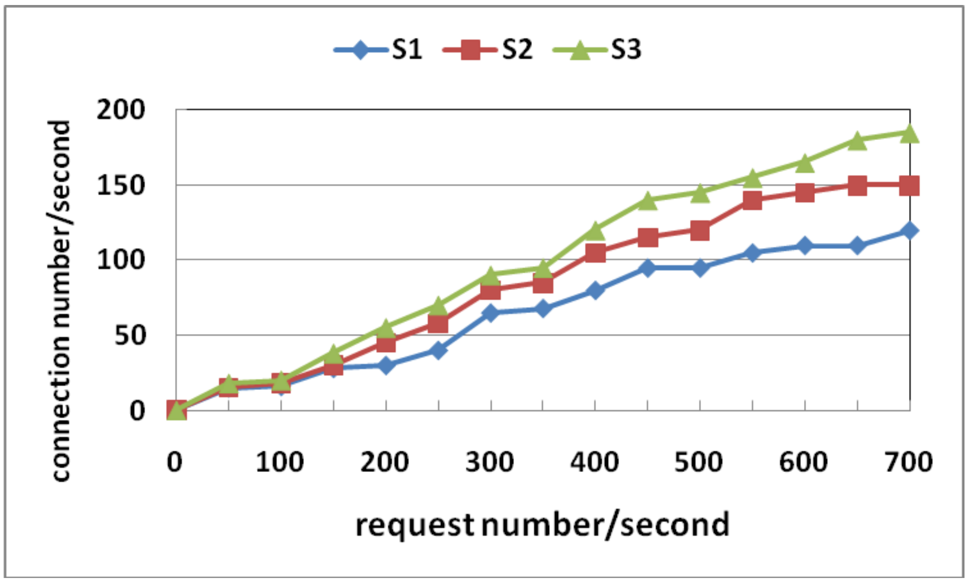


Figure 4.4: Comparison of connection numbers with capacity ratio 9:11:13.

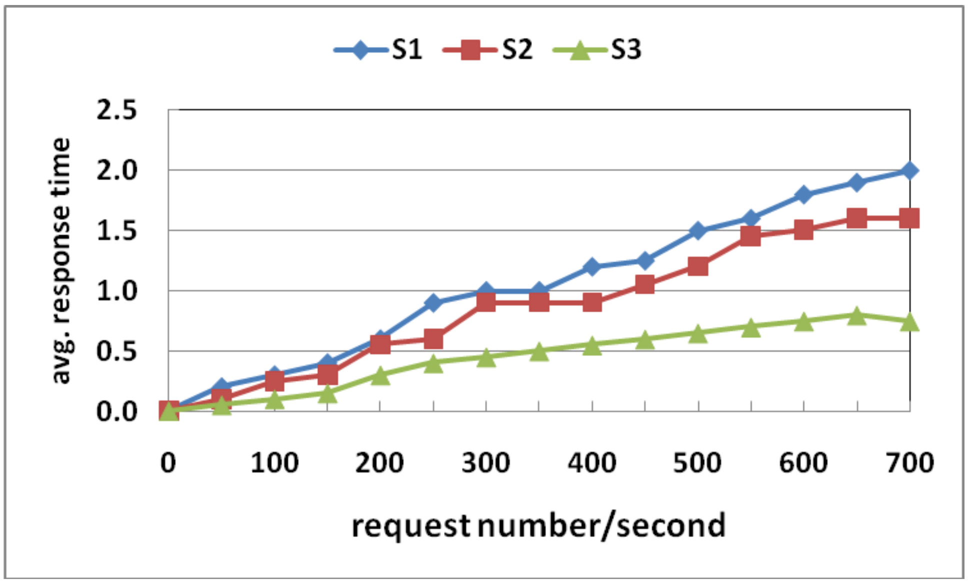


Figure 4.5: Comparison of average response time with capacity ratio 9:11:13.

We consider both drop rate and average response time to get new capacity ratio. The new capacity ratio of S_1:S_2:S_3 is *6:10:17*, as listed in Table 4.4 which is derived from the average response time under 1 second as shown in Table 4.2. Figures 4.6 and 4.7 show the number of connection and the average response time of these backend servers. The values of average response time for the three servers are very close.

Table 4.4: Capacity of each backend server with drop rate below 5% and avg. response time below 1 second.

Server	**Max. connection numbers** **- Drop rate below 5%** **- Avg. response time below 1 second**	**Capacity**
S_1	**120**	**6**
S_2	**200**	**10**
S_3	**340**	**17**

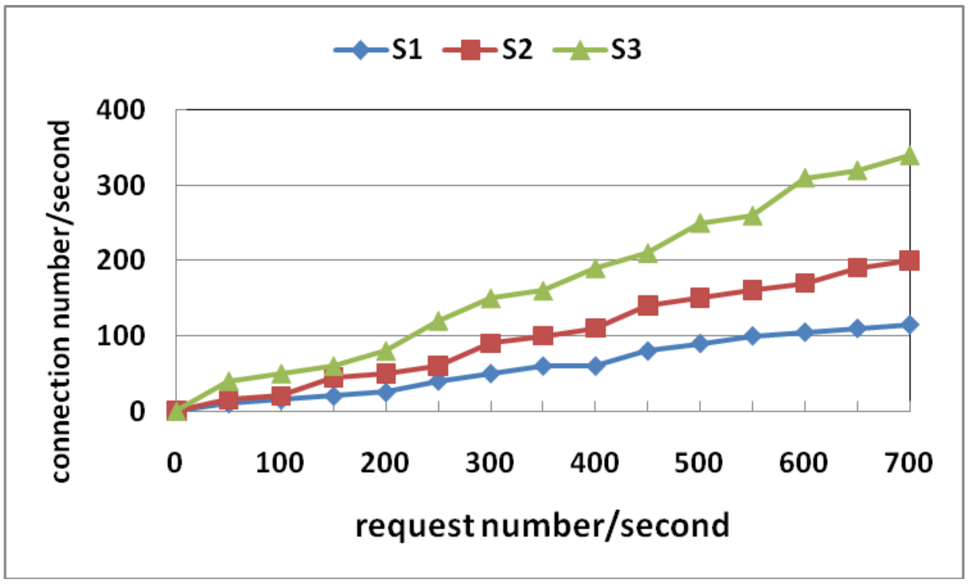


Figure 4.6: Comparison of connection number with capacity ratio 6:10:17.

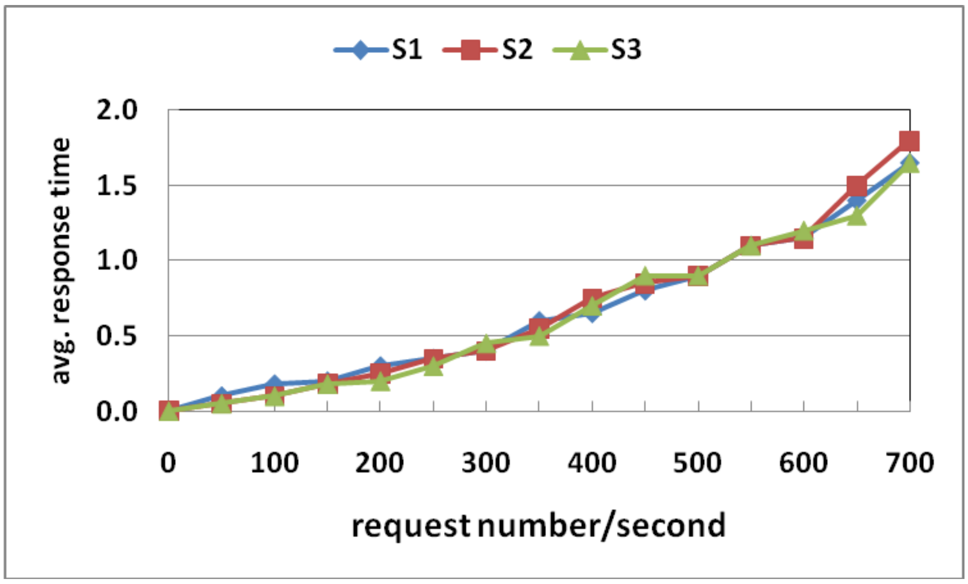


Figure 4.7: Comparison of average response time with capacity ratio 6:10:17.

