UNIVERSITY OF OSLO
Department of Informatics

Predicting Performance and Scaling Behaviour in a Data Center with Multiple Application Servers

Master thesis

Gard Undheim
Oslo University College

May 21, 2006

Predicting Performance and Scaling Behaviour in a Data Center with Multiple Application Servers

Gard Undheim
Oslo University College

May 21, 2006

Abstract

As web pages become more user friendly and interactive we see that objects such as pictures, media files, cgi scripts and databases are more frequently used. This development causes increased stress on the servers due to intensified cpu usage and a growing need for bandwidth to serve the content. At the same time users expect low latency and high availability. This dilemma can be solved by implementing load balancing between servers serving content to the clients. Load balancing can provide high availability through redundant server solutions, and reduce latency by dividing load.

This paper describes a comparative study of different load balancing algorithms used to distribute packets among a set of equal web servers serving HTTP content. For packet redirection, a Nortel Application Switch 2208 will be used, and the servers will be hosted on 6 IBM bladeservers. We will compare three different algorithms: Round Robin, Least Connected and Response Time. We will look at properties such as response time, traffic intensity and type. How will these algorithms perform when these variables change with time. If we can find correlations between traffic intensity and efficiency of the algorithms, we might be able to deduce a theoretical suggestion on how to create an adaptive load balancing scheme that uses current traffic intensity to select the appropriate algorithm. We will also see how classical queueing algorithms can be used to calculate expected response times, and whether these numbers conform to the experimental results. Our results indicate that there are measurable differences between load balancing algorithms. We also found the performance of our servers to outperform the queueing models in most of the scenarios.

Acknowledgements

First and foremost I would like to thank my supervisor, professor Mark Burgess, for his invaluable help and guidance during this semester. He has given me inspiration, motivation and input beyond what I expected. I would also like to express my gratitude towards Espen Braastad, Jon Henrik Bjørnstad, Ilir Bytyci, Sven Ulland and all the other master students at Oslo University College for valuable discussions and help, and also for being good friends. Furthermore I would like to thank my beloved fiancée Cecilie Moen for being so tolerant during my late nights in front of the computer, and for her unconditional love and friendship. Last but not least I would like to thank my family and friends for all the support and guidance I have received during this stressful period.

Oslo, May 2006

Gard Undheim

Preface

A technical paper, *Predictable Scaling Behaviour in the Data Centre with Multiple Application Servers*, based on the results from this thesis has recently been submitted for review to the *17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006)* conference. The paper is written in collaboration with my supervisor, Professor Mark Burgess.

Project Background

The idea for this project was developed during the fall of 2005, when my professor Mark Burgess told me he would like me to look at possible assignments surrounding the subject high volume webservers. Later that fall the school bought a Nortel Alteon Application switch for our lab. The Alteon switch is designed for load balancing traffic between servers. Just before Christmas 2005 we also received 6 IBM bladeservers for our lab, which gave me an excellent opportunity for setting up a perfect load balancing environment in the lab. A quick search in well known libraries such as IEEE and ACM revealed that experiments already had been done to study the performance of load balancing algorithms, but not all the results were coherent. We decided to do another performance analysis of available load balancing solution and also investigate the questions of whether or not queueing theory could be used to predict service requirements such as response time and bandwidth capacity.

Target Audience

Knowledge about networks and basic protocols is an advantage when reading this thesis. Technologies used will be briefly discussed in the opening chapters, but will not be sufficient for fully comprehending the material without prior knowledge of basic protocols such as IP, TCP/IP and HTTP.

All experiments including their purpose will be explained to the reader before the results are presented. Configurations and setup procedures are documented either in the methodology chapter or in the appendix for reproduction purposes.

Terminology

Abbreviations will be avoided where it is reasonable. The abbreviations used in this document will be explained in their first occurence, e.g. Server Load Balancing (SLB). The terms *server*, *webserver* and *fileserver* will be used interchangeably throughout this document. These terms refer to machines serving content to clients. The terms load balancer, application switch and dispatcher are all used throughout the document to describe the unit doing load balancing.

Thesis Outline

The thesis outline contains a coarse description of the sections in this thesis. Refer to the table of contents for a more detailed overview of the structure.

Chapter 1: Introduction
In the first section we describe the motivation for this thesis, and what advantages we can gain from server load balancing. In the second section a coarse description of load balancing techniques are given and we explain what areas we are about to investigate. In the end the reader is given a broad overview of the experimental approach taken in this thesis.

Chapter 2: Background and Previous Research
This part include background material that is necessary for comprehension of the experiments and results described in the following chapters. A survey of previous research in the field is provided along with the theory. The former research will be subject to comparison and discussion.

Chapter 3: Hypotheses
This is a short chapter where we define our hypothesis and shortly discuss their meaning.

Chapter 4: Experimental Design
This chapter starts by defining the constraints of our system. Then we describe our system and the tools used during the experiments.

Chapter 5: Methodology
We start by describing the workflow for our experiments. Then we describe some statistical theories used to analyze our results. In the end we represent a description of our test plan.

Chapter 6: Results and Analysis
This is where we present our results, mostly with tables and graphs. We start by analysing the performance of the servers. Then we compare

queueing models with results from experiments. In the end we look at scalability and algorithm performances.

Chapter 7: Conclusions and Discussion

This chapter is dedicated for discussions and conclusions drawn from our results. Further work will also be proposed.

Appendix

This part of the document includes scripts, graphs and programs developed during this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We believe in the possibility of a theory which is able to give a complete description of reality, the laws of which establish relations between the things themselves and not merely between their probabilities ... God does not play dice. - Albert Einstein

1.1 Load Balancing - A Valuable Asset

Downtime often has a direct dollar value associated with it. Large sites can lose thousands of dollars or more in revenue every minute their site is unavailable. By using server load balancing (SLB) or global server load balancing (GLSB) it is possible to provide better reliability since load balancing algorithms can facilitate fault resilience. It also enables the possibility to do server upgrades and maintenance without interrupting the service offered. If servers crash and need to be turned off due to maintenance, it will only cause a performance degradation [15], while the service is still available on the servers remaining active. Research by Bhatti [6] shows that the average tolerance for delay is around 8 seconds for a normal user loading a web page. With SLB, user perceived Quality of Service (QoS) can be improved by dynamically distributing load between several servers.

A Service Provider often needs to conform to a Service Level Agreements (SLAs). SLAs describe limits for the service delivered, such as response time, uptime, bandwidth etc. A SLA is usually created in mutual cooperation between client and provider. These agreements are often static over periods of time, and are renegotiated on regular basis. Since network traffic is subject to rapid change (e.g. slashdot effect, where a popular website can cause highly increased traffic to smaller sites by linking to them) it is of great importance for the service provider to be able to adapt to changes that might occur due to increased traffic against the servers. Load balancing provide the possibility to adapt the network to increasing demands, or redirect traffic based on dynamic

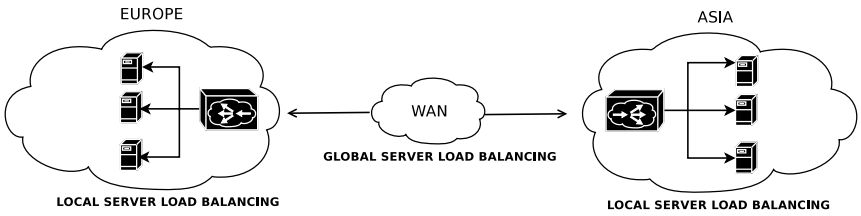

Figure 1.1: **Overview of GLSB and SLB:** *The figure shows that GSLB can distribute requests to different physical locations, while SLB distributes requests among different servers usually located on the same location*

or static metrics.

1.2 Predicting Service Scalability

Meeting SLA requirements with variable demand and resources has become very important during the last years [14, 32]. One common way to solve this is to overprovision, e.g. using more resources than needed, within acceptable margins. In this thesis we will look at how low level load balancing scales, and how simple queueing models can be used as estimators for response time.

1.3 Aspects of Load Balancing

Load balancing is a generic term that can be divided in two main branches: wan load balancing, also referred to as Global Server Load Balancing (GSLB) and Local Area Network (LAN) SLB. Whereas GSLB can be used for distributing load between servers all over the world, LAN SLB is used to distribute load between servers on the same LAN (see figure 1.1). In figure 1.2 we see a cause tree showing the most important factors to consider when implementing SLB. Quality of Service (QoS) refers to the characteristics of a network and the threshold values to maintain user satisfaction. QoS is closely connected to SLAs, whereas the SLA include guidelines or limits for the QoS metrics that need to be fulfilled.

The two first nodes in figure 1.2 shows *network delay* and *site delay*. The former refers to all aggregated delays between the client and gateway of the service provider, while the latter considers delay caused by factors residing in the service providers network. In this experiment we will consider the rightmost node, which is *switch/router*. Below the *switch/router* node we see key elements like SLB algorithms and topology implementation. We want to investigate how these factors affect QoS.

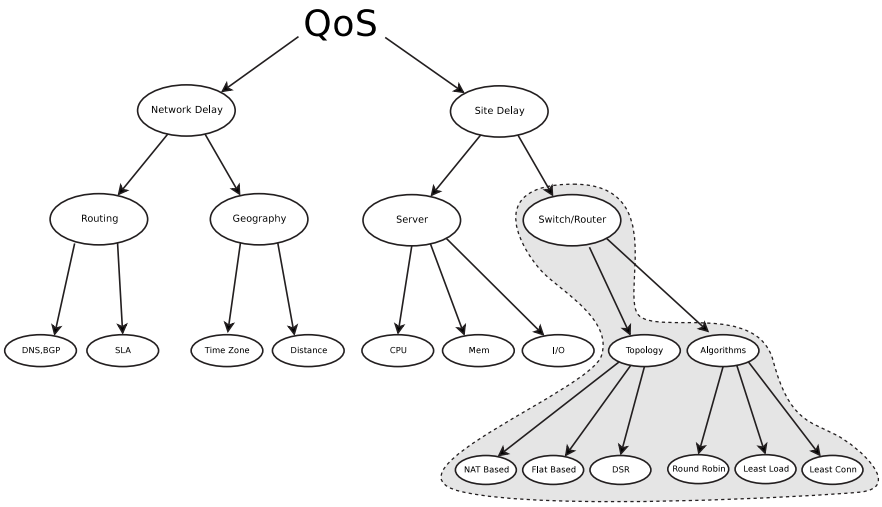

Figure 1.2: **Cause tree for QoS using load balancing** *The areas marked grey are subject to investigation in this thesis*

1.3.1 WAN Load Balancing

Network delay can be improved by doing Load Balancing on the WAN side, where possibilites such as BGP and DNS balancing are available. On the WAN side we need to consider locality, meaning the proximity between client and server. DNS load balancing can redirect requests such that clients connect to servers in near proximity, which can reduce transfer delays (for further investigation of these subjects see [9]).

1.3.2 LAN Load Balancing

Server delay refers to latency caused on the LAN side where the servers are located, and can be induced by the server itself or the network and network equipment used. A server can be overloaded in terms of memory usage, number of users, cpu utilization etc. This might affect user perceived QoS in terms of reduced response time, packet loss or even server downtime. SLB gives the opportunity to use specialized servers that serve different contents, combined the servers incorporate a complete web page. This can be convenient for reducing response time since it enables the possibility to create servers that are optimized for serving static or dynamic content, such as pictures or cgi scripts [15].

On the network side on the LAN we have switches and routers, possibly acting as dispatchers (see section 2.3). Limitations such as bandwidth, topology and if SLB is used: load balancing algorithms affect the performance and will be subject to measurements in this assignment.

1.3.3 Experimental Overview

This experiment will focus on the factors marked grey in figure 1.2, which means the network part on the LAN side. One of the objectives of this experiment is to see whether or not queueing theory can be used to predict hardware requirements for fulfilling SLAs. To investigate whether this is possible or not, a model of our network will be created. To create this model we need to find the performance of our servers. We will not investigate how to improve server performance, we will treat each server as a *"black box"*, only considering the throughput measured in requests per second. Expected response times will be calculated using measured throughput for the servers using queueing theories. These results will be compared to see if they scale with results obtained from measurements taken when generating http traffic against the servers. We will also do experiments to reveal possible relationships between performance and factors such as traffic intensity, load balancing algorithms and traffic characteristics.

Chapter 2

Background and Previous Research

Server Load Balancing (SLB) is a widely used term, and its meaning and perception might differ depending on context or author. This is because of the fierce competition between different vendors, where each producer use their own terminology. This often makes it hard to compare one product and technology against another without vendor specific knowledge. Throughout this report SLB will be referred to as a process of distributing web traffic among several servers. An illustration of a simple SLB setup is shown in figure 2.1.

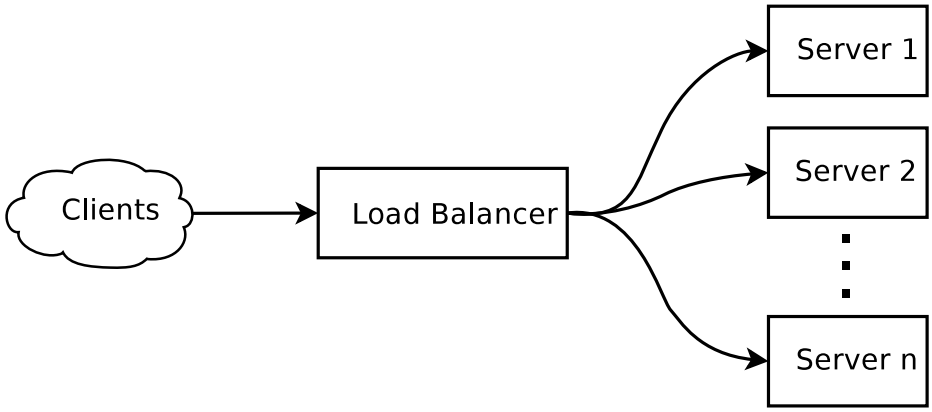

Figure 2.1: **Basic Load Balancing Scheme:** *Figure showing a load balancer (dispatcher) that distributes requests among the available servers*

Cardellini et al. divide load balancing into four schemes: *client-based, DNS-based, dispatcher-based* and *server-based* [11].

Load balancing at the client can be divided into two groups, the transparent mode and the non-transparent mode [15]. The latter requires the client software to be aware of available servers. An example of this is the way Netscape load balanced their webservers earlier. They integrated a list of their available webservers into the browser, from which the browser then selected which server to use in a round-robin style [22] (see figure 2.2). The apparent problem

with this solution is that each client needs to be updated if there is a change in the pool of available web servers. The former mode of client load balancing deploys load balancing using DNS servers. The DNS servers have several IP's for each domain name, and rotate the returning list of addresses for each request.

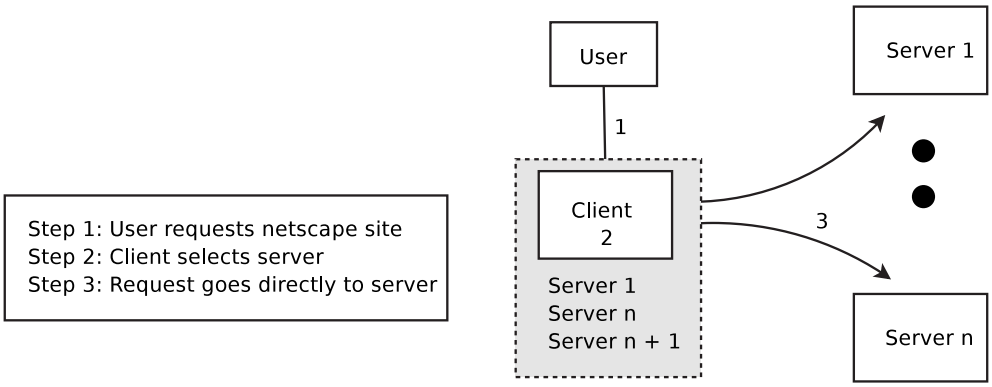

Figure 2.2: **Client load balancing:** *Client load balancing where the client browser holds several records for each domain name. For each request the client alternates which server to use according to some preconfigured algorithm*

Load balancing at the server can be achieved by setting up proxy servers that do not process content, but redirect requests to other available servers based on detailed server metrics. The problem is that requests are redirected on the application level, and the requests must traverse the entire protocol stack (see section 2.2) 4 times before it is processed. This solution is only useful if the bottleneck is the processing of content and it is not scalable [15]. Server based load balancing can also be achieved by having servers redirect packets only when its own utilization is above a certain threshold (See figure 2.3). This means that every server needs to inform the others about its own utilization, and the frequency for this information needs to be high to make this work [5, 15]. High frequency of information between servers can cause bottlenecks both in network and server utilization due to processing of interchanging information.

Load balancing in the network is achieved by using a load balancer or dispatcher that intercepts and redirects packets to suited servers (see figure 2.1) based on predefined algorithms. Load balancing web servers using a dispatcher can be divided into two parts: the entity performing the load balancing and the algorithm used to distribute client requests among the pool of available servers [30]. Network SLB is transparent to the end user, and can operate in several modes using different algorithms. Below is a list of advantages using SLB:

Flexibility: SLB allows an enterprise to add or remove servers without affect-

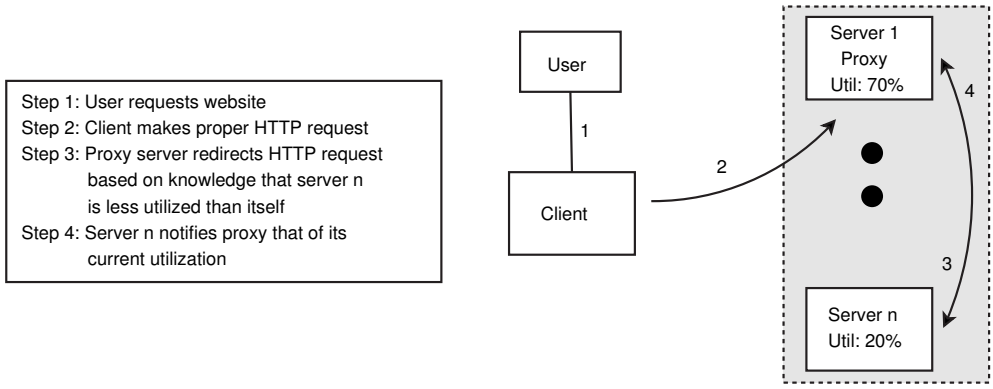

Figure 2.3: **Server side load balancing:** *Server side load balancing where one of the webservers act as a proxy and decides which of the servers that is least utilized and therefore best equipped to handle the request*

ing the users. This makes it possible to do maintenance on servers providing critical services without affecting the avalability.

Scalability: If traffic increases, it is easy to add additional servers to handle the load. Using many servers is cheaper in purchase cost than buying one high-end server.

High Availability: Most SLB implementations offer the possibility to monitor server health. This way machines that are malfuntioning can be removed from the pool of available servers, and be activated again when they are fixed. Load balancers often support redundant configurations, using a heartbeat protocol between the devices. This removes the problem with having a single point of failure within the load balancer itself.

Increased Server Utilization: SLB can be implemented for several protocols. This makes it possible to load balance protocols that might peak in usage at different times of day across several servers. This can be used to reduce number of servers.

2.1 The Beginning

The first attempts to balance traffic between multiple webservers were done by Netscape. They hardcoded the addresses of all their servers into the Netscape browser. The browser then used a round robin algorithm to alternate which address to use. This is not a very flexible solution, and the next attempt of load balancing was achieved through DNS load balancing.

DNS servers have zone files that include mappings between hostnames and ip addresses. When a user tries to access `www.example.net` the host-

name will be resolved to the appropriate ip address through a request made to a nameserver which is specified on a particular machine. The drawback is that many clients and local DNS servers cache these responses. This can cause a skewed load on the clustered web servers [15]. Below is a description of how the correct ip address is obtained by a client:

1. User tries to access `www.example.net`
2. DNS request is sent to configured DNS server(often provided by ISP).
3. The DNS server checks whether it has the address of `www.example.net` cached. If not, it sends another DNS request to one of the root DNS servers.
4. The root server will respond with the address of the autorative DNS server for the requested hostname.
5. The DNS server will then ask the authorative DNS server for the address, and receives an address in response.
6. The web browser will connect to `www.example.net` using the ip address received from the DNS server.

