UNIVERSITY OF OSLO
Department of Informatics

Presenting a Prototype for Pull Based Load Balancing of Web Servers

Master thesis

Arild Berggren
Oslo University College

23rd May 2007

Presenting a Prototype for Pull Based Load Balancing of Web Servers

Arild Berggren
Oslo University College

23rd May 2007

Abstract

The traffic and the number of users on the Internet are increasing, and service providers must respond to this demand if they are not to loose both customers and revenue. In order to provide a satisfactory level of QoS, the providers must increase server performance by aggregating multiple physical host into a web farm, which by working together, sharing the total load of the requests, will act as one unified server.

Various algorithms and technology exists for performing this load sharing, and this thesis will introduce a new approach in this field. In the traditional load balancing (LB) strategies, the requests are pushed to the individual servers, which are passively forced to accept the requests. The new approach presented in this paper, differs from the traditional techniques, in that the servers themselves are more involved in the decision making associated with load balancing. Requests are stored in a central queue, and the individual requests is processed by the servers at the server's own convenience, leaving the server with full control over it's own resources.

The paper will present and compare this algorithm with the traditional load balancing algorithms, and investigate possible benefits of this strategy.

Preface

The work in this thesis originated from an idea conceived during participation in a course in high availability services, as part of the obligatory courses in the Master degree in Network and System Administration at Oslo University College. This course, in addition to a trip to Washington DC where I attended the LISA (Large Installation System Administration) conference, illuminated me in the various load balancing techniques that are being used toady, and it helped me realize the importance and significance of load balancing when it comes to issues such as availability and performance.

This thesis is a contribution to the body of knowledge, and will be my first go at science. My goal has been, when I look back at this work, to be able to state that I explored my own ideas, made my own choices when necessary; and ultimately did the thesis my way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the web traffic and the number of users on the Internet increases, service providers struggle to keep up with Service Level Agreements (SLAs). User perceived Quality of Service (QoS) is especially important when it comes to e-Commerce. If a web server has more requests than it can handle, either due to a generally high request rate, or due to a flash crowd effect [1] this will results in slow or non responsive web servers, and some customers might even be denied access to the service completely. Under either of these circumstances, a user will take his business elsewhere, and the results is a loss in both customers and revenue. In order for a company to maintain customer loyalty, the servers must provide a consistent level of QoS [1].

If a server gets more requests than it can handle, this can be combatted by using multiple hosts to provide the same service. A web farm of multiple physical hosts grouped together will share the total load of the client requests. This will reduce the response time, thus increase the QoS, ultimately resulting in satisfied customers.

Internet traffic is a random process with a long tailed distribution [2], which means that traffic often comes in bursts. A way to combat short periods with heavy load, is to over-provision the total resources. This means to have more computers than you normally need, in order to respond to situations such as the flash crowd effects.

Another important issue amongst service providers is their degree of up-time of the servers. This is also refereed as server availability, which in marketing documents are given as a certain number nines. An availability of 99.99 per cent is refereed to as an availability of four nines. This means that the server should only be down .1 per cent of the time, which over a duration of one year contributes to about 52 minutes of unavailability.

Another benefit of having a web farm is redundancy, which helps to achieve

[1] An event happens that triggers an unusual amount of people to visit a specific web page at the same time.

both high availability as well as high performance. It is possible to perform upgrades on a host without shutting down the total service. By performing upgrades on only one server at a time, it is possible to still have a functional service running, however, with a slight performance decrease, due to the loss of potential processing power. The same situation will be true in an unforeseen event, such as a computer crash; the service will still be available due to the other operational hosts in the web farm.

As we see, load balancing (LB) addresses important issues such as high performance, availability, and redundancy.

Several techniques and algorithms exists for performing the actual load balancing of the servers. Research have been done to compare different load balancing algorithms, with the goal as to find the optimal algorithm, and the optimal algorithm during given traffic situations. This paper will introduce yet another load balancing technique, in which the individual hosts in the web farm will play a more direct role, and thus have more control over the specific act of the load balancing. This architecture will be compared with a traditional load balancing technique, in order to come to the conclusions whether or not this different approach to load balancing has any performance benefits.

Chapter 2

Background

> A scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die and a new generation grows up that is familiar with it.
>
> Max Planck

2.1 Network communication

This section will describe basics about network communication on the Internet, and talk about fundamentals, such as the TCP (Transmission Control Protocol) and IP (Internet Protocol) protocols, as well as the OSI (Open Systems Interconnect) model.

TCP and IP are the most important protocols used on the Internet [3], and they are often abbreviated as TCP/IP, which only helps to show their importance and mutual inter dependencies. A protocol is simply a set of rules which states how network equipment should talk to each other.

TCP and IP work on different layers, meaning that they are used for dealing with different aspects and solve different problems of network communication. Table 2.1 shows the seven different layers of the OSI reference model, created by the International Standards Organization (ISO) [4]. This model is a convenient way to describe the network, and shows the different responsibilities of the various protocols. A layered structure enables us to change the details of the lower levels protocols without changing the upper layers. This means that we can improve network communication on the low levels without rewriting software [4], which works at the high levels.

Burgess [4] describes the layers in more details as follows:

1. *Physical layer*. This is the problem of sending a binary signal across a wire, amplifying it if it gets weak, removing noise etc.

7	Application layer	telnet, ssh (Application which sends data)
6	Presentation Layer	HTTP, SMTP, FTP
5	Session layer	RPC / sockets
4	Transport layer	TCP, UDP
3	Network layer	IP
2	Data link layer	Ethernet (MAC layer)
1	Physical layer	Fiber optics, Coaxial cable

Table 2.1: The seven layers of the OSI reference model.

2. *Data link layer* This layer checks to make sure that the data that was send across a wire actually arrived at the other end; also known as *handshaking*.

3. *Network layer* This is the layer of software that remembers which machines are talking to each other, and based upon network addresses, sets up connections and addresses data to the right destinations.

4. *Transport layer* This layer builds packets so that the network layer knows what is data and how to get the data to their destination. Because many machines may use the same network at the same time, data are broken into short 'bursts', which is refereed to as time sharing multiplexing.

5. *Session layer* This layer helps to set up connections, with the use of sockets or RPC (Remote Procedure Call).

6. *Presentation layer* This layer defines how data should be presented, using protocols (i.e HTTP) or RPC.

7. *Application layer* is the program that sends data. E.g. a web browser or a telnet client.

In practice, the OSI model is often shortened down to a four layer TCP/IP model, in which layers 5 to 7 is abbreviated into Application layer. The network layer becomes the Internet layer, and the data link and physical layer becomes the network layer. Additionally, we also see that the physical layer is omitted.

One of the reasons of this abbreviation, is that it is not always clear what is contained in the different layers of the OSI model. Some consider it too theoretical, and feel that it does not apply to the modern networking protocols such as TCP/IP. The layers of this shortened version of the OSI model is shown table 2.2.

When a packet is transmitted on the Internet, all of the four layers of the TCP/IP model are involved. Figure 2.1 shows a figure of a HTTP request along with how the contents of this request is contained in the different layers. We

4	Application
3	Transport
2	Internet
1	Network layer

Table 2.2: The four layers of the TCP/IP model.

see that the HTTP content is wrapped inside a TCP packet, which is wrapped inside a IP packet, which again is wrapped inside an ethernet packet. The headers contain, among others, information about how the packets are to traverse the network [5], as well as information used for error handling.

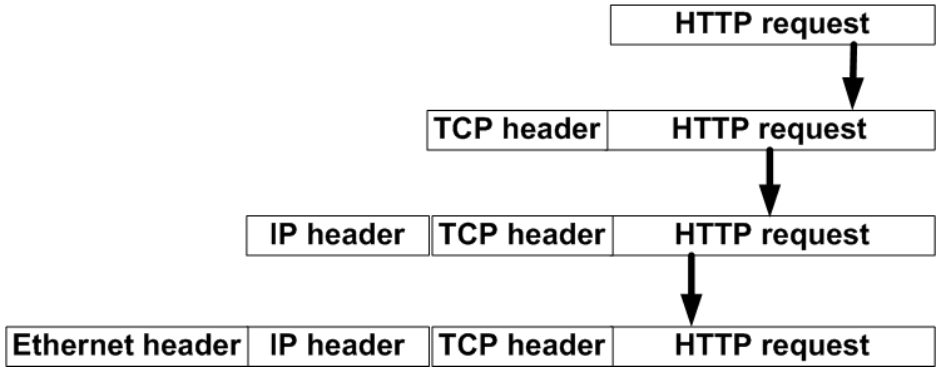


Figure 2.1: Encapsulation of a HTTP request in the four layers of the TCP/IP model.

When a client sends a HTTP request to a web server, the data of the requests is first wrapped inside a TCP packet. The TCP protocol is responsible for assuring that the delivery of the data from the client to the server were correct. It detects transmission errors, and will retransmit packets if the packets contain data errors or if they are lost completely.

The TCP packet is further wrapped inside an IP packet. The *IP* protocol is responsible for sending packets between network equipment. When a packet traverses the network, it will hop between multiple routers on the path between the client and server. Based upon the IP address of the server, which is contained in the IP header, a router will decide the next hop router which it will relay the packet to. This process continues until the packet eventually reaches the web server.

Just like layer 3 uses IP addresses to route packets, a similar approach is used in layer 2. Layer 2 uses MAC addresses (Medium Access Control) to identify network interfaces. The MAC address is physically stored on the network interface, and every network interface being manufactured are given a

unique address. In order for a packet to hop between routers, the IP packet is wrapped inside an ethernet packet, and the MAC address of the next hop router is stored in the ethernet header. This address is then continually exchanged by every router on the path from the client to the server so that it will point to the next hop router.

We see that in order for the router to read the IP address in the IP header, it must first unwrap the lower level ethernet packet. Likewise, when the web server reads the content of the HTTP request, it must first ”unwrap” the headers of the three lower layers.

2.2 The HTTP protocol

The HTTP (HyperText Transfer Protocol) is an presentation layer protocol (OSI model), which is used when surfing the web. This protocol defines *how* data is sent, but not *what* type of data is sent. The HTTP protocol can therefore be used to send any data, not just web pages [3].

This protocol defines two different aspects of communication; how to query for data, and how to return the queried data. The response consists of a header followed by data. A query, however, consists of a header only. Below shows an example of what type of data is send when a web browser queries a web server, along with a description of the various parts of the request.

Example 1.
The browser asks for the URL to a website of a shareware computer game http://www.scuddendeath.com. This is the response header of this query.

```
GET / HTTP/1.1
Host: www.scuddendeath.com
User-Agent: Opera/9.01 (Windows NT 5.1; U; nb)
Connection: close
```

The web server sends it's reply. It firsts sends a response header:

```
HTTP/1.1 200 OK
Date: Fri, 04 May 2007 14:21:52 GMT
Server: Apache
Cache-Control: no-store, no-cache, must-revalidate, post-check=0,
pre-check=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Pragma: no-cache
X-Powered-By: PHP/4.4.4
Set-Cookie: PHPSESSID=36c9b811bfc753b4cfd6ffcb9c426084; path=/
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
```

After the response header comes the data. There is a blank line dividing the header and the data sections.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
   <head>
      <link rel="stylesheet" type="text/css" href="style.css">
      <link rel="SHORTCUT ICON" href="favicon.ico">
      <title>The Scudden Death 2 Website - Main</title>
      <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
   </head>
   <body>
```

More data follows, but are truncated for display purposes.

2.2.1 HTTP request

The first line of a HTTP requests consists of information about which specific resource on the web server is being requested. As we saw from the previous section, this is usually the string ”GET” followed by an URL (Uniform Resource Locator). There are three different version of the HTTP protocol, and below is an example of how this first line looks in these different versions [3]:

Example 2.
HTTP 0.9 (Also referred to as a 'simple request')
`GET` pagename
HTTP version 1.0
`GET` pagename `HTTP/1.0`
HTTP version 1.1
`GET` pagename `HTTP/1.1`

This query can be broken down into the *method type*, the *pagename*, and optionally the *version number*. In the example above, the method type is GET. However, other methods types exists, such as POST and HEAD. *POST* is used to transmit data to the server (Send values of forms, e.g username and password), where as *HEAD* will tell the web server to only return the response header, omitting the data section.

This first line is followed by the *request headers*, and a blank line to mark the end of the response. Several header types exist, and they are used to pass information to the server about the request or the client itself. Mansfield [3] describes the most important headers as follows:

- *Accept:* Contains a list of the types of data the client can accept e.g text/html, video/mpeg.
- *Host:* This header is only in use in HTTP version 1.1, and allows multiple web servers to be run on a single IP address. The value of this header tells which of the many websites this request is targeted at.
- *User-Agent:* Name and version number of the client's web browser.
- *Referrer:* If the user requests the page due to clicking a hyper link, this header will contain the URL of the referring page.

2.2.2 HTTP response

The first line of the response contains the version number of the HTTP protocol used by the server, along with the status code. The status code tells the client whether the request was successful or not. The status codes can be divided into three classes [3], and the first digit of the codes defines which class the code belongs in.

- *1xx* Informational - Request received, continuing process
- *2xx* Success - The action was successfully received and processed.
- *3xx* Redirection - Further action must be undertaken in order to complete the request.

- *4xx* The request contains bad syntax or can not be fulfilled.
- *5xx* The server failed to fulfill an apparently valid request.

Following the first status line, the server sends it's response headers. The most interesting headers are:

- *Server:* The name of the web server, e.g Apache or IIS
- *Content-type:* The current content type of the response, e.g text/html
- *Expires:* The date/time after which the content is considered to be "stale". If the data is cached, it should not be read from the cache after the expiration date.
- *Last-modified:* Date/time of when the file content was last edited.

2.2.3 HTTP 1.0 vs HTTP 1.1

The most important difference between versions 1.1 and the previous versions, are that HTTP 1.1 supports the use of *persistent connections*, in which the same TCP connection may be used to send several HTTP requests. In HTTP 0.9 and 1.0, one TCP connection was needed for each request. The use of persistent connections greatly increases performance, due to the overhead associated with setting up and tearing down TCP connections.

2.2.4 Dynamic vs static pages

When one requests a normal static HTML file, the file is simply read from the data storage of the web server, and the content is sent to the client for viewing in the web browser.

With the use of dynamic pages, the HTML that is sent to the client, is first generated on the fly by a script being executed on the server. Examples of such scripting languages are PHP (PHP Hypertext Preprocessor), ASP (Active Server Pages), and CGI (Common Gateway Interface). By the use of such languages, the content of the HTML can be derived from data stored in a database backend, and/or generated based upon user input.

The following example shows a PHP script calculating the 20 first numbers of the Fibonacci sequence, along with output of how it is displayed in the browser. The code is derived from example code found on *http://www.codecodex.com/wiki/Fibonacci_sequence*.

Example 3.

```
<?php
$n = 20;
$f = array( 1, 1, 0 );