In the experimental results above, the load balancing system can improve the performance. Most of the load balancing systems are based on homogeneous web servers. If the hardware specifications of backend servers in the system are different, the load balancing system must have a strategy to fairly dispatch the load to the backend servers. In this section, we derive a formula to define the capacities for heterogeneous backend servers. From the experimental results, the maximum number of connections with certain drop rate can be used as the capacity indicator, but it can not provide fair response time for each client requests. Thus, we must consider the capacity not only depending on drop rate but also on the response time. Using this measurement, the response time for all client requests will nearly be the same.

4.2 Remaining Capacity Load Balancing Algorithm

To evaluate the performance of the remaining capacity load balancing algorithm, we performed simulation using round robin (RR) and least connection (LC) load balancing algorithm as well as our proposed remaining capacity (RC) algorithm. The performance evaluation includes connection hit rate, server utilization, drop rate, and system scalability.

4.2.1 Connection Hit Rate and Server Utilization

In our simulation environment, we use three backend servers, denoted as S_1, S_2 and S_3. The capacities of these three backend server are C_1, C_2 and C_3, where $C_3 > C_2 > C_1$. We generate the client requests from 100 requests per second to 1000 requests per second. Figures 4.8 and 4.9 shows the results for the RR algorithm. Figure 4.8 shows the total hit number for the three heterogeneous backend servers. According to our environment, the S_1 has lowest capacity. When the arrival rate of client requests is about 300 requests per second, the S_1 server does not have enough remaining capacity. If the arrival rate increases, the S_1 server can only serve the same amount of client requests and begin to drop requests. If we continuously increase the arrival rate to about 600 requests per second, the second server S_2 will not have enough remaining capacity. If the arrival rate is up to about 900 requests per second, the third server S_3 also runs out of capacity. The utilizations of the three servers are shown in Fig. 4.9. At 300 requests per second, the utilization of S_1 goes to about 95%. If the arrival rate increases, the server S_1 will begin to drop requests.

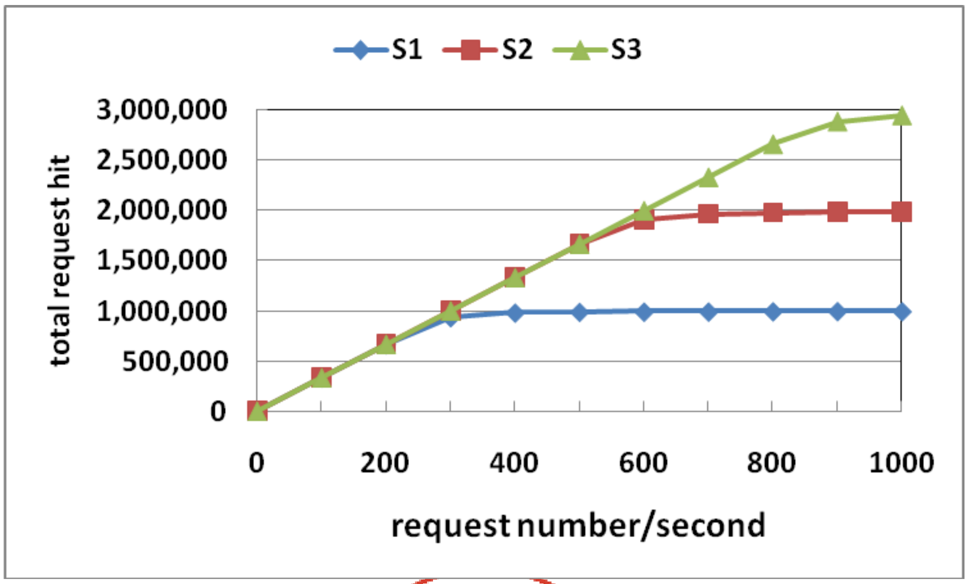


Figure 4.8: Comparison of hit rate for RR algorithm.

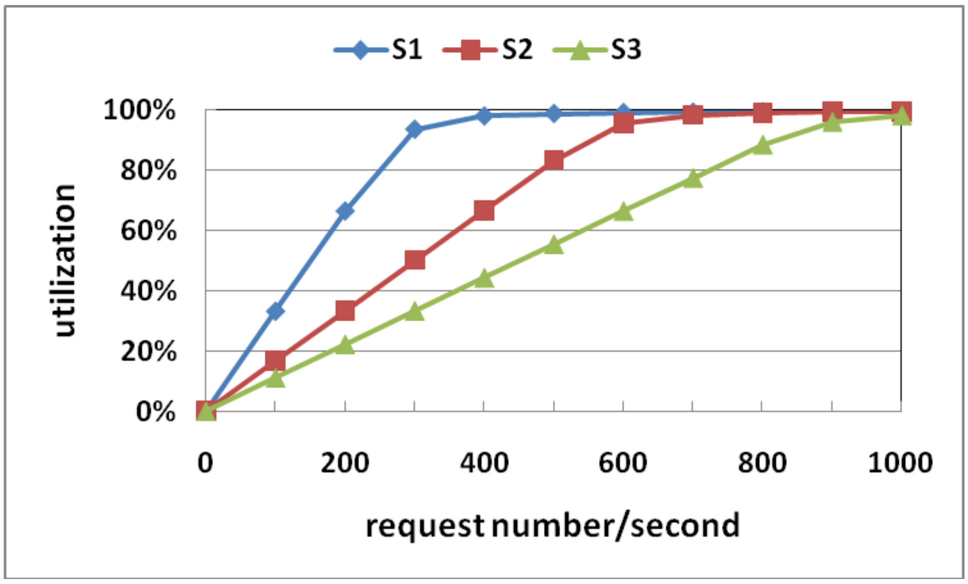


Figure 4.9: Comparison of utilization for RR algorithm.

Figures 4.10 and 4.11 show the results of the simulation of the LC algorithm. In the LC algorithm, the dispatcher will select the server with the least connection. Since we use heterogeneous backend servers; the server with least connection may not be able to serve extra request because it does not have enough remaining capacity. In this situation, if a server with least connection but with 95% loading, the dispatcher will not redirect client request to this server. Instead, it redirects the request to the next least loaded server. The hit rate of LC algorithm is similar to the RR algorithm. But in this case, the S_2 will up to 95% loading at 500 requests per second and S_3 at 700 requests per second. It seems that the RR algorithm is better than the LC algorithm. But when we consider the drop rate, we can find that the RR algorithm drops more requests than LC algorithm.

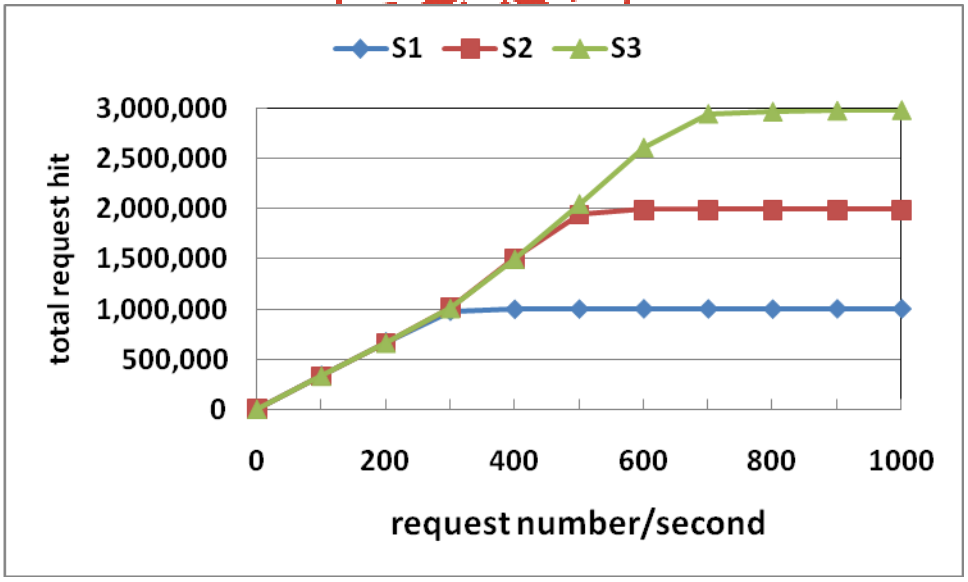


Figure 4.10: Comparison of hit rate for LC algorithm.