The procedure above is a simplification, and might include several steps before reaching the authorative DNS server. It is also two ways a DNS server can be configured to do lookups, recursive and iterative mode. The procedure described above is an iterative lookup. This is out of the scope of this experiment, for further details see [16, 9].

An authorative DNS server usually have one mapping between a hostname and a ip address, in BIND the configuration looks something like this:

```
www.example.net      IN      A      212.12.12.15
```

When a DNS server is configured to do load balancing between one URL and a set of IP addresses each entry includes pointers to all available IP addresses for that URL. BIND is also configured to hand out the available addresses for a domain name in a round robin manner. The zone file for a DNS server with several addresses mapped to one hostname will look something like this:

```
www.example.net      IN      A      212.12.12.15
                     IN      A      212.12.12.16
                     IN      A      212.12.12.17
```

There are several caveats to consider if using this solution. First there is the problem that most DNS servers cache the responses they receive from other DNS servers. In reality this means that if a client asks for the address of `www.example.net` from a local DNS (non-authorative), it might only receive an IP address cached by that DNS server. All other clients using the same non-authorative DNS server will receive the same address until the cache of the DNS server is cleared. To avoid this problem one could adjust each entry on the authorative DNS server with a short validity period, which is done through setting a Time To Live value (TTL). A cache entry can be valid from anywhere between a day to an entire week. In [15] they tested different TTL values for a DNS server handing out responses for a cluster of 8 web servers. With a TTL of 1 hour the load on the servers (measured in connections handled by each server) ranged between 10.7 to 15.8 percent. With a TTL of 24 hours the load ranged from 8.1 to 18.5 percent. Bryhni also states that DNS caching can introduce skewed load on a server farm by an average of as much as ± 40 percent of the total load. Dynamically setting TTL values for DNS through hidden load weight for each domain, client location and server condition can improve performance [5]. Bryhni [15] also mentions that if TTL values are small, the DNS traffic itself can cause significant network overhead, since it does not carry any user information. Barford and Crovella says that DNS Load balancing can not use more than 32 servers due to UDP size constraints [5]. Cardellini [11] also points out the fact that intermediate DNS servers caching responses, often ignore small TTL values to avoid unecessary DNS traffic.

Another problem using DNS based load-balancing is that there is no way for a DNS server to notice when a server goes offline, which leads to DNS servers handing out addresses that are unreachable. Even if the authorative DNS server is updated the moment a server goes online/offline, other cached DNS entries need to expire before they request a new address [15].

In [11] performance tests are performed to compare dispatch based (see section 2.3) and DNS load balancing. Results show that in their testing scenario, all DNS schemes (constant TTL, adaptive TTL and Round Robin) are outperformed by a simple Round Robin (RR) dispatch based method. The DNS approach with best performance is the one with adaptive TTL, where the results show a maximum server node utilization of 90 percent, while the dispatch based approach have a maximum of 80 percent. They consider server nodes that are utilized above 90 percent to be overloaded. The DNS RR approach shows that at least one server node is overloaded 70 percent of the time, while the DNS with static TTL overloads at least one node 20 percent of the time. They conclude by saying that network bandwidth, more than server node capacity, is likely to be a limiting factor for load balancing techniques. Therefore a combination of DNS based and dispatch based load balancing should be implemented, so that client proximity and network load can be considered.

2.2 The OSI Model

When we speak about Server Load Balancing the OSI model is often mentioned. The OSI model is meant as a framework for developing network protocols, the reference model consists of 7 layers. The OSI model is used to describe how different types of protocols work together, such as HTTP, IP, TCP and 802.3 (Ethernet). We often refer to layers in the OSI model to describe how SLB works. It is therefore important to know what each layer does. In figure 2.4 the different layers in the TCP/IP and OSI reference model is shown.

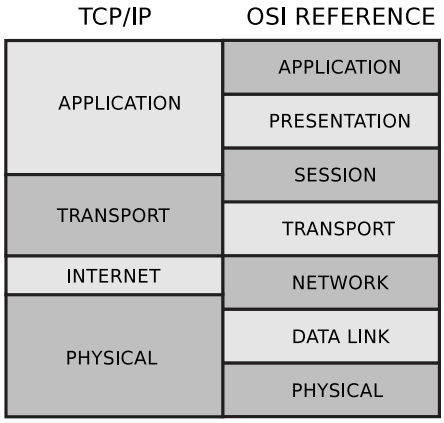

Figure 2.4: **OSI and TCP/IP model:** *The figure shows how the TCP/IP implementation of the layers conform to the OSI reference model*

Layer 1 This layer is often referred to as the ”Physical” layer. The protocols in this layer specify how 1s and 0s are transmitted over the physical medium they are operating. Load Balancing is not performed at this level.

Layer 2 On this layer it is specified how the packets are encapsulated before being transmitted out on the link. Ethernet, which is the most common Layer 2 protocol, also has error correction in its header. This layer is often involved when talking about SLB, here we can tune the maximum size for frames transmitted (usually 1500 bytes, but Jumbo Frames can support up to 9000 bytes). In some load balancing schemes we also need to take into consideration the Adress Resolution Protocol (ARP), which is used for looking up MAC addresses for devices on the same segment in the network. When using the topology scheme Direct Server Return, DSR (see section 2.4.1), we need to disable the ARP protocol on the servers.

Layer 3 On this layer the routers operate. They forward packets based on IP addresses according to static routes or routes created by routing proto-

cols such as Routing Information Protocol (RIP) and Open Shortest Path First (OSPF). The source IP address is often used in SLB context, especially when talking about session persistance.

Layer 4 On this layer we often speak of two sets of protocols; connectionless and connection oriented. Typically UDP represents the former and TCP the latter. Higher level services such as HTTP, FTP etc. all have dedicated port numbers when they are encapsulated in a TCP or UDP packet. Source port, destination port, source IP and destination IP combined make a socket, used as a unique identifier for connections. This 4-tuple is often used to track connections and separate them when implementing SLB.

Layer 5-7 These higher levels consist of protocols used directly by users. FTP, HTTP, SSH and DNS are just a few. There are also a set of presentation protocols to present information to us in ways we can understand (e.g. HTTP packets are usually presented using a web browser). Using these protocols for load-balancing is possible, but often relies on powerful devices because of the need to process each packet at such a high level in the protocol stack. This type of load balancing is often used to balance load between servers serving different types of content (e.g. separating content between static and dynamic content).

2.3 Dispatch Based Load-Balancing

There are several ways to deploy a SLB scheme, which makes it easier to adapt the solution to different needs and network architectures. The most common solution is to centralize scheduling and completely control client-request routing, an additional network component called a dispatcher will be used for this [11]. Dispatch based load balancing has several possible implementations. What characterizes these modes of operation is how the dispatcher forward packets, and how the headers are changed. Dispatch mode can operate on layer 2 and up in the OSI model, which means that it supports both switching, routing and forwarding based on higher levels such as URLs and TCP sessions.

An important term when describing load balancing is the Virtual IP Address (VIP). The VIP address is configured on the load balancing unit, and is used by the clients when connecting to the servers behind it. The VIP address can be equal to the actual address of each server, or it can be the ”external” address before a Network Address Translation (NAT) takes place. This depends on which mode we use for load balancing. All modes share a common feature, the load balancing is transparent to the clients. Clients connect to the VIP address as if it was one single server serving all the content. Behind the VIP

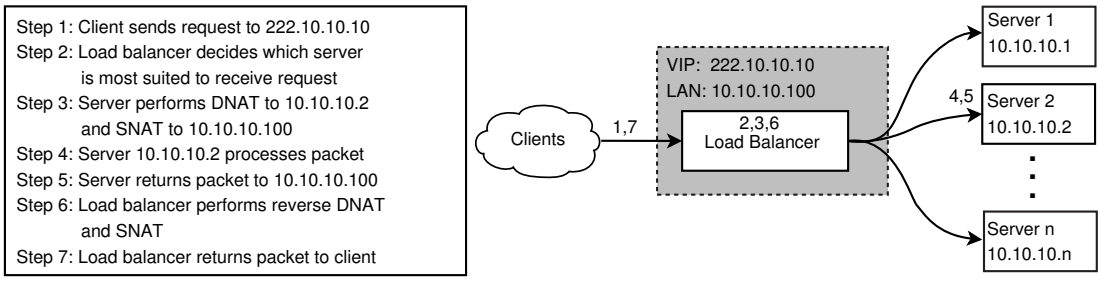

Figure 2.5: **Virtual Interface Adress:** *The load balancer acts as a proxy and intercepts all packets before they reach the servers. In this case a NAT based architecture is used where all servers have local adresses*

there can be anywhere from 1 to many servers. In figure 2.5 a setup using a VIP address is shown.

2.4 Dispatch Modes

Dispatch modes can be implemented in many ways, and the terminology used to describe these implementations is not consistent. Vendors use their own terms, and scientific articles often refer to terminology used by vendors. This makes it harder to compare different technologies and papers. In the following sections I will explain the different architectures, and refer to them using the terminology I have found to be most common and appropriate. The terms are in accordance with the ones used in the O'Reilly book: *Server Load Balancing* [7] by Tony Bourke and the article *Comparison of Load Balancing Strategies on Cluster-based Web Servers* [30]. They divide SLB into two main branches: Flat-based and NAT-based.

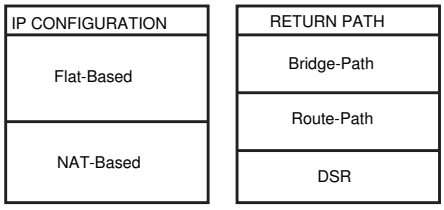

Figure 2.6: **Possible Implementations of SLB:** *2 sets of topology implementations: flat-based and NAT-based. 3 return paths that packets can be sent: bridge-path, route-path and Direct Server Return (DSR)*

All possible implementation of SLB infrastructure can be described as a variation of those shown in Figure 2.6. The first box show the two variations available for the IP topology. When implementing flat-based SLB the VIP addresses and the IPs for the real servers reside on the same segment. For NAT-

based SLB the VIP addresses and real servers are on different subnets. The second box show how the traffic can be routed or switched from the real servers back to the client. Bridge-path is when the dispatcher acts as a bridge, and is in the layer-2 path between the real server and the client (this solution only works with Flat-based SLB). Layer-2 path is the path a packet follows inside one segment, using only layer 2 information (MAC addresses) to make forwarding decisions. When using Route-path the dispatcher acts as a router, and is in the layer-3 path between the real server and the client. Layer-3 path is the path where IP addresses are used to make forwarding decisions. DSR is an abbreviation for Direct Server Return, and is implemented so that the traffic from the real servers goes directly to the client without passing the load balancer. What is common to all these implementations is that the dispatcher should be close to the servers to avoid network capacity becoming a bottleneck [5], and keep, if used, server load information as accurate as possible [15].

2.4.1 Flat-Based SLB Network Architecture

In this architecture the VIP address resides on the same segment as the IPs of the real servers. In figure 2.7 we see an example of a flat-based SLB architecture. There can be variations in the setup.

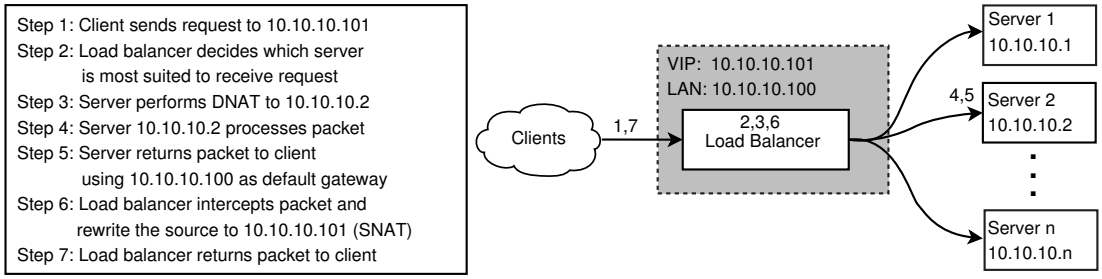

Figure 2.7: **Flat SLB Architecture:** *The load balancer acts as a proxy and intercepts all packets before they reach the servers. Load balancer uses Destination Network Address Translation (DNAT) on incoming packets and Source Network Address Translation (SNAT) on outgoing packets. When implementing this architecture the load balancer need to be in the route path between the server and the client*

2.4.2 NAT-Based SLB Network Architecture

Directed mode operates on layer 3. The load-balancer translates its' VIP address to the real server address. The answer from the real servers are then translated back to the VIP address before they are sent to the client requesting files. This process makes it possible to place the real servers on different seg-

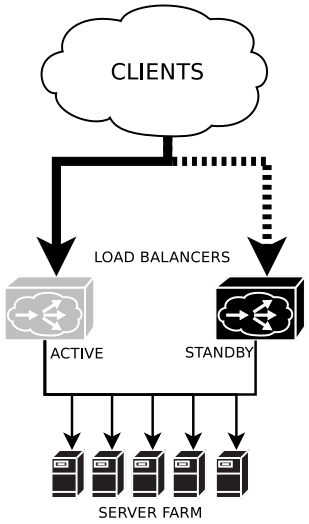

Figure 2.8: **Active - Standby scenario:** *The active load balancer receives all traffic. If it fails the backup unit will take over. Information such as connection tables, cookies, weighting information etc. is transferred from failing unit to backup unit if possible*

ments, hence make it easier to alter physical locations and better use of the IP address space.

Dispatch mode operates on layer 2. This requires the load balancer and the real server to be on the same segment connected through a switch. The VIP address on the load-balancer is the same as the ip address of the real servers. Because of this it is recommended to configure a non-arp secondary or alias interface on the real servers assigned to the VIP address, this to avoid issues surrounding duplicate ip addresses in some OS'es.

2.4.3 Redundancy

One of the important advantages of implementing SLB is to facilitate fault resilience. With one load balancer we still have a single point of failure as we also would using one webserver. Most load balancers support redundant architectures where one can employ one or several units in backup or parallell (see figure 2.8). Proprietary heart beat protocols are then used between the units to communicate. It is usual to use the Virtual Router Redundancy Protocol (VRRP) to share an adress between the units. If a unit fails, the next in line will then assume the VRRP adress and start load balancing the traffic. Proprietary protocols are usually responsible for trying to transfer state information such as connection tables, cookies etc. from the failing device to the one taking over.

2.5 Load-Balancing Algorithms

In this section I will start by describing the most ordinary load balancing algorithms, in other words the ones that rarely change due to vendor specific solutions. In the end I will summarize some of the research on the field, and compare results obtained from different experiments.

Round Robin Algorithm: This is a simple algorithm that distributes requests to different servers in a round-robin manner independent of any information about load, active connections, bandwidth usage etc. The algorithm is known to be effective in the terms of processing needed for the dispatcher, but can overload servers if the sequence of requests is nonoptimal [15]. In inhomogeneous server environments with long-range dependent traffic distribution, this algorithm should not perform better than random chance.

Least Connections Algorithm: The least connections algorithm keeps track of how many active connections each node currently has, and sends new connections to the node with least active connections. If two or more servers have the same amount of active connections, there are several ways to decide which server to use. Some implementations use the lowest IP address, or round-robin, while others use an identifier for each server, and sends the request to the server with the lowest one [15]. The last solution have the effect that the same server will receive requests each time the system is empty. This algorithm is said to be somewhat self-regulating since faster servers can process requests at a higher rate, hence receive more requests [24].

Server Response Algorithm This is a generalization of algorithms that use real time server information to balance load. Some implementation use the Round Trip Time (RTT). The RTT is calculated by sending out Internet Control Message Protocol (ICMP) echo requests, and calculate the time it takes to receive an ICMP echo reply, to get an impression of which server is least utilized. Some monitor the TCP sessions and calculate the time between a request is forwarded and to the first reply back. Servers with a small response time is considered less utilized than server with higher response times. There are a number of different solutions for selecting the appropriate server when two servers have the same value. Some use round-robin between the equal servers, while others use a least loaded scheme when servers can not be separated using the metrics given in the server response algorithm [15] More accurate system monitoring tools are available, but often in proprietary solutions. Agents on each server can serve or push information to the load balancer, which

again will use this information to make adequate decisions for distributing requests [15].

2.5.1 Performance Evaluation - Algorithms

Many experiments concerning performance testing has been performed to uncover which algorithm that is most efficient when load balancing web traffic [21,18,30,11,28,1,31,3,23,4,15]. These experiments either use a trace driven or program generated traffic. Program generated traffic is created by the use of tools like `specWEB`, `httperf`, `siege`, etc. The trace driven approach uses playback of previously captured traffic. The majority of experiments focus on response time, and how this factor vary depending on number of servers available and traffic intensity.

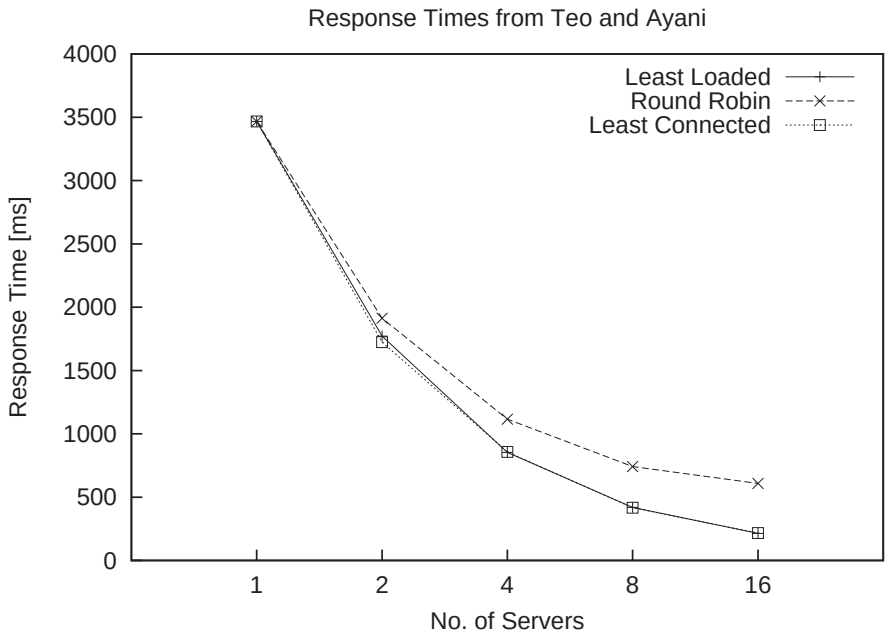

Figure 2.9: **Response Times from Teo and Ayani [30]:** *The graph shows response times using the least loaded algorithm, least connections algorithm and round robin algorithm. The number of servers range from 1 to 16, and their experiment yield that round robin is outperformed by the two other algorithms*

In [30] they claim that Least Loaded (LL) algorithm is superior to Round Robin (RR), although it is hard to implement because you need detailed information about workload on each server (see figure2.9). RR algorithms are measured to have the highest response times, especially under low to medium workload. The Least Connections (LC) algorithm performs well for medium to high workloads, but they measure it to be 2-6 times worse than the least loaded

algorithms under low loads. Their results show that all three algorithms converge when the load increases. In table 2.1 we see that average response time decreases when they increase the number of servers. Arrival rate and number of servers are increased proportionally, so that the load per server is constant during all tests. The authors in [30] give no explanations for their result. Due to the increased amount of processing needed at the dispatcher when increasing the load, one would expect a decrease in throughput.

No. of servers	LL Response Time	RR Response Time	LC Response Time
1	3467	3467	3467
2	1772.0	1913.2	1724.6
4	853.1	1115.9	856.7
8	419.7	741.4	416.0
16	213.0	608.0	215.6

Table 2.1: **Results from research by Teo and Ayani [30]:** *Response times obtained when using three different algorithms with increasing number of servers. LL equals Least Loaded, LC equals Least Connected and RR equals Round Robin*

Cardellini *et al.* show in their experiment results that contradict results shown in [30, 5], they found that the RR algorithm demonstrated better performance than other load balancing algorithms due to less need for processing power at the dispatcher unit. Bryhni [15] proposes the adaptive load balancing algorithm using the Round Trip time for ICMP messages to weigh the servers as a good alternative to the RR. Although it needs improvement to reflect current server load, especially during transient high-load conditions.