echo "0 1 1 ";
for ( $i = 2; $i < $n; $i++ )
{
  $f[ $i ] = $f[ $i - 1 ] + $f[ $i - 2];
    echo "$f[$i] ";
}
?>
```

This is what is displayed in the browser.

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

The response time using static pages are easier to predict, compared to dynamic pages. If we simplify, we can express the response time formally as follows:

$$r_{static}(\text{file size}, \text{network speed})$$

$$r_{dynamic}(\text{file size}, \text{network speed}, \text{complexity of script})$$

We see that the response time for the static page is a merely a function of the file size and the speed of the network carrying the data from the server to the client. When using dynamic pages, another factor to consider is that it also takes time to generate the page. If the page is very complex, it will take longer time to generate it. This is referred to as a *spin delay*, because the server seems to *spin* before starting to send traffic on the network. However, in practice things are more complex than these formulas would suggest. The load on the server and the server's available CPU power is a factor to confider, in addition to the fact that one can have multiple physical hosts for each server (See section 2.4 about load balancing).

2.3 Push vs pull

Message passing on the Internet are divided into two fundamental architectural models: push and pull.

When a client browser requests a web server, the client may pull off web pages from the server at the client's own convenience, leaving the web server passively accepting the client's requests. This is called *receiver-pull* [6], because the receiver pulls information from the server.

The email service (SMTP [1]) works the other way around, in that the client may send emails to the server, having the server passively accepting the data. This is called *sender-push* [6].

Figures 2.2 and 2.3 shows the difference of these two architectures.

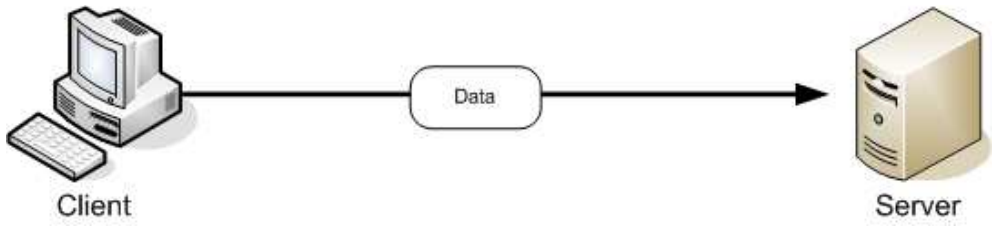


Figure 2.2: The client pushes data to the server.

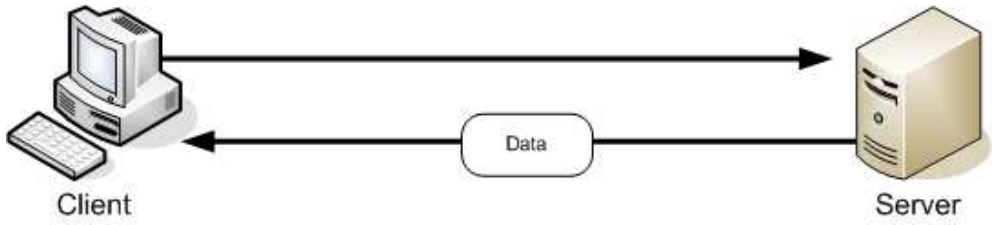


Figure 2.3: The client pulls data from the server.

A push based architecture leaves the server with little control over controlling traffic [6], and Duan et al [6] states that the choice between push or pull have fundamental implications on security, usability, and robustness, and concludes that a pull model should be selected whenever appropriate.

Burgess and Begnum [7] have addressed the problem of lack of server control by introducing the notion of *voluntary cooperation* and voluntary RPC [2]:

> In a traditional client-server model, the client drives the communication transaction ”pushing its request”. In voluntary RPC, each party controls its own resources, using only ”pull” methods.

The point to be made is that the act of providing a service is always associated with risk. Therefore communication between two parties is to be based upon trust. Burgess et al divides risk into three categories:

[1] Simple Mail Transfer Protocol
[2] Remote Procedure Call

- *Client's Service Level Agreement (SLA) expectations.* The clients do not expect to wait long for the server to process a request. The use of synchronously communication can be a source of Denial of Service (DoS) attack [3]. The server may also use up resources during peek loads, due to random variations in the network traffic.

- A server does not have control over the demand on its processing resources, but must adhere to the bidding of the clients. This also may lead to DoS attacks.

- The requests to the server, are accepted at "trusted" coming from the network, and does not check to see if the content are "secure". It may contain worms or code to create buffer overflow attacks.

2.4 Load balancing

This section will introduce different techniques and algorithms used when performing web server load balancing (LB). Load balancing can be categorized using two dimensions. The first dimension is about location, and the second dimension is about which scheme is used when the actual load balancing is performed.

Traffic can be balanced between computers in a Local Area Network (LAN) inside a data center, or it may be balanced on a global scale (Global load balancing), in which traffic is shared amongst computer centers across a Wide Area Network (WAN), which can span large geographical distances, e.g. between countries or continents.

Cardielli et al [8] divides load balancing into four schemes: Client-based, DNS-based, server based, and dispatcher based load balancing.

Using *client based* LB, the client itself performs the load balancing, and selects which server to send the request to.

Using *DNS-based* LB, the DNS servers decides which physical servers the URL will point to. The DNS server will balance load by varying the IP address returned when a DNS lookup is performed by the client.

Server based LB: The servers themselves have the possibility of redirecting the user upon requests. This will typically happen when the server load reaches a given threshold.

Dispatcher-based LB: A dispatcher functions as a proxy, hiding the different servers. The dispatcher selects which server will handle the request.

Cardellini [8] has identified pros and cons of the different LB schemes. Dispatcher based LB is the one in which the providers have the most control over

[3] A targeted attack against a server by a malicious user resulting in resource depletion, for the purpose of stopping the service.

the LB. However, dispatcher based LB will only work in a LAN environment. Client-based, DNS-based and Server-based will also work over a WAN (Wide Area Network). A more thorough list can be seen in table 2.4.

Scheme	Pros	Cons
Client	No server overhead	Limited applicability
	LAN and WAN solution	Medium-course grained balancing
DNS	No bottleneck	Partial control
	LAN and WAN solution	Coarse grained balancing
Dispatcher	Fine-grained balancing	Dispatcher bottleneck
	Full control	LAN solution
Server	Distributed control	Latency time intense
	Fine grained balancing	Packet-rewriting overhead
	LAN and WAN solution	

Table 2.3: Comparison of the pros and cons of the different load balancing schemes.

2.4.1 Load balancing algorithms

Several algorithms exists for selecting the server which is to receive the next request. These algorithms can be divided into two groups: static- and dynamic algorithms. Static algorithms tries to maximize the entropy distribution of the requests between the servers, so that each server will receive the same amount of connections on average. A common static algorithm is called Round Robin (RR), in which each server in turn is given the next request. Another static algorithm is to simply select the next server totally at random.

However, a problem emerges when using static algorithms. The computers may not be homogeneous, as they may vary in capacity, such as CPU clock are, available memory and so on. As a computer bought today is better than a computer bought six months ago, inhomogeneous computers are the situation in most computer centers today. In order to address this issue, it is possible to weight the different computers, so that a high capacity computer will get more requests than a slower computer. This enhanced version of RR is called Weighted Round Robin.

However, this algorithm does take into consideration that the requests may not be homogeneous; both when it comes to the size of the request and response, and the actual use of resources on the server. If the server provides dynamic pages, one request to a dynamic page may use significantly more resources than a request to another dynamic page, or compared to requesting a simple static HTML page.

Dynamic algorithms takes into account the the state of available resources on the servers when balancing the load. Ideally the dispatcher would have direct access to the state of each server, such as available memory, CPU etc. However, what is done in practice is to guess the load on the servers based on measure ables available to the dispatcher itself. Four dynamic algorithms have been described in scientific literature: Least connection, Round trip time, XmitByte and the baseline algorithm [9] [10].

Least Connections (LC). The server with the least number of ongoing connections is selected as the target for the next request.

Round Trip Time (RT). The response time of request is measured at certain intervals, and the server with the shortest average response time over a given interval of time is selected as the target of the next request.

XMitByte The number of bytes going to and from each server is counted, and a the server with the lowest average traffic rate is selected as the target of the next request.

Baseline algorithm (Least loaded) Assigns next request to the server that has the lowest workload; workload is defined as the sum of the service time of all requests pending on the server. However, this algorithm is difficult to use in practice [10].

Dispatcher based LB

This type of LB uses a dispatcher to forward requests to the individual servers in the web farm. The client sends HTTP requests to the IP address of the dispatcher, which acts as a proxy to the web farm. It is transparent to the client that there is actually several physical servers, as the dispatcher makes the web servers act as one unified entity. This section will describe various strategies and architectures associated with dispatcher based LB.

LB at layer 2 The dispatcher forwards packets to individual servers by setting the target address of a packet using the MAC addresses of the servers. The dispatcher does not change the IP address of the packet, only the MAC address is changed. Because of this, each individual server must be configured with the same IP address. In order for this to work, the servers must be configured so that they do not answer ARP (Address Resolution Protocol) requests.

A limitation of this LB strategy is that the servers and the dispatcher must be connected to the same physical network segment, by the use of a switch or a HUB. This means that this strategy can not be used over a WAN.

LB at layer 3 When the client sends a request to the dispatcher's IP address, the dispatcher replaces the target IP address of the request with one of the

server's IP addresses. This process is called Network Address Translation (NAT). When the server has processed the request, the server will send the reply back to the proxy, which translates the target IP address to the one of the client (Reverse NAT). However, the NAT requires that the dispatcher keeps state of each ongoing connection, and this might be a bottleneck if the traffic rate is high [8].

Another strategy is to use IP tunneling instead of NAT, in which an IP packet is wrapped inside another IP packet. In order for this to function, the servers must be spesificly configured to support tunneling. By the use of tunneling, the dispatcher can cope with higher request rate compared to when using NAT. This is due to the possibility of using DSR (Direct Server Routing).

Return path When the dispatcher has relayed a request to a server, and the server has successfully processed the request, there are various strategies of how the resulting reply is transmitted back to the client. Bourke [11] have defined three different types of return paths:

- *Bridge-path:* The dispatcher acts like a bridge, and works on layer 2. The server and the dispatcher must therefore be contained on the same network segment, in order for the switching on the MAC layer to work.
- *Route-path:* The dispatcher acts like a router, and thus works on layer 3.
- *Direct Server Routing (DSR):* The answer from the server is send back to the server directly without going through the dispatcher.

2.4.2 Redundancy

One important benefit of having load balancing, is a reduced probability that the service will be unavailable due to a server failure. If a single host fails, there are still other hosts that can provide the same service.

If the availability of a single hosts is 99 per cent (A down time just over seven hours each month), which gives the probability of failure $p = 0.01$. If we have a total of two hosts to provide the service, and assume that the probability that one computer fails is independent of the failure of another computer, the probability of both computers failing at the same time would be $q = p^2 = 0.0001$. This would give a down time of just four minutes per month.

Additionally, due to the fact that failed hosts reduces the overall resources available to the system, the loss of a host will come with a cost of increased response time.

Another issue to consider is that the load balancer itself would prove to be a single point of failure. If the load balancer fails, no traffic will reach the servers. In order to combat this, an additional load balancer may be used as a

backup. By the use of heart beat messages, the other load balancer host will be able to take over the the responsibilities of the first dispatcher in the event of a failure.

2.4.3 Queueing theory

Requests to web servers can be modelled as a stream of requests, in which these requests are added to a queue. The use of queueing theory is useful for modelling the performance of web servers [12, 13, 12]. The rate of requests to a server is a random process, which can be described as an average rate of λ requests per second. The requests which is added to the queue is processed by the server at a rate of μ jobs per second.

In the book "Handbook of Network and System Administration", to be published in late 2007, Burgess describes the following about the mathematics behind the prototype $M/M/1$ queue [14]:

> Queues are classified according to the type of arrival process for requests, the type of completion process and the number of servers. In the simplest form, Kendall notation of the form $A/S/n$ is used to classify different queueing models.
>
> - A: the arrival process distribution, e.g. Poisson arrival times, deterministic arrivals, or general spectrum arrivals.
> - S: the service completion distribution for job processing, e.g. Poisson renewals, etc.
> - n: the number of servers processing the incoming queue.
>
> The distribution of inter-arrival times for both A and S is normally considered to be a Poisson distribution in discrete time. One writes this $M/M/n$. This assumption is made in order to simplify the analysis. M stands for "memoryless" because the Poisson distribution, taken over non-overlapping time intervals is a Markov process whose behavior, at any discrete time t, is quite independent of what has happened in the past, i.e. it has no memory of the past. The provides a huge simplification of the analysis.
>
> Another reason for the Poisson assumption is that, in the limit of large numbers of independent arrivals, one would expect the limiting distribution to have a Poisson form. Suppose the probability of obtaining a result is fixed and is equal to p on each independent observation, then the probability of obtaining q positive arrivals in n observations is

$$P(q) =^{n} C_q \; p^q (1-p)^{n-q}, \tag{2.1}$$

where ${}^nC_r = n!/(n-r)!r!$ are the binomial coefficients. This is a binomial distribution whose mean value is $\overline{q} = np$. Now, suppose the probability of observing an arrival is scarce i.e. $p \to 0$, but we consider the limit of long times or many observations $n \to \infty$, taking the limit in such a way as to make $np \to \lambda$, where λ is a constant. Then, noting the limits,

$$\begin{aligned} {}^nC_r &\leq \frac{n^r}{r!} \\ {}^nC_r &\geq \frac{n^r}{r^r} \\ \lim_{\alpha\to\infty}(1-x)^\alpha &\to \mathrm{e}^{-\alpha} \quad \text{where } (x<1). \end{aligned} \tag{2.2}$$

one observes that

$$P(q=k) \leq \frac{n^k}{k!}p^k(1-p)^{n-k} \to \frac{(np)^k}{k!}(1-p)^{n-k} \to \frac{\lambda^k}{k!}\mathrm{e}^{-\lambda}, \tag{2.3}$$

which is the the Poisson distribution, for which one verifies that $\sum_{k=0}^{\infty} P(q=k) = 1$.

This widely held belief is somewhat controversial, however, as measurements of network traffic have shown evidence of considerable "burstiness", or long-tailed behavior[15, 16, 17, 18]. Other work indicates that these contradictory measurements would in fact settle into a Poisson distribution if only enough measurements were taken[12, 19]. However, it is estimated that something of the order of 10^{10} transactions might be needed to see this limiting form emerge.

Regardless of this controversy, the Poisson model survives in queueing theory for its overriding simplicity.

$M/M/s$ queues

It is possible to have multiple queues and servers, which will help to reduce the total response time of the system. If s servers have their own queue, we will have s queues in parallel. This is notated $M/M/s$, because we in essence have s number of $M/M/1$ queues. If s servers instead shear a single common queue, only the servers will be in parallel. The difference between these two strategies is shown in figure 2.4

Burgess [14] describes as follows:

> Then, let n be the number of unprocessed requests in the queue at time t, and suppose that requests for transactions arrived at a rate of λ per second, and can be processed at a rate of μ per second.

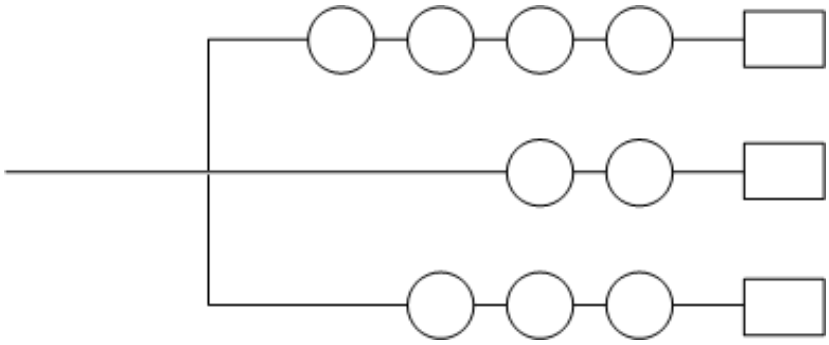


Figure 2.4: $M/M/k$: All of the servers processes the same queue.

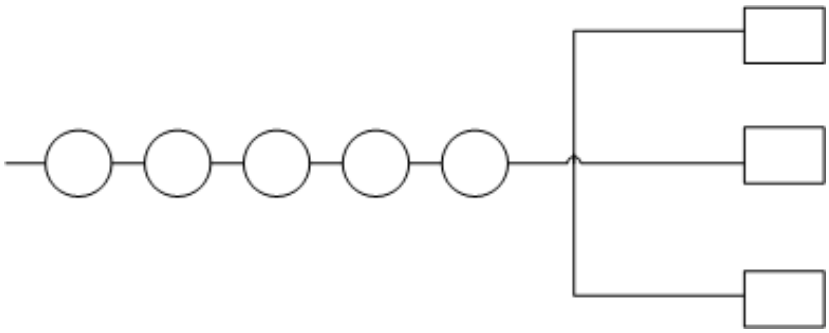


Figure 2.5: $M/M/1^k$: Each server processes requests from their own queue.

We can treat this simple case as a continuum flow approximation, using expectation values.

Consider any time t: the system makes a transition from a state of queue length $n-1$ to n at a continuous rate λ. Similarly, when it is in state n, it makes transitions to a state of length n at a rate μ. The expectation of λ arrivals per second, when the queue length is in a state $n-1$ (for any n) must therefore lead to the expectation of having μ completions when the state is n, in order to balance the queue, so we write, on balance $\lambda p_{n-1} = \mu p_n$ or

$$p_n = \rho p_{n-1}, \tag{2.4}$$

where $\rho = \lambda/\mu < 1$ is called the traffic intensity[20, 21]. This is a recurrence relation that can be solved for the entire distribution p_n, for all n. One finds that

$$p_n = (1-\rho)\rho^n, \tag{2.5}$$

and hence, the expected length of the queue is

$$\langle n \rangle = \sum_{n=0}^{\infty} p_n n = \frac{\rho}{1-\rho}. \tag{2.6}$$

Clearly as the traffic intensity ρ approaches unity, or $\lambda \to \mu$, the queue length becomes infinite, as the server loses the ability to cope.

This situation improves somewhat for s servers ($M/M/s$), where one finds a much more complicated expression. In simplified form one has

$$\langle n \rangle = s\rho + P(n \geq s)\frac{\rho}{1-\rho} \tag{2.7}$$

where the probability that the queue length exceeds the number of servers $P(n \geq s)$ is of order ρ^2 for small load ρ, which naturally leads to smaller queues.

It is possible to show that a single queue with s servers is at least as efficient as s separate queues with their own server. This satisfies the intuition that a single queue can be kept moving by any spare capacity in any of its s servers, whereas an empty queue that is separated from the rest will simply be wasted capacity, while the others struggle with the load.

2.5 Message Queues

A message queue (MQ) is an architecture strategy for asynchronous message passing, in which messages are put in a message queue, instead of being send directly to the server. Asynchronously means that the server and the client do not need to be operative at the same time in order to send a message or read a message from the queue. This is different from synchronous message passing, like web services (HTTP), in which the client sends a request to the server and then waits for a reply to come.

When the client sends a message to the message queue, it will not expect an immediate reply, but instead the message will stay in the queue until it is removed by a reader. It is possible to have several servers reading from the same queue, thus enabling load balancing [22] [23]. Message queues may also be used in a publish/subscriber pattern, in which several recipients can read the same message. The message will then be removed after a specific duration of time, and not upon the first read.

Several commercial and open source message queue frameworks has been made, e.g. Websphere MQ from IBM, and JBoss Messaging.

Chapter 3

Motivation

A prudent question is one-half of wisdom.

Francis Bacon

3.1 Motivation

The ideal load balancer would know the load on each server at all times, and use this to correctly balance the load between the hosts, so that each physical host would get a fair share of the total load. However, both the processing need of each request differs, as well as the current load of each hosts. While round robin (RR) and random selection algorithms totally ignore these problems, LC, RT and XmitByte try to address this problem by doing measurements of properties of the current traffic to the server.

The RT algorithm measures the round trip time at configurable intervals, and the job of finding the optimal selection of this value is not trivial. A too high interval might not be adaptive enough to reflect the true load on the server, while it has been shown that a low polling interval leads to a degregation in performance [24].

Because the measurements are not performed on the actual hosts, these algorithms only provide a qualified ”guess” of the current load on the server. They do not reflect the load well enough to be a good enough substitute for the simple round robin approach [9] [24].

All of the dispatcher algorithms listed in the previous section pushes requests to the server, leaving the sever with little or no control over when to process the new requests. This is clearly true with the static algorithms, in which the next server is selected purely on the basis of achieving a high entropy distribution of the traffic between the servers. The dynamic algorithms also pushes request to the server, however, after a qualified guess of the current state of load and resource utilization of the servers. In order to do this,

these algorithms must make certain assumptions of the traffic characteristics and the processing time of the server.

None of the push based algorithms can guarantee that a server will not receive requests while still under load. When a request is dispatched from the dispatcher to a server that is under high load, the request will be queued at the server. However, if there are other idle servers, it would be more beneficial to the response time if that request was handled by the idle server instead of waiting in the queue. This might lead to over-utilized and saturated servers, due to bad entropy.

3.2 An ideal Pull Based Load Balancer

3.2.1 Design

This section will introduce the design of a theoretically ideal pull based load balancer; a new approach to load balancing. It will explain how the load balancing strategy will work, and what benefits this dispatcher will potentially have over the other load balancing (LB) strategies.

In computer science, there is a principle that when one are to solve a problem, or information has to be retrieved, one should always use an expert [4]. While the dispatcher in the traditional LB strategies does a qualified prediction of the state of the individual servers, it does not know the true state of the resource utilization. The experts in reading the server's state would be the web server hosts themselves.

Our approach is as follows; instead of using a dispatcher to balance load between the servers, the load balancing is performed by the individual server hosts. This way, the authoritative decision of whether a host should receive a request or not, is moved from the dispatcher and given to the web server itself. In this approach, we exchange the dispatcher with a central queue, in which all of the HTTP requests are queued after they have been issued by the clients. Each of the servers will pull requests from this queue at the server's own convenience, which is whenever the server hosts have enough free resources to process a request.

Figure 3.1 shows a simple diagram of the components of this push based LB strategy. This system consists of three entities: The end client's web browser, the web servers of the web farm, and the proxy in the middle, which queues the requests coming from the client.

It is not clear which of Cardellini's [8] categories this load balancer belongs in, as it would be a hybrid between a dispatcher and a server based load balancer.

Figure 3.2 shows an activity diagram of how the load balancing works. This diagram is explained in detail as follows:

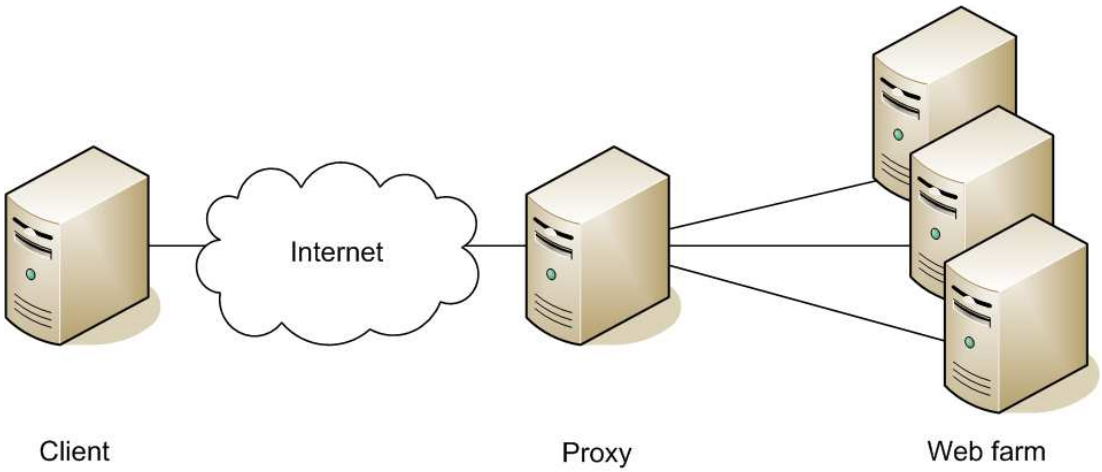


Figure 3.1: The different components of the load balancer system.

1. The client sends a message to the proxy, and it is added to the queue.
2. A web server polls the proxy.
3. The request is removed from the queue and is sent to the web server. The requests are removed from the queue in a FCFS (First Come First Served) fashion.
4. A reply is send from the web server to the proxy.
5. The proxy relays the response back to the client.

In order for a server hosts to know when to process a request, it will constantly measure it's own health, and thus use various metrics of the state of the system in order to make decisions as to whether or not the web server should process new requests, or instead wait until the health improves. If no requests are in the queue, the server will be idle. But as soon as a request is inserted into the queue, the requests will be processed by the first server which removes the requests from the queue. If the request rate surpasses the response rate, the requests will be queued on the proxy, and the queue will be processed by the servers at the server's best convenience. The health check would guarantee that the load balancing would be fair, and that no web server would have too much to do, thus becoming too saturated.

3.2.2 Theoretical model

Load balancing of web servers can be modelled using memoryless queues [25, 13]. A traditional push based load balancing strategy can be described

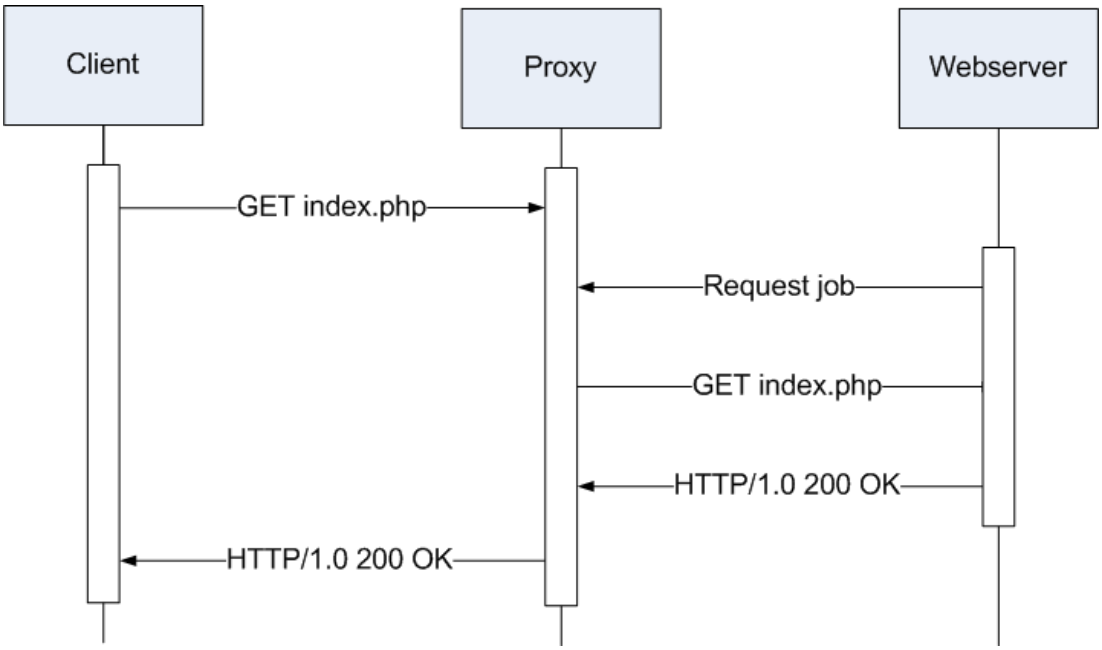


Figure 3.2: An activity diagram showing how the pull based load balancing works.

by the $M/M/1^k$ queueing model [24], in which requests are queued at each individual server, thus having several $M/M/k$ queues.

A pull based based LB strategy can be described formally by the $M/M/k$ queueing model, due to the fact that the requests are queued at a central queue, instead of each at each of the individual servers. As queueing theory and the folk theorem about redundancy states that a $M/M/k$ queue is at least as or more efficient than a $M/M/1^N$ queue [20], a pull based LB strategy should intuitive give shorter response times than a simple push based strategy.

3.2.3 Goals of an ideal pull based LB

The two previous sections can be summed up as a set of goals of which an ideal pull based load balancing strategy should satisfy.

- Jobs are queued only at central queue, as this is the most efficient queueing strategy.
- Each web server must run a health check, so that they only process new jobs when they have available resources to do so.
- The overhead of the proxy is negligible, and thus will have no significant impact on the total response time.

However, as this is an ideal strategy, a real implementation of this load balancer would not meet all of the goals 100 per cent. This is perhaps most evident in the third point. However, if the overhead of the queue proxy is no greater than the overhead of a normal dispatcher LB, this point would still be valid.

3.3 Proof of concept implementation of a pull based LB strategy

3.3.1 Introduction

This section will a address the actual implementation of a push based LB developed and used in the experiments of this thesis.

In order to compare pull and push based servers or load balancing algos, two simple implementations of a web server will be implemented. One as a traditional push based server, while another is pull based; polling the request from a central queue proxy. A simple round robin (RR) dispatcher will be implemented, in addition to the central queue proxy. All of the software is written in Java version 1.5.

3.3.2 System Components

This section will describe the responsibilities of the various components of the system. The components are divided into two main packages, one for the web servers, and another for the proxy and dispatcher components. A more thorough explanation of the system, by the use of source code, can be found in section A.

Web servers

There are two types of web servers: type A and type B. Type A works just like a traditional (push based) web server. Type B is a pull based web server.

Both types of web servers will use the same HTTP handler. This handler will be implemented only to support HTTP version 1.0. This means that the server will not support persistent connections. The HTTP handler will only be able to answer simple GET requests, and will not support HTTP methods such as POST or HEAD.

Both web server types spawn worker threads when a job needs to be processed. The workers are not preforked, and there are no logical limit as to how many worker threads that can maximally be spawned at one given time.

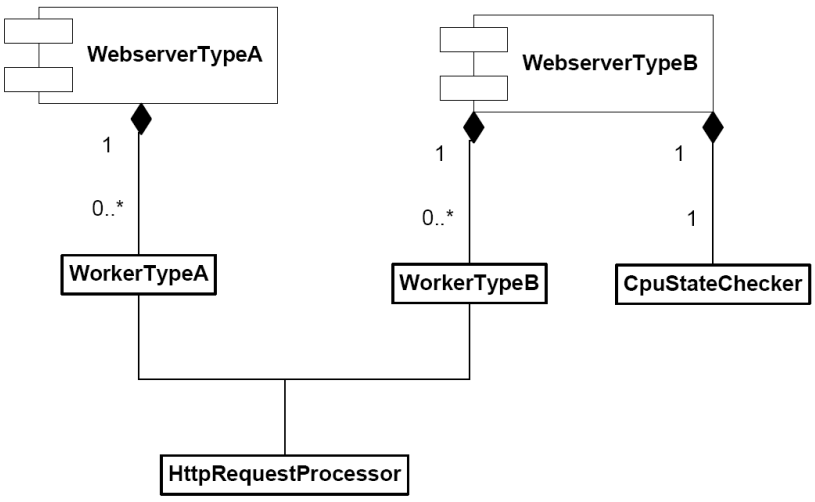


Figure 3.3: UML diagram showing the parts of the two types of web server components.

In order to simulate dynamic pages[1], the server will run a randomized spin delay with a configurable mean spin value. The spin delay is normally distributed, with a standard deviation σ, equal to $\frac{1}{3}$ of the mean spin value μ. Figure 3.4 shows the normal distribution. This will give greater variances of the spin as the average spin delay increases, producing a spin delay which roughly varies in the range $[0, 3\mu]$.

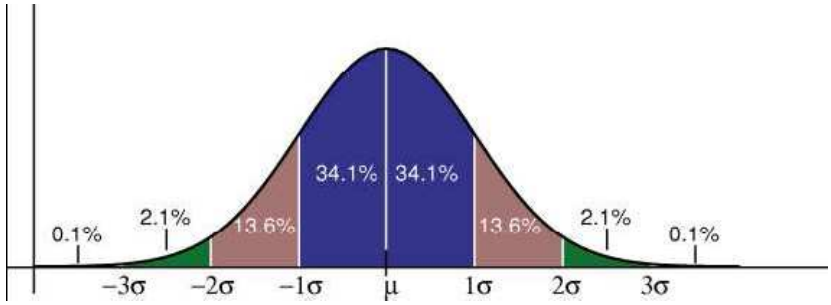


Figure 3.4: For a normally distributed dataset, 68.3 % of the values will be within one standard deviation from the mean, and 95.5 % of the values will be within two standard deviations from the mean value.

Available computer resources can be measured using both available memory and from the current CPU utilization. However, as server computers are assumed to have large amount of memory, CPU utilization will be chosen as

[1]PHP, ASP, CGI

the single metric to validate the status of the host computer's health in this design.

When a web server type B is run, a thread is spawned that constantly keeps reading output from /proc/stat [2] at configurable intervals using native OS-depended system calls, and converts this values into a per centage CPU utilization value. This metric will be used by the web server to check if it should process new requests, or wait until a currently working thread are finished processing a request.

Note that the design of how the CPU utilization is measured makes this software only function on the Linux platform, despite the fact that Java itself is meant to be platform independent. However, as there are no standard APIs in Java for measuring the CPU rate of the host computer the VM is running on [26], Mikhalenko [26] has developed a library which uses Java Native Interface (JNI) to transfer data from OS depended system calls to Java. Due to difficulties of retrieving the library, and uncertainties as to whether it supports the Linux platform (Description of library only states support for Windows and Solaris), it was chosen to implement it using direct native calls.

Proxy components

Two types of proxy components will be used. One proxy will act as a traditional round robin (RR) load balancer dispatcher, while the second proxy will be a central queue.

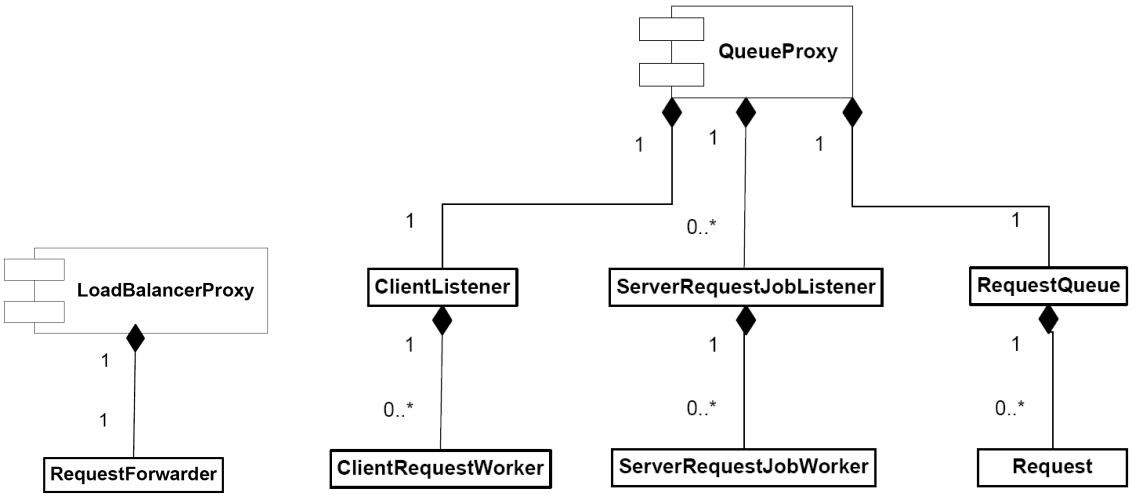
The queue proxy has two ports open all times. One port is used by the ClientListener for listening to client requests (Port number 8080 by default), and the second is used by the ServerRequestJobListener to listen to polls from the type B web servers (Port 8081 by default). Both the client and server listener spawns worker threads (ClientWorker and RequestJobWorker, respectively) upon client and server connections.

Additionally, callback ports are also opened for each request in the queue by the ClientListener. This port is used for listening to replies from the web servers, once the requests has been read from the queue and processed by a type B web server.

Activity and communication

This section will describe the communication between the queue proxy and the type B web servers, and internal activity of the two components. Three different activities will be described. How the web server's connect to the proxy, how the client queries the proxy, and how the servers process the messages in the proxy queue.

[2] Contains information about the current utilization of the CPU



(a) (b)

Figure 3.5: UML diagrams showing the parts of the two types of proxy components.

Figure 3.6 shows a sequence diagram of how the web servers connect to the proxy. This diagram is explained in detail as follows:

1. First the proxy is started, and is ready to listen to polls from the web servers.

2. A web server opens a socket connection to the proxy, and a ServerRequestJobWorker thread is spawned in order to handle future job requests from this server.

3. The web server sends a poll message stating that it is ready to process a job.

4. If a job exists in the queue, the job will be removed from the queue, and is then send to the web server. If no jobs are currently in the queue, the proxy will add the server's ServerRequestJobWorker to a list of active workers. Upon arrival of a new HTTP request from a client, the proxy will send the request to a web server based upon the ServerRequestJobWorkers that are in the list of active workers.

Below shows the content of a poll message send from a type B web server.

Output 1. `Request-job: true`

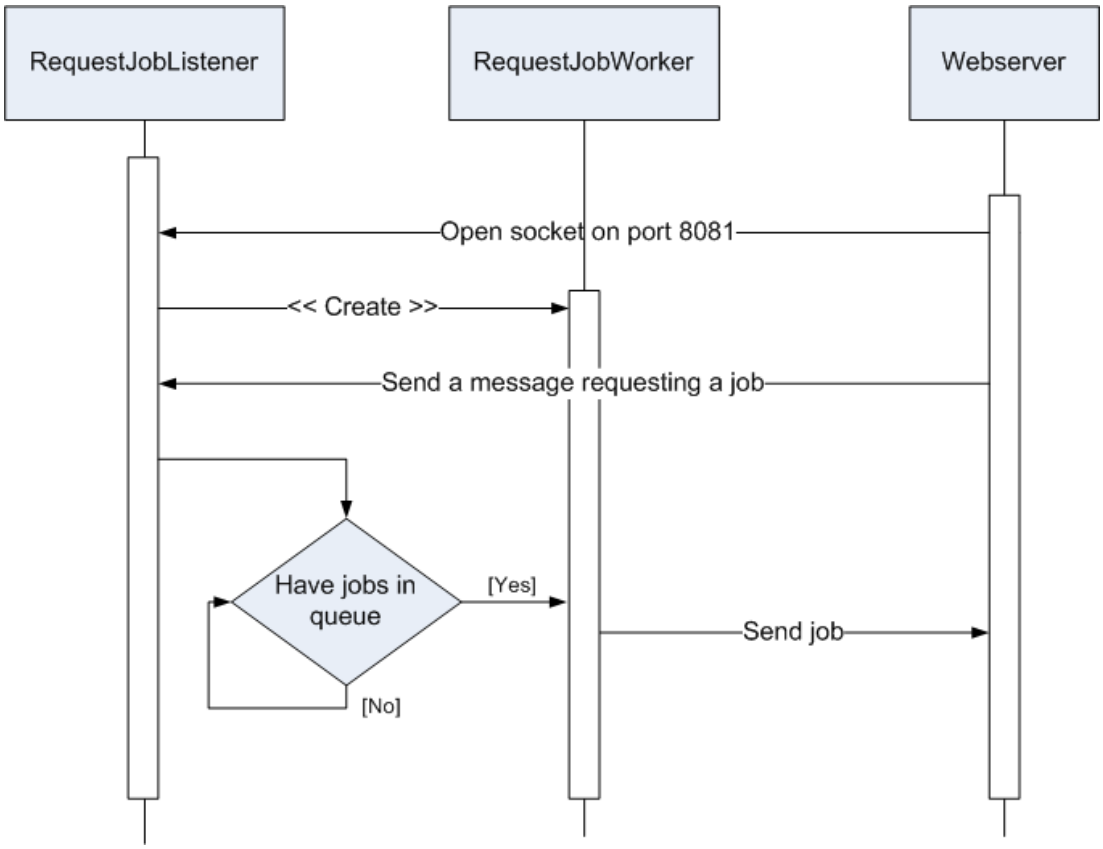


Figure 3.6: Sequence diagram showing the flow as a web server connects to the proxy

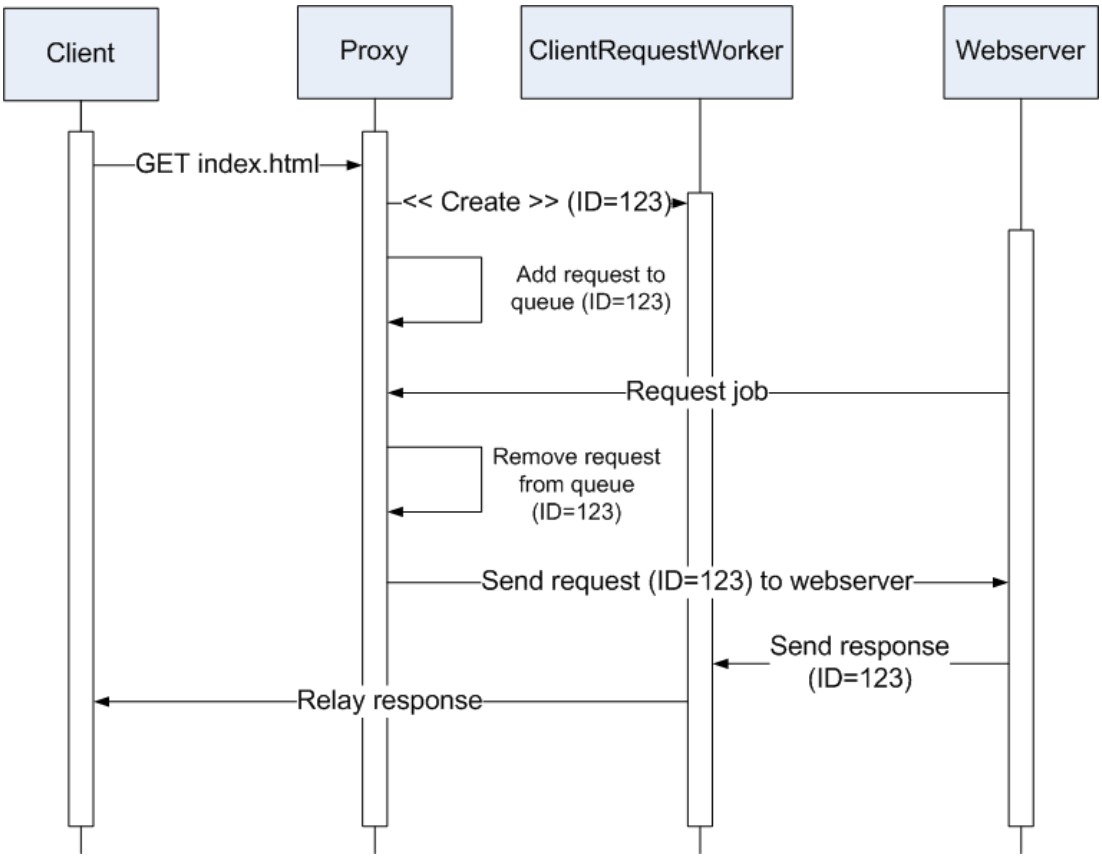


Figure 3.7: Sequence diagram showing the flow as a web server connects to the proxy

Figure 3.7 shows a sequence diagram of how a request from the client is queued by the proxy. Next is a detailed explanation of this sequence of events.

1. A client's web browser sends a HTTP request to the proxy.
2. The proxy spawns a client worker, which opens a port on which to expect an answer from the web server. This port number is to be used as a identifier, unique identifying the current request.
3. The client's request is stored in the proxy queue, and identified by the unique identifier.
4. An available web server polls the queue and removes the request from the queue.
5. The web server processes the request, and returns the answer by connection to the right port number on the proxy hosts.
6. The response is relayed to the client.

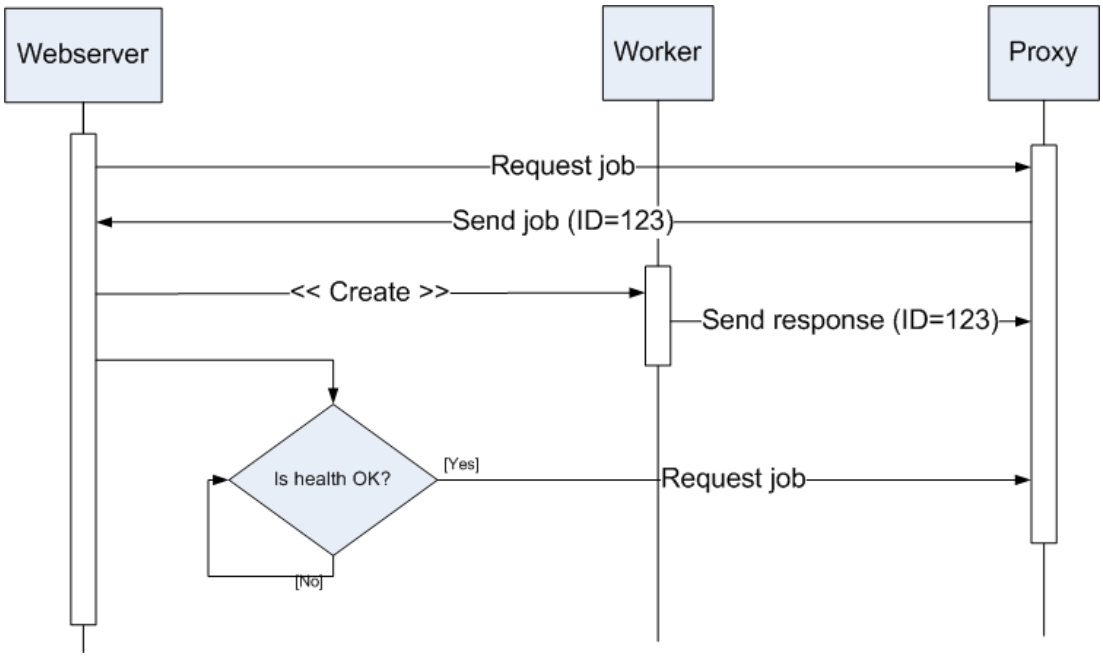


Figure 3.8: Diagram showing the flow as a web server connects to the proxy

Figure 3.8 shows the activity of how the server queries and processes jobs from the query, and how the request is relayed back to the client. Below is a thorough explanation of the steps in the flow diagram.

1. The web server polls the proxy in order to receive a new job.
2. The web server spawns a worker thread that processes the request.
3. When a worker thread is finished processing the job, it will open a connection to the proxy on the same port number as the identifier of the request, and transmit the result of the request.
4. If the computer health is good, a new poll message is sent to the proxy. If not, the web server will suspend sending a poll message until a worker thread is done with a job, and then do another health check. If the health has improved and is at a level considered to be acceptable, the server will send a new poll message. If the health has not improved, it will enter the suspension loop again.

In order for the proxy and web servers to communicate the identifier of the current request, additional headers are added to the HTTP request. Below shows an example of a HTTP header which the web server receives from the proxy, which contains an additional header variable at the top. This variable contains the request identifier, and is used in order for the web server to know which port on the proxy to connect to when the request has been processed.

Output 2.
```
Client-Worker-Port: 20001

