UNIVERSITY OF OSLO
Department of Informatics

High-Level Load Balancing for Web Services

Master thesis

Sven Ingebrigt
Ulland

20th May 2006

Abstract

A structured approach to high availability and fault tolerance is essential in a production-grade service delivery network, where delays and faults can occur for a multitude of reasons. In this thesis, we consider the high-level scheduling and load sharing properties offered by the Domain Name System, as implemented in popular DNS software packages. At this level, the scheduling mechanism can account for server availability, geographical proximity, time zones, etc. We explore the performance and capabilities of high-level DNS-based load balancing, where we draw special attention to the choice of caching policy (time-to-live) for DNS data. Our findings confirm the high performance of modern DNS server implementations, but question the use of DNS as a suitable load balancing mechanism in itself. Further, we analyse the use of a database-supported DNS service for allowing highly dynamical query responses, and show that this approach has both potentially negative (single point of failure) and positive (improved balancing flexibility) properties.

Preface

The work presented herein signifies the conclusion of a two year master's degree in network and system administration attained at Oslo University College in collaboration with the University of Oslo. The degree has spanned the years 2004 through 2006, with this thesis being the focus of the last four months in the final semester.

High level load balancing is the main topic of the thesis, motivated by a cooperative proposal to study the nature of operation critical services. Indeed, the topic was suggested as part of a set of parallel projects on load balancing for networked services, each with a different approach. Where in this paper attention is drawn towards high level properties of load balancing, the other projects address lower level characteristics of such systems.

With a basis in the results obtained in this work, Professor Mark Burgess and myself have co-written a paper entitled *Uncertainty in Global Application Services with Load Sharing Policy*, which has been submitted to the *17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006)* conference. It is currently awaiting acceptance as per May 2006.

Now at the summit of two years of demanding work, there are many people to whom I am most indebted – their help and suggestions have been and are greatly appreciated. First, I would like to extend my genuine thanks to my advisor and course instructor, Professor Mark Burgess, whose enthusiasm, dedication and understanding have guided me and my work on the right path many a time throughout the degree, especially these last months. Also, I am very grateful for the cooperation and support from my fellow students and friends, Jon Henrik Bjørnstad, Espen Tymczuk Braastad, Ilir Bytyci and Gard Undheim. Without the encouragement and mutual understanding inspired by their presence through these two years, the work would not have progressed as smoothly as it did. A special thanks to Kyrre M. Begnum for his efforts in maintaining the virtual network setup, and helping me with problems and questions in its regard. To my family and friends, your support has been very much honoured in these months of challenging endeavour.

Oslo, May 20th 2006

Sven Ingebrigt Ulland

Contents

Abstract **i**

Preface **iii**

1 Introduction **1**
1.1 The Need for Reliable Services 1
1.1.1 Quality of Service 2
1.1.1.1 Round-Trip Time 2
1.1.1.2 Throughput 3
1.1.2 The Flash Crowd Effect 3
1.1.3 High Sustained Load 4
1.2 Methods of Combatting Congestion 4
1.2.1 Topology Overview 5
1.2.1.1 Extending Topologies 6
1.3 Brief Note on Terminology 7
1.4 Paper Organisation 7

2 Overview of Load Balancing **9**
2.1 The Load Balancing Big Picture 9
2.1.1 Cause Tree Perspective 10
2.1.1.1 Levels of Scheduling 11
2.1.1.2 High-Level Scheduling 12
2.2 Modes of Operation 12
2.2.1 Load Balancer Components 13
2.2.2 Lower Level Load-Balancing 16
2.2.3 The Domain Name System 18
2.2.3.1 Example of Use 21
2.2.3.2 Challenging Caches 22
2.2.4 Routing-based Load Balancing 24
2.2.4.1 IP Anycast 24
2.2.4.2 Site Multihoming 25
2.2.5 Global Server Load Balancing 26
2.3 Related Technologies 27

2.3.1 Content Delivery Networks 28
2.3.2 Multicast 28
2.3.3 HTTP Redirect 28
2.4 Traffic Characteristics 29

3 Previous Research **31**
3.1 DNS-based Load Balancing 31
3.2 Summary 34
3.3 Traffic Characteristics 35
3.3.1 Pareto and Gaussian Distributions 35
3.4 Aim of this Study 37