2.6 Performance Evaluation - Topology

As we have seen there is more than one way to implement SLB. Which solution to choose depends on infrastructure, the need for security and traffic characteristics.

The dispatcher unit needs to process each packet before forwarding it to a server, which indicates a possible bottleneck [2]. When network load balancing is used, and we use unique IP addresses for each of the real servers, the load balancer need to recalculate the IP checksum of each packet forwarded. This means additional overhead compared to load balancing on the link layer [15]. If the client requests a large file, it means that several reply packets are sent by the server, hence several ACK packets from the client. This results in increased utilization on the dispatcher, since it needs to process more ACK replies. The process time for each request is therefore proportional to the size of the request. In [30] they hypothesize that:

$$Dispatcher\ Service\ Time = \frac{L}{K} + C \tag{2.1}$$

Where **L** denotes the size of the request in bytes, **K** is a linear factor to model the processing of ACK packets and **C** is a constant factor modelling the overhead for setting up a TCP session. By sampling request sizes between 1 byte and 2MB they found values for **L** and **C**.

As mentioned in [15] the amount of processing performed by the load balancer can be significant to the overall performance. In [2] they discuss the amount of processing needed for different modes of load balancing:

- Dispatch Mode
- Directed or server NAT
- Client NAT
- Connection spoofing
- Direct Server Return

When configuring the load balancer's mode of operation, the first thing to consider should be service requirements. A secondary consideration is performance. In [2] they have arranged the modes in a list relative to their expected performance. The performance is measured relative to the amount of processing needed for each packet. In DSR mode for example, the packet can go directly from the server to the client without alterations, which might increase performance.

1. Direct Server Return
2. Dispactch Mode
3. Directed or server NAT
4. Client NAT
5. Connection spoofing

Table 2.2 points out the tasks or changes a load balancer must perform to incoming and outgoing packets under different modes of operation.
From 2.2 it is clear that Direct Server Return is the implementation with the least amount of processing. Since outbound traffic usually exceeds inbound traffic, the advantage of DRS becomes even greater. When using connection spoofing, the packet needs to get its header rewritten in both layer 2,3 and 4 of the protocol stack, which might cause the dispatcher to become the bottleneck.

Mode/Task	Inbound Traffic	In-/Outbound Traffic	MAC Header	IP Header	TCP/UDP Header	Connection Spoofing
DSR	•	•				
Dispatch		•	•			
Directed		•	•	•	•	
Client NAT		•	•	•		
Connection Spoofing		•	•	•	•	•

Table 2.2: **Performance considerations from Cisco [2]:** *The table shows which alterations that are done to each packet using different topology solutions when implementing SLB. As we see the Direct Server Return solution needs the least alterations, while connection spoofing needs the most.*

2.7 Web-Traffic Characteristics

When doing simulations it is important to take into consideration what type of traffic that is used. Traffic characteristics can affect performance. How web traffic is distributed have been subject for discussion in many articles [5,12,26] Barford and Crovella identified 6 constraints that apply to http traffic: server file distribution, request size distribution, relative file popularity, embedded file references, temporal locality of reference and idle periods of individual users [5].

Network traffic can be self-similar, which means that traffic can show significant variability over a wide range of scales. This characteristic has been shown to have a negative impact on network performance [13,25]. In [5] they mention two methods for capturing these properties: using traces from previous traffic or an analytical approach. Analytical workload generations start with mathematical models that represent the characteristics of web traffic, and then generate output according to these. The disadvantage of using traces is that the traffic is a ”black box”, and without the knowledge of traffic characteristics, it is often harder to understand causes of behaviour on the server. Traces offer no flexibility when it comes to adjusting the load depending on the site, or if characteristics change with time. Analytical traffic generators are often more complex to create, but offer greater flexibility, such as file size adjustments, number of clients, packet inter-arrival times etc.

With HTTP 1.0 it is possible to achieve load balancing by directing requests aimed at one logical address to different machines, this is because the HTTP protocol is stateless and creates new TCP sessions for every object on a web page [15]. The Internet Engineering Task Force (IETF) has standardized a new HTTP protocol, version 1.1. This version enables several objects to be transferred per TCP session [17].

In [5] they used both exponential (see figure 2.10) and heavy-tailed traffic to model client traffic. The heavy-tailed client traffic were represented using Pareto and Weibull distributions, while the exponential traffic were created with poisson distributions. The experiments showed that using the exponential distribution model the dispatch based approach outperform both DNS based and server based load balancing. The dispatcher based method keeps the webserver utilization below 0.8. DNS with adaptive TTL, and server based load balancing manages to keep server utilization below 0.9. While DNS with constant TTL values had at least one overloaded node (utilized above 0.9) for almost 20 percent of the time. The RR DNS solution overloaded one server for more than 70 percent of the time, and turned out to be the worst solution.

When using heavy-tailed traffic distribution the results degrade for all approaches. Dispatch based and DNS with adaptive TTL are the two approaches that perform best. The server based approach does not cope with the heavy-tailed distribution and at least one server was over utilized for more than 50 percent of the time.

Traffic intensity can constrain load balancing schemes even more than server capacity. This means that LAN based webserver clusters are just a limited solution to scaling increased client load. For an optimal solution a WAN based solution should be combined with a LAN based solution. The WAN based solution needs to take proximity and client load into consideration. One of the great challenges is how to dynamically evaluate such information as it rapidly changes in the Internet environment.

2.8 Performance Model

To create a model of our system we want to create a reasonable abstraction level for our experiments. We want to look at the throughput each webserver can handle, and not necessarily how each server is configured. To find the service time for a server can be complicated, in [30] they identify three main delay sources to consider: *CPU (together with memory access time), disk access time and network delay time*. The delay caused by disk access can be avoided by storing all the content in memory instead of disk. This is a viable assumption considering the large amount of memory often found in web servers today. Network delay time is also considered to be a negible. This leaves us with only CPU and memory access time as sources for delay on a server.

We define the throughput of each server as a function $X_0(n)$, where n is the number of request present on the server. This means that we wish to look at the server as a **black box**. Figure 2.11 shows a reasonable assumption about how the throughput will change with n number of request present in the system. The throughput will stop increasing when the server is saturated, and then the server needs to start queueing jobs.

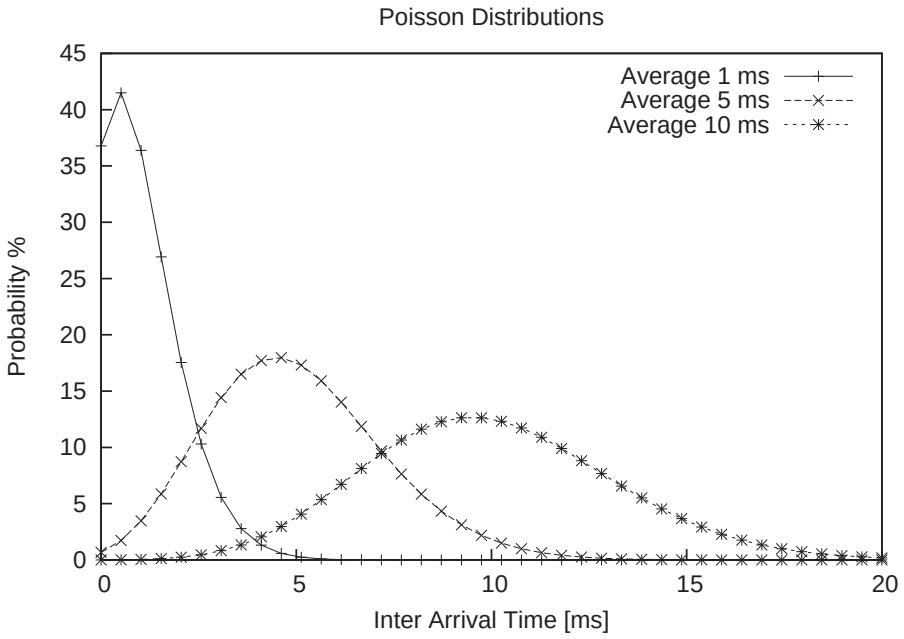

Figure 2.10: **Poisson Distribution:** *The graphs show poisson distributions with different average rates and exponential inter-arrival times. The function is only defined at integer values of x. Connecting lines do not indicate continuity.*

In [8] it is mentioned 6 parameters to consider when using queueing models:

- **Arrival time distribution process:** This is the inter-arrival time between arriving requests. A lot of research has been done to find out whether the inter-arrival time of network traffic form a Poisson or Long-tail distribution. In our experiment we will generate network traffic with Poisson distribution (see figure 2.10).

- **Process time distribution:** This is the time a client is engaged in requesting a service. Research has shown service time to be heavy-tailed in distribution [15]. Due to software limitations the service time in this experiment is static.

- **Number of servers:** This is the number of computers processing requests. In our experiment we have at maximum 6 servers available.

- **System processing limit:** If the servers have a maximum throughput this will affect the system as a hole. In our case we willl find the maximum throughput of each server measured in *KB/s* for different file sizes and cpu consuming scripts.

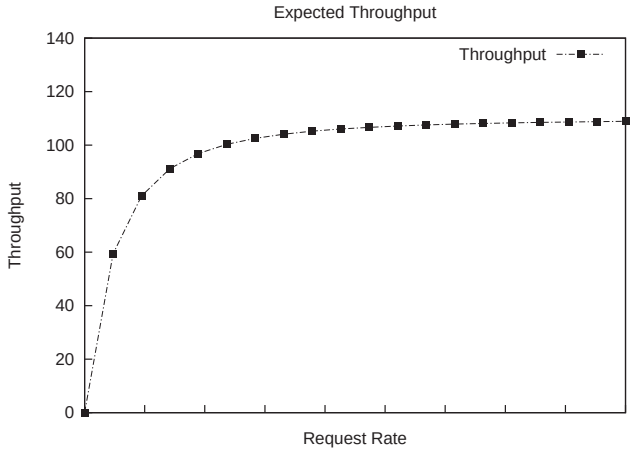

Figure 2.11: **Exptected trend for throughput:** *We expect the throughput of each server to increase proportional with the increased request rate until approaching maximum limit. Then the rate will flatten out and remain constant.*

- **Maximum population size:** Maximum number of clients that can ask for services.
- **Scheduling policy:** This refers to how the queueing policy is implemented. In our case which load balancing algorithm the dispatcher uses, and the topology of the network.

2.8.1 Infinite Queue Model

To create a model of our web server we make the assumption that there is no maximum length on the queue that our servers are capable of handling. This is of course an erroneous assumption, but it is sufficient as long as our model is only used to describe situations where the server is not utilized above its limitations [29].

We assume that all requests are statistically indistinguishable, which means that the requests present in the web server are not important, only the number of requests present. This is called the *homogeneous workload* assumption [20]. Since our web server model accepts requests regardless of how many that are present in the system, we get what we call an *infinite queue* [19].

If our model should work, it needs to be in something that is called an *operational equilibrium* [10] (this means that for a given time-interval the number of requests that goes into the web server needs to be equal to the number of requests that are processed by the server). We can now describe each web servers state by saying how many requests that are present in the server, waiting or receiving processing. This indicates that the previous states of the server are irrelevant, and are what we call *memoryless* or *Markovian* assumption.

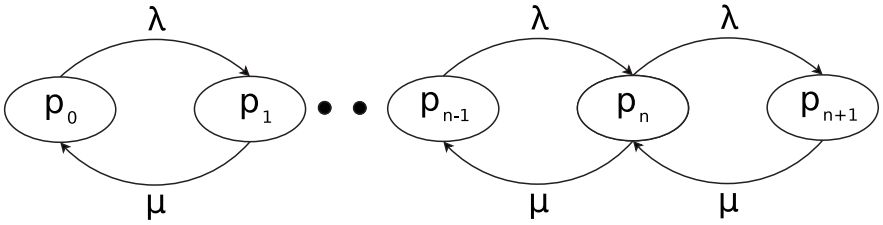

Figure 2.12: **Transition between states:** *The figure shows how incoming (λ) and processing (μ) requests take our system from one state to another. An operational equilibrium is accomplished when $\lambda = \mu$*

To visualize the states of each server, we define that a state is given by an integer: $p_0, p_1, p_2, \cdots, p_n$. The transitions between states are caused by new requests arriving at the server, this event will take the state from p_n to p_{k+1}. The rate at which these transitions occur will depend on the properties of web traffic, we use a *Poisson* distribution to generate traffic. The *Poisson* process is memoryless in distribution[1]. Each time the server completes one request, a state transition from p_n to p_{n-1} occurs (see picture2.12).

The rate at which requests arrive: $\lambda\frac{requests}{second}$ must be equal to the rate at which requests are processed: $\mu\frac{requests}{second}$. There is a probability that the server is in a given state that is given by: p_n, where $(n = 0, 1, 2, \cdots, \infty)$. We can then write that:

$$\begin{aligned} \lambda \cdot p_{n-1} &= \mu \cdot p_n, \\ p_n &= \rho p_{n-1} \end{aligned} \tag{2.2}$$

for any n, where $\rho = \frac{\lambda}{\mu}$. ρ is an estimate of the traffic intensity, and we see that if $\rho > 1$ the incoming rate is higher than the outgoing rate. We will see that if ρ approaches 1 the average number of tasks in the queue will approach infinity. Since the equation in Eq. (2.2) holds for all n we can write:

$$\begin{aligned} p_1 &= \rho p_0 \\ p_2 &= \rho^2 p_0 \\ &\cdot \\ &\cdot \\ p_n &= \rho^n p_0 \end{aligned} \tag{2.3}$$

If we can find p_0 we can find the possibility that the server is in any of the other states. At any time the server has to be in one of the states, so if we sum of the fraction of each state we get that [8]:

[1]What happened before has no significance, only the present state

$$\sum_{n=0}^{\infty} p_n = 1 \tag{2.4}$$

Since this is a geometric series we can write:

$$\sum_{n=0}^{\infty} p_n = \frac{p_0}{1-\rho} = 1 \tag{2.5}$$

This gives:

$$p_n = (1-\rho)\rho^n \tag{2.6}$$

If we assume that there is an infinite number of tasks, we can calculate the expectation value of the number of tasks in the queue:

$$\begin{aligned} E(n) &= \langle n \rangle = \sum_{n=0}^{\infty} n p_n \\ \langle n \rangle &= \sum_{n=0}^{\infty} n(1-\rho)\rho^n \\ &= \sum_{n=0}^{\infty} n\rho^n - \sum_{n=0}^{\infty} n\rho^{n+1} \end{aligned}$$

If we relabel $n \to n+1$, the second term gives us:

$$\begin{aligned} \langle n \rangle &= \sum_{n=0}^{\infty} n\rho^n - \sum_{n=1}^{\infty} (n-1)\rho^n \\ &= \sum_{n=1}^{\infty} \rho^n \\ \langle n \rangle &= \frac{\rho}{1-\rho} \end{aligned} \tag{2.7}$$

We are now able to calculate the average number of jobs in the queue, and the only information we need is the λ incoming rate and μ processing rate of the server. We will use littles law to find a formula for calculating the average response time by using $\langle n \rangle$. In [8] Burgess defines load average/utilization as the probability of finding at least one job in the queue, which equals $1 - p_0$:

$$U = 1 - p_0 = 1 - (1 - \rho) = \rho = \frac{\lambda}{\mu} \tag{2.8}$$

Little's Law about queue sizes says that the mean number of jobs $\angle n\rangle$ is equal to the product of the mean arrival rate per second λ and the mean delay time per second R. We can then calculate the expected response time:

$$\begin{aligned} R &= \frac{\langle n\rangle}{\lambda} \\ R &= \frac{1}{\mu(1-\rho)} \\ R &= \frac{1}{\mu - \lambda} \end{aligned} \tag{2.9}$$

A result of this formula is that the response time goes to infinity when $\lambda \to \mu$. This is in accordance with the formula for calculating the mean number of tasks, where the number of tasks goes to infinity when $\rho \to 1$, and $\rho = \frac{\lambda}{\mu}$.

2.8.2 M/M/k Queues

In the previous section we deduced the formula for calculating response times for $M/M/1^k$ queues. In this section we will look at a slightly different queue formula, the $M/M/k$ queue. According to the folk theorem the $M/M/k$ queue generates lower response times as it can be compared to having low redundancy (see figure 2.13), compared to the $M/M/1^k$ queue which is compared to high redundancy (see figure 2.14).

It is hard to classify our experiment in any of the two architetures shown in pictures 2.13 and 2.14. The closest architecture is probably the low redundancy, since all traffic goes through the dispatcher, and the service we offer will be available even if the dispatcher fails. The only problem is if the dispatcher fails itself, although this can be solved using backup solutions. In our experiment we will see which of the two models that best describes our system. The equations in this section will derive the formula for finding the response time using the $M/M/k$ queue model.

$$\lambda p_{n-1} = \begin{cases} n\mu p_n & (0 < n \leq k) \\ k\mu p_n & (n > k) \end{cases} \tag{2.10}$$

If we solve this, it gives us that:

$$\lambda p_n = \begin{cases} p_0 \left(\frac{\lambda}{\mu}\right)^n \frac{1}{n!} & (0 < n \leq k) \\ p_k \left(\frac{\lambda}{\mu}\right)^{n-k} \frac{1}{k^{n-k}} = p_0 \left(\frac{\lambda}{\mu}\right)^n \frac{1}{k!k^{n-k}} & (n > k) \end{cases} \tag{2.11}$$

By normalizing we can then find p_0

$$\sum_{n=0}^{\infty} p_n = p_0 \left(\sum_{n=0}^{k-1} \left(\frac{\lambda}{\mu} \right)^n \frac{1}{n!} + \sum_{n=k}^{\infty} \left(\frac{\lambda}{\mu} \right)^n \frac{1}{k! k^{n-k}} \right) = 1 \tag{2.12}$$

If we let the traffic intensity per server be $\rho = \frac{\lambda}{\mu k}$

$$p_0 = \left(1 + \sum_{n=1}^{k-1} (k\rho)^n \frac{1}{n!} + \frac{(k\rho)^k}{k!(1-\rho)} \right)^{-1} \tag{2.13}$$

κ is the probability that a task will have to wait to be performed, and is given by the probability that there are k or more tasks already in the system

$$\kappa \equiv P(n \geq k) = \sum_{n=k}^{\infty} p_n = \frac{(k\rho)^k}{k!(1-\rho)} p_0 \tag{2.14}$$

The expectation value of n represents the average number of jobs in the system

$$\langle n \rangle = \sum_{n=0}^{\infty} n p_n = k\rho + \frac{\kappa\rho}{1-\rho} \tag{2.15}$$

Little's law gives that the average response time for the system to respond is

$$R = \frac{\langle n \rangle}{\lambda} = \frac{1}{\mu} \left(1 + \frac{\kappa}{k(1-\rho)} \right) \tag{2.16}$$

We can now verify that $\kappa(k = 1) = \rho$ and that when $k = 1$ the results are the same for both $M/M/k$ and $M/M/1^k$ models. We see that the formula for response time goes towards infinity when $\lambda \to \mu$ just as the response time formula for $M/M/1^k$ queues.