GET /index.php HTTTP/1.0
Accept: image/gif, image-x-xbitmap, image/jpeg, image/pjpeg
Accept-Language: no
Accept-Encoding: gzip, defate
User-Agent: Mozilla/4.0
Host: localhost
```

3.3.3 Final notes about the design

The system was designed so that it could be used without need for the client to be altered. If the system was designed following a pure cooperative RPC design, the client would have to be altered as well. The voluntary RPC was only implemented by the proxy and end servers, and from the perspective of the client, the system would work as in a traditional RPC fashion.

The programming language Java was chosen based upon a belief that this language would yield a shorter development period, compared to using a lower level language, such as C or C++. However, Java has some shortcomings when it comes with dealing with network streams. As there are no low level functions for working with streams at a level lower than sockets, the LB

algorithms implemented was not as efficient as it might have been. Had the software been developed in C or C++ there would have been possible to manipulate packets directly, e.g. changing IP source and destination. Instead, the whole HTTP request was read before being relayed to a server.

However, there exists a Java package called *Jpcap* for capturing and sending packets on a low level scale which is based upon *libpcal* or *winpcap* Raw Socket API. It will work on any operating system which has either of these libraries implemented. However, although *Jpcap* can listen to streams, it may not alter the content of the packet data or headers [27]. It would merely work like a network logger, being similar to *tcpdump* [3].

Due to the fact that the load balancer reads the whole HTTP request when performing LB, it might be considered to be a 7 layer load balancer. However, as no information in the HTTP request is used when performing load balancing, and no mean is taken to address the issue of handling session state variables, it would be sound to consider this as a layer 3 load balancer.

[3] A commonly used networking debugging tool.

Chapter 4

Related research

> An expert is a man who has made all the mistakes which can be made in a very narrow field.
>
> Niels Bohr

Several papers have been written in the late 90s and early 2000s investigating the differences between the various load balancing (LB) algorithms and techniques. They have been trying to find the optimal load balancing strategy, and the best strategy under various degrees of load. They have mainly been focusing on the response time as a metric to describe web server performance.

Research by Bryhni et al [9] lead to the recommendation of the round-robin algorithm, however being positive to the RoundTrip algorithm, with the prerequisite that it have better prediction of the current load. Additionally, Teo et al [10] found that there was no difference between the round-robin and LC algorithm under high load. However, under low to medium load, the LC algorithm yielded faster response times. This contradicts the results by Undheims [24], in which the LC was performing worse than RT and RR at low loads. At high loads, the RT algorithm outperformed RR.

The focus has been mainly on testing the load balancing techniques on homogeneous equipment [9] [10]. However, Undheim [24] also compared the LB techniques on inhomogeneous equipment. The conclusions were that the RT and RR algorithms were similar in performance, but were both outperformed by the LC algorithm. It was concluded that that the LC algorithm distributed the load more fairly when the computers had different processing capabilities.

Not much research has been done on a load balancing strategy in which the server's themselves participate in the distribution of the requests. However, US patent 6023722 [28] describes an architecture, very much alike the one presented in this thesis, in which the messages are distributed using a centralized message queue.

While it is uncertain how much the patent is to blame of why there has not been much research on this topic, we see that Burgess notion of voluntary RPC

[7] shares the same principles that are used in the pull based LB strategy; in which the servers have superior control over their own resources.

Chapter 5

Hypothesis

> Your theory is crazy, but it's not crazy enough to be true.
>
> Niels Bohr

In this section the hypothesis of the thesis is stated. The hypothesis are deduced from theoretical thinking about the pull based load balancer; how it is designed and how it designed to function. The hypothesis will reflect what we expect the results from our experiments to be, and the possible reasons for the hypothesized results.

Hypothesis 1. *The queue proxy will introduce some overhead compared to running a RR load balancing strategy.*

As the queue proxy is considered to be more advanced than a RR dispatcher, this is a valid assumption.

Hypothesis 2. *The queue proxy of the pull based LB strategy will not become the bottleneck, as long as the processing time of the server is longer than the processing time of the proxy.*

As long as the dynamic pages are more resource consumptive than the overhead of the proxy, most of the response time measured by the client will be introduced by the web server. Therefore the servers will become the bottleneck of the system before the queue proxy does.

Hypothesis 3. *The pull based LB strategy will give a more fair weighting of the web-servers, and thus utilize the total resources of the web-servers better.*

As the individual servers are only processing requests when they have available resources to do so, other servers with available resources will process the job instead, thus utilizing the total resources of the web-servers better.

Hypothesis 4. *Due to the overhead of the proxy, push based LB is better when the average processing time of the dynamic pages are low, but as the average processing time increases, pull based LB will at some point surpass the push based LB strategy, due to more fair weighting.*

When the average processing time is low, the probability of having a busy server when a request is issued from a client is low, but as the average processing time increases, this probability increases. As this probability increases, the need to address the requests to an idle server presents itself. Because the processing time on a web server is a random process, it will at times both be shorter and longer than the average value. Thus, there would be idle servers (Or servers soon-to-be idle) that are better suited to process the request. Because a new requests is not processed by a server that is busy, but instead by an idle server, the response time will be improved.

Hypothesis 5. *When the individual web server hosts have inhomogeneous hardware, the differences between push and pull based LB will be even greater than when the hardware is inhomogeneous.*

As the capacity of each individual hosts is not equal, a simple RR LB strategy would not weight fairly. A pull based LB strategy would consider the capacity of each individual server when performing the LB.

Chapter 6

Experimental Design

> Science must begin with myths, and with the criticism of myths.
>
> Karl Popper

6.1 The Scientific Method

Humans are curious by nature, and philosophers have always asked questions about the world. We might have certain believes, or hypothesis, of why things happen, or how things works or will work.

Research is a process in which we try to find answers to these questions. As science can never know the absolute truth about the world, this process is merely about finding *suitably idealized approximations* to given problems [14]. However, in order to deal with criticism with some authority, one states the degree of uncertainty of these approximations.

The first step in the scientific method is to theorize. Here we state our believes, as what in science is refereed to as hypothesis, and we also create models of the system we are researching.

Figure 6.1: The process of the scientific method.

A model can be created for two reasons; we try to test if our assumptions lead to the outcomes we imagine, through a chain of cause and effect, or we try to predict new scenarios that has never been observed before, based on past observations and modelling [14].

The second step is to collect measurements of the system. As one of the philosopher of science Karl Popper notion of falsification states; one can never really prove a theory, but one can falsify a theory by a single counter example based upon measurements.

All measured data will include some degree of some uncertainty. This uncertainty is due to errors, which is categorized by two types: Random or systematic error. Random error affects the measured data randomly, adding or subtracting the measured value. An example of this is background noise. Systematic error is a constant shift in the value of the measured data, due to wrong calibration. This would make all measured values wrong by the same amount.

The last step is to process and analyze our data. Based upon this analysis, we state our conclusion. We can either state that the measured results were predicted by our model, or that it was falsified from the evidence of the measured data.

The scientific method can also be referred to "The research loop" [29], in which one follows the following sequence of actions.