4 Hypotheses **39**

5 Experimental Design **41**
5.1 System Modelling 41
5.1.1 Queues and Delays 41
5.2 Hardware Considerations 43
5.2.1 Virtual Network with Xen 43
5.2.2 Emulating WANs – NetEm and IProute 45
5.2.2.1 Delay Distributions 46
5.3 Software 48
5.3.1 Operating System Details 49
5.3.2 DNS Server – BIND 50
5.3.3 DNS Server – PowerDNS 51
5.3.4 Apache Webserver 52
5.3.5 Traffic Generator Tools 53
5.3.6 QueryPerf – DNS testing 53
5.3.7 Flood – HTTP testing 53
5.3.8 HTTPerf – HTTP testing 54
5.3.9 Clock Synchronisation – NTP 55
5.3.10 Measuring Load 55
5.3.11 Analysis Tools 55

6 Methodology **57**
6.1 DNS Server Performance 57
6.2 RRset Ordering Implications 58
6.3 BIND Scheduling Entropy 59
6.4 Impact of TTL on Response-time 60
6.4.1 Static Back-end 60
6.4.2 Dynamic Back-end 62

7 Experiment Results **65**
7.1 DNS Server Performance 65
7.2 BIND RRset Ordering 67
7.3 BIND Scheduling Entropy 68
7.4 Impact of TTL on Response-time 71
7.4.1 Static Back-end 71
7.4.2 Dynamic Back-end 73
7.4.3 Dynamic Back-end Equilibrium 77

8 Conclusions and Discussion **81**
8.1 Review of Hypotheses 81
8.1.1 DNS Performance 81
8.1.2 RRset Ordering 82
8.1.3 Scheduling Entropy 83
8.1.4 Effect of TTL 83
8.2 On DNS as a Balancing Mechanism 83
8.3 Future Work 84
8.3.1 Alternative Protocols 84
8.3.2 Proactive Caching 85

A Configuration Files **91**
A.1 MLN/Xen Configuration 91
A.2 Flood Configuration 94
A.2.1 Flood static 94
A.2.2 Flood dynamic 95

B Scripts **97**
B.1 Analysis Scripts 97
B.2 Web Server Scripts 103

List of Figures

1.1 Insertion points for load balancing. 7

2.1 Simplified QoS cause tree. 11
2.2 Brief overview of basic queueing models. 13
2.3 Essential components in a load balancer. 14
2.4 Dispatcher-based topology setup. 17
2.5 Standard DNS lookup procedure. 19
2.6 Potential caching-points for DNS lookups. 23
2.7 Simplified schema of a multihomed site. 25
2.8 Global server load balancing. 27

3.1 Examples of Pareto and Gaussian distributions. 36

4.1 Supposed effect of TTL on RTT. 40

5.1 Simplified Wide-Area Network Model. 42
5.2 Xen – conceptual design. 44
5.3 NetEm delay distributions. 47
5.4 Pareto Generator Phenomenon. 48
5.5 Logical Network Setup. 49

6.1 Experiment Setup – DNS Performance. 58
6.2 Experiment Setup – RRset Ordering. 59
6.3 Supposed effect of TTL on RTT. 61
6.4 Experiment setup, RTT vs TTL. 61

7.1 DNS server performance. 66
7.2 Normalised request distributions. 68
7.3 Flood static – Total page load time. 71
7.4 Flood static – DNS lookup times. 72
7.5 Flood static – DNS lookup times (scaled). 73
7.6 Flood static – HTTP round-trip times. 74
7.7 Flood dynamic – HTTP response time. 75
7.8 Web server utilisation with varying TTL. 76

7.9 Flood dynamic – Cumulative HTTP response time. 77
7.10 Flood dynamic – TTL equilibrium. 78

List of Tables

2.1 Typical metrics used in load-balancing algorithms. 15

3.1 A selection of DNS load-balancing approaches. 32

5.1 Traffic Generator Tools. 53

6.1 Flood static – NetEm settings. 62

7.1 RRset ordering implications. 67
7.2 Normalised request distributions. 69

Chapter 1

Introduction

Since its infancy, the Internet has experienced a near exponential growth in user base, infrastructure, content size and resources like low-latency, high throughput network links. According to the Internet World Stats initiative, Internet users now total over one billion – approximately 16 percent of the world's population. [1] This explosive increase means that high traffic sites offering e-commerce, community and other resource intensive services, face an enormous challenge when it comes to ensuring high availability and fault tolerance for their services. This paper examines how load balancing is used as a central concept to achieve these goals while ensuring transparency and interoperability with existing technology.