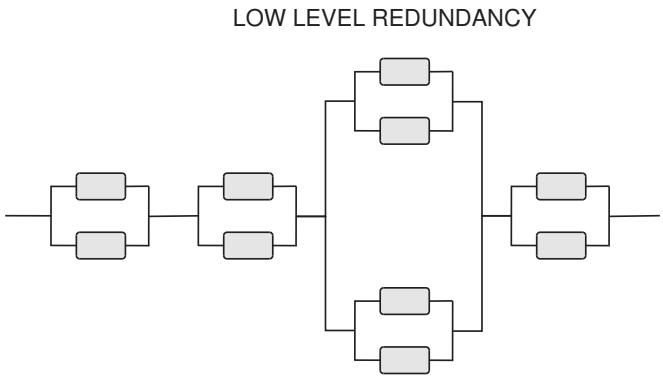

Figure 2.13: **Low Redundancy Architecture:** *If we imagine the blocks in the figure being hardware components, we might say that each component is in parallell, and if one fails the other takes over. This means that no matter which path a "task" takes through the system, it is enough if one of the components is functional*

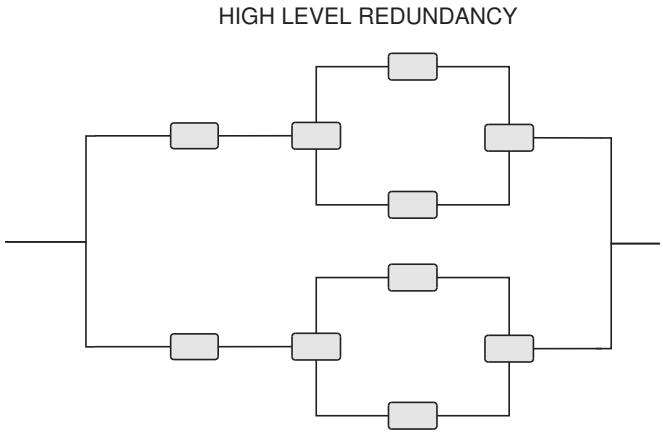

Figure 2.14: **High Redundancy Architecture:** *If we imagine the blocks in the figure being hardware components, we see that the entire system as a whole is ducplicated. This means that if a "current" chooses the upper path and one component in that path is broken, the current fails to pass through the system.*

Chapter 3

Hypotheses

This chapter is dedicated to state our hypotheses. First the hypotheses are stated. After each hypothesis there is a short discussion where we describe how we will test the hypothesis and what results we are expecting.

3.1 Expected Results

In this chapter, we present our hypotheses, and during our experiment our goal is to falsify or justify them. Karl Popper, considered one of the most important philosophers of science since Francis Bacon, suggested once that we cannot be certain of what we see, but maybe we can can prove whether or not we are wrong [8]. In other words this means that since we can always do one more test, and we can never know whether this test will deviate from the others or not, we cannot say that our hypotheses are right. But if our hypotheses fail one time, we can at least say that it is wrong.

Hypothesis 1: *Increasing number of servers available to a load balancer should decrease the response time.*

It is a valid assumption that an increase of servers will decrease the response time as the total processing power increases. We would also like to see if the response time decreases linearly proportional with the increasing number of servers added. We also suspect that with some types of traffic (e.g. multimedia traffic) the constraint is not within the servers but in network or load balancer capacity. Tests with 1-5 servers will be performed to see how the response time is affected.

Hypothesis 2: *The decrease in response time should follow the same trend as a theo-*

retical model based on a $M/M/1^k$ queue, where k is the number of computers available to the load balancer. We also believe that the $M/M/k$ queue will generate too optimistic results compared to our experiments

This assumption is very optimistic since these queueing models assume perfect load balancing between available processing units. We would like to see whether our simulations and experimental results scale proportionally. We will generate response rates from queueing algorithms to compare with actual results.

Hypothesis 3: *Response time varies depending on load balancing algorithm used and number of servers available.*

We would like to see how different load balancing algorithms perform compared to each other. Response time might not vary in all situations, for example when the load is low or the network is heavily utilized. Tests will be performed using different algorithms and traffic intensities.

Hypothesis 4: *Inhomogeneity among the servers varies the relative efficiency of the algorithms*

How does algorithms perform when balancing load between equal servers compared to different ones. We will test this scenario by reducing CPU power on some of our servers, and then do tests to compare the relative efficiency of algorithms compared to results obtained when servers had equal capabilities.

Hypothesis 5: *There are correlations between algorithms and traffic intensity which can be used to create an adaptive solution for selecting appropriate algorithms depending on current load.*

If we are able to find relative correlations between algorithms and traffic intensity we can propose limits for when to change algorithms to increase performance. Excisting algorithms that use current load are not necessarily the most efficient at all intensities.

Chapter 4

Experimental Design

4.1 System constraints

According to Burgess [8] a constraint defines the limitations of the variables and parameters in the system. A constraint usually takes the form of a rule or a parameter inequality. When doing measurements on the performance of load balancing it is important that we are aware of the constraints. Some of the constraints that apply to this experiment are:

Processing power: Each of the blades in use have limited processing power. When this limit is exceeded it might be difficult to acquire accurate measurements from the blades.

Network Bandwith: All blades are connected to the network with GigaBit (GB) interfaces, but since all blades are connected through one CAT6 Twisted Pair (TP) cable, they share 1GB of bandwidth.

Load Balancer: The load balancer must process packets, and the amount of processing depends on topology and algorithms used. The load balancer could become a limitation in extreme cases.

Apache Webserver: The apache webserver is configured to spawn a limited number of child processes to handle incoming requests (see section 4.3.1). When this limit is reached, it starts to put requests in a queue, and when the queue is full the server do not accept more incoming requests.

Client limitations: Our traffic generator act as a client generating traffic towards the webservers. The client can only generate a limited amount of connections based on a filedescriptor value (each connection needs a filedescriptor).

4.2 Experimental Configuration

In this section some brief descriptions of hardware and experimental setup will be given and discussed. One of the main objectives when configuring the lab was to get the best possible bandwidth between all units to avoid network congestion due to traffic intensity.

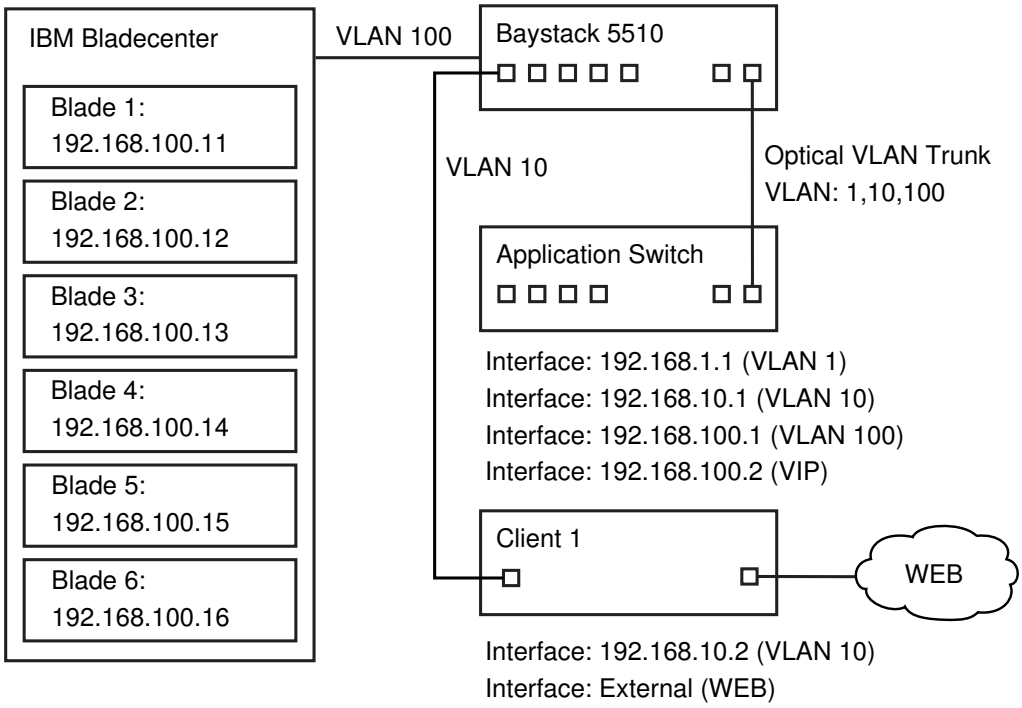

Figure 4.1: **Experimental Setup:** *The picture shows how the setup in the lab is, and the ip addresses used. All lines except to WEB has GB bandwidth.*

An overview of the lab can be seen in figure 4.1. We have divided the setup into three VLANs:

VLAN 1 - Administration VLAN (192.168.1/24): This VLAN is for administration purposes only. The network equipment used is configured with management interfaces on this VLAN for easy access through the network. The baystack switch is configured with management interface 192.168.1.2 and the application switch has the address 192.168.1.1. Both these units can be accessed through ssh or telnet depending on configuration. The default gateway for this vlan is 192.168.1.1, which is the interface configured on the application switch.

VLAN 10 - Client VLAN (192.168.10/24): This VLAN is the client VLAN. This is the VLAN which gives us access to the network from the outside. The

machines accessible through internet have two interfaces, one of them is located on VLAN 10 with address 192.168.10.2 (see figure 4.1) and uses 192.168.10.1 as gateway. 192.168.10.1 is configured on the application switch.

VLAN 100 - Server VLAN (192.168.100/24): This VLAN is created for the servers, which have ip addresses in the range 192.168.100.11-16. They all use 192.168.100.1 as default gateway, this address is configured on the application switch.

4.2.1 Hardware Specifications

All 6 blades used in the experiment are identical. The specification is listed in Table 4.1, they are of the type *IBM BladeCenter HS20*. The blades are located in a BladeCenter Chassis, specifications are listed in Table 4.2.

HS20 Blade 1-6	
CPU	Intel Xeon Processor - 2.8GHz
Memory	1GB PC2-3200 DDR2
Network	Dual Gigabit Ethernet
Internal Hard Drive	32GB Ultra320 SCSI
Operating System	Debian, Kernel 2.6.12-

Table 4.1: **Description of HS20 Blades:** *The blades are internally connected to a GB switch which again is connected to the baystack switch (see figure 4.1). All blades share the same disk, usb, cdrom, mouse, keyboard and screen through an internal KVM switch.*

BladeCenter Chassis	
Blade Bays	14 2-processor blades or 7 4-processor blades
Media	DVD-ROM and USB drive available on all blades
Networking	Nortel Networks Layer 2/3 Copper Gigabit Ethernet
Management Software	IBM Director

Table 4.2: **Description of BladeCenter Chassis:** *The BladeCenter chassis has its own network interface for management. It is also possible to control and monitor through SNMP. The SNMP protocols supports automatic shutdown with signalling from UPS if desired.*

4.3 Tools

In this section the tools used during the experiment will be described. We also included the configuration of servers and programs, although we tried to keep

as much as possible in the default state.

4.3.1 Apache 2.0 - Webserver

All our webservers are running a default installation of Apache 2.0. We need to make some adjustements to increase their performance. Apache have a set of modules implementing how multi-processing should be performed. Only one of the modules are active when running apache, and it is usually either the *Apache MPM worker* module or the *Apache MPM prefork* module. There are other modules available, but these are either under development or deprecated. The *prefork* and *worker* modules have a set of common directives to control them, and some of them need to be adjusted. Below is the output showing that our webservers are using the *Apache MPM prefork* module.

```
blade1:/etc/apache2# apache2 -V
Server version: Apache/2.0.54
Server built:   Sep  5 2005 11:15:09
Server's Module Magic Number: 20020903:9
Architecture:   32-bit
Server compiled with....
 -D APACHE_MPM_DIR="server/mpm/prefork"
 -D APR_HAS_SENDFILE
 -D APR_HAS_MMAP
 -D APR_HAVE_IPV6 (IPv4-mapped addresses enabled)
 -D APR_USE_SYSVSEM_SERIALIZE
 -D APR_USE_PTHREAD_SERIALIZE
 -D SINGLE_LISTEN_UNSERIALIZED_ACCEPT
 -D APR_HAS_OTHER_CHILD
 -D AP_HAVE_RELIABLE_PIPED_LOGS
 -D HTTPD_ROOT=""
 -D SUEXEC_BIN="/usr/lib/apache2/suexec2"
 -D DEFAULT_PIDLOG="/var/run/httpd.pid"
 -D DEFAULT_SCOREBOARD="logs/apache_runtime_status"
 -D DEFAULT_LOCKFILE="/var/run/accept.lock"
 -D DEFAULT_ERRORLOG="logs/error_log"
 -D AP_TYPES_CONFIG_FILE="/etc/apache2/mime.types"
 -D SERVER_CONFIG_FILE="/etc/apache2/apache2.conf"
```

The *prefork* module implements a non-threaded, preforking web server. In apache2.conf some of the directives controlling the *prefork* module have been set to default values:

```
<IfModule prefork.c>
StartServers         5
MinSpareServers      5
MaxSpareServers     10
MaxClients          20
MaxRequestsPerChild  0
</IfModule>
```

`StartServers`, `MinSpareServers`, `MaxSpareServers` and `MaxClients` control how apache spawn processes to serve new requests. We want to be able to serve as many requests per second as possible, but not so many that our

server starts swapping, as this is too time consuming. `MaxRequestsPerChild` defines how many requests each child process will process before it is killed, setting this value to zero means that the process will never die. We would like to have the possibility of 300 simultaneous requests, to do this we adjust the `MaxClients` directive to 300. Any connection attempts above this value will be queued up, and the directive `ListenBacklog` defines the maximum queue length. By default Apache is configured with a directive called `ServerLimit`, that defines the maximum number of processes apache can start. Therefore when we set `MaxClient` to 300 we also need to adjust `ServerLimit` accordingly.

4.3.2 Atsar - System Activity Report

Atsar is a program that deliver statistics about a system. It gathers information from files under the /proc directory. We will use atsar on the servers when doing performance tests to gather information about the CPU load. By doing this we can see how the load on each server increases when the requests per second is increasing.

4.3.3 httperf - HTTP performance measurement tool

Httperf[1] is a tool for measuring web server performance. Httperf suppoprts both HTTP 1.1 and 1.0. One of the main advantages of using httperf for traffic generation is that it can generate web traffic with Poisson inter-arrival times. An example of using httperf:

```
httperf --hog \
        --server=192.168.10 \
        --period=e0.1 \
        --num-conn=400 \
        --num-calls=1 \
        --uri=/index.html2 \
        --timeout=10
```

The `--hog` option specifies that httperf should open as many TCP connections as needed, otherwise httperf is limited to ephemeral port[2]. The `--server` option specifies the domain name or IP address of the server we are about to test. `--period=e0.1` specifies that we want to generate requests with a Poisson distribution, with an average inter-arrival time `\lambda` equals 0.1 seconds. The `--num-conn` option specifies how many connections we wish to generate on total, while `--num-calls` specify how many sessions we wish to generate for each connection. With the `--uri` option we select which file to

[1] Homepage: http://www.hpl.hp.com/research/linux/httperf/
[2] Ports in the range 1024 until 5000

get in our HTTP GET request. The last option `--timeout` sets a limit for how long the client will wait for an answer. If the server is loaded it might be wise to increase the timeout value to give the server enough time to answer.

```
Total: connections 400 requests 400 replies 400 test-duration 44.354 s

Connection rate: 9.0 conn/s (110.9 ms/conn, <=30 concurrent connections)
Connection time [ms]: min 89.5 avg 1523.1 max 12091.4 median 1070.5 stddev 1464.3
Connection time [ms]: connect 113.2
Connection length [replies/conn]: 1.000

Request rate: 9.0 req/s (110.9 ms/req)
Request size [B]: 75.0

Reply rate [replies/s]: min 6.8 avg 10.0 max 12.2 stddev 2.0 (8 samples)
Reply time [ms]: response 131.2 transfer 1278.7
Reply size [B]: header 253.0 content 10000000.0 footer 0.0 (total 10000253.0)
Reply status: 1xx=0 2xx=400 3xx=0 4xx=0 5xx=0

CPU time [s]: user 5.80 system 34.95 (user 13.1% system 78.8% total 91.9%)
Net I/O: 88073.1 KB/s (721.5*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

4.3.4 Autobench - Perl wrapper for httperf

Autobench is a perl wrapper to automize a series of tests using httperf. Autobench also parse the output, and summarises the results in a CSV[3] or an TSV[4] file. The problem with autobench is that it does not enable all the functionalities from httperf to be used. The possibility of using Poisson distributed requests is not enabled in autobench. We solved this by manually editing the autobench perl script to accept the Poisson distribution option. Another useful feature in autobench is that you can run an autobench daemon on other machines, and launch distributed tests against a web server. This will be useful if we find out that one client machine can not generate enough traffic for the blades. An example of running autobench against the blades:

```
autobench_mod --single_host \
--host1 192.168.100.1 \
--quiet \
--low_rate 10 \
--high_Rate 150 \
--poisson \
--num_call 1 \
--rate_step 10 \
--uri1 /index.html
--const_test_time 60
--timeout 10
--file result.tsv
```

[3]Comma Separated Variables
[4]Tab Separated Variables

With the `--single_host` option we specify that we only wish to test one host, and `--host` defines the domain or IP address of the host we wish to test. We do not wish to generate any output to STDOUT[5] since we write our results to file with the option `--file`. The `--high_rate` and `--low_rate` specify which rate autobench should start running httperf on, and on which rate it should end. In the example above, autobench start the tests with a rate of 10 packets per second. Since we have specified that we wish to use poisson distributed inter-arrival times with `--poisson`, the mean inter-arrival time will be set to `1/low_rate`. One of the more important features of autobench is the option `--const_test_time`[6], which defines the time each iteration should run. In httperf we can only specify how many connections each test should run, which causes a skewed time distribution when increasing the load[7].The option `--rate_step` specifies how much we should increase the rate for each iteration. In this case we increase with 10 requests/second, so that the next iteration will be run with 20 requests/second. Autobench runs until it reaches `--high_rate`, which in this case is 150 requests/second, hence autobench will run httperf 15 times with increasing rates. `--num-call` specify how many sessions to establish per TCP connection, `--file` specify the output file and `--timeout` defines how long the client should wait for an answer. Below is a sample output from one of our tests:

```
1       1.0     1.0     1.0     1.0     1.0     0.0     18.9    0.3     0
6       6.0     6.0     6.0     6.0     6.0     0.0     18.7    2.0     0
11      11.0    11.0    11.0    11.0    11.0    0.0     18.8    3.7     0
16      16.0    16.0    16.0    16.0    16.0    0.0     18.8    5.4     0
21      21.0    21.0    21.0    21.0    21.0    0.0     18.8    7.0     0
26      26.0    26.0    26.0    26.0    26.0    0.0     18.8    8.7     0
31      31.0    31.0    31.0    31.0    31.0    0.0     18.9    10.4    0
36      36.0    36.0    36.0    36.0    36.0    0.0     18.7    12.0    0
41      41.0    41.0    41.0    41.0    41.0    0.0     18.7    13.7    0
46      46.0    46.0    46.0    46.0    46.0    0.0     18.7    15.4    0
51      51.0    51.0    50.8    51.0    51.0    0.1     18.7    17.0    0
56      53.5    53.5    52.6    53.4    53.8    0.4     737.2   17.9    0.1191895
61      50.8    50.8    44.2    52.1    53.6    3.5     1727.0  17.0    0.0546747
66      48.0    48.5    18.2    49.0    54.4    12.5    2030.5  16.0    0.9688934
```

We are mainly interested in column 1,5,8 and 10. Column 1 shows the request rate demanded at the command line. Column 5 shows the average response rate in responsen per second. Column 8 shows the average response time and column 10 shows the error rate.

[5]Standard Output, file descriptor 1

[6]It is recommended to run each iteration for at least 300 seconds, that will provide you with 30 samples as httperf take measurements every 10th second

[7]Tests with high rate will finish faster than the tests with lower rates

4.3.5 Gnuplot - Plotting program

Gnuplot is a command-driven plotting program. It supports both CSV and TSV files. In this experiment we have created gnuplot scripts to generate encapsualted postscript files (EPS), this file format is vector based and easy to integrate in latex.

4.3.6 Scripts

Scripts were developed to aid us in the work with this project. Some of them are created for testing purposes and others are for merging and formatting test result to the desired format.

stddev.pl

The stddev.pl script can take a number of result files generated by autobench and calculate mean and standard deviations for each column from the original files. A file called processed will be generated and contains the result. Only the columns containing: *number of connections*, *response time*, *netio* and *errors* are processed. The script must be located in the same directory as the result files, and the files must have this naming convention: *result1.tsv result2.tsv result3.tsv ... result**n***.