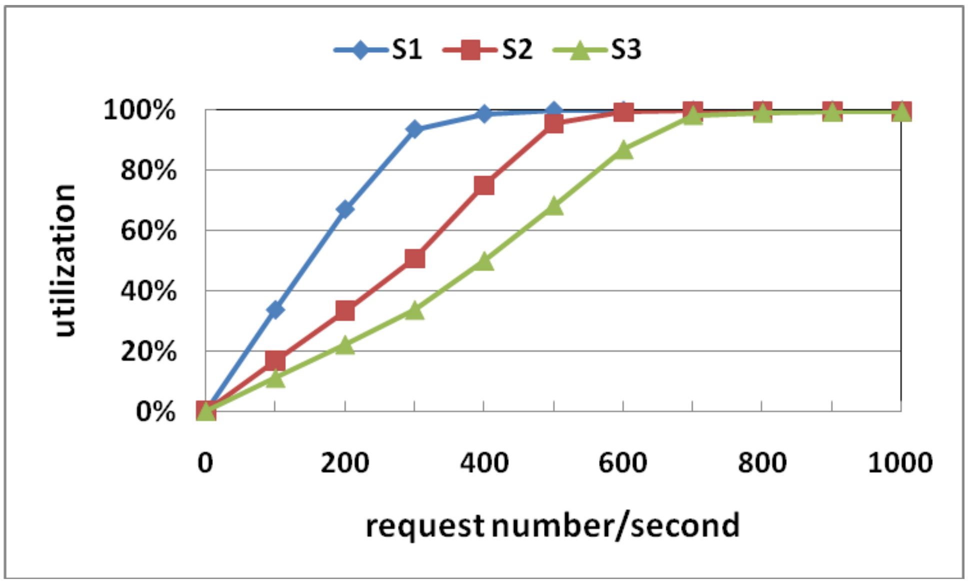


Figure 4.11: Comparison of utilization for LC algorithm.

With our proposed RC algorithm shown in Figures 4.12 and 4.13, all the three servers will up to 95% loading in 700 requests per second. This means that the three heterogeneous backend servers will with 95% loading at the specific arrival rate. In other word, the system can serve as much requests as possible with minimum drop rate. When the arrival rate is less then 300, the least capable server S_1 will serve less connection and its utilization will be lower. In this situation, the average response time will be shorter than the RR and LC algorithm.

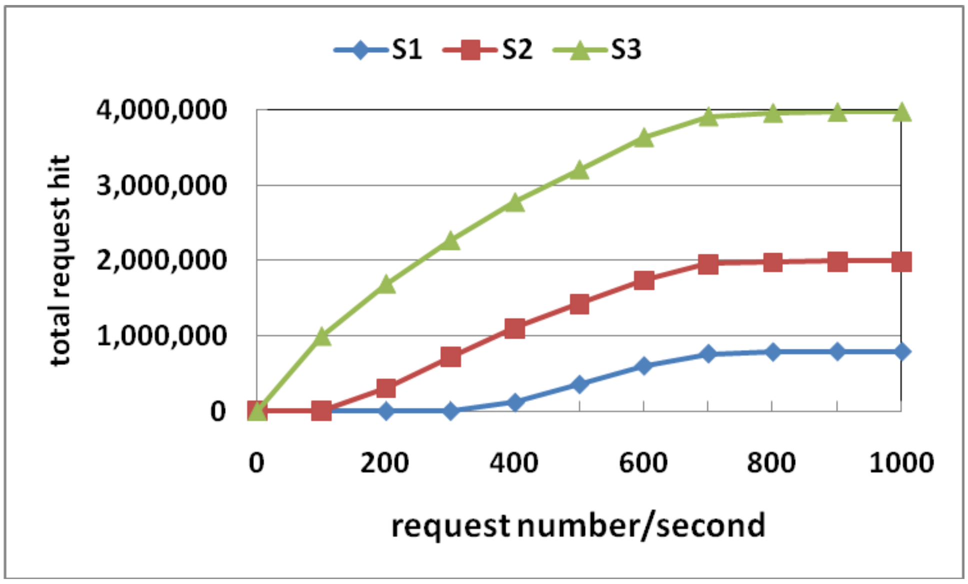


Figure 4.12: Comparison of hit rate for RC algorithm.

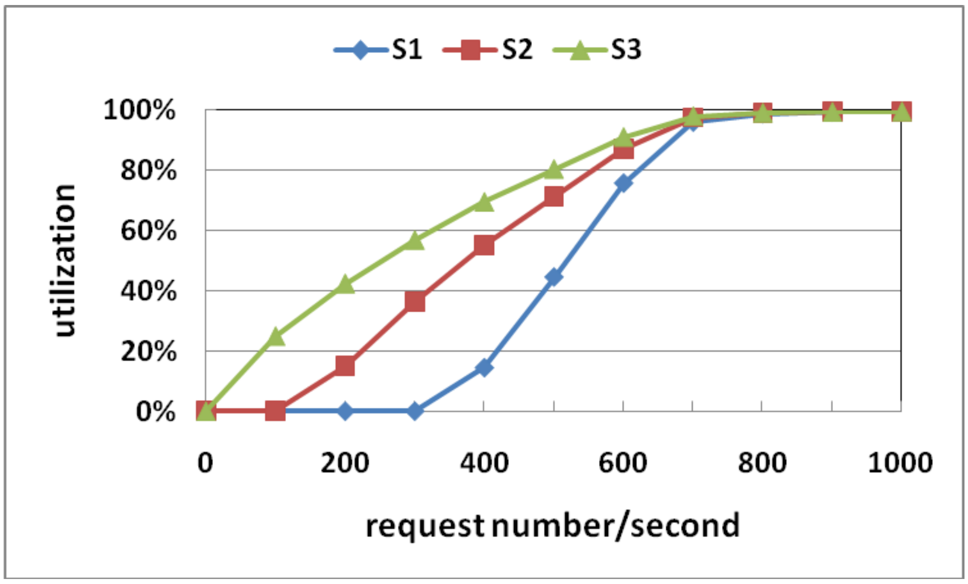


Figure 4.13: Comparison of utilization for RC algorithm.

4.2.2 Drop Rate Comparison

When the server reaches its service capacity but the dispatcher still redirects the client request to that server, the request will be dropped. An optimal load balancing system must have as little drop rate as possible. Now we compare the drop rate for these three load balancing algorithms. As shown in Fig. 4.14, the RR algorithm begins to drop client request at 300 requests per second, but in the LC and RC algorithm, they begin to drop requests at 600 and 700 requests per second, respectively. Because the LC algorithm considers only the number of connections, it redirects the request to the backend server according to the current number of connections. But with RC algorithm, the remaining capacity will be considered, so the request will be redirected to the maximum remaining capacity server. Thus the drop rate is less than LC algorithm.

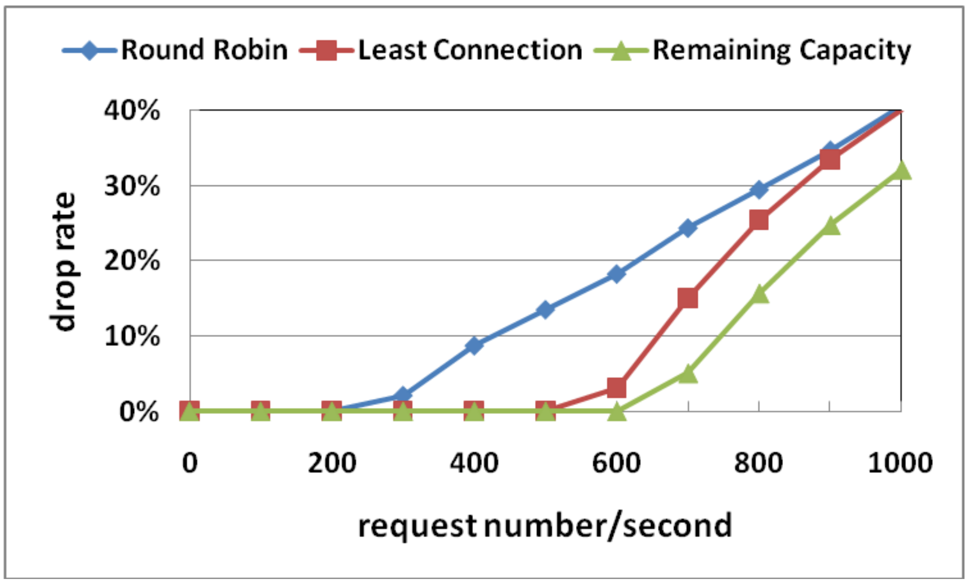


Figure 4.14: Comparison of drop rate for RR, LC, RC algorithms.