```
ignorant=true;

while (ignorant || alive)
   {
   Assess Motivation and Subject;
   DoMeasurements/Experimentation
   Interpret results
   Criticize interpretation
   if (results interesting)
      {
         Communicate results
      }
   }
```

Note that this loop does not end, because research is a constantly ongoing process. As another philosopher as science, David Hume, says; measurements collected from one experiment might disprove conclusions drawn from measurements collected from another experiment. This shows us that science must be humble, in that we must make do with the most suitable approximation to describe given phenomenon, as we can never be sure that what we have measured is the "truth".

6.2 Setup

This section will describe the hardware and network setup for the experiments undertaken during the work in this thesis. Figure 6.2 shows a network dia-

gram of the setup in the lab.

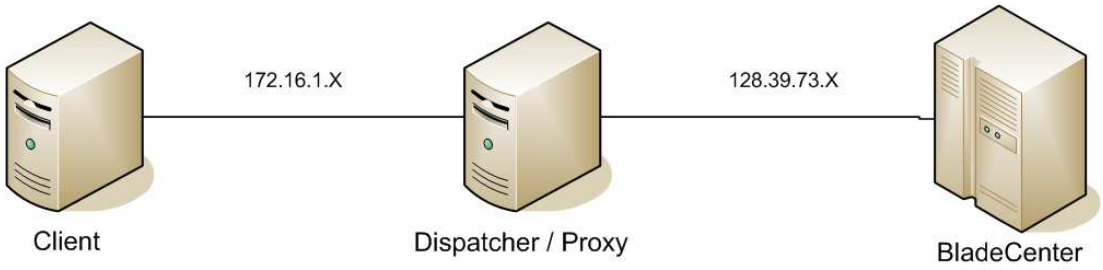


Figure 6.2: Basic lab setup

All of the computers are connected using gigabit ethernet, in order to avoid network congestion during tests. The computers in the lab consists of a client computer, a computer acting as a dispatcher/proxy, and a web farm.

The web farm is created using an IBM BladeCenter. The BladeCenter consists of six blades, whereas only five were functioning at the time of the tests. The client and dispatcher/proxy computers are regular computers. A detailed explanation of the different computers can be seen in figure 6.2

Computer	CPU	Mem
Blade server	Intel Xeon 2.8 GHz	1 GB
Client	Pentium 4 1.7 GHz	512 MB
Proxy/Dispatcher	Pentium 4 2.8 GHz	2.5 GB

Table 6.1: The hardware specifications of the hosts in the lab.

6.3 Web server type A and B

For the tests, we use the Java based web servers, described in section 3.3. We use both the pull- and push based types (Web server type A and B). This section will describe the command line parameters to all of the runnable components of the system.

Below shows how to start the push based web server (Type A) and the pull based web server (Type B).

Example 4.

```
java webserver.WebserverTypeA --port 8080
                              --spin 100000 -v
```

Example 5.
```
java webserver.WebserverTypeB -a 192.168.1.5 -p 8181
                              -s 10000 -m 100 -i 250 -v
```

–port or -p sets the server's port number, which is used for listening to client requests.

–spin or -s sets the Poisson distributed average number of iterations that the server will spin each request. The standard deviation is 1/3 of the value of the spin.

-m or –maxcpuload sets the maximum CPU load threshold before the server stops accepting more jobs as a per centage value.

-i or –cpumeasureinterval sets the interval in milliseconds of which to calculate the average CPU load of the server host.

Below shows how to start the queue proxy.

Example 6.
```
java proxy.QueueProxy -c 8080 -r 8181 -v
```

-c or –clientlistenerport sets the server's port number, which is used for listening to client requests.

-r –serverrequestjobport sets the server's port number, which is used to listen for job requests from the web servers.

Below shows how to start the load balancing proxy.

Example 7.
```
java proxy.LoadBalancingProxy -c 8080 -a RR
                              -s blade1:8080,blade2:8080
```

-c or –clientlistenerport sets the server's port number, which is used for listening to client requests.

-a –loadbalancingalgorithm states which LB algorithm to use. Currently only Round-Robin is supported.

-s –servers states which web servers to perform load balancing between.

For all of these components, *-v, -vv or -vvv* will make the component produce various degrees of verbatim output.

6.4 Tools

6.4.1 httperf

Httperf is a tool developed by HP Research Labs [30], which use is to measure web server performance. It supports both HTTP version 1.0 and 1.1, and it's basic operation is to generate a fixed number of HTTP GET requests and to measure the reply rate from the server [31]. Next is an example of usage:

Example 8.

```
httperf --hog
        --server mywebserver
        --port 8080
        --num-conns 1500
        --num-calls 10
        --rate 100
        --timeout 5
```

The description of each parameter is taken from the httperf man page [31]:

–hog tells httperf to use as many TCP ports as necessary. Without this option, httperf is limited to using ephemeral ports (in the range 1024 to 5000).

–num-conns specifies the total number of connections to create.

–num-calls specifies the number of calls to issue on each connection. If this value is more than 1, the server must support persistent connections (HTTP/1.1).

Here is an example of how the output from httperf looks like:

Example 9.

```
Total: connections 30000 requests 29997 replies 29997
test-duration 299.992 s
Connection rate: 100.0 conn/s (10.0 ms/conn, <=14
concurrent connections)
Connection time [ms]: min 1.4 avg 3.0 max 163.4 median 1.5
stddev 7.3
Connection time [ms]: connect 0.6
Request rate: 100.0 req/s (10.0 ms/req)
Request size [B]: 75.0
Reply rate [replies/s]: min 98.8 avg 100.0 max 101.2
stddev 0.3 (60 samples)
Reply time [ms]: response 2.4 transfer 0.0
Reply size [B]: header 242.0 content 1010.0 footer 0.0
(total 1252.0)
Reply status: 1xx=0 2xx=29997 3xx=0 4xx=0 5xx=0
CPU time [s]: user 94.31 system 205.26 (user 31.4% system 68.4%
total 99.9%)
Net I/O: 129.6 KB/s (1.1*106 bps)
Errors: total 3 client-timo 0 socket-timo 0 connrefused 3
connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

There are six groups of statistics: overall results (”Total”), connection related results (”Connection”), results relating to the issuing of HTTP requests (”Request”), results relating to the replies received from the server (”Reply”), miscellaneous results relating to the CPU (”CPU”) and network (”Net I/O”) utilization and, last but not least, a summary of errors encountered (”Errors”).

6.4.2 Autobench

Autobench is a perl wrapper around httperf, which purpose is to run a series of tests using httperf. Autobench will aggregate the results from the tests and display them in nice column formatted files (TSV [1] or CSV [2]) [32]. Below is an example of running Autobench. The explanation of the flags is taken from the Autobench man file.

Example 10.

```
autobench --single_host
          --host1 masterproxy
          --port1 8080
          --uri1 /index2.html
          --low_rate 10
          --high_rate 200
          --rate_step 10
          --num_call 1
          --const_test_time 60
          --timeout 5
          --file results.tsv
```

–file filename Send output to filename instead of STDOUT

–high-rate hrate The number of connections per second to open at the end of the test

–low-rate lrate The number of connections per second to open at the start of the test

–rate-step step Autobench runs httperf numerous times against the target machines, starting at lrate connections per second, increasing the number of connections per second requested by step until hrate is reached.

–host1 hostname The hostname of the first server under test

–const-test-time length Used instead of –num-conn, this causes Autobench to calculate a value for nconn for each test to make the test last length seconds. It is recommended that this be used instead of –num-conn. For each test, nconn is set to (current-rate * length). Note that no results will be obtained if length is set to less than 10 seconds (since httperf samples only once every 10 seconds), and meaningful results (reproducible results derived from a significant number of samples) will require each test to last at least 60 seconds.

–single-host Only test a single server.

–timeout time time is the time in seconds for which httperf will wait for a response from the server - responses received after this time will be counted as errors.

–uri1 uri The URI to test on the first server (e.g. /foo/bar/index.html)

[1] Tab separated values

[2] Comma separated values

Chapter 7

Methodology

> It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.
>
> Franklin D. Roosevelt

7.1 Bottlenecks

Certain properties of a system, such as variables or parameters, may be configured freely. This makes the system have a certain degree of freedom. However, when we add constraints to this system, we limit this freedom by adding rules which states the range of values the parameters may be chosen from.

Constraints are caused by certain bottlenecks of the system, and mathematically these bottlenecks creates rules which can be described as equalities or inequalities [20]. Figure 7.1 shows this scenario graphically.

In this graph we have two degrees of freedom: x and y. However, the functions f and g limits this freedom , and leaves us with a limited set of the xy plane of which to choose the values for x and y.

$$y \geq 2x$$
$$y \leq -3x + 10$$

Possible bottlenecks during the thesis experiments can be located at or caused by different entities. These entities are: the blade server, the proxy/dispatcher, the network bandwidth, or the client computer it elf. When performing benchmarks of a web server, it is important that the web server itself is the bottleneck. Because we want to stress test the web server, and find out it's maximum performance, it is important that other factors do not limit the overall throughput of the system.

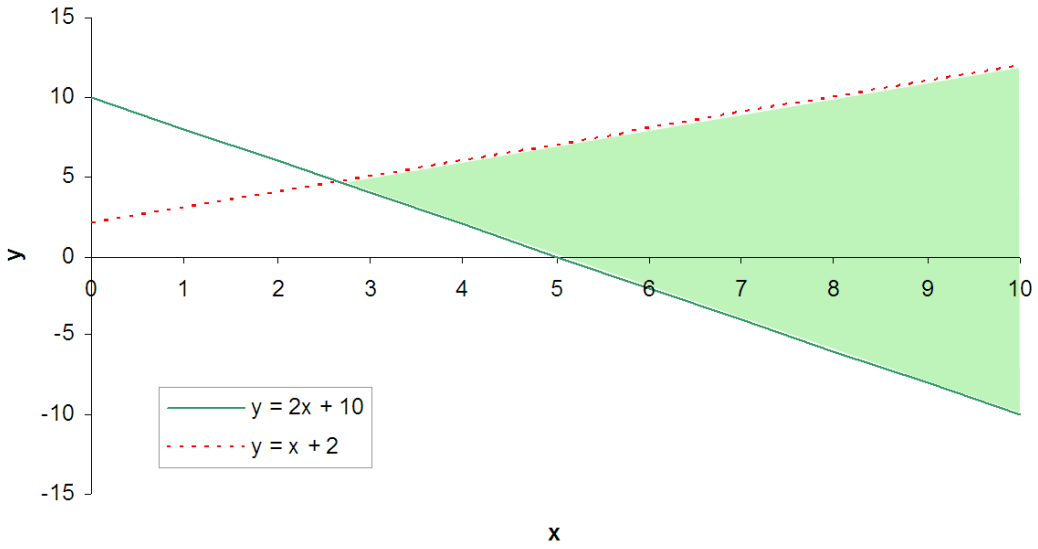


Figure 7.1: Constraints issued by f and g limits our freedom to the colored area of the xy-plane.

In order to find the constraints applied to the system, a three step test will suffice [24]:

1. Here we use httperf and autobench to automate a series of tests. The traffic is directed directly against the blade, meaning that it does not go through the load balancer. The CPU load on the blade serving as a server will be monitored during the test.

2. We repeat test 1, only this time the traffic goes through the load balancer. By doing this we can eliminate the load balancer being a bottleneck if we get the same results as in test 1.

3. The client is also a possible source of error in our tests. To eliminate this possibility we will run the same tests as described above, but now using two clients where the rate each client produces is halved, meaning that the sum of request per second is the same as using one client.

7.2 The experiments

This section will describe the experiments undertaken, and their methods and purpose. When performing experiments, it is important to vary one variable at

a time, so that recorded variations of the results can be pinpointed to a specific change of a single parameter. If more than one variable is changed between tests, it is difficult to know which change in the system caused the change in the measured results.

Several tests will be performed to compare push and pull based LB. This section will ex pain how each individual test is performed in detail. The setup will differ a between each of two LB strategies. In order to focus the experiment to the variation of the spin delay, a low size is selected for the HTML page that is queried by the measurement tool.

7.2.1 Parameters of web server type B

The web server type B (Pull based web server) has a set of configurable parameters governing the functionality for measuring the health of the computer. These parameters are the CPU utilization threshold value; a value, if surpassed, will cause the system to stop accepting new jobs until health improves, and the time interval between which to take samples of the current CPU utilization.

The web server's performance can be described mathematically as a black box, having two degrees of freedom, using the parameters u_{max} for the per centage CPU utilization threshold value, and Δt for the interval between measurements, given in milliseconds.

$$x(u_{max}, \Delta t)$$

An analytical approach will be used in order to find the optimum values for these parameters, using both tests and logical deduction. One guess is that there might be an optimum value for the CPU utilization point; A too high value of the CPU utilization threshold might downgrade performance, due to the server being overloaded, but a too low value will leave CPU resources unused. The point of this test is to find a point in which balance these two extremes.

Tests will also be done to find an optimum value for the measurement interval. The interval must be low enough for the load balancing to quickly adapt to changes in the CPU utilization. On the other hand, frequent requests might lead to performance decrease. The point of this test is also to find a balance between two extremes.

7.2.2 Benchmarking push based LB

When performing tests using the push based load balancer, we run WebserverTypeA on each of the blades, and the Round Robin load balancer is run on the

dispatcher host. Several tests will be run, varying the spin delays of the web servers. The system is benchmarked using Autobench.

Example 11.
Starting the RR load balancer.
```
java proxy.LoadBalancerProxy -c 8080 -a RR \\
  -s blade1:8080,blade2:8080;blade3:8080,blade5:8080,blade6:8080
```
Starting the push based web server
```
java webserver.WebserverTypeA -p 8080 -s 1500000
```

7.2.3 Benchmarking pull based LB

When performing tests using the pull based LB, we run WebserverTypeB on the blades, and QueueProxy on the dispatcher/proxy host. We run tests varying the spin delay of the web server.

Example 12.
Starting the pull based LB
```
java proxy.QueueProxy -c 8080 -r 8081
```
Starting the pull based web server
```
java webserver.WebserverTypeB -a 128.39.73.99 -p 8081 -m 100 -vv
```

7.3 Comparing homogeneous and inhomogeneous hardware

Blade	CPU Rate
1	1,4 GHz
2	1.75 GHz
3	2.10 GHz
4	2.45 GHz
5	2.80 GHz

Table 7.1: The different CPU speeds of each individual blade

The same tests of the push and pull based web servers will be performed while using inhomogeneous hardware. This is done by adjusting the CPU speed of the blades using *cpufreq* [1]. Table 7.1 shows the CPU rate of each of the individual blades.

[1] A Linux kernel module, maintained by Dave Jones, which is used for adjusting the clock rate of a computer's CPU at runtime.

Chapter 8

Results and Analysis

> No amount of experimentation can ever prove me right; a single experiment can prove me wrong.
>
> Albert Einstein

This section describes the results of each of the experiments, along with analysis of the results. Two types of metrics have been used for measuring server performance. The most common metric [8, 10, 24, 9] is the average response time, which measures the mean time between a request has been issued to the server and until the first packet of the reply is received by the client. The second metric that is used, is the response rate [30], which states the amount of responses the server can sustain during a specific period of time. This is used to find the point of which a web server is saturated.

$$u_{responserate} = \frac{n \text{ replies}}{\Delta t \text{ sec}}$$

8.1 Finding bottlenecks

This section will describe the results from exploring the constraints of the system.

The client computer can only sustain a request rate of about 1200 requests per second. A single blade running web server A, can sustain 2500+ requests per second, however we found that the web server becomes unstable during high number of connections (1000+ req/sec), as exceptions were thrown during high request rates.

It was also discovered that the proxy, while running the QueueProxy, was only able to process about 60 connection per second, meaning that the pull based LB proxy is a massive bottleneck of the system. This means that the

system is constrained by the proxy, and we must keep within the limits of this constrain when performing the experiments.

By setting high spin delays of the web servers, we can make the system operate within the limits issued by the queue proxy. Initially, our hypothesis prompted for the use of high spin delays, in order to simulate the generation of complex and CPU demanding dynamic pages. This means that adjusting our tests to fit the constraint will not have a too big high impact on the original test plan.

8.2 Finding values for parameters of the pull based web server

A set of different values for the CPU threshold and the measurement interval was issued on the web server, and the response rate was measured.

Figure 8.1 shows the results of the test which varies the CPU threshold value. At request rates higher than 13 requests per second, the variation of the data starts to increase. However, at 13 requests per second we see that as the CPU threshold decreases, the total response rate also increases. This suggests that there is no sweet spot in which the server work best at a certain CPU utilization rate, in which there is a weight between CPU utilization and performance, but instead that any threshold below 100 per cent only limits the use of available resources, thus degrading performance.

Figure 8.2 shows the results of the tests while changing the CPU utilization measurement interval. The results show a very high variation of the data, which leaves us unable to see any difference of performance between the different choices of interval values. Instead, a value is chosen based purely on theoretical intuition. A value is chosen that is low enough to be able to collect a sample during the lifetime of a working thread. This way, the web server will be able to measure the changes in CPU utilization when a request is finished processing the HTTP request. The value chosen for the time interval is 250 milliseconds, which conveniently also is the median of the set of intervals that were tested.

8.3 Main results

8.3.1 Homogeneous hardware

Results

This section will show the results of the two load balancing strategies using homogeneous hardware configuration on the blades. Figures 8.3 to 8.6 show

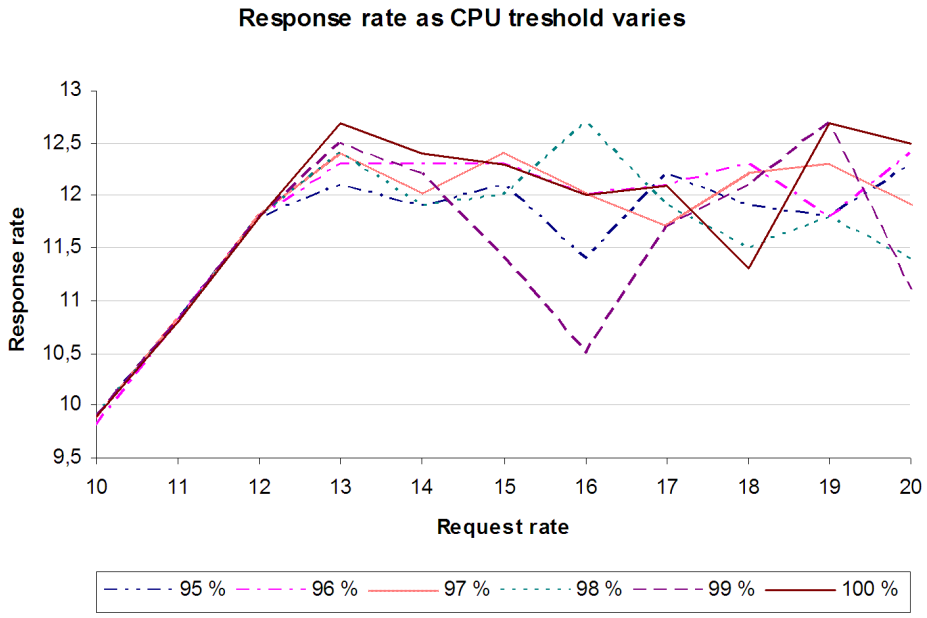


Figure 8.1: Comparison of the response rate while varying the CPU threshold value.

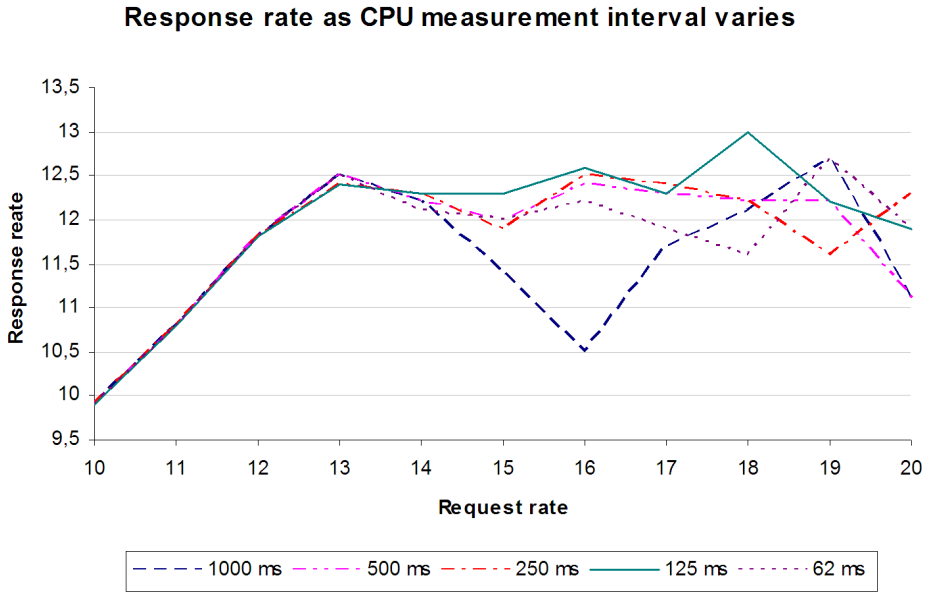


Figure 8.2: Comparison of the response rate while varying the CPU measurement interval.

the response rate using spin delays of $500 \cdot 10^3$, $1 \cdot 10^6$, $1.5 \cdot 10^6$, and $1.75 \cdot 1^6$ iterations. Figures 8.7 to 8.10 shows the average response times, using the same set of spin delays. All of the tests were run using a five second client timeout value.

When looking at the results of the spin delays above 500K (Figures 8.5 and 8.6), we see that the push based system can cope with a higher request rate than the pull based system. As with the 500K spin delay (Figure 8.3), we see that they both peek at the same request rate. Due to the resolution of the request rate (The request rate increased with a step of 5) we can not differ the peek rate of the two strategies.