```
Usage: stddev.pl <number of result files>
user@blade1: ./stddev.pl 10
```

iterate.sh

This is a shell script that iterates autobench *n* times. To keep things simple, the only option that can be used with this script is the number of iterations to perform. THe autobench parameters must be changed in the script code. Between each iteration the script restarts apache2 webserver on all the blades and waits for 10 minutes. This is because we want all connections from last run to be timed out, so that the next test is not affected by the one before. The script generates files for each iteration of this format: *result1.tsv result2.tsv result3.tsv ... result**n***, which makes it easy to use *iterate.sh* to extract information and calculate mean and standard deviations.

```
Usage: iterate.sh <number of iterations>
user@blade1: ./iterate.sh 10
```

merge.sh

merge.sh is a shell script created to merge results obtained at the servers using atsar and the results created by the autobench script. The merge.sh script also calculates the standard deviations of the cpu load measured at the servers.

autobench_mod

Autobench is just a perl wrapper to automize the use of httperf. Although autobench lacks some of the functionality offered by httperf. We had to alter the autobench script to allow us to use Poisson distributed traffic. We also added start and stop times for each test, which was necessary to be able to compare load data gathered with `atsar`.

Gnuplot scrips

For each of the graphs in the result section, there is created a gnuplot script. These scripts are merely a set of the gnuplot commands to create a certain graph. By storing the commands we can easily alter things such as: output format, range of axis, labels, input files etc.

```
Usage: gnuplot [gnuplot script]
user@blade1: gnuplot ./blade1_resuilt.gnuplot

Example script:
## Setting the name that will show above the graph
set title "Least connections algorithm"

## Setting the key in the upper left corner
set key under

## Setting the labels for my axis
set xlabel "Requests/Second"
set ylabel "Response Time "

## Need to disable mirroring of y-axis when using
## different yaxis
set ytics nomirror

## I do not wish to have ticks on the x2axis
set xtics nomirror

# Set range for x and y axis
set xrange[0:450]
set yrange[0:]

## Setting output parameters
set terminal postscript eps enhanced color
set output "result-respons.eps"

## Plotting the graph
plot "processed" using 1:2 title 'Response Time Mean' w l, \
"processed-error" using 1:2:3 title 'Response Time Std. Dev.' with yerrorbars
```

Chapter 5

Methodology

5.1 Workflow

In Figure 5.1 there is a workflow chart showing the course of the experiment. The workflow is divided into 7 states: *theorize*, *model*, *test*, *process*, *analyze*, *visualize* and *conclude*:

Theorize: In this part of the thesis it is important to acquire enough knowledge in the field of load balancing to be able to conduct sane experiments. It is also of great importance to do a survey to find other similar experiments and results.

Model: This area includes the configuration of the equipment and the selection of hardware and software tools. In this part we should also have enough knowledge to make some hypothesis about what we wish to prove with our experiments.

Test: Here we perform the actual tests, using the equipment and software we found suitable in the previous phase.

Process: Some of the results need additional calculations, like mean and standard deviation values. We also need to merge and adapt our results so that they can be used for visualization and easy interpretation.

Analyze: Here are the results analyzed. This includes trying to find the cause and effect for the values obtained from the test. It is important that we know how different factors affect our results. By keeping all variables in the system constant except one, we can isolate causes for change.

Visualize: Visualization is often a great aid to help interpret results, therefore we will in this phase use graphs to create time series of our result. This will help us to find trends and analyze the results further.

Conclude: This is the final phase of the project, where we look at our analysis and see whether the results conform to our hypothesis or not.

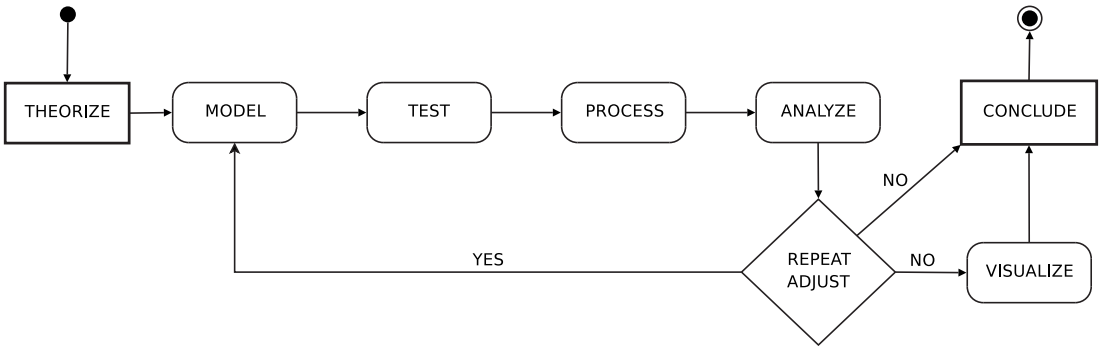

Figure 5.1: **Workflow chart:** *Showing the different states during the thesis, and the transitions between them*

In figure 5.1 there is a decision point, where an arrow goes back to the model phase. Empiricism requires three iterative main phases: *observe, model* and *explain*. From Figure 5.1 the arrow going from the decision point back to model represents the iterations needed to do empirical research. For each iteration we will have gained knowledge about our software and equipment, and can therefore create better models and do more precise testing to receive more accurate results.

5.2 Assessment of Error and Uncertainty

When doing scientific experiments we can not expect to find absolute results, therefore we need to calculate with probabilities. Through the use of statistics, it is possible to say what the chances are that our results yield the truth and can be reproduced. In this experiment we use the quantities mean and standard deviation, also called the first and second moments of the collected data.

Assuming that there are no systematic errors, meaning that all errors have independent random causes, we can define the value <n> to be the mean arithmetic value:

$$\langle n \rangle = \frac{v_1 + v_2 ... v_N}{N} = \frac{1}{N} \sum_{i=1}^{N} v_i \tag{5.1}$$

The mean value is only a rough description of the dataset, and two datasets can have the same mean value, but differ in distribution of values. The mean value can not tell whether the distribution is skewed or how wide the distribution is. So to be sure that the mean value yields a sane representation of our data

we need to calculate the second moment of our data as well, the standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(x_i - \langle x \rangle)^2} \tag{5.2}$$

The standard deviation tells us about the spread in our dataset, and whether the mean value is a valid representation for our set. The standard deviation gives us a measure of the scatter in the data due to random influences. If we assume that the errors in the experiment are caused by independent random causes, we can assume that they also are normally distributed. For normally distributed datasets, with $\overline{x}$ and standard deviation σ, 68.26% of the values will be within $[\overline{x} - \sigma, \overline{x} + \sigma]$ and 95.46% will be within $[\overline{x} - \sigma, [\overline{x} + \sigma]$ (See Figure 5.2)

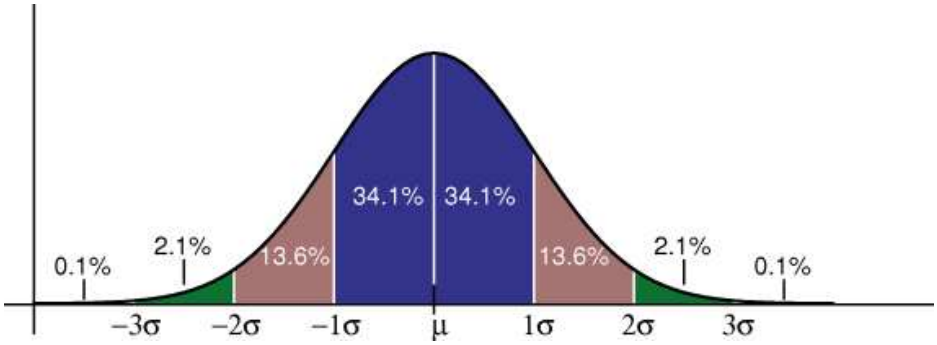

Figure 5.2: **Normal Distribution:** *Curve illustrating the percentage distribution of values for a normally distributed set of values. Where μ represents the mean and σ is the standard deviation.*

5.3 Test Plan

A series of tests will be executed to find correlations between the load balancing algorithm used and the perceived quality of service (QoS) for the users. In this experiment we will define QoS as the response time of the packets generated toward the servers. The tests will be performed in specific order in accordance with the proposed workflow chart, which is important to uncover the properties of the system.

5.3.1 Blade properties

All 6 blades used in our experiments have the same hardware and software, which makes it possible for us to find the limitations of the blades by benchmarking one of them. The first thing we are interrested in, is how much traffic

one server can cope with, and how much memory and cpu that is utilized while serving content, both static and dynamic, at different rates.

We start by creating a test that generates HTTP GET requests toward one of the blade servers at pre-defined rates. This rate will be Poisson distributed to assimilate regular web traffic. To eliminate some sources of error we decided to run the first test in three different ways:

1. **Blade Performance, test 1:** Here we use httperf and autobench to automize 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. **Blade Performance, 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. **Blade Performance, test 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.

5.3.2 Benchmarking Algorithms

To be able to compare the efficiency of different load balancing algorithms, we will perform a series of tests on each of them and compare the results. We will look at properties such as response time and load distribution. To be able to compare our results we will only use one topology configuration while testing the different algorithms.

Here we use blade1-5 as webservers, blade6 is used as a traffic generator. All generated traffic is sent towards the VIP address on the dispatcher, which in turn distributes the requests among the blades. We will use a small shell script to run autobench 20 times with rates from 2-600 requests per second. After the test, mean and standard deviations will be calculated from the result files and plotted in graphs. We will test the performance of three algorithms: Round Robin (RR), Least Connected (LC) and Respone Time (RT)

5.3.3 Experiment vs. Queueing models

In these experiments we will look at how our blades perform compared to $M/M/k$ and $M/M/1^k$ queueing models. We use a small c program to generate results from the models. Then we perform the same tests at our blades. We

will test how response varies with 1-5 blades using autobench as a traffic generator. All results will be plotted in graphs together with the results from our queueing program.

5.3.4 Scalability

In these experiments we are interested in seeing how performance scales when adding additional servers. We will run tests with rates: 100, 200, 400, 600 requests per second. All these tests will be repeated using 1-5 blades, and we will plot response times and see if there is a linear increase in performance when adding blades.

Chapter 6

Results and Analysis

6.1 Describing the Results

In this section the results we obtained will be discussed, and causes deducted.

6.1.1 Apache Webserver Performance

It is important to identify the saturation point of the servers used in the experiment [30]. Given a precise value for the throughput of our blades, the result can be used to predict response times using the formulas for $M/M/1^k$ and $M/M/k$ queues. A reasonable assumption is that all blades have the same performance, as they are equal in both software and hardware. We define the throughput of our server as:

$$Throughput\ \mu_i \quad = \quad \frac{No.of completions}{Time} \tag{6.1}$$

To eliminate some sources of error tests were executed from more than one location, using both *ab* and *httperf*. To be able to utilize the blades, a php script had to be made. For each request the server ran a PHP loop 10000 times: `GET index.php?iter=10000`.

```
<?php
        $start = microtime(TRUE);