4.2.3 Scalability of the Load Balancing System

In our load balancing system, we can scale the system by using the remaining capacity of other part-time servers. Figure 4.15 is the utilization of only one web server. The web server cannot handle over 350 request per second. If we add another web server in the system, the utilization for these two web servers is shown in Fig. 4.16, which can serve more requests per second. In our proposed architecture, we use the remaining capacities of other servers like DNS or MAIL server. Consider using the remaining capacity of DNS server, we need to reserve some capacity for its original service, DNS service. In this simulation, we reserve 10% of capacity for DNS service, and the other capacity can be used as web service. The utilization for one web server plus one DNS server is shown in Fig. 4.17. It seems that the performance is not as good as the one shown in Fig. 4.16. Now we continue to use the remaining capacity of MAIL server. We must reserve more capacity for the MAIL service, say 50%, so the remaining 50% capacity can be used for web service. The utilization for one web server plus one DNS server plus one MAIL server is shown in Fig. 4.18.

Figure 4.19 shows the hit rate for one web server, two web servers, one web server plus one DNS server, and one web server plus one DNS server plus one MAIL server. We can find that when we use the remaining capacity of both DNS and MAIL server, the hit rate is higher than the hit rate of the one with two web servers. It means that if we can use the remaining capacities, the performance will be better than use another dedicate web server. The drop rate is also compared in Fig. 4.20. When using the remaining capacity of both DNS and MAIL servers, the system will begin to drop request about 700 requests per second which is also better than adding another dedicated web server.

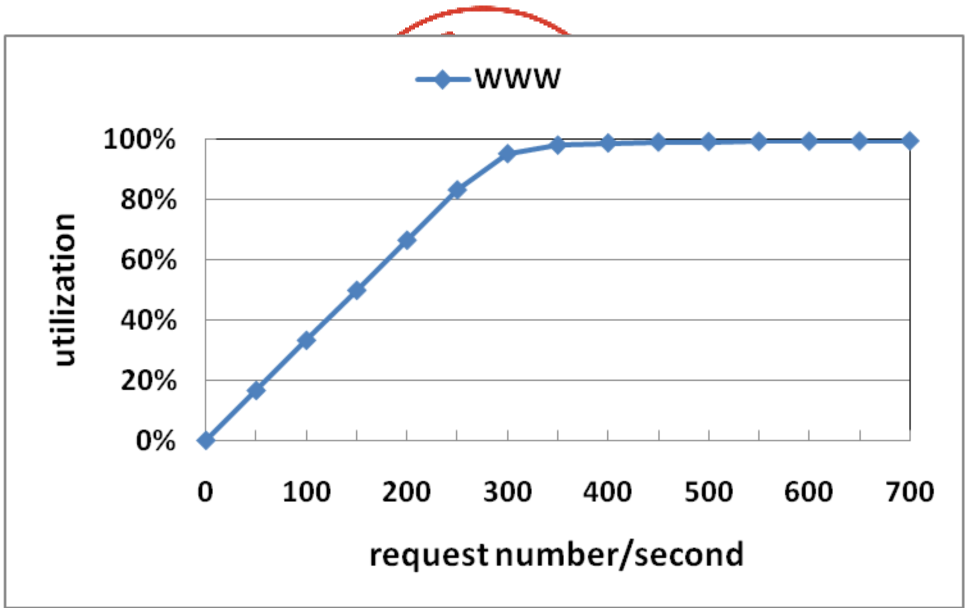


Figure 4.15: Utilization for one web server.

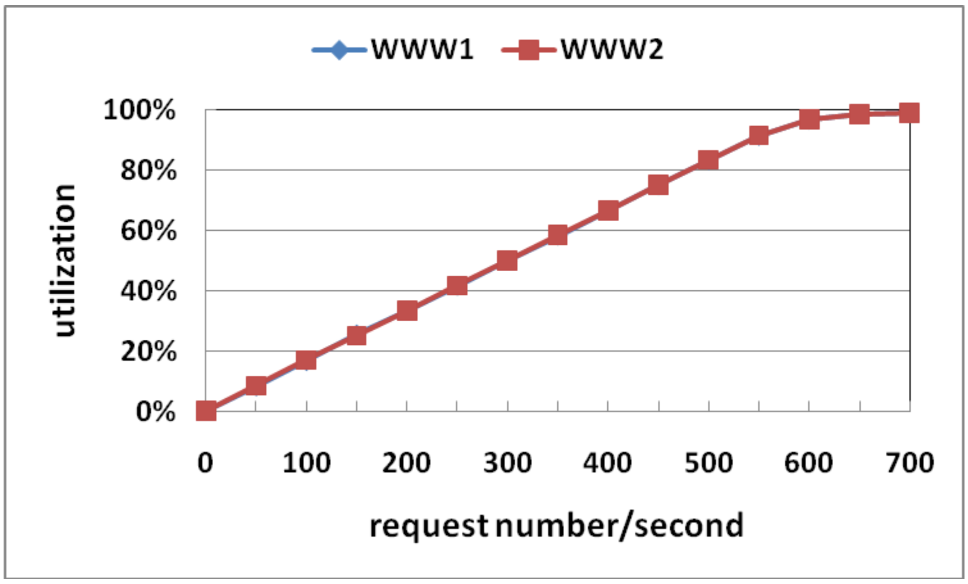


Figure 4.16: Utilization for two web servers.

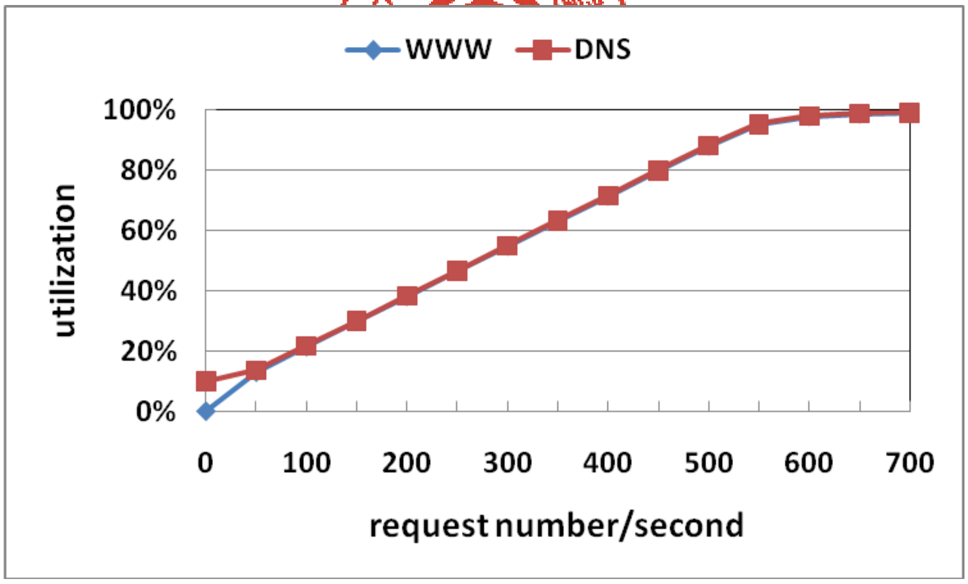


Figure 4.17: Utilization for Web + DNS.