1.1 The Need for Reliable Services

With online business-to-business and business-to-customer revenues soaring to ever new heights each passing year, it is trivial to deduce that for a company that relies heavily on online sales and service provision, even a small amount of downtime could have a disastrous impact on its economy. To put this in perspective, consider Google's total revenues for the first quarter of 2006 (92 days): 2,254 billion USD. [2] Assuming a uniform distribution of traffic over the entire period, this translates to $24,497 million per day, or roughly $284 per second.

Now, even with a seemingly high uptime rating of 99.99% (0.01% downtime), a company like Google would face a quarterly loss of around $22.5 million. The repercussions of downtime not only affect direct revenue lost to unavailable services; it could also potentially hurt a business' reputation. However, high-profile enterprises like Google and Amazon have, through their long term service quality, established strong trust relationships with customers. Accordingly, they would not experience massive customer migration, at least not for one-time occurences. Also, dissatisfied customers do not always have alternatives to turn to, and may very well decide to stay

with their current service provider, despite a degrade in service quality. It should be noted that the examples of world-class service systems, like the ones mentioned by name here, are not directly applicable to the cases of smaller companies or ventures that have not yet established a sustainable market share. In such situations, the quality of a service could mean the difference between growth and, in the worst case, bankruptcy.

The keywords here have already been mentioned: Service quality – more commonly coined *Quality of Service*, or QoS. This is the essence of what we would like to achieve with the diversity of techniques and solutions for high availability and load balancing.

1.1.1 Quality of Service

In a technical and quantitative context, QoS refers to the level of quality in a service, represented by several measurable factors. These factors typically include network metrics or properties such as latency, jitter, throughput, error rate, availability, etc. Through analysis of past and present communication information, a service provider is able to calculate several metrics which are then used to determine the QoS level in business-to-business or business-to-customer *service level agreements*, which are contracts detailing the QoS guarantees given by the provider.

Depending on scope, the term QoS is applied to different parts of a network. Often, it is related to the entire end-to-end communication path, that is, from a customer's local computer to the very server from which the content is retrieved. In other scenarios it might be applied to a single network link between two businesses. For a service provider, it is impossible to guarantee QoS levels of networks outside its jurisdiction. For example, a business serving audio streams to the public cannot be responsible for failures and limitations in the end-users' networks.

1.1.1.1 Round-Trip Time

In the context of load balancing and QoS, one factor is particularly interesting: The response time when querying a resource. Response time is often referred to as round-trip time (RTT) or latency. For example, we are interested in knowing how long a user has to wait for a general web page to load. This response time should be kept as low as possible: Research has shown that poor web site performance relates to degraded corporate image, and even has a poor impact on users' perception of site security. [3] As a user browses through a site, the tolerance for latency decreases with every page click. In other words, a user might tolerate a high initial delay, but will tend to not accept a static or increasing delay. This is an important point that will be discussed later.

1.1.1.2 Throughput

Another factor to consider for large sites is throughput. Though it is not necessarily as important as response time, it does have distinct applications. Throughput is a measure of how much data can be pushed through a connection over a given time interval. Common denominators for network throughput are bits and bytes per time unit, often preceded by SI or IEC binary prefixes; for example kilobits (10^3 bits) per second or mebibytes (2^{20} bytes) per second – kbps and MiBps respectively. Certain applications, like streaming live media and real-time interactive systems, normally have throughput requirements, e.g. 128kbps per connection for streaming audio. In such scenarios, response time does not play an important part as long it is kept below acceptable levels. Therefore, a company should determine what factor to focus on when designing their highly available, load-balanced infrastructure.

In general, we can take a step back and observe what is the cause of the high demands on server performance. Somewhat simplified, we observe two possible causes that we want to combat with load balancing: The flash crowd effect and high sustained load.

1.1.2 The Flash Crowd Effect

The relatively new concept of a "flash crowd" surfaced with the growing popularity of large commercial and community news sites, with prominent examples including Slashdot.org and Digg.com, together with world events such as war reports, the Olympics and various other cultural happenings. It refers to the effect of hundreds and thousands of users simultaneously accessing a certain web site or page, thus overloading the server capacity. The word "simultaneous" is not very accurate, however, because it implies that events happen at the exact same time. In this context, we use the term a bit more loosely to mean "over a relatively short time interval." Real-world intervals typically range from seconds and minutes to days and even weeks.