We also see that the variation of the measured data is great when the request rate is beyond the saturation point, and we also see that the error bars overlap most places.

We also see that the pull based strategy constantly yields higher response times compared to the push based strategy at low request rates. However, if we look at the results of the response time using the lowest spin delay (500M) in figure 8.7, we see that the pull based LB strategy produces lower response times at high response rates than the push based strategy. When we increase the spin delay to 1M, we see that the tables have turned (Figure 8.8). As we increase the spin delay even further (Figure 8.9 and 8.10, the performance of the two strategies at high response rates seem to converge.

Analysis

The hypothesis states that the push based strategy would improve it's relative performance as the spin delay increases (Hypothesis 4, however, from the results we see evidence of the contrary.

All of the results showed that the pull based strategy performed poorer than the push based strategy, with the exception of the measurements of the response time using a low spin delay of $500 \cdot 10^6$ iterations. Results using this spin delay, suggests that a pull based strategy performs better than the push based strategy during high load. This might seem like an anomaly, but a possible explanation is given: Jobs that take shorter time to process by a server is easier to for the queue proxy to shear amongst the multiple servers, and this benefit becomes apparent first during high load. As the spin delay increases, the requests become harder to shear, thus resulting in higher queueing time in the central queue on the proxy.

The difference of the response times of the two strategies seemed to converge during high load, suggests that when using a high spin delay, the added overhead of the proxy would merely be a small part of the total response time. As the spin delay increases, the time fraction of the overhead of the proxy gets smaller and smaller.

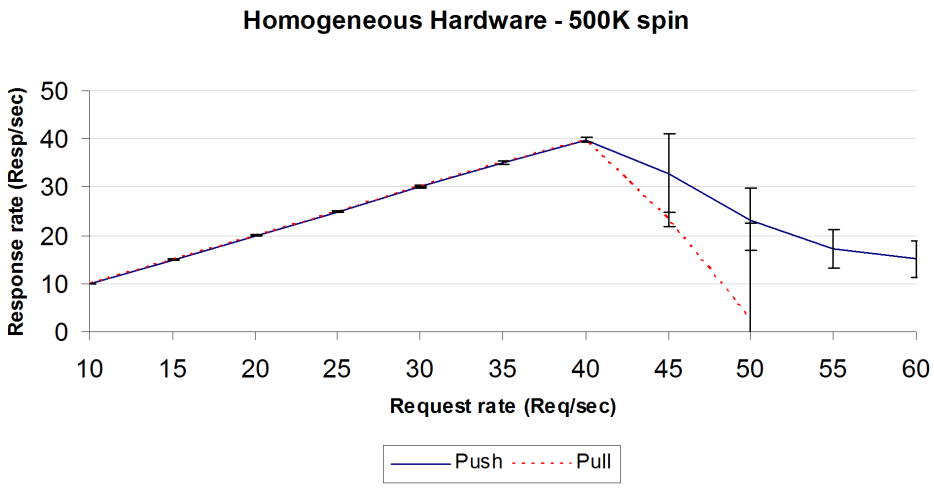


Figure 8.3: Comparison of the average response rates between the two load balancing strategies, using homogeneous hardware.

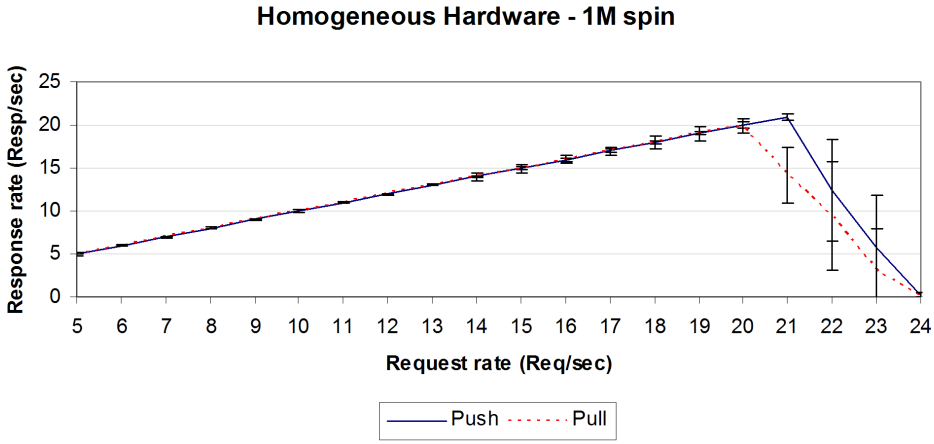


Figure 8.4: Comparison of the average response rates between the two load balancing strategies, using homogeneous hardware.

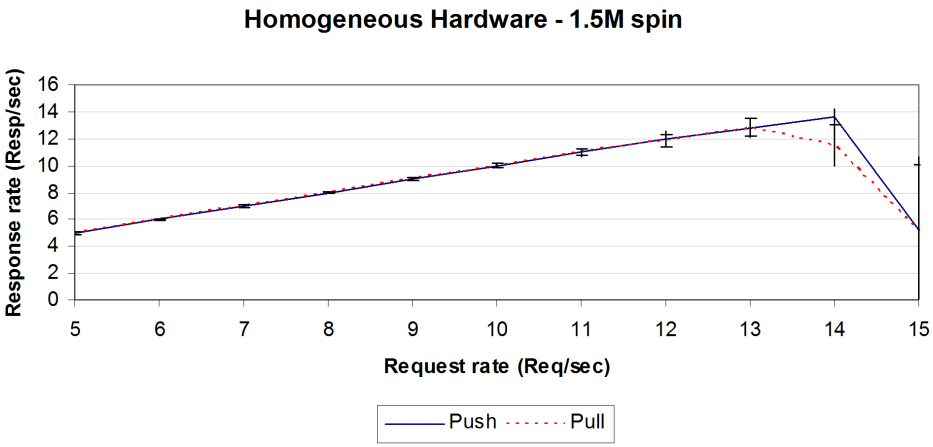


Figure 8.5: Comparison of the average response rates between the two load balancing strategies, using homogeneous hardware.

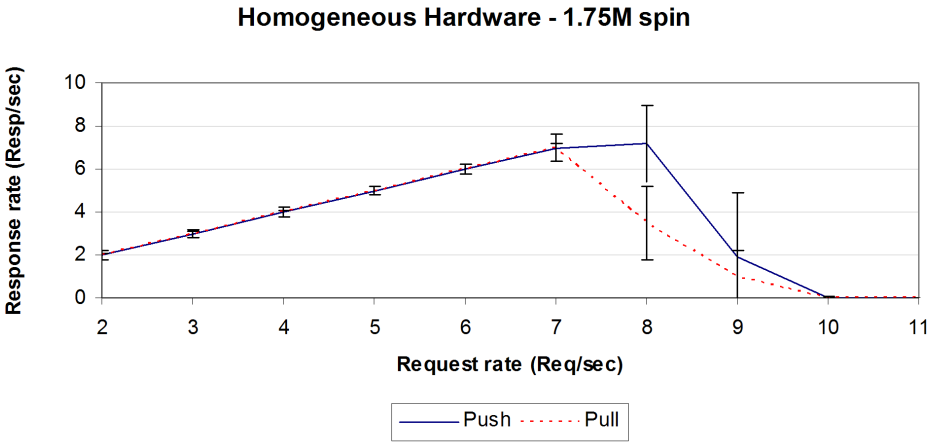


Figure 8.6: Comparison of the average response rates between the two load balancing strategies, using homogeneous hardware.

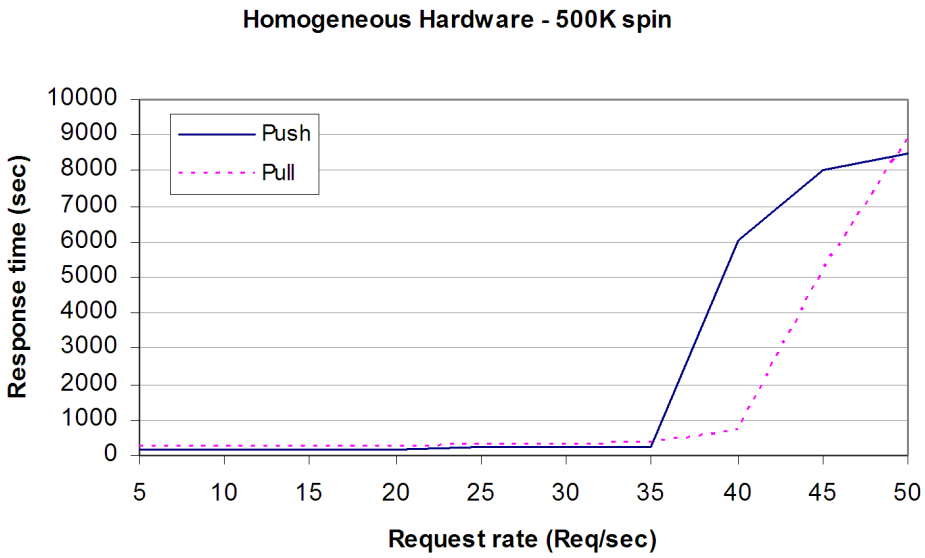


Figure 8.7: Comparison of the average response times between the push and pull based LB strategy, using homogeneous hardware.

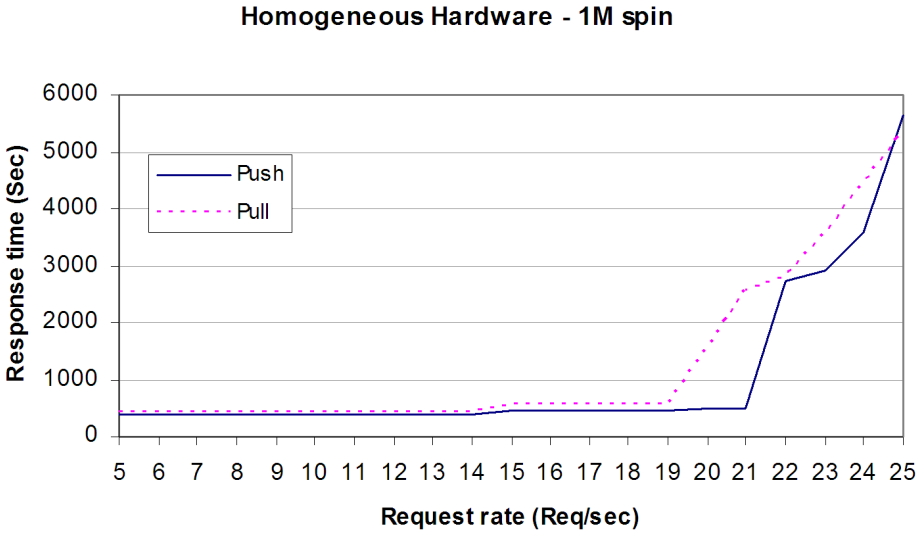


Figure 8.8: Comparison of the average response times between the push and pull based LB strategy, using homogeneous hardware.

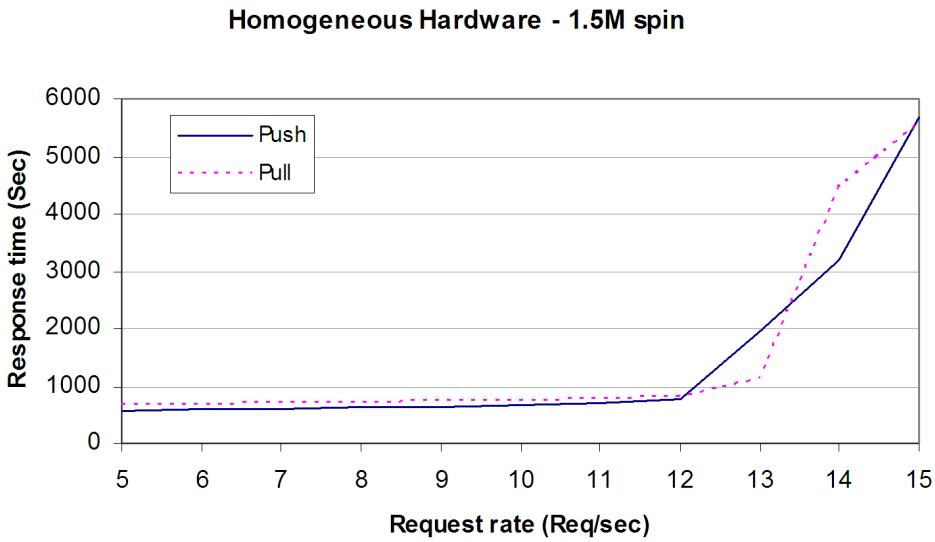


Figure 8.9: Comparison of the average response times between the push and pull based LB strategy, using homogeneous hardware.

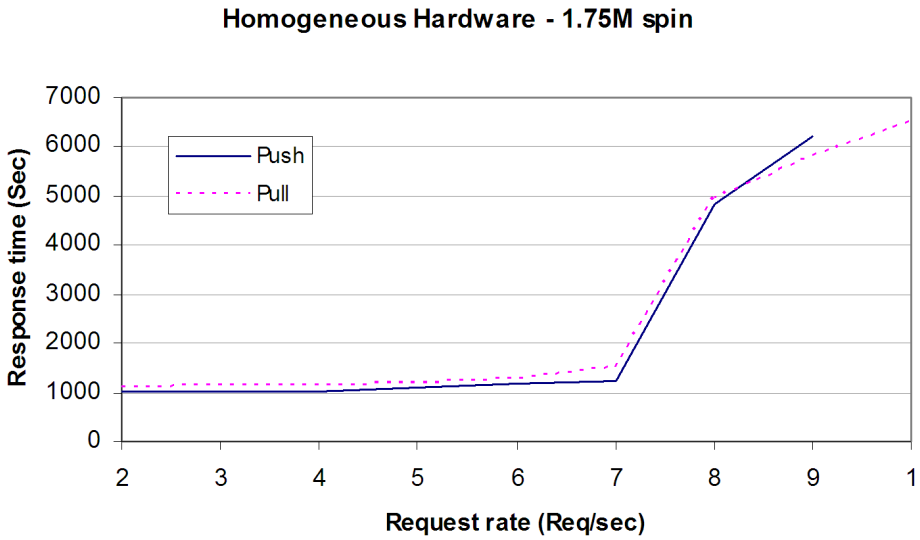


Figure 8.10: Comparison of the average response times between the push and pull based LB strategy, using homogeneous hardware.

The pull based LB strategies poor performance, using homogeneous hardware, suggests that any possible benefits of having a pull based web server, due to a hypothetically more fair load balancing, is out weighted due to the overhead associated with the use of the queue proxy.

8.3.2 Inhomogeneous hardware

Results

This section will show the results of the two load balancing strategies using inhomogeneous hardware configuration on the blades. Figure 8.11 to 8.14 shows the results of the response rates using spin delays of $250 \cdot 10^3$, $500 \cdot 10^3$, $1 \cdot 10^6$, and $1.5 \cdot 10^6$ iterations. Figures 8.15 to 8.18 shows the average response times, using the same set of spin delays. All of the tests were run with a five second client timeout of the HTTP requests.

When looking at the results for the 250K spin delay (Figure 8.11), we see that the push based LB outperforms the push based strategy. However, at a request rate of 55 and above, we see a steep decline of the values for the pull based system.

When we look at the 500K spin delay results (Figure 8.12), we see that the pull based LB are better than the push based one, yielding a request rate of 4 requests per second before saturating. We see that the push based has more variation than the pull based strategy, and we also see a small decline in the response rate of the pull based strategy.

When we look at the results when the spin delay increases (1M to 1.5) (Figures 8.13 and 8.14), we see that the differences between the strategies decreases and that the error bars overlap. However, at both figures we see that the average value still is better for the pull based strategy. In both figures we also see a sharp decline at high request rates, even performing worse than the push based strategy.

We see that response times using the push based LB strategy during low request rates is constantly higher than the push based LB strategy. We also see that the response times of the push based LB strategy do not seem to increase as much during high load compared to first strategy; This is perhaps most evident in figure 8.18.

Using low and medium spin delays (Figures 8.15 to 8.17) we see that the pull based strategy maintains a flat response time longer, and thus outperforms the push based strategy in the borderline between medium and high request rates.

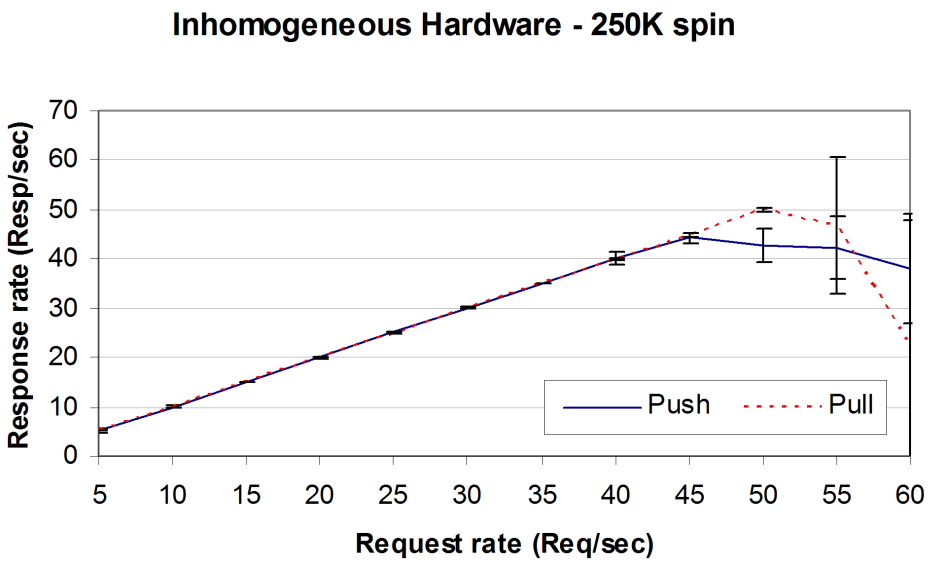


Figure 8.11: Comparison of the average response rates between the two load balancing strategies, using inhomogeneous hardware.

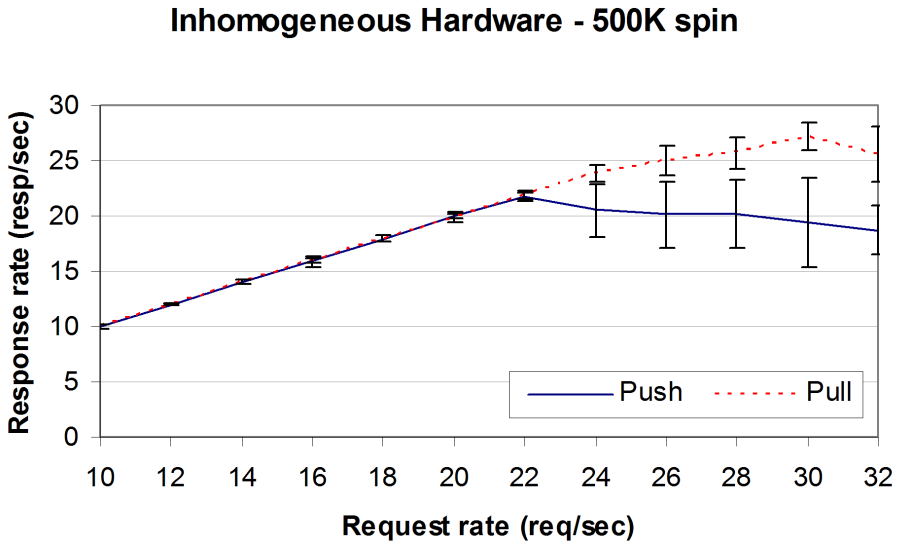


Figure 8.12: Comparison of the average response rates between the two load balancing strategies, using inhomogeneous hardware.

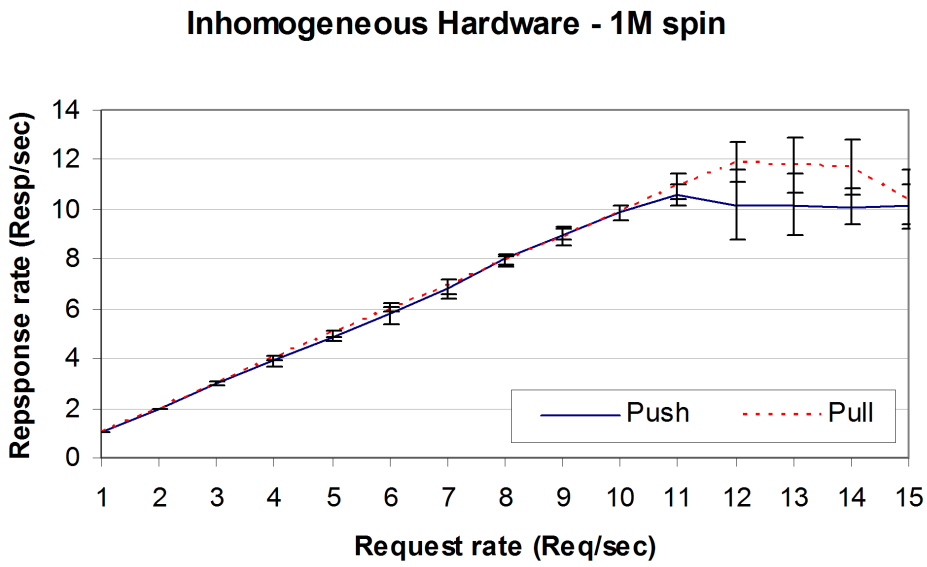


Figure 8.13: Comparison of the average response rates between the two load balancing strategies, using inhomogeneous hardware.

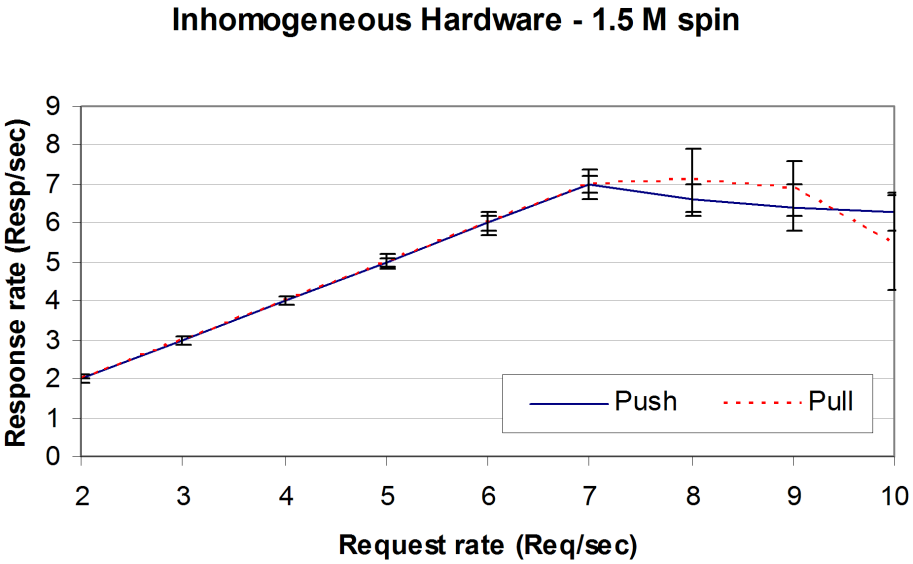


Figure 8.14: Comparison of the average response rates between the two load balancing strategies, using inhomogeneous hardware.