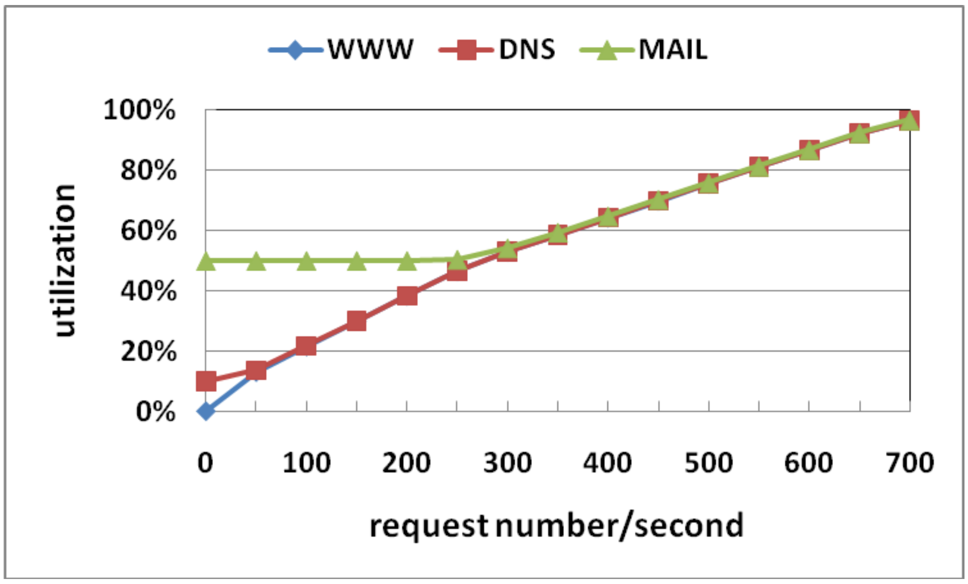


Figure 4.18: Utilization for Web + DNS + MAIL.

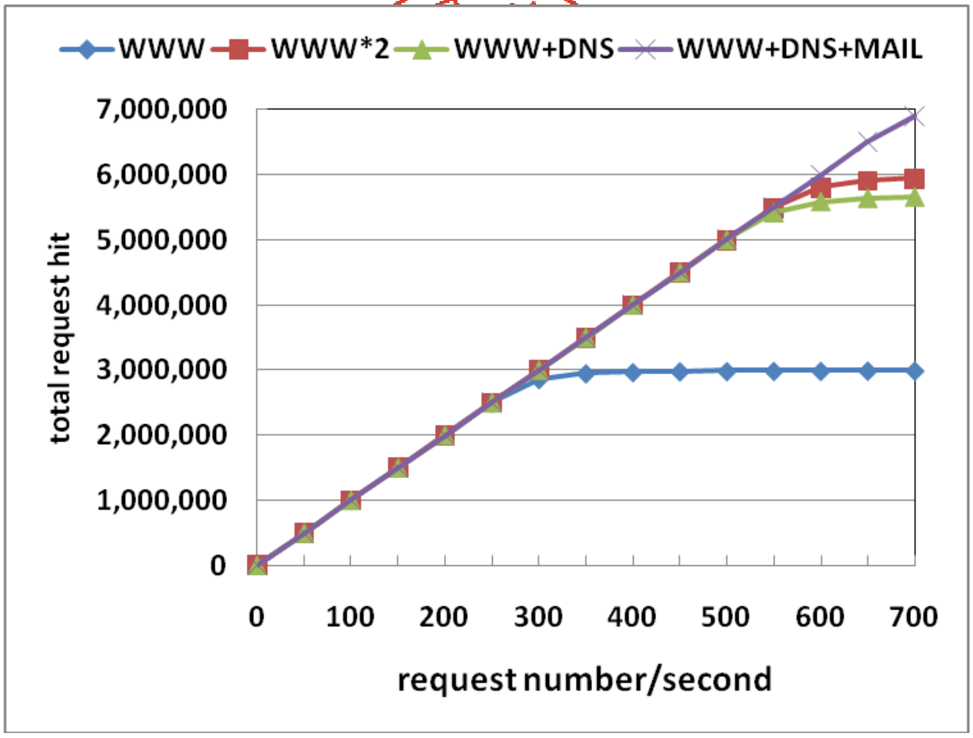


Figure 4.19: Comparison of hit rate for different schemes.

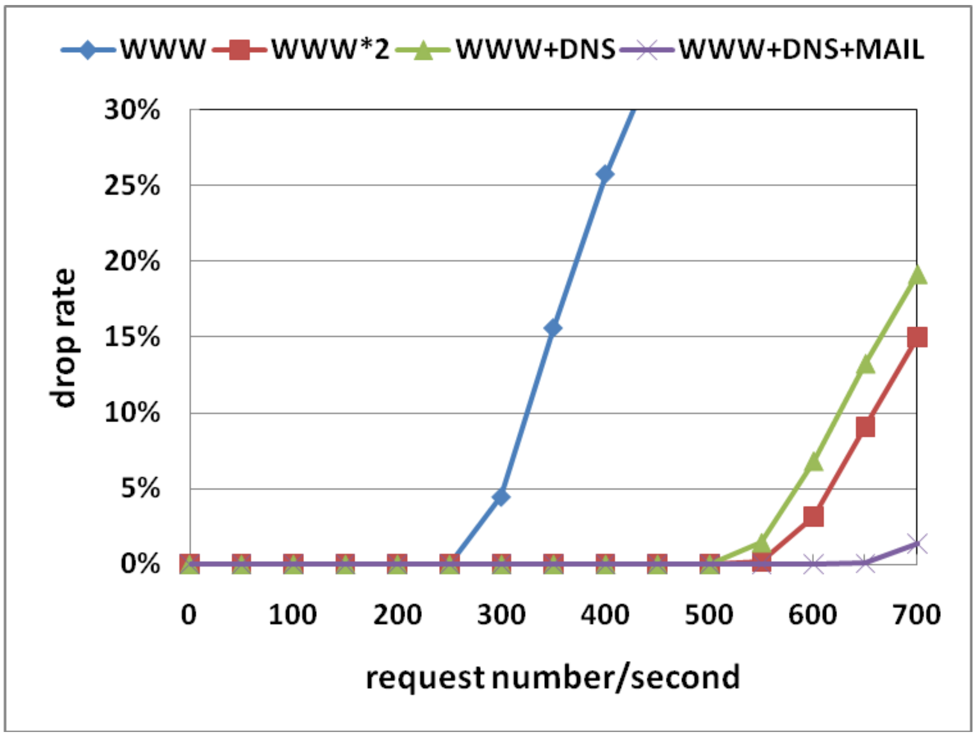


Figure 4.20: Comparison of drop rate for different schemes.

4.3 Fuzzy Decision Load Balancing Algorithm

In this section, we will present the experimental results for fuzzy decision load balancing algorithm. In our first experiment, we use the fuzzy decision mechanism to dispatch the client requests. When there is only one web server, the response time at 800 connections per second is a little bit over 1.5 seconds. In order to reduce the response time, we tried to use two web servers, and the response time at 800 connections per second is reduced to about 0.85 seconds. In this architecture, we need bring another dedicated server to the load balancing system. If the high traffic volume does not happen regularly, using another dedicated web server is not cost effective. Therefore we try to use the DNS server as the service-on-demand server to join the load balancing system. The response time at 800 connections per second is about 1.17 seconds. The results showed that it is no better than the case of using another dedicated web server. Then, the MAIL server also being brought into the load balancing system, and the response time at 800 connections per second is only about 0.55 seconds. In Fig. 4.21, when the DNS server and the MAIL server joined the load balancing system, we achieved a lower average response time when compared to using another dedicated web server.