        for($i = 0; $i < $_GET['iter'];$i++)
        {}

        $stop = microtime(TRUE);
        echo "Start: $start; Stop: $stop; Total time:".($stop-$start);
?>
```

The graph in Figure 6.1 shows that the response time stays low until we reach approximately 100 requests/second. Above 110 requests/second the response

time stays between 1200 and 1400 ms. From these results it is obvious that the server can handle somewhere between 100 and 110 requests per second. We also see that the CPU utilization increases linearly until around 105 requests/second, then it drops. This is a strange result since the server should be heavily utilized above this rate. Looking at the standard deviation of the results obtained for CPU load, we see that it is very high when the request rate is above 110 requests/second. This implies that the results obtained for CPU load when requests per second exceeds the maximum threshold, is useless due to process thrashing at the server side. Thrashing is a term describing situations where two or more processes try to access a shared resource repeatedly, and the overhead of accessing that resource becomes so high that no real work is done. This also tells us that we should stay clear of the thrashing region at all costs.

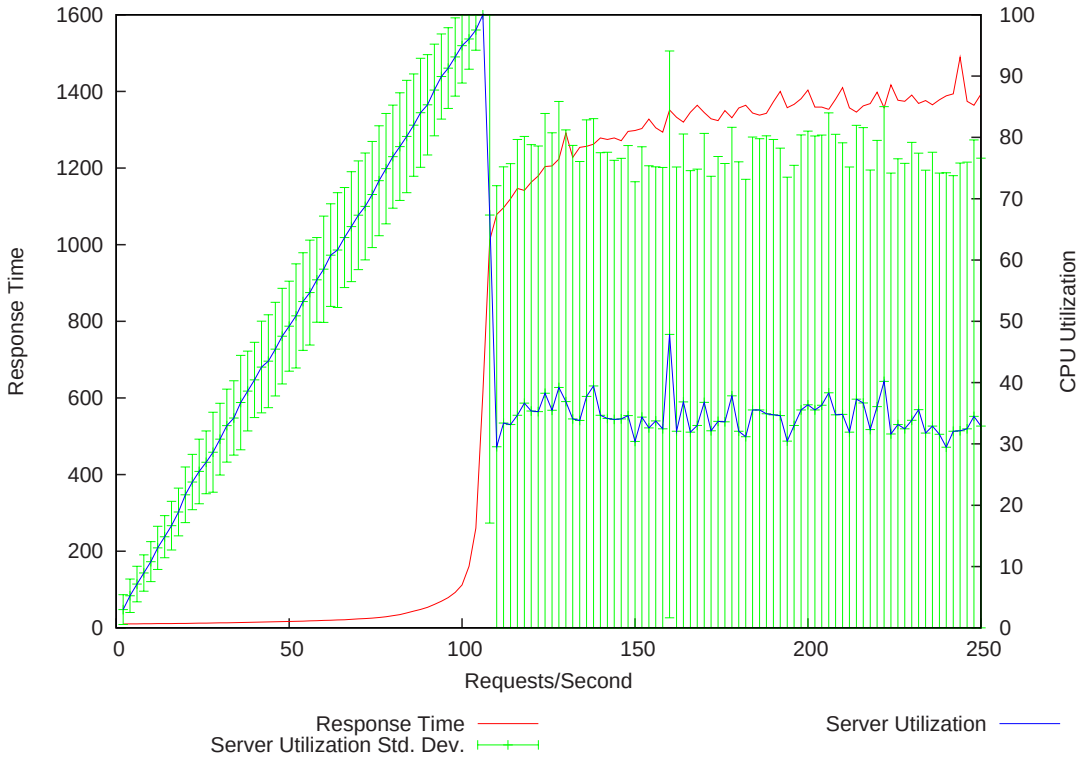

Figure 6.1: **Throughput of single webserver - Poisson:** *Response time and CPU utilization as a function of requests per second, using Poisson distribution with exponential inter-arrival times.*

From the graph in figure 6.1 the response time flattens out when the requests per second exceeds approximately 105 requests per second. Above this rate the server starts to refuse connections instead of accepting all of them. From table 6.1 the numbers show that the error rate above 100 requests per second

increases proportionally with the increasing request rate. This indicates that the server can not process requests at any higher rate.

No. of requests	CPU utilization	CPU Std. Dev.	No. of errors
2	2.98	2.43	0
10	10.80	3.26	0
30	30.78	5.85	0
50	49.22	7.34	0
70	67.30	8.88	0
90	85.32	8.18	0
100	94.96	6.09	0
120	35.36	43.55	5.91
140	34.67	42.85	19.45
160	47.87	46.24	37.63
180	32.07	43.93	55.01

Table 6.1: **Throughput of single webserver - Poisson:** *Table showing a summary of CPU utilization and corresponding request rate when using Poisson distributed inter-arrival times*

The reason for the results shown in table 6.1 is the queue length configured on the apache webserver. Apache is only spawning a limited number of child processes to handle incoming requests, and when all these processes are busy, it needs to queue up waiting requests. This queue has a maximum length, in our case defined by the Apache2 `ListenBacklog` directive, although this can often be OS dependent. The errors start to increase when the server has filled up its queue. 120 requests/second gives an error rate of 5.91 packets, and when the request rate increases further, we see that the increase in errors is almost the same as the increase in requests per second. We are mainly interested in the results before the server reaches full utilization.
In this experiment we will use exponential inter-arrival times with poisson distribution. This is the closest to real-life traffic we get without altering the source code of `httperf`. Research has shown [26, 17] that packet switched networks tend to be best characterized with long-tailed self similar distributions when looking at large amount of traffic. In figure 6.2 the results of using both poisson distribution and static inter-arrival times are visualized. When the load is low we see that both distributions give approximately the same results, but when the load increases, the poisson distribution causes greater stress to the server, and this results in increasing response times.

To enlarge the differences between stress caused on servers using exponential and deterministic inter-arrival times, figure 6.3 shows a graph with response times as a function of request rates between 0 and 100. This graph shows only a part of the data collected, but the selection makes it easier to see the gap in response time using poisson inter-arrival times compared to using static inter-arrival times. We can clearly see that using the posisson

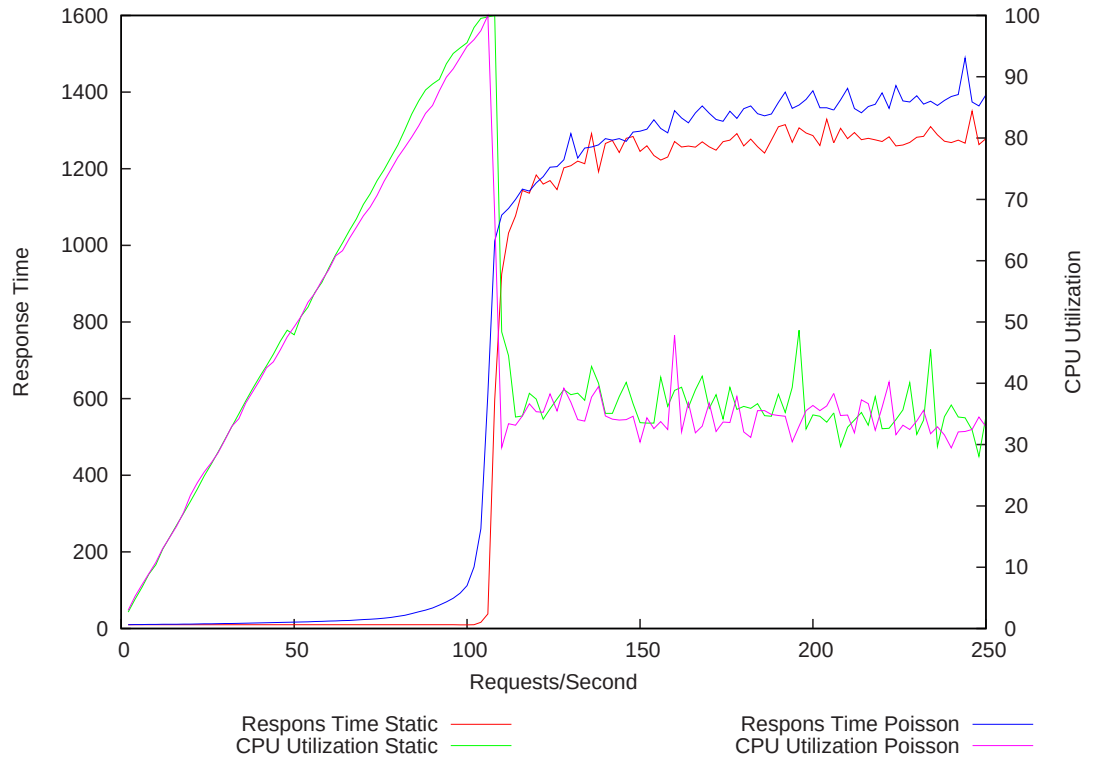

Figure 6.2: **Comparison - Poisson vs Static:** *Graph showing results obtained from using both Poisson and Deterministic inter-arrival times*

inter-arrival distribution causes significant higher amount of stress on the web server, which results in the response time to peak at lower request rates than when using the static inter-arrival distribution.

To eliminate some sources of error we ran the same performance tests from 1 and 2 clients. Often when doing such performance tests the client can be a limiting factor, and it is not always easy to discover. In figure 6.4 we have compared the results obtained from using both 1 and 2 clients. It is evidential that the client is not a limiting factor in this scenario since the results show approximately the same results. Next we also did an experiment where the traffic was sent directly from the client to the server. This was done to see if the load balancer caused any delay. The results showed that the dispatcher did not cause any significant increase in response times.

From the extensive tests we have run through httperf and autobench we also want to be sure that our test tool is providing accurate results. From our graphs and result files we now know that each webserver can approximately process 105 requests per second when requesting the index.php file with 10000 iterations for each request. To verify these results we also used ab (Apache HTTP benchmarking tool). `ab` calculates things such as the processing time and re-

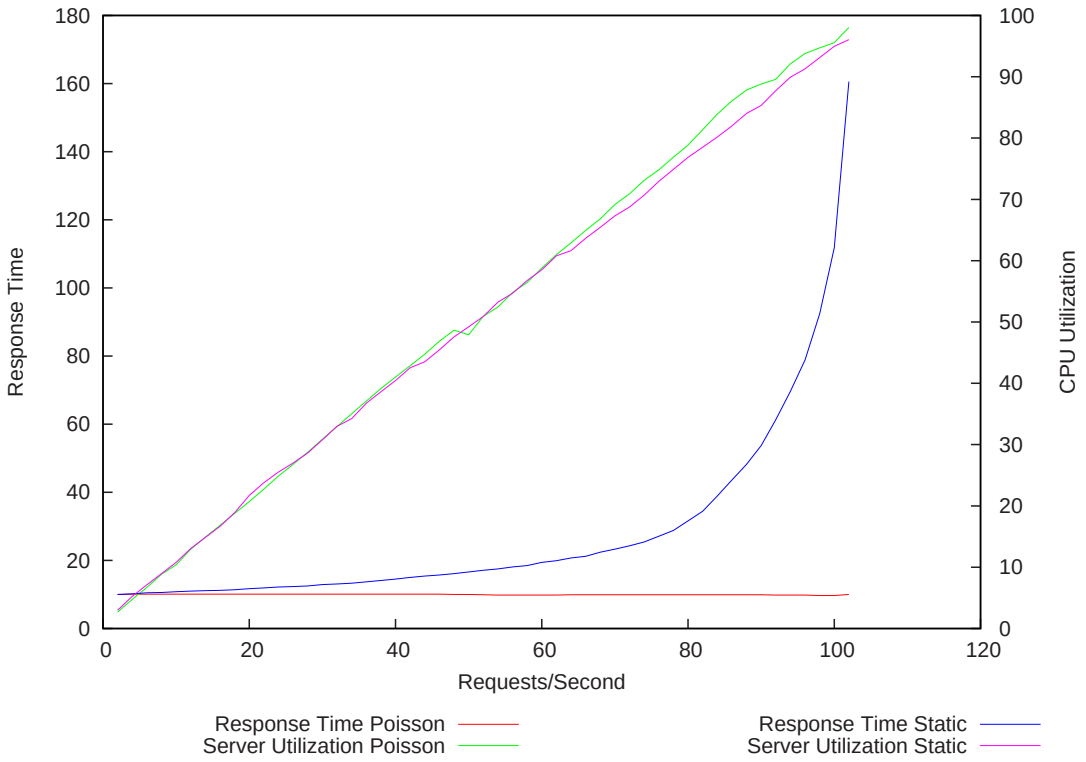

Figure 6.3: **Comparison2 - Poisson vs Static:** *Graph showing results when request rate is between 0 and 100 obtained from using both Poisson and Deterministic inter-arrival times.*

sponse time. We ran `ab` in a shell script 100 times to find standard deviation and mean value. The result from our run with `ab` showed that the average value of 100 testruns, each run doing 1000 lookups, gave us a mean value of 104.13 requests/second with a standard deviation of ± 2.16.

6.1.2 Queueing Models vs. Real Life

In this part we will look at how queueing models perform compared to real results obtained from our lab. We use a c script to calculate response times using both $M/M/1^n$ and $M/M/n$ queues. From the previous sections we have calculated the performance of a single webserver, and we use this result in our queueing simulation. Below is a list of parameters we need for calculating expected response times using queueing theory:

- μ This is the variable we found when doing performance tests against one server. It is the processing capability of the server, also called the service rate and is often expressed in completions per millisecond.

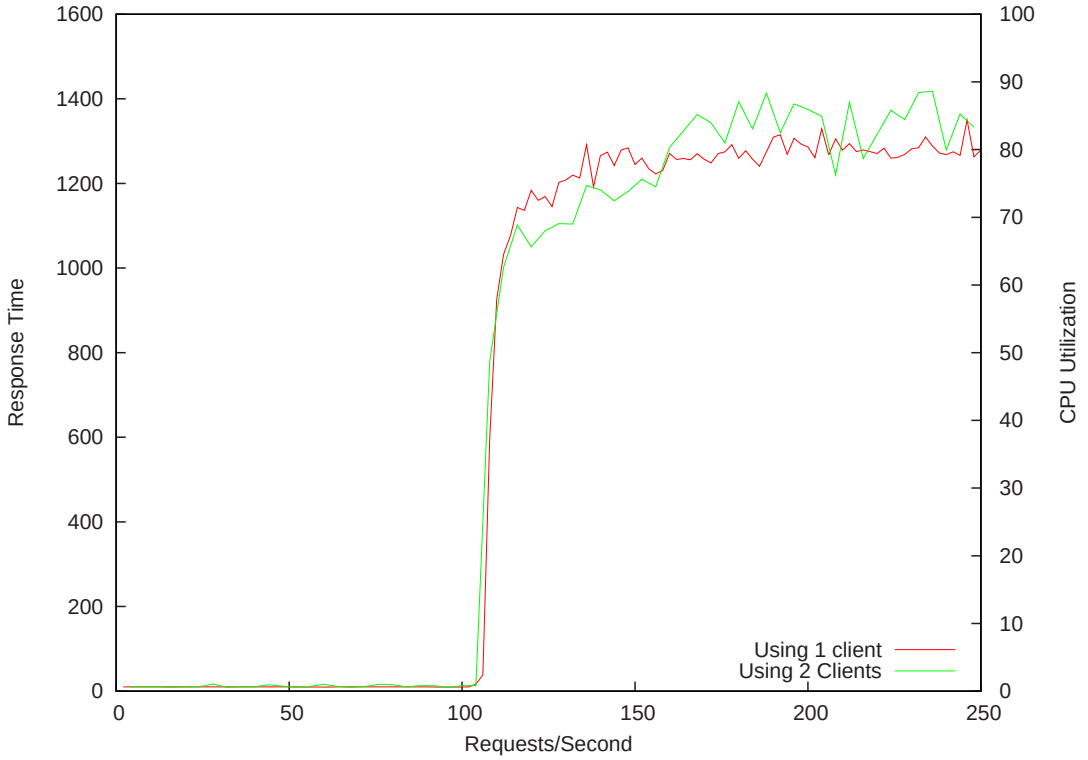

Figure 6.4: **2 Clients Generating Traffic:** *Graphs showing the response time when using both 1 and 2 clients to generate the traffic.*

- λ This is the actual rate, at which the traffic arrives. It is usually referred to as the arrival rate, and is expressed in arrivals per millisecond
- **n** Number of servers available. In our experiments we had at the most 5 available servers to balance the load between.

In figure 6.5 we see the comparison of the response times acquired through simulation, and the response times we calculated using our c program. The input parameters used were:

- $\mu = 0.10413$ This value is calculated from the fact that each server is able to process 104.13 requests per second, which means 0.10413 requests per millisecond
- $\lambda = 0.002$ This means that our c program will start simulations with 2 requests per second, and it will increase this value until it reaches the value specified by *max_rate*.
- *n* Number of computers to simulate, in figure 6.5 we load balance between 2 computers.
- *max_rate* Specifies the limit at which we will stop our simulation. This is usually equal to $n \cdot \mu$ since the queueing algorithms do not produce sane results when the servers are overloaded. In figure 6.5 max_rate equals $2 \cdot \mu = 0.20826$.

In figure 6.5 we used the RR algorithm to balance the load between two servers, requesting the cpu intensive php script described in 6.1.1. We see that the order of performance from best to worse is: our experimental results, $M/M/2$, and then the $M/M/1^2$ queue. It is quite surprising that our experimental results outperformes the response times from the queueing algorithms. The queueing algorithms are based on the assumption that we have perfect load balancing, meaning that requests are always forwarded to the unit that is least loaded. One of the reasons for our result is probably that the same object is always requested from our traffic generator, which means that the service time on the servers is static. The queueing algorithms assume that service time at the servers are poisson distributed. It is also important to remember the fact that internet traffic is considered to be heavy-tailed in nature, which will cause greater variance in inter-arrival times which leads to an increase in response time. In table 6.2 a summary of the results are shown. We see that when the request rate exceeds 200 requests per second, the standard deviation starts to increase more rapidly, and we also get some errors. Above this rate it is hard to make any conclusions because of the high standard deviation and the errors caused by the servers dropping requests. We see that in the range between 0-150 requests per second the $M/M/2$ queueing model can be used as a good

predictor for response times. The sudden peak in response times also suggest that it would be preferable to avoid rates above 150 requests per second per server, since it is very hard to predict sudden bursts of traffic that might cause servers to be overloaded. In the table 6.2 we see that the servers managed to handle 200 requests per second before any error occured. If we use the graph in 6.5 for 2 servers, we see that the point of maximum curvature is approximately at 180 requests per second. 180 requests per second is around 87% percent of the ideal performance of 208 requests per second.

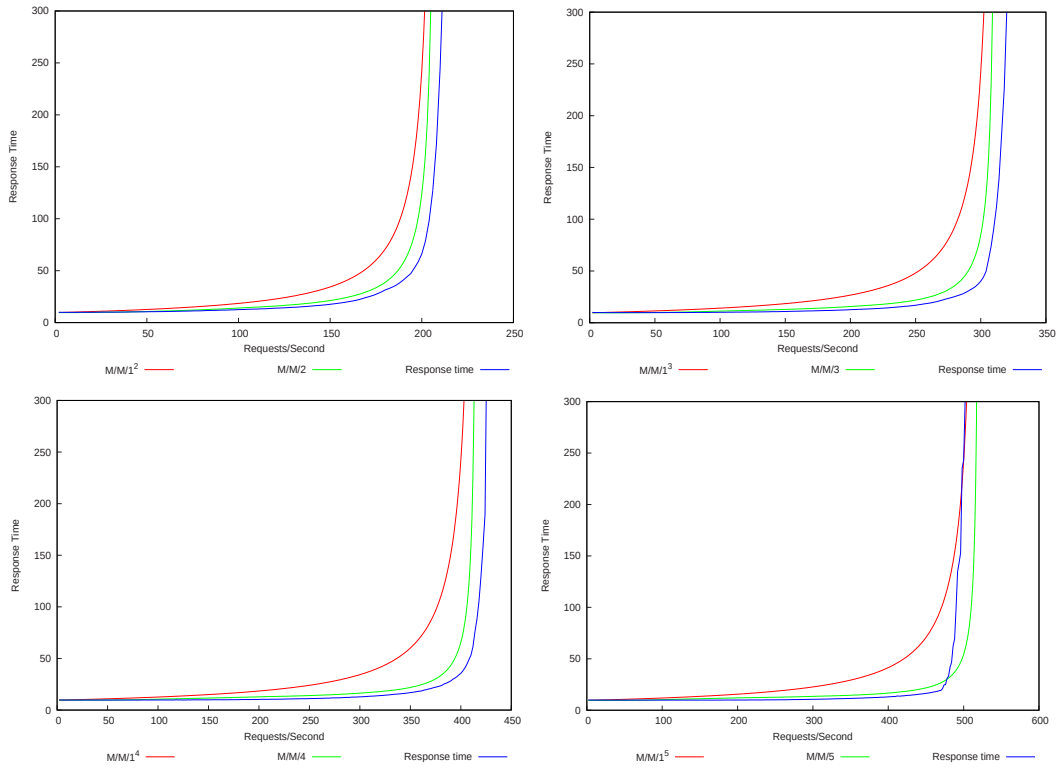

Figure 6.5: **Simulation vs. experiment:** *The graph shows how response rates scale when adding extra servers. In the top left corner we used 2 servers, in the top right corner 3 machines, in bottom left 4 machines and in bottom right 5 machines.*

Figure 6.5 also shows the simulation where we used 5 servers. The parameters are the same except for:

- n Number of computers to simulate, in figure 6.5 we see the results using 5 computers.
- max_rate Specifies the limit at which we will stop our simulation. This is usually equal to $n \cdot \mu$ since the queueing algorithms do not produce sane

No. of requests	Response Time	Std. Dev.	No. of errors
2	9.89	0.09	0
10	9.87	0.12	0
50	10.45	0.13	0
100	12.49	0.12	0
150	17.64	0.36	0
160	20.12	0.67	0
170	24.40	1.00	0
180	31.40	1.72	0
190	41.08	3.27	0
200	66.04	18.35	0
202	78.44	34.21	0.002
204	97.91	77.31	0.003
220	853.56	76.73	0.1

Table 6.2: **Error and CPU Utilization:** *Table showing a summary of CPU utilization and corresponding request rate when using Poisson distributed inter-arrival times, and load balancing between 2 servers*

results when the servers are overloaded. In figure 6.5 max_rate equals $5 \cdot \mu = 0.52065$.

From the graphs in figure 6.5 we see that our experimental response rates using 5 servers are lower than the $M/M/1^5$ and $M/M/5$ queue in performance until the request rate reaches approximately 480 requests/second. After this the response rate peaks following the same trend as the $M/M/1^5$ queue. In our formulas we have calculated the max rate for *n* number of servers by multiplying with the processing rate we found for one server in (see section 6.1.1). This means that 5 servers should be able to process approximately 520 requests/second. From the graph we see that our results peak around 480 requests/second, which means that the servers perform at around 92% of what is expected according to the queueing models. Before the rate reaches 480 requests per second we can use the $M/M/1^5$ queue as an estimator for response time, which also gives us some tolerance since our results show that the $M/M/1^5$ queue gives slightly higher response times than what our experiment gave us. If we compare this result to what we saw in our experiments with two computers, we see that the relative performance has dropped from 96% to 92%. Other experiments performed with 3 and 4 servers show relative performance similar to using 2 servers (see figure 6.6. The 2,3,4-server solutions outperform the M/M/2,3,4 predictions at all rates, even in the thrashing regime. The reason for this might be because the servers are handling the requests using parallel processing threads, not First Come First Server (FCFS) as the queue models assume [27]. The results show that when using 5 servers, the transition from low response times to process thrashing is steeper than for 2,3,4 servers. This could be a sign that the load balancer itself is the bottleneck. We were not able

to investigate this anomaly further since we had no more identical servers to deploy in our experiment.

6.1.3 SLB Scalability

We would like to investigate how perfomance scales when increasing the number of servers. If performance scales linearly when adding servers, it means that we can easily predict the effect of adding more servers. This is especially useful when considering QoS and maintaining SLA agreements. In figure 6.6 we have 4 graphs, where each graph represents a simulation with one traffic intensity, in this case: 100, 200, 400 and 600 requests per second. We have measured performance at all rates with 1-5 servers available.

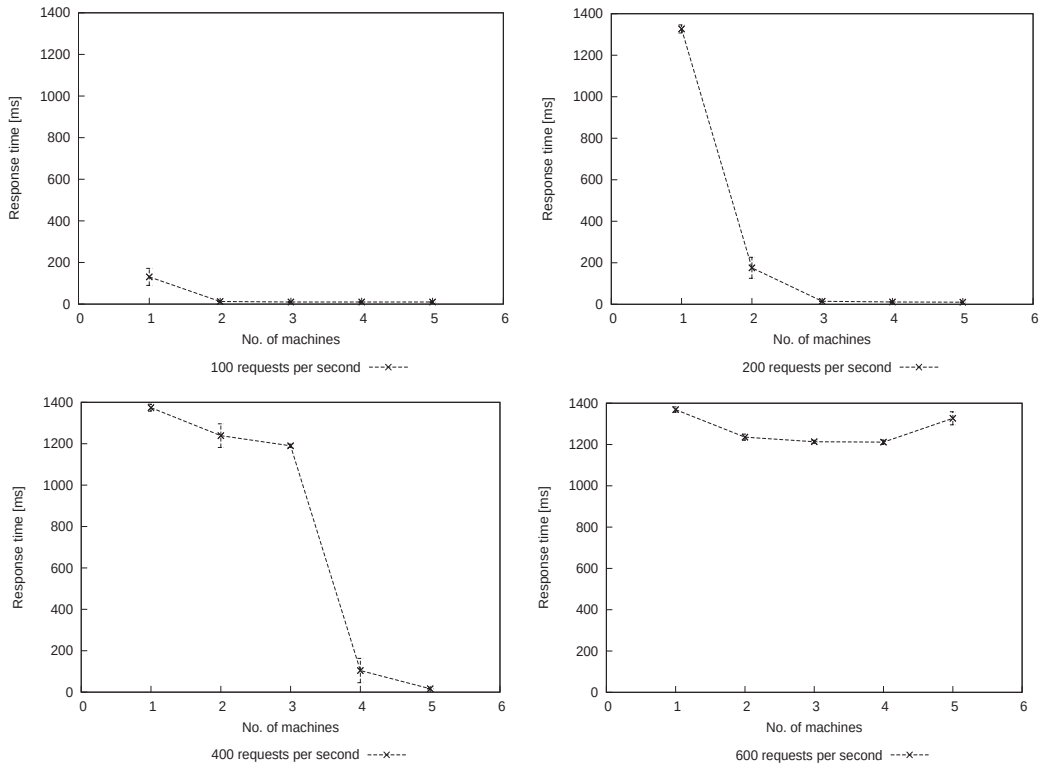

Figure 6.6: **Scalability - SLB:** *The graph shows how response rates scale when adding extra servers. Adding servers at low rates seems to result in a discontinuous improvement. Adding new servers the highest rates seems to reduce performance, as the limitations of the bottleneck dispatcher become apparent*

We know from investigation that each server should be able to process approximately 104 requests per second. If we use maximum point of curvature in the

graphs in figure 6.6 we find these values for their performance relative to the max rate:

Performance 2 servers: $\frac{200}{(104 \cdot 2)} \cdot 100\% \approx 96\%$

Performance 3 servers: $\frac{300}{(104 \cdot 3)} \cdot 100\% \approx 96\%$

Performance 4 servers: $\frac{400}{(104 \cdot 4)} \cdot 100\% \approx 96\%$

Performance 5 servers: $\frac{480}{(104 \cdot 5)} \cdot 100\% \approx 92\%$

In the first graph in figure 6.6 we see that the response time is kept low in all scenarios. This fits with the assumption that 1 server should be capable of handling 104 requests per second. The next graph show that all scenarios except the one with 1 server is capable of handling 200 requests. This is correct according to the observation that 2 servers should be able to handle approximately 200 requests per second. The other graphs also conform to our calculations of relative performance.

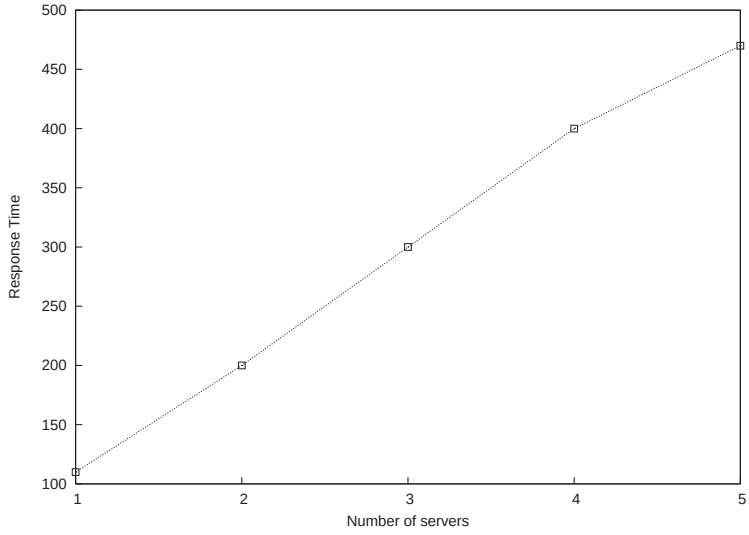

Figure 6.7: **Scalability2- SLB:** *The graph shows the maximum processing capability as a function of number of servers*

In figure 6.7 we see how the maximum processing capability changes when adding more servers. From figure 6.7 we see that the scaling is not linear, and that there are some overhead when going from having 1 server to start load balancing between 2 or more servers. We also see that the increase in performance is starting to drop, which indicates that at this point the load balancer becomes the bottleneck.

6.1.4 Comparison of Algorithms

In this section we will look at how the different algorithms perform when increasing the request rate. A load balancer is usually preconfigured to use one type of algorithm, it is therefore important that this algorithm performs good both when the load is low and when it is high.

Homogeneous server environment

In figure 6.8 we see the results from using least connected, round robin, and response time algorithms to balance load between 5 servers. The graphs show that the algorithms perform approximately the same under low stress (0-420 requsts per second). Althoug the LC algorithm have slightly higher response times.

Our traffic generator use poisson distributed inter-arrival times, but the files requested and processed by each server is the same each time, this means that the server's service time is deterministic. This indicates that as long as the incoming request rate for each server is below its threshold of processing capability, the roundrobin algorithm is a very effective algorithm due to low processing costs. Although when the request rate starts to exceed the threshold of requests per second, the RR algorithm does not take into consideration that some servers might be under more stress than others because of the poisson distributed inter-arrival times.

If we take a closer look at the response times in the range 0-450 requests per second, we see that the LC algorithm peaks around 260 requests per second, and then starts to approach the efficiency of the RR and RT algorithms. In the region 400-430 requests per second the LC algorithm surpasses the RR and RT algorithms (see figure 6.9).

Inhomogeneous server environment

In the previous section we investigated how the load balancing algorithms performed relative to each other. In this section we have manually adjusted the CPU speed on some of the blades to create an inhomogenous environment. Below is the CPU speed of each blade specified:

Blade 1: 2800000 MHz

Blade 2: 2450000 MHz

Blade 3: 2100000 MHz

Blade 4: 1750000 MHz

Blade 5: 1400000 MHz

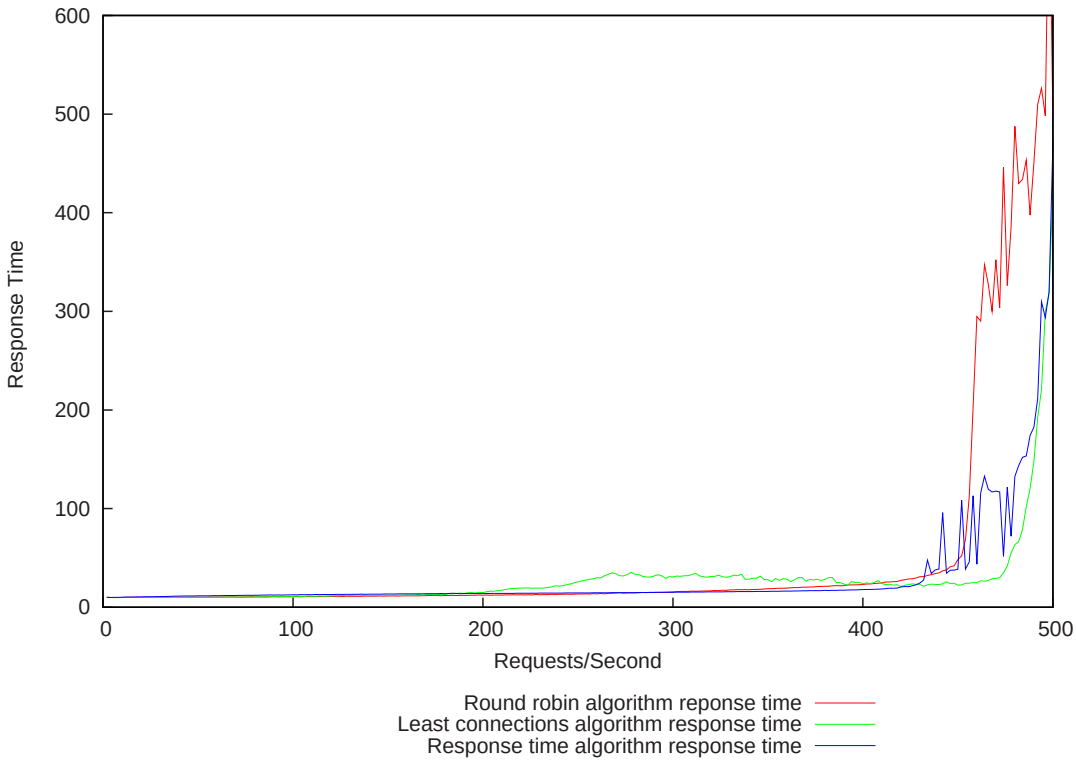

Figure 6.8: **Algorithm Performance - Homogenous:** *The graphs show the performance of the RT, LC and RR algorithms in a homogeneous server environment*

In the inhomogeneous environment the RT and RR strategies perform equally well (based on actual measurements of system performance). The LC algorithm sustain low response times the longest (see figure 6.10. We believe that when the servers enter the thrashing regime they can no longer provide us with useful information about load, as the RT algorithm requires. The RT algorithm becomes no better than RR (random chance) in practice, and we see that RR and RT succumb to noise about the same point. We suspect that LC works at higher rates because it relies on state information in the dispatcher, thus creating an equilibrium where the fastest servers are receiving most requests. Real traffic would not have static service times, and we believe that LC would not perform significantly better than RR or LC in such a scenario.

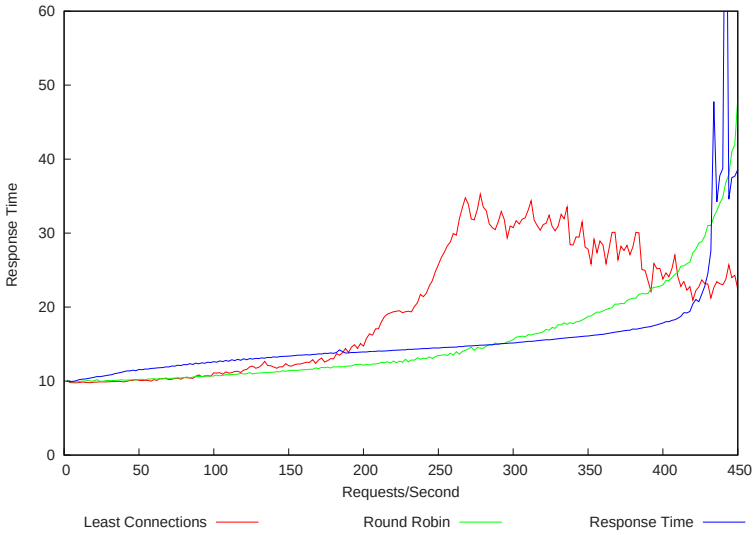

Figure 6.9: **Algorithms Performance 2 - Homogenous:** *The graphs show the performance of the RT; LC and RR algorithms in a homogeneous environment. The range is limited to 0-450 requests per second to enhance the differences between the algorithms*

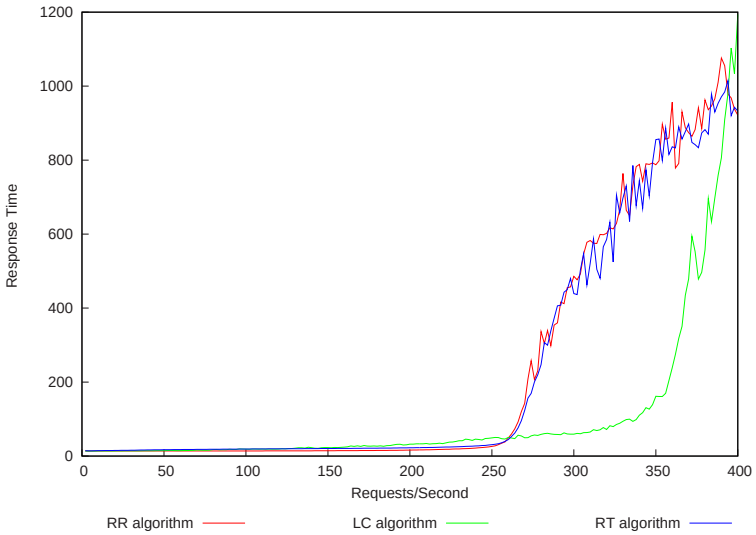

Figure 6.10: **Algorithms Performance - Inhomogeneous:** *The graphs show the performance of the RT; LC and RR algorithms in an inhomogeneous environment*

Chapter 7

Conclusions and Further Work

7.1 Conclusions

In this thesis we have investigated the performance of different load balancing algorithms under various traffic intensities and types. We have measured the performance in homogeneous and inhomogeneous server environments. Our results show measurable differences in algorithm performances for different testing scenarios.

In hypothesis 1 we stated that the response time should decrease when we add additional servers. What we found was that response time did not decrease linearly, instead it dropped very quickly as soon as we had enough processing power. This means that we got from saturation to comfort zone fairly easy by adding an additional server. In this regard we can see no reason for not having extra capacity available at the service centre. We also saw that going from 4 to 5 servers when the traffic intensity was high, only caused a degradation in performance, which we believe is caused by the dispatcher becoming a bottleneck. This non-linear performance scale contradicts results shown in [30].

We also stated that we believed the response rates would follow the $M/M/1^k$ queueing model, and that the $M/M/k$ model would be too optimistic. As it turned out the results from our experiments show that for low to medium utilization our experiment using the RR algorithm outperforms both queueing models. When using 1-4 servers we also saw that our experiment gave better response times under medium to high workload as well, but when using 5 servers the response time from our experiment converged with the $M/M/1^5$ queue under high loads. We also found that the service rate used in the queueing algorithms was very sensitive to minor adjustments. This implies that we need a very accurate measurement of the servers we wish to simulate.

Our next hypothesis suggested that there are differences in performance when using different load balancing algorithms. Other researches has found

the RR algorithm to be worst under all conditions [5, 30]. Teo [30] also suggests that the LC, RT and RR algorithms converge at high loads. These results contradict our own and the ones found in [11]. Our results show that the RR algorithm performs best under low to medium load. Under high loads we saw that the LC algorithm was able to sustain the response time longer than the RR and RT algorithm.

We also performed tests checking whether inhomogeneous server environments would alter the relative performance between the three load balancing algorithms. We found that under low to medium loads the RR algorithm gave the lowest response times. In the high load regime we saw that LC outperformed the RR and RT algorithms. The RR and RT algorithms succumb to noise at the same time, indicating that the polling used by the RT algorithm did not reflect the server status (we polled server status every 5th second). We tried altering the inter-polling time used by the RT algorithm without any effect, although we found that polling servers each second gave us a degradation in performance due to the increased amount of connections each server needed to establish. The fact that LC is superior in the high load regime is probably due to the operational equilibrium that is created by the dispatcher keeping track of open connections, and therefore sending more requests to the fastest machines.

In our last hypothesis we stated that we wanted to find an optimal solution for switching between different algorithms that performed optimal for current traffic intensity. When considering the steep curve taking us from comfort zone to complete saturation, it would be better to increase the number of servers rather than switching between algorithms. Since network traffic is considered to be bursty and unpredictable [13] it would be very hard to predict performance when the servers enter the thrashing regime. An optimal approach to maximize performance would be to use the RR algorithm under low to medium workloads, and switch to the LC algorithm when the workload is high. This can be implemented by having a load threshold telling the dispatcher when to switch between the algorithm.

Although one could implement a safety switch that changed from RR to the LC algorithm if the traffic intensity exceeded some threshold, at least if the server park is inhomogeneous, since the differences under high loads become more significant under such conditions.

7.2 Further Work

The scope of this thesis needed some adjustements during the course of the semester. Some of the experiments originally planned had to be dropped due to time considerations.

One of the things that need further investigation is when the dispatcher

becomes a bottleneck. From our experiments we found that going from 4 to 5 servers under high loads would not increase relative performance in the same magnitude as going from 3 to 4 servers. It would be interesting to use more servers and see if we could find a trend describing the performance increase when adding more servers.

The fact that our experiment outperformed the queueing models in most scenarios was rather surprising, and we suspect that the static service time and Poisson inter-arrival time might be the reason. Further tests could be performed using more realistic traffic patterns. We would like to use heavy-tailed distributions both in file sizes requested and in the inter-arrival time of requests. Heavy-tailed traffic distributions would probably have negative impact on server performance, and we could hypothesize that our experimental results would converge with the $M/M/1^k$ queue as first predicted.

In the section where we tested relative performance between algorithms there are also some unanswered questions. Further investigation could help us find the cause of why the RT algorithm performed no better than random chance. The RT algorithm is dependent on state information from the server, which in these experiments was gathered by measuring the time it took to setup and teardown a tcp session (this was performed every 5th second). There are many other available polling methods available: ICMP, SNMP, HTTP, ARP, etc. Some of these might give us better indications of current load on a server. The time between polls could also be further investigated.

If we want to create an adaptive scheme that change queueing algorithms based on load or current traffic intensity, we need to investigate further to find a more exact point where the RR algorithm succumb to noise. If we can predict this point with greater accuracy, a good suggestion would be to use this as a threshold for switching to the LC algorithm, which would endure higher loads and perhaps avoid congestion.

Bibliography

[1] Redirection algorithms for load sharing in distributed web-server systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 528, Washington, DC, USA, 1999. IEEE Computer Society.

[2] M. Arregoces and M. Portolani. *Data Center Fundamentals*. Cisco Press, Indiapolis, IN 46240 USA, 2004.

[3] Luis Aversa and Azer Bestavros. Load balancing a cluster of web servers using distributed packet rewriting. In *Performance, Computing, and Communications Conference, 2000. IPCCC '00. Conference Proceeding of the IEEE International*, pages 24–29, Boston, MA, USA, 1999. Boston University.

[4] Jaiganesh Balasubramanian, Douglas C. Schmidt, Lawrence Dowdy, and Ossama Othman. Evaluating the performance of middleware load balancing strategies. In *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*, pages 135–146, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 151–160, New York, NY, USA, 1998. ACM Press.

[6] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 1–16, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[7] Tony Bourke. *Server Load Balancing*. O'Reilly & Associates, 101 Morris Street, Sebastopol, CA 95472, first edition edition, August 2001.

[8] M. Burgess. *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester, 2004.

[9] M. Burgess and S. Ulland. Uncertainty in global application services with load sharing policy. In *17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006), volume submitted*. Springer, 2006.

[10] J. P. Buzen. Operational analysis: An alternative to stochastic modelling. In *Performance of Computer Installations*, pages 175–194, North Holland, 1978.

[11] V Cardellini and M Colajanni. Dynamic load balancing on web-server systems. *Internet Computing IEEE*, 3(4):28–39, 1999.

[12] Mark E. Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Trans. Netw.*, 5(6):835–846, 1997.

[13] Ashok Erramilli, Onuttom Narayan, and Walter Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Trans. Netw.*, 4(2):209–223, 1996.

[14] J. Sauv et al. Sla design from a business perspective. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*. IEEE Press, 2006.

[15] E Klovning H Bryhni and O Kure. A comparison of load balancing techniques for scalable web servers. 14(4):58–64, 2000.

[16] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. *IEEE/ACM Trans. Netw.*, 10(5):589–603, 2002.

[17] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key differences between http/1.0 and http/1.1.

[18] Hwa-Chun Lin and C. S. Raghavendra. A dynamic load-balancing policy with a central job dispatcher (lbc). *IEEE Trans. Softw. Eng.*, 18(2):148–158, 1992.

[19] D. A. Menasc and V. A .F Almeida. *Capacity Planning for Web Services*. Prentice Hall, Upper Saddle River, New Jersey 07458, first edition edition, 2002.

[20] D. A. Menasc, V. A .F Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Prentice Hall, Upper Saddle River, New Jersey 07458, first edition edition, 1994.

[21] Michael Mitzenmacher. On the analysis of randomized load balancing schemes. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 292–301, New York, NY, USA, 1997. ACM Press.

[22] Dan Mosedale, William Foss, and Rob McCool. Lessons learned administering netscape's internet site. *IEEE Internet Computing*, 1(2):28–35, 1997.

[23] Sucheta Nadimpalli and Shikharesh Majumdar. Techniques for achieving high performance web servers. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 233, Washington, DC, USA, 2000. IEEE Computer Society.

[24] Nortel Networks. Nortel application switch operating system: Application guide. January 2006.

[25] Kihong Park, Gitae Kim, and Mark Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *ICNP '96: Proceedings of the 1996 International Conference on Network Protocols (ICNP '96)*, page 171, Washington, DC, USA, 1996. IEEE Computer Society.

[26] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Trans. Netw.*, 3(3):226–244, 1995.

[27] J.H. Bjørnstad and M. Burgess. On the reliability of service level estimators in the data centre. In *17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006), volume submitted*. Springer, 2006.

[28] Milan E. Soklic. Simulation of load balancing algorithms: a comparative study. *SIGCSE Bull.*, 34(4):138–141, 2002.

[29] Francois Spies. Modeling of optimal load balancing strategy using queueing theory. *Microprocess. Microprogram.*, 41(8-9):555–570, 1996.

[30] YM Teo and R Ayani. Comparison of load balancing strategies on cluster-based web servers. *Transactions of the Society for Modeling and Simulation*, 2001.

[31] G. Teodoro, T. Tavares, B. Coutinho, Jr. W. Meira, and D. Guedes. Load balancing on stateful clustered web servers. In *SBAC-PAD '03: Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, page 207, Washington, DC, USA, 2003. IEEE Computer Society.

[32] S. Singhal W. Xu, X. Zhu and Z. Wang. Predictive control for dynamic resource allocation in enterprise data centers. In *10th IFIP/IEEE Network Operations and Management Symposium (NOMS 2006)*, pages 115–126. IEEE Press, 2006.

Appendix A

Scripts

This part of the appendix will include the source code of the scripts used for testing and processing of results. The scripts consists of perl, bash and c code. The functioning of each script will be described in the source code, as will the functions etc.

A.1 process.pl