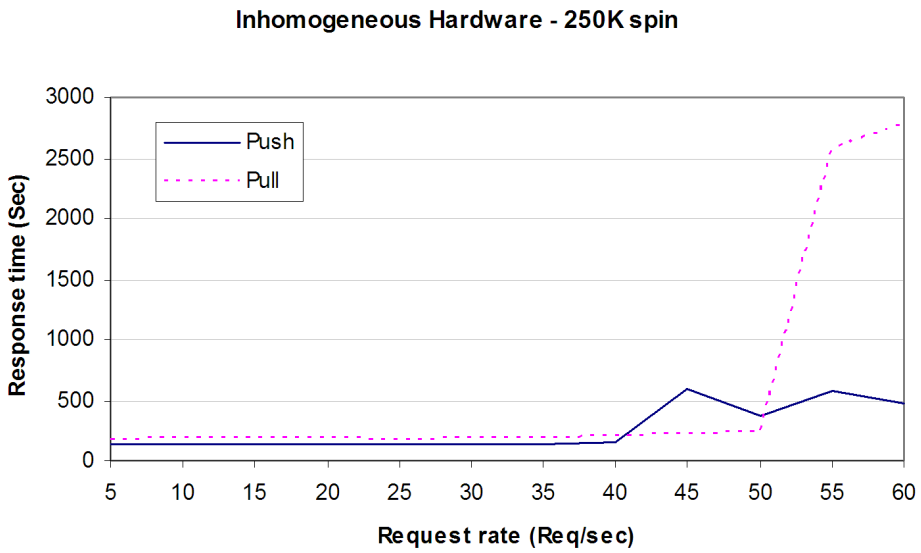


Figure 8.15: Comparison of the average response times between the push and pull based LB strategy, using inhomogeneous hardware.

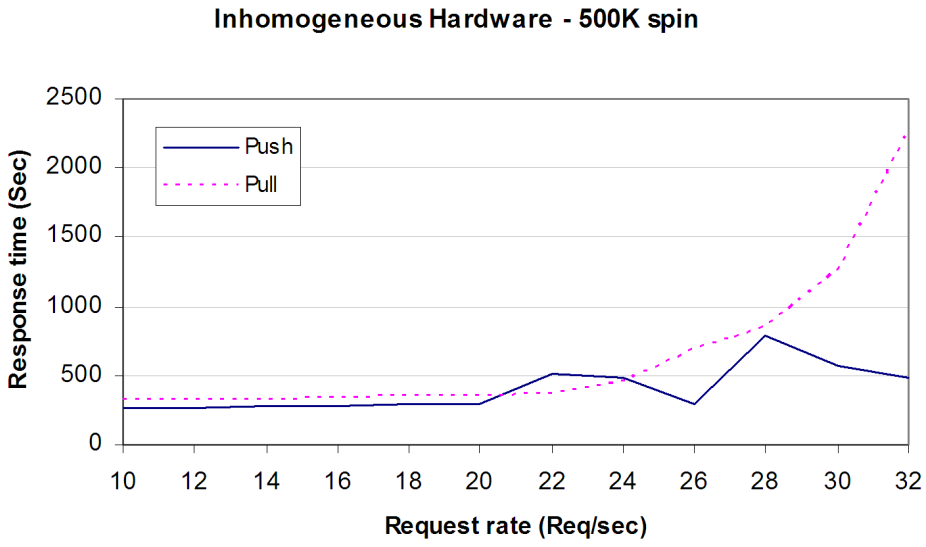


Figure 8.16: Comparison of the average response times between the push and pull based LB strategy, using inhomogeneous hardware.

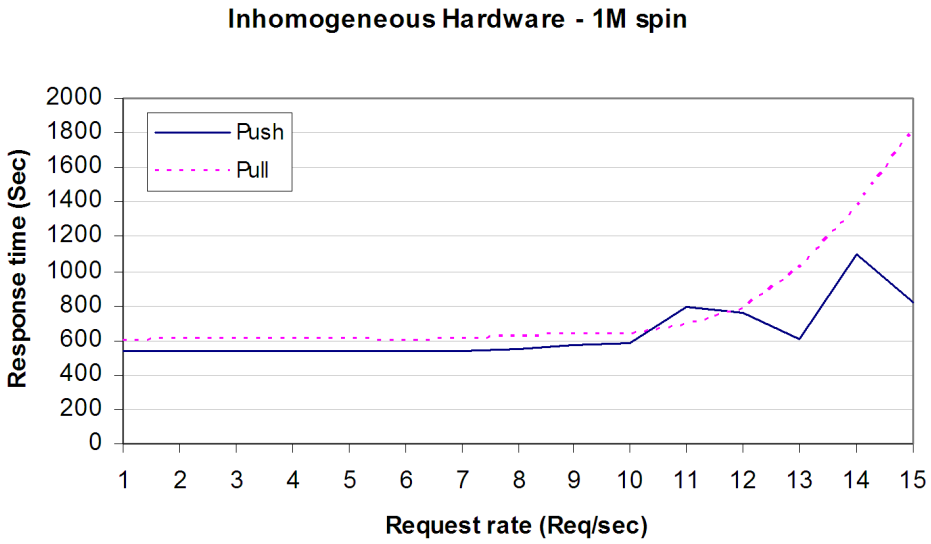


Figure 8.17: Comparison of the average response times between the push and pull based LB strategy, using inhomogeneous hardware.

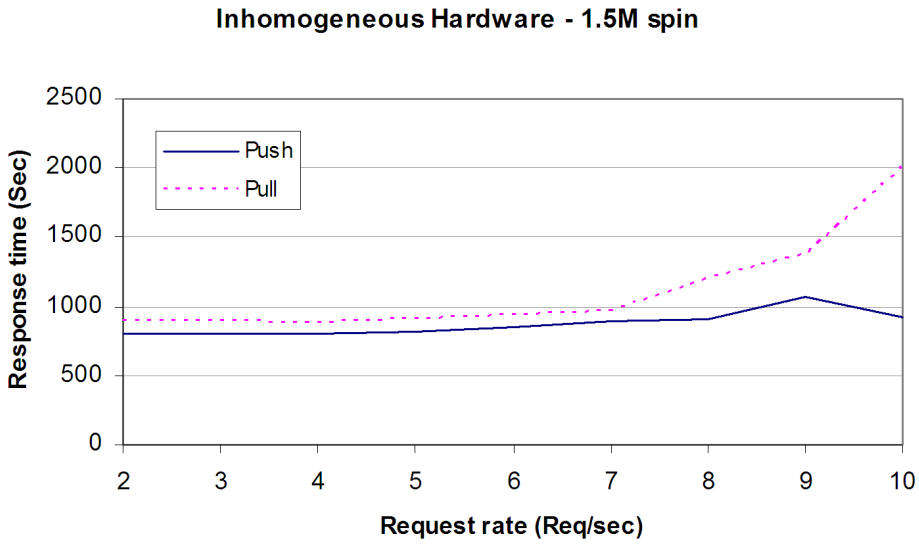


Figure 8.18: Comparison of the average response times between the push and pull based LB strategy, using inhomogeneous hardware.

Analysis

The results of the response rate shows that the pull based strategy can sustain a higher response rate than the push based strategy, which suggests that there is a better distribution of the load amongst the multiple servers. The sharp decline in the response rates of the pull based LB strategy suggests that the proxy or the servers are too saturated. The results using the smallest spin (Figure 8.11, which has the sharpest decline, is saturated at 60 requests per second. This was found as the maximum request rate that the queue proxy could manage in section 8.1.

When we compare the results of the response rates with the results of the response times for the pull based load balancer strategy, we see that the decline in the response rate is followed with an increase in the response time. However, when using the spin delay of 500K, we see somewhat contradictory data.

We see that the pull based strategy is constantly better than the other strategy from 24 to 32 requests per second (Figure 8.12). When we look at the corresponding results of the response times in figure 8.16, we see that the pull based strategy performs worse of the two. However, the reason for the increase in response time at high rates, might also be attributed to how *httperf* measures the response time.

Httperf only measures response time of requests that did not time out or receive a connection error [30]. A web server which receives n requests may, as an example, yield an error rate of 50 per cent. This means that only one half of the requests issued were actually answered successfully. A web server which is saturated may yield good response times on the requests that are actually processed, but might simply refuse to process the other 50 per cent, giving a time out value or connection refused error. As shown in the httperf manual [30], a server will typically sustain a flat response time after the point of which the server reach the saturation point. This way, the server might yield low response times on the responses that are actually processed, and at the same time yield low reply rates.

A pull based web server differs from a push based server, in that the only error in which a pull based web server may yield, is client timeout. They will not give connection refused errors, which normally happens when a server is saturated, because the servers will only poll the queue when they have free capacity to process a request. Because a request may have been stored in the queue some time before being processed, the pull based strategy might yield longer response times compared to the push based strategy, despite that the response rate is higher.

In order to investigate this further, one would have to have information about the number of error and the total number of connections. Due to the fact that *autobench* only outputs the number of errors, and not the per centage

value, nor the number of connections, there is simply not enough information. However, this information would be available from the output from each run of *httperf* by autobench. Unfortunately, this information was not stored on file, but merely written to the console.

8.4 Additional comments

In retrospect, the methods undertaken during the experiments could have been done differently. In order to more specifically pinpoint the reasons for the bottleneck, the total response time could be divided into chunks, so as to see how much each part of the system contributes to the total response time. Although the experiments show that the proxy is clearly a system constraint, by adding performance counters in various parts of the system, the overhead associated with these parts could be looked at in more detail.

$$t_{response} = t_{server} + t_{proxy} + t_{transmissiontime}$$

By splitting up the total response time, the overhead of the proxy and network transmission time could be subtracted, and it could be shown whether the pull based servers them selves yield shorter processing time compared to the push based servers.

The performance counters could be implemented by adding timers directly into the source code of the prototype, but due to time constraints it was not implemented in practice.

Chapter 9

Conclusion

The scientific work in this thesis have been to develop a prototype for a new pull based load balancing technique, and to to compare this strategy with a traditional pull based load balancing strategy by using a web farm of both homogeneous and inhomogeneous hardware. Of the set of hypothesis, most were falsified by the results of the testing.

However, we verified that the proxy added overhead compared to a Round Robin (RR) load balancing strategy, and this overhead had a significant impact on the results. The results generally showed that the pull based load balancing strategy performed poorer and gave larger response times than the RR algorithm, and this is generally attributed to the high overhead associated with the central queue proxy.

The push based strategy performed better than RR occasionally, when using low spin delays and medium to high request rates. This is the opposite as stated in hypothesis 4, which projected that the push based strategy would perform better when using high values for the spin delays.

It was also stated that the pull based strategy would share the load more evenly when using inhomogeneous hardware. However, results found that that the response times were generally higher using the pull based strategy, compared to the RR algorithm. However, at medium to high request rates, the pull based strategy occasionally yielded the lowest response times of the two.

The hypothesis that the pull based load balancing strategy would give a more fair weighting of the web servers, and utilize the total resources of the web servers better, have not been confirmed from these experiments. The queue proxy introduced a overhead that had a too big impact on the results. Further work is needed to help reduce this overhead, if the strategy is ever going to be a liable load balancing strategy. Lower overhead might simply be achieved by developing the push based load balancing prototype using a more low-level language such as C or C++, which have more direct access to the network stream, and more direct capabilities of manipulating network packets, compared to when using Java.

It might seem as the only reason for selecting a pull based strategy would be the principles of voluntary cooperation, in which the web servers are autonomous and may decide them selves when to process requests from the client. However, it is doubtful that the poor performance measured in the experiments would be justified by these principles alone.

Bibliography

[1] Allan Kuchinsky Nina Bhatti, Anna Bouch. Integrating user-perceived quality into web server design. *Computer Networks, Elsevier Science B.V.*, 2000.

[2] Xiaoyun Zhu, Jie Yu, and John Doyle. Heavy-tailed distributions, generalized source coding and optimal web layout design.

[3] Niall Mansfield. *Practical TCP/IP, Designing, Using, and Troubleshooting TCP/IP Networks on Linux and Windows*. Addison-Wesley / Pearson Education, 2003.

[4] Mark Burgess. *Principles of Network and System Administration*. Wiley, 2000.

[5] Alberto Leon-Garcia and Indra Widjaja. *Communication Networks*. McGraw-Hill, 2nd edition, 2004.

[6] Kartik Gopalan Yingfei Dong Zhenhai Duan. Push vs. pull: Implications of protocol design on controlling unwanted traffic. *USENIX: Proceedings SRUTI (Steps to Reducing Unwanted Traffic on the Internet Workshop) 05*, pages 25–30, July 2005.

[7] Mark Burgess and Kyrre Begnum. Voluntary cooperation in pervasive computing services. In *Proceedings of the 19th Large Installation System Administration Conference*. Usenix, December 2005.

[8] Philips S. Yu Valeria Cardellini, Michele Colajanni. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, May-June 1999.

[9] ivind Kure Haakon Bryhni, Espen Klovning. A comparison of load balancing techniques for scalable web servers. *IEEE Network*, 2000.

[10] Rassul Ayani Yong Ment Teo. Comparison of load balancing strategies on cluster-based web servers. *Transaction of the Society for Modeling and Simulation*, 2001.

[11] Tony Bourke. *Server Load Balancing*. OReilly & Associates, 1st edition, August 2001.

[12] C. Nyberg M. Kihl J. Cao, M. Andersson. Web server performance modeling using an m/g1/k*ps queue. *10th International Conference on Telecommunications*, 2:1501–1506, February - March 2003.

[13] R. Hariharan R.D van der Mei and P.K. Reeser. Web server perforamnce modeling. *Telecommunication Systems*, 16(3-4):361–378, March 2001.

[14] Mark Burgess. *Handbook of Network and System Administration*, chapter System Administration and the Scientific Method. Elsevier, 2007.

[15] W. Willinger and V. Paxson. Where mathematics meets the internet. *Notices of the Am. Math. Soc.*, 45(8):961, 1998.

[16] V. Paxson and S. Floyd. Wide area traffic: the failure of poisson modelling. *IEEE/ACM Transactions on networking*, 3(3):226, 1995.

[17] W. Willinger, V. Paxson, and M.S. Taqqu. Self-similarity and heavy tails: structural modelling of network traffic. *in* A practical guide to heavy tails: statistical techniques and applications, pages 27–53, 1996.

[18] W.E. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, pages 1–15, 1994.

[19] K.I. Hopcroft, E. Jakeman, and J.O. Matthews. Discrete scale-free distributions and associated limit theorems. *Journal of Mathematical Physics*, A37:L635–L642, 2004.

[20] Mark Burgess. *Analytical Network and System Administration - Managing Human-Computer Systems*. J. Wiley & Sons, 2004.

[21] R. Jain. The art of computer systems performance analysis. 1991.

[22] Gerald Sabin Jan Prins P. Sadayappan James Dinian, Stephen Oliver and Chaun-Wen Tseng. Dynamic load balancing of unbalanced computations using message passing. *Proceedings of 6th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS 2007)*, March 2007.

[23] Christopher Baldwin and Carl Nolan. Message queueing: A scalable, highly available load-balancing solution. *Microsoft Developer Network (MSDN)*, July 2000.

[24] Gard Undheim. Predicting performance and scaling behaviour in a data center with multiple application servers. Master's thesis, Oslo University College, May 2006.

[25] Ph.D. Louis P. Slothouber. A model of web server performance. 2005.

[26] Peter V. Mikhalenko. Measure cpu and memory consumption of a java application. http://articles.techrepublic.com.com/5100-3513_11-6067049.html, May 2006.

[27] Keita Fujii. Jpcap. http://netresearch.ics.uci.edu/kfujii/jpcap/doc/, 6 2006.

[28] Adrian Mark Colyer. United states patent 6023722. www.freepatentsonline.com/6023722.html, February 2000.

[29] Mark Burgess. System administration research, part 2: Analythical system administration. *;login:*, 25(4), July 2000.

[30] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. *Performance Evaluation Review*, 26(3):31–37, December 1998.

[31] David Mosberger and Tai Jin. httperf - http performance measurement tool. Linux man file http://www.hpl.hp.com/research/linux/httperf/httperf-man.txt.

[32] Jilian T. J. Midgley. autobench - automates the benchmarking of web servers using httperf. linux man file http://www.xenoclast.org/autobench/man/autobench.html, March 2001.

Appendix A

Source code

This appendix contains the complete source code for the system developed during the work of this master thesis. The files are organized in sections based upon the source code Java package structure.

A.0.1 Common

This section Contains source code of classes shared amongst the proxy and server components.

CpuState.java

```
package common;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.*;

public class CpuState
{
   public int User, Nice, System, Idle;

   public static CpuState getCurrentState() throws Throwable
   {
      // Reads output from linux /proc/stat
      Runtime runtime = Runtime.getRuntime();
      Process proc = runtime.exec("cat /proc/stat");
      BufferedReader reader = new BufferedReader(new InputStreamReader(proc
            .getInputStream()));
      // Read values from proc.
      // Parse and return a new object.
      String cpuinfo = reader.readLine();
```

```
        StringTokenizer tokenizer = new StringTokenizer(cpuinfo);
        reader.close();

        tokenizer.nextToken();

        // Format of the first line:
        // cpu <user> <nice> <system> <idle> <irq> <soft irq>
        CpuState stat = new CpuState();
        stat.User = Integer.parseInt(tokenizer.nextToken());
        stat.Nice = Integer.parseInt(tokenizer.nextToken());
        stat.System = Integer.parseInt(tokenizer.nextToken());
        stat.Idle = Integer.parseInt(tokenizer.nextToken());

        return stat;
    }

    /**
     * Returns the cpu utilization as a value between 0 and 1, based upon the
     * last measurement and the current state. The measurements must be of at
     * least one hundreth of a second for this method to return a real value.
     *
     * @param last
     * @return
     */
    public static float getCpuUtilization(CpuState lastState) throws Throwable
    {
        CpuState now = getCurrentState();
        CpuState diff = new CpuState();
        diff.User = now.User - lastState.User;
        diff.System = now.System - lastState.System;
        diff.Nice = now.Nice - lastState.Nice;
        diff.Idle = now.Idle - lastState.Idle;

        return getCpuUtilization(lastState, now);
    }

    public static float getCpuUtilization(CpuState first, CpuState second)
    {
        CpuState diff = new CpuState();
        diff.User = second.User - first.User;
        diff.System = second.System - first.System;
        diff.Nice = second.Nice - first.Nice;
        diff.Idle = second.Idle - first.Idle;

        return (float) (diff.User + diff.System + diff.Nice)
```

```
            / (float) (diff.User + diff.System + diff.Nice + diff.Idle);
   }

   public static float getCpuUtilization() throws Throwable
   {
      CpuState lastState = getCurrentState();
      // We must sleep so that the cpu can be averaged over at leas one
      // hundreth of a second, because
      // /proc/stat has units of one 100th second.
      Thread.sleep(11);
      CpuState now = getCurrentState();

      return getCpuUtilization(lastState, now);
   }
}
```

CpuStateChecker.java

```
package common;

/**
 * This thread constantly measures the CPU utilization based on an average over
 * a fixed interval.
 *
 * @author Ocine
 *
 */
public class CpuStateChecker implements Runnable
{

   public static int sleepTime = 500;

   private static float cpuUtilization;

   public static float GetCurrentCpuUtilization()
   {
      return cpuUtilization;
   }

   public void run()
   {
      try
      {
         cpuUtilization = CpuState.getCpuUtilization();
      } catch (Throwable t)
      {
         t.printStackTrace();
      }
```

```
      do
      {
         try
         {
            CpuState start = CpuState.getCurrentState();
            Thread.sleep(sleepTime);
            cpuUtilization = CpuState.getCpuUtilization(start);
         } catch (Throwable t)
         {
            t.printStackTrace();
         }

      } while (true);
   }

}
```

Logger.java

```
package common;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * A wrapper for output of debug information.
 *
 * @author Ocine
 *
 */
public class Logger
{
   private static DateFormat dateFormat = new SimpleDateFormat(
         "dd/MM/yyyy HH:mm:ss");

   private String componentName;

   public Logger(String componentName)
   {
      this.componentName = componentName;
   }

   public void writeOutput(String componentName, String message)
   {
      Date now = new Date();
      System.out.println(dateFormat.format(now) + "-" + componentName + ": "
            + message);
   }

   public void writeOutput(String message)
```

```
   {
      writeOutput(this.componentName, message);
   }
}
```

A.0.2 Proxy

This section contains the source code for the two types of load balancing components.