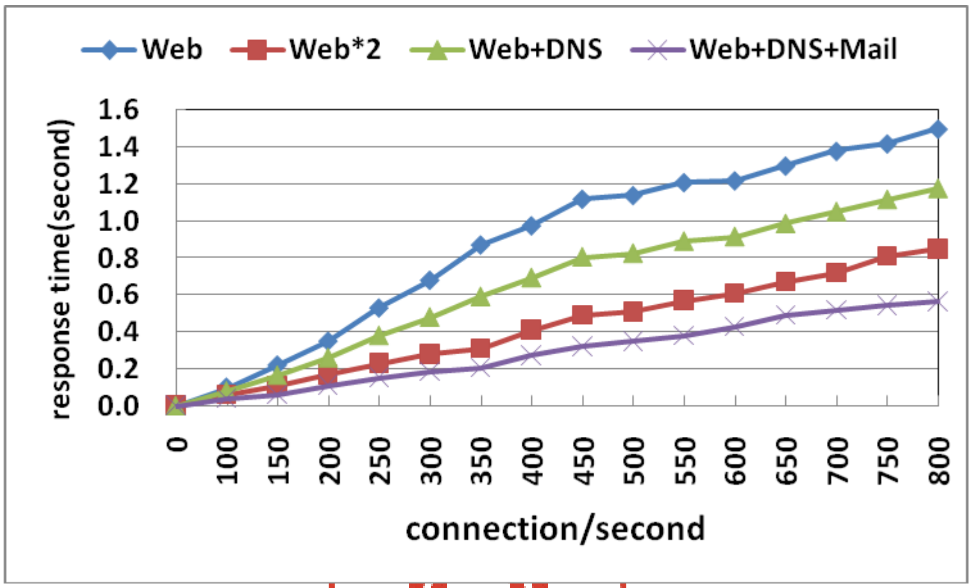


Figure 4.21: Comparison of response time for service-on-demand servers.

The fuzzy decision and the other load balancing algorithms are compared and the results are shown in Figures 4.22 and 4.23. In these experiments, the DNS server and the MAIL server joined the load balancing system. In Fig. 4.22, the round robin, least connection, and hash algorithms dispatched the client requests without considering the loading information of backend servers, so the average response times are much higher than that of the proposed fuzzy decision mechanism. The response time and bandwidth algorithms, which take the current status of backend servers into consideration, shall have better performance than the round robin, least connection, and hash. But when comparing these two algorithms with our proposed fuzzy decision algorithm, the fuzzy decision algorithm is still better than these two, as shown in Fig. 4.23. Because only partial capacity of the DNS server and the MAIL server are used to serve the requests from clients, our fuzzy decision mechanism takes the CPU, memory, and connection of backend servers into consideration. So we can reduce the average response time.

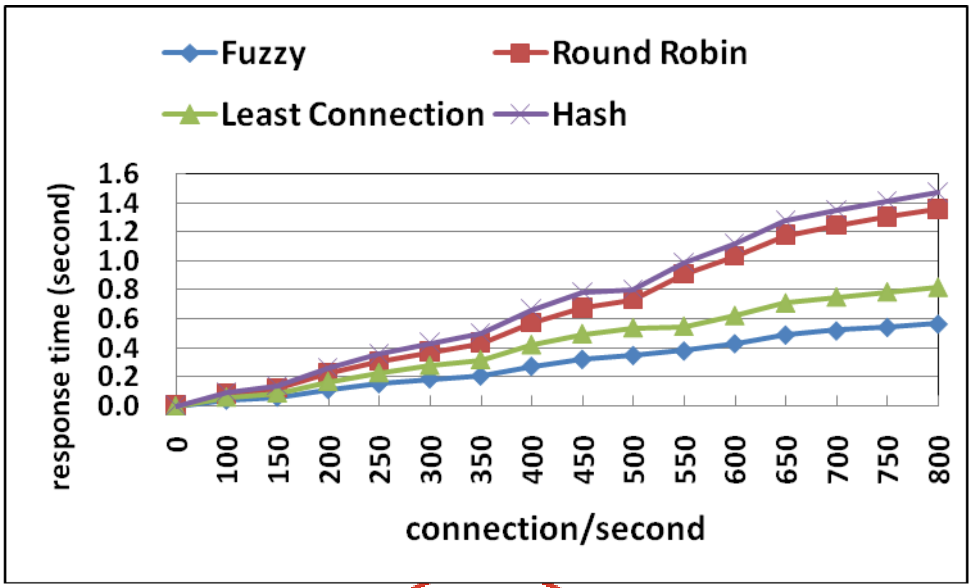


Figure 4.22: Comparison of response time for fuzzy, round robin, least connection, and hash algorithms.

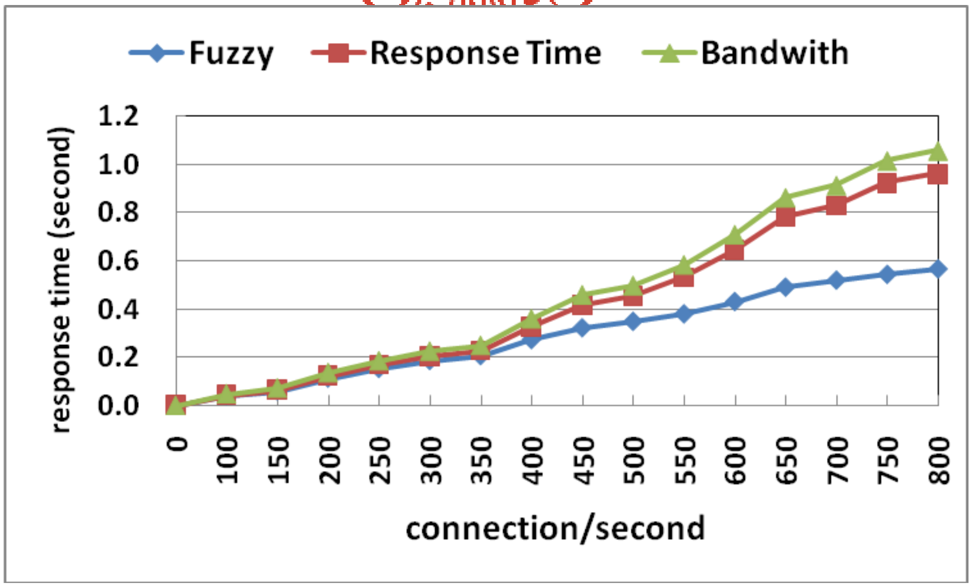


Figure 4.23: Comparison of response time for fuzzy, response time, and bandwidth algorithms.

CHAPTER 5

CONCLUSIONS

In this dissertation, we propose some load balancing algorithms to improve the performance of web sites. Many researches already proposed useful load balancing architectures. Our proposed algorithms are focused on some specific working environments. Firstly, if the processing power or the memory of each backend servers are different, the initial weights for each backend server are very important. We can use the ratio of drop rate as the initial weights to distribute the client request. But the experimental results show that the perceive latency for each client are not fair. In order to be fair in the perceive latency for each client, we re-define the initial weights according to the response time. With this modification, we can see that all of the client can obtain a fair average response time regardless of the backend server who serves it.

Secondly, we want to solve the burst web traffic problem without extra cost because most web sites do not have heavy traffic all the time. The service-on-demand server, such as DNS server or MAIL server can join the load balancing system when needed. When the service-on-demand servers join the system, we use the remaining capacity to find out which backend server is the most appropriate one to serve the client request. And then dispatch the client request to that backend server.

In addition to the remaining capacity load balancing algorithm, we want to use another intelligent method to decide which backend server is the most appropriate one. The fuzzy decision mechanism was adopted to dispatch the client request in our proposed fuzzy decision load balancing algorithm. With this intelligent algorithm, we can reduce the average response time for all the client requests. The experimental results also show that the part-time DNS server plus a MAIL server can achieve higher performance than by adding an additional dedicated web server.

REFERENCES

[1] Eunmi Choi, Yoojin Lim, and Dugki Min, “Performance Comparison of Various Web Cluster Architectures,” *Lecture Notes in Computer Science*, vol. 3398, pp. 617-624, May 2005.

[2] Dan Mosedale, William Foss, and Robert Martin McCool, “Lessons Learned Administering Netscape’s Internet Site,” *IEEE Trans. Internet Computing*, vol. 1, no. 2, pp. 28-35, March/April 1997.

[3] Eric Dean Katz, Michelle Butler, and Robert McGrath, “A Scalable HTTP Server: The NCSA Prototype,” *Trans. Computer Networks and ISDN Systems*, vol. 27, no.2, pp. 155-164, 1994.