```
#!/usr/bin/perl
#
# This script will take several result files from autobench
# and calulate mean and standard deviation values
#
# Example usage: process.pl <number of resultfiles>
#
# The above example have a couple of prerequisuites. First
# the program needs to be executed in the same directory as
# the resultfiles. Second the resultfiles have names like:
# result1.tsv, result2.tsv, result3.tsv.....

unless ($#ARGV + 1 == 1 )
{
print STDERR "Incorrect number of arguments\n";
print STDERR "Usage: $0 <number of resultfiles>\n\n";
exit 1;
}

$num_files = $ARGV[0];

#check whether files exists
check_files();

#open files
open_files(input);

my %output;

my @result1_lines = <RESULT1_FH>;
my @result2_lines = <RESULT2_FH>;
my @result3_lines = <RESULT3_FH>;
my @result4_lines = <RESULT4_FH>;
my @result5_lines = <RESULT5_FH>;
```

```
my @result6_lines = <RESULT6_FH>;
my @result7_lines = <RESULT7_FH>;
my @result8_lines = <RESULT8_FH>;
my @result9_lines = <RESULT9_FH>;
my @result10_lines = <RESULT10_FH>;

#close files
close_files(input);

#calculate mean and std.dev from data
process();

#print results to new file
print_output();

sub process
{
$line_counter = 0;

        # remove trailing carriage return characters if they exists
chomp($result1_lines1[$#result1_lines]);
chomp($result2_lines1[$#result2_lines]);
chomp($result3_lines1[$#result3_lines]);
chomp($result4_lines1[$#result4_lines]);
chomp($result5_lines1[$#result5_lines]);
chomp($result6_lines1[$#result6_lines]);
chomp($result7_lines1[$#result7_lines]);
chomp($result8_lines1[$#result8_lines]);
chomp($result9_lines1[$#result9_lines]);
chomp($result10_lines1[$#result10_lines]);