QueueProxy.java

```
package proxy;

/**
 * A proxy for the webserver. This proxy queues up all of the requests and
 * stores them in a messagequeue. The webserver can then eat requests from this
 * queue, and thus handle them at the webserver's own convenience.
 *
 * @author Ocine
 *
 */
public class QueueProxy
{
 public static int debugLevel;

 public static int clientListenerPort;

 public static int serverDoneJobListenerPort;

 public static int serverRequestJobListenerPort;

 private static RequestQueue messageQueue;

 private static common.Logger log = new common.Logger("QueueProxy");

 public static void main(String[] args) throws Exception
 {
  debugLevel = 0;
  clientListenerPort = 81;
  serverRequestJobListenerPort = 82;

  if (!HandleCommandLineParameters(args))
   return;

  log.writeOutput("QueueProxy started.");

  messageQueue = new RequestQueue();
  ClientListener clientListener = new ClientListener(clientListenerPort,
    messageQueue);
```

```
 ServerRequestJobListener serverRequestListener = new ServerRequestJobListener(
   serverRequestJobListenerPort, messageQueue);
 messageQueue.SetServerRequestJobListener(serverRequestListener);
 Thread threadA = new Thread(clientListener);
 Thread threadC = new Thread(serverRequestListener);
 threadA.start();
 threadC.start();

 while (true)
 {
  Thread.yield();
  Thread.sleep(15000);
 }
}

private static boolean HandleCommandLineParameters(String[] args)
{
 // Handle command line parameters.
 if (args.length > 0)
 {
  for (int i = 0; i < args.length; i++)
  {
   if (args[i].compareToIgnoreCase("--help") == 0)
   {
    System.out
      .println("Usage: QueueProxy <params>\n\nThe parameters are:\n\n"
        + "-c --clientlistenerport    "
        + "The port number used to listen for client requests.\n"
        + "-r --serverrequestjobport  "
        + "The port number used to listen for server job requests.\n"
        + "-v -vv -vvv                "
        + "Run the proxy in verbatim mode.\n\n");
    return false;
   }
   if (args[i].compareToIgnoreCase("-v") == 0)
    debugLevel = 1;
   else if (args[i].compareToIgnoreCase("-vv") == 0)
    debugLevel = 2;
   else if (args[i].compareToIgnoreCase("-vvv") == 0)
    debugLevel = 3;
   else if (args[i].compareToIgnoreCase("--clientlistenerport") == 0
     || args[i].compareToIgnoreCase("-c") == 0)
   {
    if (args.length == i + 1)
    {
     System.err.println("No port number is given for client listener port.");
     return false;
    }

    try
    {
```

```
      clientListenerPort = Integer.parseInt(args[i + 1]);
     } catch (Exception e)
     {
      System.err.println("Invalid port number given for client listener port.");
      return false;
     }

     i++;
    } else if (args[i].compareToIgnoreCase("--serverrequestjobport") == 0
      || args[i].compareToIgnoreCase("-r") == 0)
    {
     if (args.length == i + 1)
     {
      System.err
        .println("No port number is given for server done job listener port.");
      return false;
     }

     try
     {
      serverRequestJobListenerPort = Integer.parseInt(args[i + 1]);
     } catch (Exception e)
     {
      System.err
        .println("Invalid port number given for server done job listener port.");
      return false;
     }

     i++;
    } else
    {
     System.err.println("Invalid argument '" + args[i]
       + "'. Use --help to see a list over valid arguments. ");
     return false;
    }
   }
  }
  return true;
 }
}
```

RequestQueue.java

```
package proxy;

public class RequestQueue
{
 public java.util.Queue<String> requests;

 public java.util.Queue<Integer> requestIds;
```

```
private common.Logger log = new common.Logger("RequestQueue");

private ServerRequestJobListener listener;

public RequestQueue()
{
 requests = new java.util.LinkedList<String>();
 requestIds = new java.util.LinkedList<Integer>();
}

public void SetServerRequestJobListener(ServerRequestJobListener listener)
{
 this.listener = listener;
}

public synchronized void AddToQueue(String request, int id)
{
 if (listener == null)
  throw new java.lang.NullPointerException(
    "The ServerRequestJobListneer reference is null.");

 requests.add(request);
 requestIds.add(new Integer(id));
 if (QueueProxy.debugLevel >= 2)
  log.writeOutput("Added request to queue. ");

 // Notify the topmost requestjobworker in the workerthreadlist.
 ServerRequestJobWorker worker = listener.workers.peek();
 if (worker != null)
 {
  synchronized (worker)
  {
   worker.notify();
   if (QueueProxy.debugLevel >= 2)
   {
    log.writeOutput("Notified worker " + worker.workerId
      + " about new job in queue.");
   }
  }
 } else
 {
  if (QueueProxy.debugLevel >= 2)
  {
   log
     .writeOutput("No workers to notify about new job. All workers are buzy. ");
  }
 }
}

/**
 * Gets the next request in the queue, and removes the worker thread from the
```

```
 * free worker threads queue. It is the responsibility of the caller to add the
 * worker back into the queue again after processing the request.
 *
 * @param worker
 *         The worker that requests a new request.
 * @return Null if no request exists in queue.
 * @throws Exception
 */
public synchronized Request GetNextRequest(ServerRequestJobWorker worker,
  boolean removeCallerThreadFromQueue) throws Exception
{
 String httpRequest = requests.poll();
 if (httpRequest == null)
  return null;

 int id = requestIds.poll();

 Request request = new Request();
 request.httpRequest = httpRequest;
 request.requestId = id;

 if (removeCallerThreadFromQueue && listener.workers.size() == 0)
  throw new Exception("Worker queue is empty in GetNextRequest for caller "
    + worker.workerId + ".");

 if (QueueProxy.debugLevel >= 2)
 {
  System.out.println("Current workers in queue: ");
  Object[] objects = listener.workers.toArray();
  for (int i = 0; i < objects.length; i++)
   System.out.println("i=" + i + ":" + objects[i]);
 }

 boolean b = listener.workers.remove(worker);
 if (!b)
 {
  if (QueueProxy.debugLevel >= 2)
   log
     .writeOutput("Could not find jobworker no "
       + worker.workerId
       + " while removing it from queue: "
       + "Most probable reason is that the requestworker found "
       + "job at startup.");
 } else
 {
  if (QueueProxy.debugLevel >= 2)
   log.writeOutput("Removed RequestJobWorker " + worker.workerId
     + " from the queue.");
 }

 return request;
```

```
 }
}
```

Request.java

```
package proxy;

/**
 * A dataclump for the HTTP request, containing the id of the request for use by
 * the request queue.
 *
 * @author Ocine
 *
 */
public class Request
{
   public String httpRequest;

   public int requestId;
}
```

ClientListener.java

```
package proxy;

import java.net.ServerSocket;
import java.net.Socket;
import java.io.*;

/**
 * This thread listenes to HTTP requests from the client, and delegates the
 * request to a RequestWorker.
 *
 * @author Ocine
 *
 */
public class ClientListener implements Runnable
{
   public int port;

   public final int MINIMUM_PORT_NUMBER = 10000;

   public final int MAXIMUM_PORT_NUMBER = 65535;

   public int currentRequestWorkerPort;

   private RequestQueue messageQueue;

   private common.Logger log = new common.Logger("ClientListener");
```

```
   public ClientListener(int port, RequestQueue messageQueue)
   {
      this.port = port;
      this.messageQueue = messageQueue;
   }

   public void run()
   {

      ServerSocket listener;
      try
      {
         listener = new ServerSocket(port);
      } catch (IOException ioe)
      {
         System.err.println(ioe.toString());
         return;
      }
      log.writeOutput("Ready to accept client connections on port " + port
            + ".");

      int workerId = MINIMUM_PORT_NUMBER;

      while (true)
      {
         try
         {
            Socket socket = listener.accept();
            if (QueueProxy.debugLevel >= 1)
               log.writeOutput("ClientListener: Ingoing connection from "
                     + socket.getRemoteSocketAddress());

            ClientRequestWorker worker = new ClientRequestWorker(socket,
                  workerId, messageQueue);
            Thread thread = new Thread(worker);
            thread.start();
         } catch (java.net.BindException be)
         {
            System.err.println(be.toString());
         } catch (IOException ioe)
         {
            System.err.println(ioe.toString());
         }

         if (workerId == MAXIMUM_PORT_NUMBER)
            workerId = MINIMUM_PORT_NUMBER - 1;

         workerId++;
      }
   }
}
```

ClientRequestWorker.java

```
package proxy;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * This worker's job is to retrieve the clients HTTP request, and to return the
 * response to the client after the response has been handled by the webserver.
 *
 * After putting the request in the message queue, the thread will listen to
 * response from the server until passing the same response back to the client.
 *
 * @author Ocine
 *
 */
public class ClientRequestWorker implements Runnable
{

   final static String LINEBREAK = "\r\n";

   private RequestQueue messageQueue;

   private Socket socket;

   private common.Logger log = new common.Logger("ClientRequestWorker");

   /**
    * The id of the worker, as well as the port number that the worker will
    * listen for response from the webserver.
    */
   private int workerId;

   public ClientRequestWorker(Socket socket, int workerId,
         RequestQueue messageQueue) throws java.io.IOException
   {
      this.socket = socket;
      this.workerId = workerId;
      this.messageQueue = messageQueue;
   }

   public void run()
   {
      try
      {
         handleRequest();
      } catch (Exception e)
```

```
   {
      System.err.println(e);
   }
}

private void handleRequest() throws Exception
{
   // 1. First read the contents of the HTTP request.
   InputStream inputClient = null, inputServer = null;
   DataOutputStream outputClient = null;
   Socket serverReply = null;
   try
   {
      inputClient = socket.getInputStream();
      outputClient = new DataOutputStream(socket.getOutputStream());

      BufferedReader reader = new BufferedReader(new InputStreamReader(
            inputClient));

      int i = 0;
      String s = "";
      String request = "";
      do
      {
         s = reader.readLine();
         if (QueueProxy.debugLevel >= 3)
            System.out.println("Request line " + i + ": " + s);
         i++;
         request += s + LINEBREAK;
      } while (s.length() > 0);

      // 2. Add the request to the queue, so that the servers can process
      // it.
      // Then listenen for a connection from the webserver.
      ServerSocket listener = new ServerSocket(workerId);

      messageQueue.AddToQueue(request, workerId);
      if (QueueProxy.debugLevel >= 2)
         log.writeOutput("RequestWorker " + this.workerId,
               "Added request to queue. ");

      serverReply = listener.accept();
      if (QueueProxy.debugLevel >= 1)
         log.writeOutput("RequestWorker " + this.workerId,
               "Got connection from webserver ("
                     + serverReply.getRemoteSocketAddress() + ").");

      // 3. Read the respose and send the response back to the client.

      inputServer = serverReply.getInputStream();
```

```
         byte[] buffer = new byte[4096];
         int n;
         while ((n = inputServer.read(buffer)) > 0)
            outputClient.write(buffer, 0, n);

         outputClient.flush();

         if (QueueProxy.debugLevel >= 1)
            log.writeOutput("RequestWorker " + this.workerId,
                  "Closed the connection to the webserver ("
                        + serverReply.getRemoteSocketAddress() + ")");

         if (QueueProxy.debugLevel >= 1)
            log.writeOutput("RequestWorker " + this.workerId,
                  "Closed connection to client ("
                        + socket.getRemoteSocketAddress() + ")");
      } catch (java.net.BindException be)
      {
         System.err.println(be.toString());
      } catch (java.net.SocketException se)
      {

         System.err.println(se.toString());
      } finally
      {
         if (inputServer != null)
            inputServer.close();
         if (serverReply != null)
            serverReply.close();
         if (outputClient != null)
            outputClient.close();
         socket.close();
      }
   }
}
```

ServerRequestJobListener.java

```
package proxy;

import java.net.ServerSocket;
import java.net.Socket;
import java.util.Queue;

/**
 * This thread listens to requests from the webserver asking if there are any
 * new HTTP connections in the message queue. The response to the webserver
 * contains the port number to send the webserver resonse to, along with the
 * actual HTTP request.
 *
 * @author Ocine
```

```
 *
 */
public class ServerRequestJobListener implements Runnable
{
   private int port;

   Queue<ServerRequestJobWorker> workers;

   private RequestQueue messageQueue;

   private int currentId = 0;

   private common.Logger log = new common.Logger("JobListener");

   public ServerRequestJobListener(int port, RequestQueue messageQueue)
   {
      this.port = port;
      this.messageQueue = messageQueue;
      workers = new java.util.LinkedList<ServerRequestJobWorker>();
   }

   public void run()
   {
      try
      {
         ServerSocket listener = new ServerSocket(port);
         log.writeOutput("Ready to accept server connection on port " + port
               + " (Request job) ");

         while (true)
         {
            Socket socket = listener.accept();
            if (QueueProxy.debugLevel >= 1)
               log.writeOutput("Webserver (" + socket.getRemoteSocketAddress()
                     + ") connecting to tracking port " + port + ".");

            ServerRequestJobWorker worker = new ServerRequestJobWorker(
                  currentId, socket, messageQueue, workers);

            Thread thread = new Thread(worker);
            thread.start();

            if (currentId > Integer.MAX_VALUE)
               currentId = -1;

            currentId++;
         }

      } catch (java.io.IOException ioe)
      {
         System.err.println(ioe.toString());
```

```
      }
   }
}
```

ServerRequestJobWorker.java

```
package proxy;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.Socket;
import java.util.Queue;

public class ServerRequestJobWorker implements Runnable
{
   final static String LINEBREAK = "\r\n";

   public int workerId;

   private Socket socket;

   private RequestQueue messageQueue;

   private Queue<ServerRequestJobWorker> workers;

   private common.Logger log = new common.Logger("RequestJobWorker");

   public ServerRequestJobWorker(int workerId, Socket socket,
         RequestQueue messageQueue, Queue<ServerRequestJobWorker> workers)// ,
   // Queue<ServerRequestJobWorker>
   // workers
   // )
   {
      this.workerId = workerId;
      this.socket = socket;
      this.messageQueue = messageQueue;
      this.workers = workers;
   }

   public void run()
   {
      try
      {
         handleRequest();
      } catch (Exception e)
      {
         System.err.println(e);
      }
   }
```

```
private void handleRequest() throws java.lang.InterruptedException,
      Exception
{
   try
   {
      // 1. First wait for a poll message from the server.
      InputStream input = socket.getInputStream();
      DataOutputStream output = new DataOutputStream(socket
            .getOutputStream());
      BufferedReader reader = new BufferedReader(
            new InputStreamReader(input));

      while (true)
      {
         // Reading the poll message. Assuming first line is
         // 'Request-job: true'.
         int i = 0;
         String s = "";
         // String pollMessage = "";
         do
         {
            s = reader.readLine();
            if (QueueProxy.debugLevel >= 3)
               System.out.println("Webserver request line " + i + ": " + s);
            i++;
         } while (s.length() > 0);

         if (QueueProxy.debugLevel >= 2)
            log.writeOutput("RequestJobWorker" + this.workerId,
                  "Received poll message from webserver ("
                        + socket.getRemoteSocketAddress() + ").");

         // 2. Check for a message in the messagequeue at startup.

         Request request = messageQueue.GetNextRequest(this, false);
         if (request != null)
         {
            if (QueueProxy.debugLevel >= 2)
               log.writeOutput("RequestJobWorker " + this.workerId,
                     "Found job in queue at startup.");

            outputJob(request, output);
         } else
         {
            if (QueueProxy.debugLevel >= 2)
               log.writeOutput("RequestJobWorker " + this.workerId,
                     "Found no job at startup.");
            do
            {
               // 3. If no message found, wait for a notification about
```

```
                // a new entry in
                // the message queue.
                synchronized (this)
                {
                   if (!workers.contains(this))
                   {
                      if (QueueProxy.debugLevel >= 2)
                         log.writeOutput("RequestJobWorker " + this.workerId,
                               "Added itself to worker queue.");
                      workers.add(this);
                   }
                   if (QueueProxy.debugLevel >= 2)
                      log.writeOutput("RequestJobWorker " + this.workerId,
                            "Waiting for notification about a job.");
                   this.wait();
                }

                request = messageQueue.GetNextRequest(this, true);

                if (QueueProxy.debugLevel >= 2)
                   log.writeOutput("RequestJobWorker " + this.workerId,
                         "Got alerted about new job in queue.");

                // If request is null, then there has been a race
                // condition, in which another
                // newly spawned thread have just pulled the request
                // from the queue.
             } while (request == null);

             if (QueueProxy.debugLevel >= 2)
                log.writeOutput("RequestJobWorker " + this.workerId,
                      "Found job in queue.");

             if (QueueProxy.debugLevel >= 2)
                log.writeOutput("RequestJobWorker " + this.workerId,
                      "Sending job to server "
                            + socket.getRemoteSocketAddress() + ".");

             outputJob(request, output);
          }
       }
    } catch (java.io.IOException ioe)
    {
       if (QueueProxy.debugLevel >= 1)
          System.err.println(ioe.toString());

       workers.remove(this);
       throw ioe;
    } catch (Exception e)
    {
       workers.remove(this);
```

```
            throw e;
        }
    }

    public void outputJob(Request request, DataOutputStream output)
            throws java.io.IOException
    {
        if (QueueProxy.debugLevel >= 2)
            log.writeOutput("RequestJobWorker " + this.workerId,
                  "Sending request to server.");

        output.writeBytes("Client-Worker-Port: " + request.requestId + LINEBREAK
              + LINEBREAK);

        output.writeBytes(request.httpRequest);
    }

    public String toString()
    {
        return "RequestJobWorker " + this.workerId + " at your service!.";
    }
}
```

LoadBalancerProxy.java

```
package proxy;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * A proxy for the webserver. This proxy load balances in the traditional way,
 * and supports RR and LC.
 *
 * @author Ocine
 *
 */

public class LoadBalancerProxy
{
 public static int debugLevel;

 /*
  * The port that the proxy listens to requests on.
  */
 public static int clientListenerPort;

 private static int loadBalancingAlgorithm;

 private static int LB_RR = 1, LB_LC = 2;
```

```
private static java.util.Vector<java.net.InetSocketAddress> hosts =
 new java.util.Vector<java.net.InetSocketAddress>();

private static common.Logger log = new common.Logger("LoadBalancerProxy");

public static void main(String[] args) throws Exception
{
 debugLevel = 0;
 clientListenerPort = 81;
 loadBalancingAlgorithm = LB_RR;

 if (!HandleCommandLineParameters(args))
  return;

 log.writeOutput("Dispacher started.");

 String algo = "";
 if (loadBalancingAlgorithm == LB_RR)
  algo = "RR";
 else if (loadBalancingAlgorithm == LB_LC)
  algo = "LC";
 else
  algo = "unsupported.";

 log.writeOutput("Load balancing algorithm is " + algo + ".");

 if (loadBalancingAlgorithm == LB_RR)
  LoadBalanceUsingRR();
 else if (loadBalancingAlgorithm == LB_LC)
  LoadBalanceUsingLC();
 else
  return;
}

private static void LoadBalanceUsingLC() throws Exception
{
 throw new java.lang.Exception("LC is not yet implemented! ");
}

private static void LoadBalanceUsingRR()
{

 ServerSocket listener;
 try
 {
  listener = new ServerSocket(clientListenerPort);
 } catch (IOException ioe)
 {
  System.err.println(ioe.toString());
  return;
```

```
 }
 log.writeOutput("Ready to accept client connections on port "
   + clientListenerPort + ".");

 int i = 0;
 while (true)
 {
  try
  {
   java.net.InetSocketAddress currentHost = hosts.get(i++);
   if (i >= hosts.size())
    i = 0;

   Socket socket = listener.accept();
   if (debugLevel >= 1)
    log.writeOutput("Ingoing connection from "
      + socket.getRemoteSocketAddress());
   if (debugLevel >= 2)
    log.writeOutput("Current server is " + currentHost.toString());

   RequestForwarder worker = new RequestForwarder(socket, currentHost);
   Thread thread = new Thread(worker);
   thread.start();
  } catch (java.net.BindException be)
  {
   System.err.println(be.toString());
  } catch (IOException ioe)
  {
   System.err.println(ioe.toString());
  }
 }

}

private static boolean HandleCommandLineParameters(String[] args)
{
 boolean hasSetServers = false;
 // Handle command line parameters.
 if (args.length > 0)
 {
  for (int i = 0; i < args.length; i++)
  {
   if (args[i].compareToIgnoreCase("--help") == 0)
   {
    System.out
      .println("Usage: QueueProxy <params>\n\nThe parameters are:\n\n"
        + "-c --clientlistenerport                  "
        + "The port number used to listen for client requests.\n"
        + "-a --loadbalancingalgorithm <algo>       "
        + "Which LB algorithm to use. Supported values are RR and LC.\n"
        + "-s --servers <host[:port][,host[:port]]> "
```

```
      + "A commaseparated list of the webservers to balance load between.\n"
      + "-v -vv -vvv                                "
      + "Run the proxy in verbatim mode.\n\n");
  return false;
 } else if (args[i].compareToIgnoreCase("-v") == 0)
  debugLevel = 1;
 else if (args[i].compareToIgnoreCase("-vv") == 0)
  debugLevel = 2;
 else if (args[i].compareToIgnoreCase("-vvv") == 0)
  debugLevel = 3;
 else if (args[i].compareToIgnoreCase("--clientlistenerport") == 0
   || args[i].compareToIgnoreCase("-c") == 0)
 {
  if (args.length == i + 1)
  {
   System.err.println("No port number is given for client listener port.");
   return false;
  }

  try
  {
   clientListenerPort = Integer.parseInt(args[i + 1]);
  } catch (Exception e)
  {
   System.err.println("Invalid port number given for client listener port.");
   return false;
  }

  i++;
 }

 else if (args[i].compareToIgnoreCase("--servers") == 0
   || args[i].compareToIgnoreCase("-s") == 0)
 {
  if (args.length == i + 1)
  {
   System.err.println("No servers are given.");
   return false;
  }

  java.util.StringTokenizer tokenizer = new java.util.StringTokenizer(
    args[i + 1], ",");
  java.util.Vector<String> servers = new java.util.Vector<String>();

  while (tokenizer.hasMoreTokens())
   servers.add((String) tokenizer.nextToken());

  for (int j = 0; j < servers.size(); j++)
  {
   String serverAndPort = servers.get(j);
   java.util.StringTokenizer t = new java.util.StringTokenizer(
```

```
    serverAndPort, ":");
  if (!t.hasMoreElements())
  {
   System.err.println("Invalid server given.");
   return false;
  }
  String hostname = t.nextToken();
  // Standard port number
  int portNumber = 80;
  if (t.hasMoreElements())
  {
   try
   {
    i++;
    String s = t.nextToken();
    portNumber = Integer.parseInt(s);

   } catch (Exception e)
   {
    System.err.println("Invalid port number given for server.");
    return false;
   }
  }

  hosts.add(new java.net.InetSocketAddress(hostname, portNumber));

  hasSetServers = true;
 }
} else if (args[i].compareToIgnoreCase("--loadbalancingalgorithm") == 0
  || args[i].compareToIgnoreCase("-a") == 0)
{
 if (args.length == i + 1)
 {
  System.err.println("No LB algorithm is given.");
  return false;
 }

 String algorithm = args[i + 1];

 if (algorithm.compareToIgnoreCase("RR") == 0)
  loadBalancingAlgorithm = LB_RR;
 else if (algorithm.compareToIgnoreCase("LC") == 0)
  loadBalancingAlgorithm = LB_LC;
 else
 {
  System.err
    .println("Invalid load balancing algorithm. Valid values are RR or LC. ");
  return false;
 }

 i++;
```

```
    } else
    {
     System.err.println("Invalid argument '" + args[i]
       + "'. Use --help to see a list over valid arguments. ");
     return false;
    }
   }
  }

  if (hasSetServers == false)
  {
   System.err.println("You must set at least one server.");
   return false;
  }

  return true;
 }

}
```

RequestForwarder.java

```
package proxy;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.Socket;

/**
 * This class receives a socket connection, and redirects the input to another
 * target. This is used to redirect HTML requests when load balancing.
 *
 * @author Ocine
 *
 */
public class RequestForwarder implements Runnable
{

   final static String LINEBREAK = "\r\n";

   private Socket socket;

   private java.net.InetSocketAddress target;

   private common.Logger log = new common.Logger("RequestForwarder");

   public RequestForwarder(Socket inputSocket, java.net.InetSocketAddress target)
         throws java.io.IOException
   {
```

```
      this.socket = inputSocket;
      this.target = target;
   }

   // Implement the run() method of the Runnable interface
   public void run()
   {
      try
      {
         handleRequest();
      } catch (Exception e)
      {
         System.err.println(e);
      }
   }

   private void handleRequest() throws Exception
   {
      // 1. First read the contents of the HTTP request.
      InputStream inputServer = null;
      BufferedReader readerClient = null;
      DataOutputStream outputClient = null, outputServer = null;

      // 1. Read request from client.
      // 2. Output request to selected server
      // 3. Read response from server
      // 4. Output response to client again.

      Socket serverSocket = new Socket();
      try
      {
         serverSocket.connect(target);
      } catch (java.net.ConnectException ce)
      {
         System.err.println(ce.toString());
         socket.close();
         return;
      }

      if (LoadBalancerProxy.debugLevel >= 1)
         log.writeOutput("Connected to server.");

      try
      {
         readerClient = new BufferedReader(new InputStreamReader(socket
               .getInputStream()));
         outputClient = new DataOutputStream(socket.getOutputStream());
         inputServer = serverSocket.getInputStream();
         outputServer = new DataOutputStream(serverSocket.getOutputStream());

         // Read the request
```

```
            String s;
            int i = 0;
            do
            {
               s = readerClient.readLine();

               if (LoadBalancerProxy.debugLevel >= 3)
                  System.out.println("Request line " + i + ": " + s);

               i++;

               outputServer.writeBytes(s + LINEBREAK);
            } while (s.length() > 0);

            outputServer.flush();

            if (LoadBalancerProxy.debugLevel >= 2)
               log.writeOutput("Request has been forwarded to server.");

            // Read the reply
            byte[] buffer = new byte[1024 * 4];
            int m = 0;

            // StringBuffer stringBuffer = new StringBuffer();
            while ((m = inputServer.read(buffer)) > 0)
            {
               outputClient.write(buffer, 0, m);

            }
            outputClient.flush();

            if (LoadBalancerProxy.debugLevel >= 2)
               log.writeOutput("Response has been forwarded to client.");

            readerClient.close();
            outputClient.close();
         } catch (java.net.BindException be)
         {
            System.err.println(be.toString());
         } catch (java.net.SocketException se)
         {
            System.err.println(se.toString());
         } finally
         {
            socket.close();
            if (serverSocket != null)
               serverSocket.close();
         }
      }
   }
```

A.0.3 Webserver

This section contains the source code for the pull and push based web servers.