[4] Tsang-Long Pao, Jian-Bo Chen, and I-Ching Cheng, “An Analysis of Server Load Balance Algorithms for Server Switching,” *Proc. Ming-Chung University International Academic Conference*, Taiwan, March 2004.

[5] Jeffery S. Chase, “Server Switching: Yesterday and Tomorrow,” *Proc. Second IEEE Workshop on Internet Applications*, pp. 114-123, San Jose, CA, 2001.

[6] Notel Networks, “Alteon Link Optimizer Application Guide,” Release 1.0, 2002.

[7] Notel Networks, http://www.nortelnetworks.com.

[8] Cisco System, http://www.cisco.com.

[9] Extreme Network, http://www.extremenetworks.com.

[10] F5 Networks, http://www.f5.com.

[11] Scot Hull, “Content Delivery Networks: Web Switching for Security, Availability, and Speed,” *McGraw Hill*, 2002.

[12] Microsoft, http://www.microsoft.com.

[13] Linux Virtual Server, http://www.linuxvirtualserver.org.

[14] Jiani Guo and Laxmi N. Bhuyan, “Load Balancing in a Cluster-Based Web Server

for Multimedia Applications," *IEEE Trans. Parallel and Distributed Systems,* vol. 17, no. 11, pp. 1321-1334, November 2006.

[15] Richard Martin, Amin Vahdat, David Culler, and Thomas Anderson, "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc. 24th International Symposium on Computer Architecture*, pp. 85-97, Denver, Co, June 1997.

[16] Huican Zhu, Hong Tang, and Tao Yang, "Demand-Driven Service Differentiation in Cluster-Based Network Servers," *Proc. Joint Conference between IEEE Computer and Communications Societies (INFOCOM)*, pp. 679-688, Anchorage, Alaska, USA, April 2001.

[17] Chang Li, Gang Peng, Kartik Gopalan, and Tzi-cher Chiueh, "Performance Garantees for Cluster-Based Internet Services," *Proc. 23th International Conference on Distributed Computing Systems*, pp. 378-385, May 2003.

[18] Devarshi Chatterjee, Zahir Tari, and Albert Zomaya, "A Task-Based Adaptive TTL Approach for Web Server Load Balancing," Proc. *10th IEEE Symposium on Computers and Communication*, pp. 877-884, Murcia, Cartagena, Spain, June 2005.

[19] Valeria Cardellini, Michele Colajanni, and Philip S. Yu., "DNS Dispatching Algorithms with State Estimators for Scalable Web-server Clusters," *Trans. World Wide Web*, vol. 2, no. 3, pp. 101-113, 1999.

[20] Kai-Hau Yeung, Kam-Wa Suen, and Kin-Yeung Wong, "Least Load Dispatching Algorithm for Parallel Web Server Nodes," *Proc. IEE Communications*, vol. 149, no. 4, pp 223-226, August 2002.

[21] DNS rfc, http://www.dns.net/dnsrd/rfc.

[22] Philips S. Yu, Valeria Cardellini, and Michele Colajanni, "Dynamic Load

Balancing on Web-server Systems,” *IEEE Trans. Internet Computing*, pp. 28-39, May/June 1999.

[23] Michele Colajanmi and Philip S. Yu, ”Adaptive TTL schemes for Load Balancing of Distributed Web Servers”, *ACM Trans. Sigmetrics Performance Evaluation Review*, vol. 25, no. 2, pp. 36-42, September 1997.

[24] Michele Colajanni, Philip S. Yu, and Daniel M. Dias, “Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems,” *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 6, pp. 585-600, June 1998.

[25] Daniel M. Dias, William Kish, Rajat Mukherjee, and Renu Tewari, “A Scalable and Highly Available Web-Server,” *Proc. 41st IEEE International Conference on Computer Society*, pp. 85-92, Santa Clara, CA, February 1996.

[26] Ashish Singhai, Swee Boon Lim, and Sanjay R. Radia, “The SunSCALR Framework for Internet Servers,” *Proc. IEEE Fault-Tolerant Computing Systems*, pp. 108-116, Munich, Germany, June 1998.

[27] Roland J. Schemers, “lbmnamed: A Load Balancing Name Server in Perl,” *Proc. 9th USENIX Conference on Systems Administration*, pp. 1-12, Monterey, CA, 1995.

[28] lbnamed, A Load Balancing name server written in Perl, http://www.stanford.edu/~schemers/docs/lbnamed/lbnamed.html.

[29] Internet System Consortium, http://www.isc.org.

[30] Trevor Schroeder, Steve Goddard, and Byrav Ramamurthy, “Scalable Web Server Clustering Technologies,” *IEEE Trans. Network*, vol. 14, no. 3, pp. 38-45, May/June 2000.

[31] Xin Liu and Andrew A. Chien, “Traffic-based Load Balance for Scalable Network Emulation,” *Proc. ACM/IEEE Supercomputing*, pp. 40-50, November 2003.

[32] W3C World Wide Web Consortium, http://www.w3c.org.

[33] Michael Garland, Sebastian Grassia, Robert Monroe, and Siddhartha Puri, “Implementing Distributed Server Groups for the World Wide Web,” *Technical Report CMUCS-95-114*, January 1995.

[34] Valeria Cardellini, Michele Colajanni, and Philip S. Yu, “Redirection Algorithms for Load Sharing in Distributed Web-Server Systems,” *Proc. 19th IEEE International Conference on Distributed Computing Systems*, pp. 528-535, 1999.

[35] Valeria Cardellini, Michele Colajanni, and Philip S. Yu, “Request Redirection Algorithms for Distributed Web Systems,” *IEEE Trans. Parallel and Distributed Systems*, vol 14, no 4, pp. 355-368, April 2003.

[36] Liming Liu and Yumao Lu, “Dynamic Traffic Controls for Web-server Networks,” *Trans. Computer Networks*, vol. 45, no. 4, pp. 523-536, 2004.

[37] John Chiasson, Zhong Tang, Jean Ghanem, Chaouki T. Abdallah, J. Douglas Birdwell, Majeed M. Hayat, and Henry Jerez, “The Effect of Time Delays on the Stability of Load Balancing Algorithms for Parallel Computations,” *IEEE Trans. Control Systems Technology*, vol. 13, no. 6, pp. 932-942, November 2005.

[38] Zhang Xiayu, Yu Yongquan, Chen Baixing, Ye Feng, and Xingxing Tan, “An Extension-Based Dynamic Load Balancing Model of Heterogeneous Server Cluster,“ *Proc. IEEE International Conference on Granular Computing*, pp. 675-679, November 2007.

[39] Haiying Shen and Cheng-Zhong Xu, “Hash-based proximity clustering for load balancing in heterogeneous DHT networks,” *Proc. International Conference on Parallel and Distributed Processing Symposium*, pp. 22-31, Rhode Island, Greece, April 2006.

[40] Nehra Neeraj and Patel R.B., “Towards Dynamic Load Balancing in Heterogeneous Cluster Using Mobile Agent,” *Proc. International Conference on*

Computational Intelligence and Multimedia Applications, pp. 15-21, December 2007.

[41] Di Wu, Ye Tian, and Kam-Wing Ng, "On the Effectiveness of Migration-based Load Balancing Strategies in DHT Systems," *Proc. International Conference on Computer Communications and Networks*, pp. 405-410, October 2006.

[42] Karim Y. Kabalan, Waleed W. Smari, and Jacques Y. Hakimian, "Adaptive Load Sharing in Heterogeneous Systems: Policies, Modifications, and Simulation," *Trans. Simulation Systems Science and Technology*, vol. 3, no. 1-2, pp. 89-100, 2002.

[43] Ruchir Shah, Bharadwaj Veeravalli, and Manoj Misra, "Estimation Based Load Balancing Algorithm for Data-Intensive Heterogeneous Grid Environments," *Proc. 13th International Conference on High Performance Computing*, pp. 72-83, 2006.