# assuming all files have the same amount of lines, we iterate
# through each line and extracts the information needed
for my $result1_line (@result1_lines)
{

my @respons;
my @netio;
my @errors;

@result1_fields = split ("\t", $result1_line);

if($result1_fields[0] eq 'begin')
{
$line_counter++;
next;
}


$result2_line = $result2_lines[$line_counter];
@result2_fields = split ("\t", $result2_line);

$result3_line = $result3_lines[$line_counter];
@result3_fields = split ("\t", $result3_line);

$result4_line = $result4_lines[$line_counter];
@result4_fields = split ("\t", $result4_line);

$result5_line = $result5_lines[$line_counter];
@result5_fields = split ("\t", $result5_line);

$result6_line = $result6_lines[$line_counter];
@result6_fields = split ("\t", $result6_line);

$result7_line = $result7_lines[$line_counter];
```

```
@result7_fields = split ("\t", $result7_line);

$result8_line = $result8_lines[$line_counter];
@result8_fields = split ("\t", $result8_line);

$result9_line = $result9_lines[$line_counter];
@result9_fields = split ("\t", $result9_line);

$result10_line = $result10_lines[$line_counter];
@result10_fields = split ("\t", $result10_line);

push (@respons, $result1_fields[9]);
push (@respons, $result2_fields[9]);
push (@respons, $result3_fields[9]);
push (@respons, $result4_fields[9]);
push (@respons, $result5_fields[9]);
push (@respons, $result6_fields[9]);
push (@respons, $result7_fields[9]);
push (@respons, $result8_fields[9]);
push (@respons, $result9_fields[9]);
push (@respons, $result10_fields[9]);

push (@netio, $result1_fields[10]);
push (@netio, $result2_fields[10]);
push (@netio, $result3_fields[10]);
push (@netio, $result4_fields[10]);
push (@netio, $result5_fields[10]);
push (@netio, $result6_fields[10]);
push (@netio, $result7_fields[10]);
push (@netio, $result8_fields[10]);
push (@netio, $result9_fields[10]);
push (@netio, $result10_fields[10]);

push (@errors, $result1_fields[11]);
push (@errors, $result2_fields[11]);
push (@errors, $result3_fields[11]);
push (@errors, $result4_fields[11]);
push (@errors, $result5_fields[11]);
push (@errors, $result6_fields[11]);
push (@errors, $result7_fields[11]);
push (@errors, $result8_fields[11]);
push (@errors, $result9_fields[11]);
push (@errors, $result10_fields[11]);


# calculating mean and std.dev and putting the results
# into our output hash
$output{$line_counter}{req_rate} = $result1_fields[2];
$output{$line_counter}{resp_time_mean} = calculateMean(@respons);
$output{$line_counter}{resp_time_stddev} = calculateStdDev(@respons);
$output{$line_counter}{netio_mean} = calculateMean(@netio);
$output{$line_counter}{errors_mean} = calculateMean(@errors);

$line_counter++;
}

}

sub print_output
{
open_files(output);

print RESULTS_FH "req_rate\t";
print RESULTS_FH "resp_mean\t";
print RESULTS_FH "resp_stddev\t";
```

```
print RESULTS_FH "netio_mean\t";
print RESULTS_FH "errors_mean\t";
print RESULTS_FH "\n";

# printing the content of our output hash to file
for my $key (sort {$a <=> $b} keys %output)
{
print RESULTS_FH "$output{$key}{req_rate}\t";
print RESULTS_FH "$output{$key}{resp_time_mean}\t";
print RESULTS_FH "$output{$key}{resp_time_stddev}\t";
print RESULTS_FH "$output{$key}{netio_mean}\t";
print RESULTS_FH "$output{$key}{errors_mean}\t";
print RESULTS_FH "\n";
}
close_files(output);
}

sub open_files
{
if($_[0] eq 'input')
{
open (RESULT1_FH, "< result1.tsv") || die "Cannot open file, $!";
open (RESULT2_FH, "< result2.tsv") || die "Cannot open file, $!";
open (RESULT3_FH, "< result3.tsv") || die "Cannot open file, $!";
open (RESULT4_FH, "< result4.tsv") || die "Cannot open file, $!";
open (RESULT5_FH, "< result5.tsv") || die "Cannot open file, $!";
open (RESULT6_FH, "< result6.tsv") || die "Cannot open file, $!";
open (RESULT7_FH, "< result7.tsv") || die "Cannot open file, $!";
open (RESULT8_FH, "< result8.tsv") || die "Cannot open file, $!";
open (RESULT9_FH, "< result9.tsv") || die "Cannot open file, $!";
open (RESULT10_FH, "< result10.tsv") || die "Cannot open file, $!";
}
if($_[0] eq 'output')
{
open (RESULTS_FH, "> processed") || die "Cannot open file, $!";
}
}

sub close_files
{

if($_[0] eq 'input')
{
close (RESULT1_FH);
close (RESULT2_FH);
close (RESULT3_FH);
close (RESULT4_FH);
close (RESULT5_FH);
close (RESULT6_FH);
close (RESULT7_FH);
close (RESULT8_FH);
close (RESULT9_FH);
close (RESULT10_FH);
} elsif ($_[0] eq 'output')
{
close (RESULTS_FH);
}
}

sub check_files
{
# Check to see whether all result files are present
for ($i = 1; $i < $num_files + 1; $i++)
{
$missing_bool = 0;
```

```
if (! -f "result$i.tsv")
{
print STDERR "Missing file result$i.tsv..\n";
$missing_bool = 1;
}
}
if ($missing_bool == 1)
{
print STDERR "One or several files missing.. Quitting\n";
exit 1;
}
}

sub calculateMean
{
$sum = 0;
$n = 0;

foreach $x (@_)
{
$sum += $x;
$n++;
}
$mean = ($sum / $n);
return $mean;
}

sub calculateStdDev
{
    my $n;               # the number of values
    my $stddev;          # the standard deviation, calculated by this
                         # subroutine, from the variance
    my $sum;             # the sum of the values
    my $sumOfSquares;    # the sum of the squares of the values
    my $variance;        # the variance, calculated by this subroutine
    my $x;               # set to each value

    #   Initialize variables.
    $n = 0;
    $sum = 0;
    $sumOfSquares = 0;

    #   Use a foreach loop to get each value from the array @_, which
    #   is the argument array and which contains the values.
    foreach $x ( @_ )
    {
        #   Add the value to the growing sum.
        $sum += $x;

        #   Increment $n, the number of values.
        $n++;

        #   Add the square of the value to the growing sum of the squares.
        $sumOfSquares += $x * $x;
    }

    #   Calculate the variance.
      $variance = ( $sumOfSquares - ( ( $sum * $sum ) / $n ) ) / ( $n );

    #   Calculate the standard deviation, which is the square root of the
    #   variance.
      $stddev = sqrt( $variance );
      #   Return the calculated standard deviation.
      return $stddev;
```

A.2 merge_results.pl

This script is only a modification of a script made by Matt Disney. The script originally merges result files created by windows versions of sar and autobench, which has different formatting. It is also added functionality to perform standard deviation calculations.

```
#!/usr/bin/perl
#
# This script combines the output from autobench and sar, using timestamps
# as the relational key.
#
#
# IMPORTANT: This script can *not* handle tests that span midnight!
#
# It expects the "name" of a test run. The output files are expected to have
# this name as well.
#
# Example usage:
# merge_results.sh test1
#
# The above example will combine the results of ~/results/autobench/test1.dat
# and ~/results/sardata/test1.sar into ~/results/merged/test1.dat. The merged
# data should be tab-delimited and easily machine-readable.

# Check command-line arguments. There can be only one!
# ($#ARGV is a reference to the index number of the last element in the ARGV array,
#  which contains all the command-line arguments *except* the program name itself.)

unless ( $#ARGV + 1 == 1 )
{
  print STDERR "Incorrect number of arguments\n";
  usage();
}

$project_name = $ARGV[0];

# Here we will setup our directory variables inside a hash (associative
# array), allowing us to loop through them later using the "keys" method.
my %dirs;

$dirs{'home'}        = "/home/gard/school/thesis";
$dirs{'results'}     = $dirs{home}."/results";
$dirs{'ab_results'}  = $dirs{results}."/autobench";
$dirs{'sar_results'} = $dirs{results}."/sardata";
$dirs{'merged'}      = $dirs{results}."/merged";

# Create a hash of the data filenames.
my %files;

%files =
(
  autobench => $dirs{'ab_results'}."/$project_name.dat",
  sar       => $dirs{'sar_results'}."/$project_name.sar",
  merged    => $dirs{'merged'}."/$project_name"."_merged.dat"
);


check_exist();

open_files(input);

# Read the file contents into arrays so that we don't have to worry about
```

```
# closing the files at the right time.
my @ab_lines  = <AB_FH>;
my @sar_lines = <SAR_FH>;

close_files(input);

# Now let's setup a hash of arrays that we will use to store the output lines.
my %output;

for my $ab_line (@ab_lines)
{
  my @abfields;
  my @matchlines;
  my @sarfields;
  my $nowepoch = 0;
  my $endepoch = 0;
  my @nowary;
  my @endary;

  # Break the $ab_line, delimited by tabs (\t), into an array
  @abfields = split ("\t", $ab_line);

  if ( $abfields[0] eq "begin" ) { next; }

  my $now   = $abfields[0];
  @nowary   = split (':', $now);
  # Hours
  $nowepoch += ($nowary[0] * 60 * 60);
  # Minutes
  $nowepoch += ($nowary[1] * 60);
  # Seconds
  $nowepoch += $nowary[2];

  chomp($abfields[$#abfields]);

  $output{$now}{end}   = $abfields[1];
  $output{$now}{dem_req_rate}  = $abfields[2];
  $output{$now}{req_rate}  = $abfields[3];
  $output{$now}{conn_rate}  = $abfields[4];
  $output{$now}{min_rep_rate}  = $abfields[5];
  $output{$now}{avg_rep_rate}  = $abfields[6];
  $output{$now}{max_rep_rate}  = $abfields[7];
  $output{$now}{stddev_rep_rate}  = $abfields[8];
  $output{$now}{resp_time}   = $abfields[9];
  $output{$now}{net_io}   = $abfields[10];
  $output{$now}{errors}   = $abfields[11];


  @endary   = split (':', $output{$now}{end});
  # Hours
  $endepoch += ($endary[0] * 60 * 60);
  # Minutes
  $endepoch += ($endary[1] * 60);
  # Seconds
  $endepoch += $endary[2];

  my $accumulating     = 0;
  my $cpu_busy_accum   = 0;
  my $cpu_busy_counter = 0;
  my $sar_time         = 0;
  my $sarepoch         = 0;
  my $sdeviation       = 0;
  my @sar_time_ary;
  my @epochvalues;
```

```
  # Now, loop through our grep matches.
  for my $sar_line (@sar_lines)
  {
    # Split the $sar_line, delimited by tabs, into an array
    # Here we use the regex match (//) representation of whitespace (\s).
    @sarfields = split (/\s+/ , $sar_line);

    $sar_time = $sarfields[0];

    @sar_time_ary   = split (':', $sar_time);
    # Hours
    $sarepoch = ($sar_time_ary[0] * 60 * 60);
    # Minutes
    $sarepoch += ($sar_time_ary[1] * 60);
    # Seconds
    $sarepoch += $sar_time_ary[2];

    # We are only interrested in lines with cpu load
    if ( $sarfields[1] =~ /all/ )
    {
      # If appropriate, start the accumulation necessary to calculate averages.
      if ( ($nowepoch <= $sarepoch) && ($accumulating == 0) )
      {
        # Start accumulating!
        $accumulating = 1;
        print "Now processing the test $now - $output{$now}{'end'} ... "
      }


      # If we're accumulating, then we want to increment stuff...
      if ( ( $accumulating == 1 ) && ( $sarepoch <= $endepoch ) )
      {
          $cpu_busy_counter += 1;
          $cpu_busy_accum   += (100 - $sarfields[8]);
  push ( @epochvalues, $sarfields[8] );
  next;
      }

      # If we have reached the end of our test period, then we
      # want to reset our variables and perform the average computations.
      if ( $sarepoch > $endepoch )
      {
        print "done.\n";
        if ( $cpu_busy_counter != 0 )
{
   $output{$now}{'cpu_busy'} = $cpu_busy_accum / $cpu_busy_counter;
          $output{$now}{'deviation'} = calculateStdDev( @epochvalues );
} else
{
   $output{$now}{'cpu_busy'} = 100;
          $output{$now}{'deviation'} = 0;
}
$cpu_busy_accum   = 0;
        $cpu_busy_counter = 0;
$accumulating     = 0;
@epochvalues   = 0;
        last;
      }
    }
  }
}

print_output();
```

```
exit 0;

sub calculateStdDev
{
    my $n;               #   the number of values
    my $stddev;          #   the standard deviation, calculated by this
                         #   subroutine, from the variance
    my $sum;             #   the sum of the values
    my $sumOfSquares;    #   the sum of the squares of the values
    my $variance;        #   the variance, calculated by this subroutine
    my $x;               #   set to each value

    #   Initialize variables.
    $n = 0;
    $sum = 0;
    $sumOfSquares = 0;

    #   Use a foreach loop to get each value from the array @_, which
    #   is the argument array and which contains the values.
    foreach $x ( @_ )
    {
        #   Add the value to the growing sum.
        $sum += $x;

        #   Increment $n, the number of values.
        $n++;

        #   Add the square of the value to the growing sum of the squares.
        $sumOfSquares += $x * $x;
    }

    #   Calculate the variance.
    $variance = ( $sumOfSquares - ( ( $sum * $sum ) / $n ) ) / ( $n - 1 );

    #   Calculate the standard deviation, which is the square root of the
    #   variance.
    $stddev = sqrt( $variance );

    #   Return the calculated standard deviation.
    return $stddev;
}

sub print_output
{
  open_files(output);
  print MERGED_FH "begin\t";
  print MERGED_FH "end\t";
  print MERGED_FH "dem_req_rate\t";
  print MERGED_FH "req_rate\t";
  print MERGED_FH "conn_rate\t";
  print MERGED_FH "min_rep_rate\t";
  print MERGED_FH "avg_rep_rate\t";
  print MERGED_FH "max_rep_rate\t";
  print MERGED_FH "stddev_rep_rate\t";
  print MERGED_FH "resp_time\t";
  print MERGED_FH "net_io\t";
  print MERGED_FH "cpu_busy\t";
  print MERGED_FH "deviation\t";
  print MERGED_FH "errors\t";
  print MERGED_FH "\n";

  for my $key (sort keys %output)
  {
    print MERGED_FH "$key\t";
    print MERGED_FH "$output{$key}{end}\t";
```

```
    print MERGED_FH "$output{$key}{dem_req_rate}\t";
    print MERGED_FH "$output{$key}{req_rate}\t";
    print MERGED_FH "$output{$key}{conn_rate}\t";
    print MERGED_FH "$output{$key}{min_rep_rate}\t";
    print MERGED_FH "$output{$key}{avg_rep_rate}\t";
    print MERGED_FH "$output{$key}{max_rep_rate}\t";
    print MERGED_FH "$output{$key}{stddev_rep_rate}\t";
    print MERGED_FH "$output{$key}{resp_time}\t";
    print MERGED_FH "$output{$key}{net_io}\t";
    print MERGED_FH "$output{$key}{cpu_busy}\t";
    print MERGED_FH "$output{$key}{deviation}\t";
    print MERGED_FH "$output{$key}{errors}\t";
    print MERGED_FH "\n";
  }
  close_files(output);
}


sub check_exist
{
  # Check to make sure all our directories exist.
  for my $key (keys %dirs)
  {
    if ( ! -d $dirs{$key} )
    {
      print STDERR "Directory $dirs{$key} does not exist! Quitting.\n";
      exit 1;
    }
  }


  # Check to make sure the files do or don't exist, as appropriate.
  for my $key (keys %files)
  {
    if ( $key eq 'merged' ) {
      # We do *not* want the merged data file to exist already, otherwise
      # we risk overwriting important data.
      if ( -f $files{'merged'} )
      {
        print STDERR "Merged data file $files{'merged'} already exists! Quitting.\n";
        exit 1;
      }
    } else
    {
      # If one of the data files doesn't exist, we have nothing to do.
      if ( ! -f $files{$key} )
      {
        print STDERR "Data file $files{$key} does not exist! Quitting.\n";
        exit 1;
      }
    }
  }
}


sub usage
{
  # $0 in Perl is always the program name that has been run.
  print STDERR "Usage: $0 testname\n\n";
  exit 1;
}

# Open up the data files:
sub open_files
{
```

```
  if ($_[0] eq 'input')
  {
    # The autobench is read-only.
    open (AB_FH, "< $files{'autobench'}") || die "Cannot open file, $!";
    # The sar file is read-only.
    open (SAR_FH, "< $files{'sar'}")  || die "Cannot open file, $!";
  } elsif ($_[0] eq 'output')
  {
    # The merged file is read-write (not concat).
    open (MERGED_FH, "> $files{'merged'}")  || die "Cannot open file, $!";
  }
}

sub close_files
{

  if ($_[0] eq 'input')
  {
    close (SAR_FH);
    close (AB_FH);
  } elsif($_[0] eq 'output')
  {
    close (MERGED_FH);
  }
}
```

A.3 queues.c

This script is used to generate response times using the well known queueing models $M/M/k$ and $M/M/1^k$

```
/*****************************************************************************/
/*                                                                           */
/* File: queues.c                                                            */
/*                                                                           */
/* Created: Mon Feb 20 11:26:32 2006                                         */
/*                                                                           */
/* Author: Mark Burgess, Jon Henrik Bjrnstad and Gard Undheim               */
/*                                                                           */
/* Revision: 2                                                               */
/*                                                                           */
/* Compile: gcc -o queue queue.c -lm                                         */
/*                                                                           */
/* Purpose:  This scripts is used for calculating response times using       */
/*           queueing models: n*M/M/1 and M/M/k                              */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <math.h>

extern double strtod();

int main (int argc, char **argv)
{
        /* variable and function definitions */
        double rate_ms,rate,max_rate,rho,mu,lambda,n,p0,kappa,sum1,sum2;
        double mm1best, mmk, factorial();
        int max_servers;

        /* verify correct number of arguments */
if(argc != 5)
```

```
        {
                printf("Usage: %s <lambda> <mu> <number-of-servers> <max-rate>\n", argv[0]);
                return 1;
        }

        /* assigning input parameteres to variables */
        lambda = strtod(argv[1],NULL);
        mu = strtod(argv[2], NULL);
        max_servers = atoi(argv[3]);
        max_rate = strtod(argv[4],NULL);


        /* measure everything in bytes per second */
        printf("k\t\tmm1best\t\tmmk\n");
        printf("------------------------------\n");

        /* increases start rate until it reaches max_rate */
        for (rate = lambda*1000; rate <= max_rate*1000; rate++)
        {
                rate_ms = rate/1000;
                rho = rate_ms/(max_servers*mu);
                sum1 = 0;
                /*mm1 = 1.0/(mu-rate_ms);*/
                mm1best = 1.0/(mu-rate_ms/max_servers);

                for (n = 1; n < max_servers; n++)
                {
                        sum1 += pow(max_servers*rho,n)/factorial(n);
                }
p0 = 1.0/(1.0 + sum1 + pow(max_servers*rho,max_servers)/((factorial(max_servers)*(1-rho))));
                kappa = pow(rho*max_servers,max_servers)*p0/(factorial(max_servers)*(1-rho));
                mmk = (1.0 + kappa/(max_servers*(1-rho)))/mu;
                printf("%g\t%g\t%g\n",rate,mm1best,mmk);
        }
        return 0;
}

/*****************************************/

double factorial(double n)
{
        double f,i;
        f = 1;
        i = n;

        while (i > 1)
        {
                f *= i--;
        }
        return f;
}
```