WebserverTypeA.java

```
package webserver;

import java.net.*;
import java.text.*;

/**
 * A normal pull based webserver. Listens to TCP/HTTP connections and delegates
 * the request to a worker thread.
 *
 * @author Ocine
 *
 */
public class WebserverTypeA
{
 public static int debugLevel;

 public static int port;

 static DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

 private static common.Logger log = new common.Logger("WebserverTypeA");

 public static void main(String[] args) throws Exception
 {
  debugLevel = 0;
  port = 81;

  if (!HandleCommandLineParameters(args))
   return;

  log.writeOutput("Webserver started. \nReady to accept connections on port "
    + port + ".");

  ServerSocket listener = new ServerSocket(port);

  while (true)
  {
   Socket socket = listener.accept();
   if (debugLevel >= 1)
    log.writeOutput("Ingoing connection from "
      + socket.getRemoteSocketAddress() + ".");

   WorkerTypeA worker = new WorkerTypeA(socket);
   Thread thread = new Thread(worker);
   thread.start();
```

```
 }
}

private static boolean HandleCommandLineParameters(String[] args)
{
 // Handle command line parameters.
 if (args.length > 0)
 {
  for (int i = 0; i < args.length; i++)
  {
   if (args[i].compareToIgnoreCase("-v") == 0)
    debugLevel = 1;
   else if (args[i].compareToIgnoreCase("-vv") == 0)
    debugLevel = 2;
   else if (args[i].compareToIgnoreCase("-vvv") == 0)
    debugLevel = 3;
   else if (args[i].compareToIgnoreCase("--port") == 0
     || args[i].compareToIgnoreCase("-p") == 0)
   {
    if (args.length == i + 1)
    {
     System.err.println("No port number is given.");
     return false;
    }

    try
    {
     port = Integer.parseInt(args[i + 1]);
    } catch (Exception e)
    {
     System.err.println("Invalid port number given.");
     return false;
    }

    i++;
   } else if (args[i].compareToIgnoreCase("--spin") == 0
     || args[i].compareToIgnoreCase("-s") == 0)
   {
    if (args.length == i + 1)
    {
     System.err.println("No spin delay is given.");
     return false;
    }

    try
    {
     HttpRequestProcessor.spinDelay = Integer.parseInt(args[i + 1]);
    } catch (Exception e)
    {
     System.err.println("Invalid number given.");
     return false;
```

```
     }

     i++;
    } else if (args[i].compareToIgnoreCase("--help") == 0)
    {
     System.out
       .println("Usage: WebserverTypeA <params> \nThe parameters are: \n\n"
         + "--port -p             "
         + "Sets the server port used for listening to client requests. \n"
         + "--spin -s             "
         + "Sets the average number of iterations that the server will "
         + "spin each request.\n"
         + "-v -vv -vvv           "
         + "Runs the webserver in verbatim mode.\n ");
     return false;
    } else
    {
     System.err.println("Invalid argument '" + args[i]
       + "'. Use --help to see a list over valid arguments. ");
     return false;
    }
   }
  }
  return true;
 }

}
```

WorkerTypeA.java

```
package webserver;

import java.net.*;
import java.io.*;

/**
 * A worker thread for the webserver. This will process the actual HTTP request,
 * and will also return data to the client.
 *
 * The worker thread does not support HTTP version 1.1 with persistent
 * connections, and will close the connection after each request.
 *
 * @author Ocine
 *
 */
public class WorkerTypeA implements Runnable
{

   final static String LINEBREAK = "\r\n";

   private Socket socket;
```

```
   private HttpRequestProcessor httpProcessor;

   private common.Logger log = new common.Logger("WorkerTypeA");

   public WorkerTypeA(Socket socket) throws Exception
   {
      this.socket = socket;
      httpProcessor = new HttpRequestProcessor();
   }

   // Implement the run() method of the Runnable interface
   public void run()
   {
      try
      {
         handleRequest();
      } catch (Exception e)
      {
         System.out.println(e);
      }
   }

   private void handleRequest() throws Exception
   {

      InputStream input = socket.getInputStream();
      DataOutputStream output = new DataOutputStream(socket.getOutputStream());

      BufferedReader reader = new BufferedReader(new InputStreamReader(input));

      httpProcessor.processRequest(reader, output, "Webserver A");

      output.flush();

      if (WebserverTypeA.debugLevel >= 1)
         log.writeOutput("Closed connection. ");
   }
}
```

WebserverTypeB.java

```
package webserver;

import java.net.*;
import java.util.*;
import java.io.*;

/**
 * A pull based webserver. This server pulls HTTP requests from the proxy, and
 * returns the request to the proxy again.
```

```
 *
 * @author Ocine
 *
 */
public class WebserverTypeB implements Runnable
{
 public static int debugLevel;

 public static int proxyPort;

 public static String proxyAddress;

 public static int maxThreads;

 final static String LINEBREAK = "\r\n";

 public static WebserverTypeB runningWebserver;

 private static common.Logger log = new common.Logger("WebserverTypeB");

 static int numActiveThreads;

 static float maxCpuThreshold;

 public static void main(String[] args) throws Exception
 {
  maxThreads = 10;
  numActiveThreads = 0;
  debugLevel = 0;
  proxyPort = 82;
  proxyAddress = "127.0.0.1";
  maxCpuThreshold = 0.95f;

  if (!HandleCommandLineParameters(args))
   return;

  common.CpuStateChecker checker = new common.CpuStateChecker();
  Thread threadChecker = new Thread(checker);
  threadChecker.start();

  WebserverTypeB webserver = new WebserverTypeB();
  runningWebserver = webserver;

  Thread threadServer = new Thread(webserver);
  threadServer.start();
 }

 public void run()
 {
  try
  {
```

```
  doRequestLoop();
 } catch (Exception e)
 {
  System.err.println(e);
 }
}

public void doRequestLoop() throws Exception
{
 log.writeOutput("Webserver started. Polling " + proxyAddress + ":"
   + proxyPort + ".");
 if (WebserverTypeB.debugLevel >= 1)
  log.writeOutput("Max cpu load threshold is " + (maxCpuThreshold * 100)
    + "%.");

 while (true)
 {
  try
  {
   Socket poll = new Socket(proxyAddress, proxyPort);

   if (WebserverTypeB.debugLevel >= 1)
    log.writeOutput("Connecting to proxy " + proxyAddress + " on port "
      + proxyPort + ".");

   DataOutputStream output = new DataOutputStream(poll.getOutputStream());
   BufferedReader reader = new BufferedReader(new InputStreamReader(poll
     .getInputStream()));

   while (true)
   {
    // 1. First send a poll message
    output.writeBytes("Request-job: true" + LINEBREAK + LINEBREAK);
    if (WebserverTypeB.debugLevel >= 2)
     log.writeOutput("Sending poll message.");

    int proxyRequestWorkerPort = -1;

    // 2. Wait for a reply
    String s = reader.readLine();

    if (WebserverTypeB.debugLevel >= 1)
     log.writeOutput("Found a request in proxy queue. ");

    try
    {
     // Read the port number to send the reply to the proxy
     // on.
     StringTokenizer tokenizer = new StringTokenizer(s);
     String u = tokenizer.nextToken(":");
```

```
    if (WebserverTypeB.debugLevel >= 3)
     System.out.println("Token one: " + u);

    String t = tokenizer.nextToken(":").trim();

    if (WebserverTypeB.debugLevel >= 3)
     System.out.println("Token two: " + t);

    proxyRequestWorkerPort = Integer.parseInt(t);

    // Read LB + CR
    s = reader.readLine();

    // Run a working thread to deal with the actual
    // response.
    String requestedFile = HttpRequestProcessor
      .parseHttpRequestAndReturnRequestedFile(reader);

    WorkerTypeB worker = new WorkerTypeB(this, requestedFile, proxyAddress,
      proxyRequestWorkerPort);
    Thread thread = new Thread(worker);
    thread.start();

    // If the health is bad, then wait for a thread to
    // complete before
    // we handle more requests.
    synchronized (this)
    {
     while (!isHealthGood())
     {
      if (WebserverTypeB.debugLevel >= 1)
      {
       log
         .writeOutput("Health is not good. Suspending until health improves...");
       log.writeOutput("Num used threads is " + numActiveThreads + ".");
      }
      if (numActiveThreads == 0)
       break;
      if (numActiveThreads > 0)
       this.wait();
     }
    }
   } catch (java.util.NoSuchElementException nse)
   {
    System.err.println(nse.toString());
   }
  }
 } catch (java.net.ConnectException ce)
 {
  System.err.println(ce.toString());
```

```
  } catch (java.io.IOException ioe)
  {
   System.err.println(ioe.toString());
  }
 }
}

public static boolean isHealthGood()
{
 float utilization = common.CpuStateChecker.GetCurrentCpuUtilization();

 if (WebserverTypeB.debugLevel >= 2)
  log.writeOutput("Utilization = " + utilization);

 return utilization < maxCpuThreshold;
}

private static boolean HandleCommandLineParameters(String[] args)
{
 // Handle command line parameters.
 if (args.length > 0)
 {
  for (int i = 0; i < args.length; i++)
  {
   if (args[i].compareToIgnoreCase("-v") == 0)
    debugLevel = 1;
   else if (args[i].compareToIgnoreCase("-vv") == 0)
    debugLevel = 2;
   else if (args[i].compareToIgnoreCase("-vvv") == 0)
    debugLevel = 3;
   else if (args[i].compareToIgnoreCase("--proxyPort") == 0
     || args[i].compareToIgnoreCase("-p") == 0)
   {
    if (args.length == i + 1)
    {
     System.err.println("No port number is given.");
     return false;
    }

    try
    {
     proxyPort = Integer.parseInt(args[i + 1]);
    } catch (Exception e)
    {
     System.err.println("Invalid port number given.");
     return false;
    }

    i++;
   } else if (args[i].compareToIgnoreCase("--maxcpuload") == 0
     || args[i].compareToIgnoreCase("-m") == 0)
```

```
{
 if (args.length == i + 1)
 {
  System.err.println("No threshold is given.");
  return false;
 }

 try
 {
  maxCpuThreshold = (float) Integer.parseInt(args[i + 1]) / 100f;
 } catch (Exception e)
 {
  System.err.println("Invalid number is given.");
  return false;
 }

 i++;
}

else if (args[i].compareToIgnoreCase("--spin") == 0
  || args[i].compareToIgnoreCase("-s") == 0)
{
 if (args.length == i + 1)
 {
  System.err.println("No spin delay is given.");
  return false;
 }

 try
 {
  HttpRequestProcessor.spinDelay = Integer.parseInt(args[i + 1]);
 } catch (Exception e)
 {
  System.err.println("Invalid number given.");
  return false;
 }

 i++;
}

else if (args[i].compareToIgnoreCase("--proxyaddress") == 0
  || args[i].compareToIgnoreCase("-a") == 0)
{
 if (args.length == i + 1)
 {
  System.err.println("No proxy address is given.");
  return false;
 }

 proxyAddress = args[i + 1];
```

```
     i++;
    } else if (args[i].compareToIgnoreCase("--cpumeasureinterval") == 0
      || args[i].compareToIgnoreCase("-i") == 0)
    {
     if (args.length == i + 1)
     {
      System.err.println("No inverval is given.");
      return false;
     }

     try
     {
      common.CpuStateChecker.sleepTime = Integer.parseInt(args[i + 1]);
     } catch (Exception e)
     {
      System.err.println("Invalid number given.");
      return false;
     }

     i++;
    } else if (args[i].compareToIgnoreCase("--help") == 0)
    {
     System.out
       .println("Usage: WebserverTypeB <params>\n\nThe parameters are:\n\n"
         + "-a --proxyaddress      "
         + "The ip address or hostname of the proxy.\n"
         + "-p --proxyport         "
         + "The port number used to listen for new requests on the proxy.\n"
         + "-s --spin              "
         + "Sets the average number of iterations that the server will "
         + "spin each request.\n"
         + "-m --maxcpuload        "
         + "Sets the maximum cpu load in per cent, before the server stops "
         + "accepting more jobs.\n"
         + "-i --cpumeasureinterval "
         + "The inverval in milliseconds of which calculate the avg CPU load "
         + "of the server.\n"
         + "-v -vv -vvv            "
         + "Run the webserver in verbatim mode.\n\n");
     return false;
    } else
    {
     System.err.println("Invalid argument '" + args[i]
       + "'. Use --help to see a list over valid arguments. ");
     return false;
    }
   }
  }
  return true;
 }
}
```

WorkerTypeB.java

```
package webserver;

import java.net.*;
import java.io.*;

/**
 * A worker thread for the webserver type B. This will process the actual HTTP
 * request, and will also return data to the client.
 *
 * The worker thread does not support HTTP version 1.1 with persistent
 * connections, and will close the connection after each request.
 *
 * @author Ocine
 *
 */
public class WorkerTypeB implements Runnable
{
   final static String LINEBREAK = "\r\n";

   private String proxyAddress, requestedFile;

   private int proxyRequestWorkerPort;

   private common.Logger log = new common.Logger("WorkerTypeB");

   private HttpRequestProcessor httpProcessor;

   private WebserverTypeB webserver;

   public WorkerTypeB(WebserverTypeB webserver, String requestedFile,
         String proxyAddress, int proxyRequestWorkerPort) throws Exception
   {
      this.webserver = webserver;
      this.requestedFile = requestedFile;
      this.proxyAddress = proxyAddress;
      this.proxyRequestWorkerPort = proxyRequestWorkerPort;
      httpProcessor = new HttpRequestProcessor();
   }

   public void run()
   {
      try
      {
         WebserverTypeB.numActiveThreads++;
         handleRequest();

      } catch (Exception e)
      {
         System.out.println(e);
      } finally
```

```
      {
         if (WebserverTypeB.debugLevel >= 2)
            log.writeOutput("Notifying server about done job. ");
         synchronized (this.webserver)
         {
            WebserverTypeB.numActiveThreads--;
            this.webserver.notify();
         }
      }
   }

   private void handleRequest() throws Exception
   {
      Socket push = new Socket(proxyAddress, proxyRequestWorkerPort);
      DataOutputStream outputAnswer = new DataOutputStream(push
            .getOutputStream());

      httpProcessor.processRequest(requestedFile, outputAnswer,
            "Webserver Type B");

      outputAnswer.close();
      push.close();

      if (WebserverTypeB.debugLevel >= 1)
         log.writeOutput("Response sent to the proxy on port "
               + proxyRequestWorkerPort + ".");
   }
}
```

HttpRequestProcessor.java

```
package webserver;

import java.util.StringTokenizer;

public class HttpRequestProcessor
{

   final static String LINEBREAK = "\r\n";

   final static String HTDOCS = "htdocs/";

   private String serverName;

   public static int spinDelay = 100000;

   private static common.Logger log = new common.Logger("HttpRequestProcessor");

   /**
    * Reads the http header request directly from the input stream, and then
    * returns the name of the requested file.
```

```
 *
 * @param in
 *           The http request.
 * @return The name of the requested file.
 */
public static String parseHttpRequestAndReturnRequestedFile(
      java.io.BufferedReader in) throws java.io.IOException
{
   int i = 0;
   // Read the GET /file HTTP/1.1

   String httpRequest = "";
   String s = in.readLine();
   if (WebserverTypeB.debugLevel >= 3 || WebserverTypeA.debugLevel >= 3)
      System.out.println("Request line " + i++ + ": " + s);

   StringTokenizer tokens2 = new StringTokenizer(s);
   tokens2.nextToken();
   String fileToRequest = tokens2.nextToken();
   httpRequest += s + LINEBREAK;

   // Read the rest of it.
   do
   {
      s = in.readLine();

      if (WebserverTypeB.debugLevel >= 3 || WebserverTypeA.debugLevel >= 3)
         System.out.println("Request line " + i + ": " + s);
      i++;

      httpRequest += s + LINEBREAK;
   } while (s.length() > 0);

   if (WebserverTypeB.debugLevel >= 2 || WebserverTypeA.debugLevel >= 2)
      log.writeOutput("Client requesting file '" + fileToRequest + "'.");

   if (fileToRequest.startsWith("/"))
      fileToRequest = fileToRequest.substring(1, fileToRequest.length());

   return fileToRequest;
}

public void processRequest(String fileToRequest,
      java.io.DataOutputStream out, String serverName)
      throws java.io.IOException
{
   this.serverName = serverName;

   doSpinDelay();

   outputFile(fileToRequest, out);
```

```
   out.flush();
}

/**
 * Reads the http header request directly from the input stream, out outputs
 * the results directly on an output stream.
 *
 * @param in
 * @param out
 * @param serverName
 *           The name of the webserver as it will show in the http header.
 * @throws java.io.IOException
 */
public void processRequest(java.io.BufferedReader in,
      java.io.DataOutputStream out, String serverName)
      throws java.io.IOException
{
   this.serverName = serverName;

   String fileToRequest = HttpRequestProcessor
         .parseHttpRequestAndReturnRequestedFile(in);

   doSpinDelay();

   outputFile(fileToRequest, out);

   out.flush();

}

private void outputHeader(java.io.DataOutputStream out, boolean ok)
      throws java.io.IOException
{
   String s = null;

   if (ok)
      s = "HTTP/1.0 200 OK " + LINEBREAK;
   else
      s = "HTTP/1.0 404 Not Found " + LINEBREAK;

   s += "Connection: close " + LINEBREAK + "Server: " + this.serverName
         + " (JAVA) " + LINEBREAK;

   out.writeBytes(s);
}

/**
 * Runs some calculations in order to induce som spin delay, so as to emulate
 * a dynamic page.
 *
```

```
 */
public static void doSpinDelay()
{
   java.util.Random rnd = new java.util.Random();
   double d = rnd.nextGaussian();

   int average = spinDelay;// 100000;//0;//0;
   int stdev = average / 3;
   int num = average + (int) ((d) * (double) stdev);

   if (WebserverTypeB.debugLevel >= 2 || WebserverTypeA.debugLevel >= 2)
      log.writeOutput("Spinning for " + num + " iterations. ");
   for (int i = 0; i < average; i++)
   {
      double f = Math.sqrt(i) * Math.sin(average - i);
   }
}

private void outputFile(String filename, java.io.DataOutputStream out)
      throws java.io.IOException
{
   if (filename.length() == 0)
      filename = "index.html";

   filename = filename.replace("../", "You'r Busted!");

   String contentType;
   if (filename.endsWith(".html") || filename.endsWith(".htm"))
      contentType = "text/html";
   else if (filename.endsWith(".gif"))
      contentType = "application/octet-stream";
   else if (filename.endsWith(".png"))
      contentType = "image/png";
   else if (filename.endsWith(".jpg"))
      contentType = "image/jpg";
   else if (filename.endsWith(".jpeg"))
      contentType = "image/jpeg";
   else
      contentType = "text/plain";

   if (WebserverTypeB.debugLevel >= 2 || WebserverTypeA.debugLevel >= 2)
      System.out.println("HttpRequestProcessor: Outputing file.");

   try
   {
      java.io.FileInputStream fileIn;

      try
      {
         fileIn = new java.io.FileInputStream(HTDOCS + filename);
      } catch (java.io.FileNotFoundException fnf)
```

```
         {
            fileIn = new java.io.FileInputStream(HTDOCS + "404.html");
            contentType = "text/html";
         }

         outputHeader(out, true);

         String stringOut = "Content-Length: " + (fileIn.available())
               + LINEBREAK + "Content-Type: " + contentType + " " + LINEBREAK
               + LINEBREAK;

         out.writeBytes(stringOut);

         byte[] buffer = new byte[4096];
         int n;
         while ((n = fileIn.read(buffer)) > 0)
         {
            out.write(buffer, 0, n);
         }

         fileIn.close();
      } catch (java.io.FileNotFoundException fnf)
      {
         // This is only found if the 404.html file is also not found.
         outputHeader(out, false);

         String stringOut = "Content-Length: "
               + (72 + filename.length())
               + LINEBREAK
               + "Content-Type: text/html "
               + LINEBREAK
               + LINEBREAK
               + "<HTML><BODY><H1>Error 404 !</H1><p>File not found! : '"
               + filename
               + "</p></BODY></HTML>";

         out.writeBytes(stringOut);
      }
   }

}
```