[44] Kai Lu and Albert Y. Zomaya, "A Hybrid Policy for Job Scheduling and Load Balancing in Heterogeneous Computational Grids," *Proc. Sixth International Symposium on Parallel and Distributed Computing*, pp. 19-27, July 2007.

[45] Yigal Bejerano, Seung-Jae Han, and Li Li, "Fairness and Load Balancing in Wireless LANs Using Association Control," IEEE/ACM *Trans. Networking*, vol. 15, issue 3, pp. 1-14, 2007.

[46] Y. Bejerano, Seung-Jae Han, and Li Li, "Fairness and Load Balancing in Wireless LANs Using Association Control," *Proc. ACM International Conference on Mobile Computing and Networking*, pp. 315-329, 2004.

[47] Jon Kleinberg, Yuval Rabani, and Eva Tardos, "Fairness in Routing and Load Balancing," *Proc. 40th Annual Symposium on Foundations of Computer Science*, pp. 568-578, New York, NY, USA, October 1999.

[48] Parameswaran Ramanathan and Prathima Agrawal, "Adapting Packet Fair Queueing Algorithms to Wireless Networks," *Proc. ACM International Conference*

on Mobile Computing and Networking, pp. 1-9, Dallas, Texas, USA, October 1998.

[49] Yijiao Chen, Xicheng Lu, and Zhigang Sun, "MSF: A Session-Oriented Adaptive Load Balancing Algorithm," *Proc. International Conference on Network and Parallel Computing Workshops*, pp. 657-663, September 2007.

[50] Chhabra Amit and Singh Gurvinder, "Qualitative Parametric Comparison of Load Balancing Algorithms in Distributed Computing Environment," *Proc. International Conference on Advanced Computing and Communications*, pp. 58-61, December 2006.

[51] Gregg Petrie, G. Fann, E. Jurrus, B. Moon, K. Perrine, C. Dippold, and D. Jones, "A Distributed Computing Approach for Remote Sensing Data," *Proc. 34th Symposium Interface*, April 2002.

[52] Seyed Mahdi Bouzari, Mohammad Reza Javan, and Ahmad Salahi, "Efficient Algorithm for Load Balancing," *Proc. International Symposium on Signals, Circuits and Systems*, vol. 2, pp. 1-4, July 2007.

[53] L. Anand, D. Ghose, and V. Mani, "ELISA: An Estimated Load Information Scheduling Algorithm for Distributed Computing Systems," *Trans. Computers and Mathematics with Applications*, vol. 37, no. 8, pp. 57-85, April 1999.

[54] Yoshitomo Murata, Hiroyuki Takizawa, Tsutomu Inaba, and Hiroaki Kobayashi, "A Distributed and Cooperative Load Balancing Mechanism for Large-Scale P2P Systems," *Proc. International Symposium on Applications and Internet Workshops*, pp. 23-27, January 2006.

[55] David B. Shmoys and Eva Tardos, "An Approximation Algorithm for the Generalized Assignment Problem," *Trans. Matematical. Programming*, vol. 62, no. 3, pp. 461-474, 1993.

[56] Rudiger Martin, Michael Menth, and Michael Hemmkeppler, "Accuracy and

Dynamics of Multi-Stage Load Balancing for Multipath Internet Routing," *Proc. IEEE International Conference on Communications*, pp. 6311-6318, June 2007.

[57] Zeng Zeng and Bharadwaj Veeravalli, "Design and Performance Evaluation of Queue-and-Rate-Adjustment Dynamic Load Balancing Policies for Distributed Networks," *IEEE Trans. Computers*, vol. 55, no. 11, pp. 1410-1422, November 2006.

[58] Waraich Sandeep Singh, "Classification of Dynamic Load Balancing Strategies in a Network of Workstations," *Proc. Fifth International Conference on Information Technology: New Generations*, pp. 1263-1265, April 2008.

[59] Nehra Neeraj, Patel R.B., and Bhat V. K., "A Multi-Agent system for Distributed Dynamic Load Balancing on Cluster," *Proc. International Conference on Advanced Computing and Communications*, pp. 135-138, December 2006.

[60] Zhiling Lan, Valerie E. Taylor, and Greg Bryan, "Dynamic Load Balancing for Adaptive Mesh Refinement Application," *Proc. International Conference on Parallel Processing*, Valencia, Spain, 2001.

[61] Sagar Dhakal, Biliana Paskaleva, Majeed M. Hayat, Edl Schamiloglu, and Chaouki T. Abdallah, "Dynamical Discrete-Time Load Balancing in Distributed Systems in the Presence of Time Delays," *Proc. IEEE Conference on Decision and Controls*, vol. 5, pp. 5128-5134, December 2003.

[62] Ana Cortes, Ans Ripoll, Miquel Senar, and Emilio Luque, "Performance Comparison of Dynamic Load-Balancing Strategies for Distributed Computing," *Proc. 32nd Hawaii Conference on System Sciences*, vol. 8, pp. 1-10, 1999.

[63] Yucai Feng, Dong Li, Hengshan Wu, and Yi Zhang, "A Dynamic Load Balancing Algorithm Based on Distributed Database System," *Proc. Fourth International Conference on High-Performance Computing in the Asia-Pacific Region*, vol. 2, pp.

949-952, May 2000.

[64] Marc H. Willebeek-LeMair and Anthony Reeves, “Strategies for Dynamic Load Balancing on Highly Parallel Computers,” *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 4, pp. 979-993, September 1993.

[65] Hwa-Chun Lin and C. Raghavendra, “A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC),” *IEEE Trans. Software Engineering*, vol. 18, no. 2, pp. 148-158, Feburary 1992.

[66] Jerrell Watts and Stephen Taylor, “A Practical Approach to Dynamic Load Balancing,” *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 3, pp. 235-248, March 1998.

[67] Mohammed J. Zaki, Wei Li, and Srinivan Parthasarathy, “Customized Dynamic Load Balancing for a Network of Workstations,” *Trans. Parallel and Distributed Computing*, vol. 43, no. 2, pp. 156-162, 1997.

[68] Suman Das, Harish Viswanathan, and Gee Rittenhouse, “Dynamic Load Balancing Through Coordinated Scheduling in Packet Data Systems,” *Proc. IEEE INFOCOM*, vol. 1, pp. 786-796, April 2003.

[69] Tsang-Long Pao and Jian-Bo Chen, “Remaining Capacity Based Load Balancing Architecture for Heterogeneous Web Server System,” *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 58-63, Las Vegas, USA, June 2006.

[70] Jaroslav Ramik, “A Decision System Using ANP and Fuzzy Inputs,” *Trans. International Journal of Innovative Computing, Information and Control*, vol.3, no.4, pp. 825-837, August 2007.

[71] Lily Lin and Huey-Ming Lee, “A Fuzzy Decision Support System for Selecting the Facility Site of Multinational Enterprises,” *Trans. International Journal of*

Innovative Computing, Information and Control, vol.3, no.1, pp. 151-162, February 2007.

[72] Nyan Win Aung and Eric W. Cooper, Yukinobu Hoshino, Katsuari Kamei, “A Proposal of Fuzzy Control Systems for Trailers Driven by Multiple Motors in Side Slipways to Haul Out Ships,” *Trans. International Journal of Innovative Computing, Information and Control*, vol.3, no.4, pp. 799-812, August 2007.

[73] Shimpei Matsumoto, Nobuyuki Ueno, Koji Okuhara, and Hiroaki Ishii, “Decision of Optimal Load Leveling Point and Effect of Unofficial Announcement for Implementing Mass Customization,” *Trans. International Journal of Innovative Computing, Information and Control*, vol.3, no.1, pp. 53-69, February 2007.

[74] William V. Wollman, Harry Jegers, Maureen Loftus, and Caleb Wan, “Plug and Play Server Load Balancing and Global Server Load Balancing for Tactical Networks”, *Proc. IEEE Military Communications Conference*, vol. 2, pp. 933-937, October 2003.

[75] KNOPPIX, http://www.knoppix.net/.

[76] Jin-Long Wang and Shih-Ping Huang, “Fuzzy Logic Based Reputation System for Mobile Ad Hoc Networks,” *Lecture Notes in Computer Science*, vol. 4693, pp. 1315-1322, 2007.