There are many relevant questions to ask when it comes to dealing with the flash crowd effect. Is it possible to safely expect when such an event will occur, or could it be modelled in a probabilistic manner? Are we able to determine the size of the crowd, and its requirements from our services? How could internal resources be regrouped to deal with the load, or would we end up with lots of unused or underutilised resources after the crowd? On-demand resource reallocation using virtualisation could be an alternative. Maybe co-location[1] is a temporary solution to a temporary problem?

[1] Co-location refers to the practice of renting server space and hardware at remote data centres and establishing service level agreements with one or more providers to offload the company's own resources.

These questions are not always straightforward to answer, and present a challenge to any undertaker willing or needing to complete projects that might face the effect.

1.1.3 High Sustained Load

Many popular sites experience a more or less continual high rate of traffic. Typical examples like Google, Microsoft, Amazon, BBC News and CNN are accessed by users from all over the world and thus at all times of the day. Being the largest search engine on the Internet, Google faces over 200 million queries per day, or roughly 2300 queries per second. Given an average query result of 20kB, this translates to around 3.6 terabytes of outgoing data per day. Again, Google is an extraordinary case, but it serves as a good example of the possibilities and what might lay ahead for smaller companies in the future, assuming a continuing growth of the online service market.

A service provider capable of handling a high sustained load will typically reserve more bandwidth and resources than is currently needed, simply to be able to cope with variations in traffic intensity – a technique often referred to as "over-provisioning." Deploying this kind of buffer or safeguard means that the flash crowd problem is also effectively dealt with, as long as the crowd size and demands remain below a certain threshold.

In this paper, we are interested in the latter case of congestion, where a service is put under relatively continuous pressure from users, though it also encompasses the possibility of flash crowds.

1.2 Methods of Combatting Congestion

When a service provider has identified and analysed the challenges of availability, the questions arise on how to mitigate the effects of high loads on infrastructure hardware and software. Solving such challenges is all about recognising and improving on bottlenecks, a process realised through the concept of *network engineering*, and its derivative, *traffic engineering*. These are basic yet complex building blocks underlying the QoS paradigm.

- Network engineering often refers to the practice of pinpointing specific service requirements and implementing both hardware and software solutions to meet these requirements.

- Traffic engineering (TE), on the other hand, is about introducing performance-enhancing optimisations in already operational networks. [4, 5] Normally, such optimisations are carried through using statistical and scientific principles oriented towards the four phases of TE: measurement, modelling, characterisation and control of Internet traffic. The main objective of the optimisations is simply to achieve

given QoS requirements with the intentional side-effect of lowered costs through utilising existing resources.

To summarise, traffic engineering seeks to effectively balance traffic load throughout existing networks, thus achieving QoS demands and minimising typical costs of adding hardware and software implementations, common to network engineering. Ideally, the two approaches complement each other in a manner satisfying both project requirements and budgets.

When dealing with real-world cases of load balancing, both network and traffic engineering are general purpose tools used throughout all steps of an implementation. However, they can only be applied to a limited scope of the network topology; typically that under administrative authority of the provider.

1.2.1 Topology Overview

Actual network and traffic engineering techniques could and *should* be introduced on multiple levels in any given topology: It is well known that a chain is only as strong as its weakest link. This holds very true for computer networking, which is why providers seek to strengthen each "link" by introducing redundant and load-balanced systems. Consider a company launching a campaign involving distribution of a streaming movie from their web pages. Even if they have a very capable cluster of web servers in their data centre, their network connection to the outside world could be disproportionate to the demand, and thus create a bottleneck or weak link.

It is apparent that multi-layered balancing and redundancy is required for any crucial online service. Details on the layered approach will follow in the next chapter. For now, consider the somewhat simplified topology in figure 1.1 to get a quick overview of the involved parties. Some of the most common spots for introducing redundancy and load balancing are enumerated.

1. Client-side load balancing is not a normal practice, but it is indeed possible. For some time, Netscape incorporated a simple balancing algorithm in their Navigator browser, making it choose a random Netscape web-server when visiting www.netscape.com. [6]

2. Core Internet routing uses protocols and agreements that allow for automated load balancing and fail-over mechanisms. These are commonly based on the Border Gateway Protocol, used for data-exchange between large Internet operators.

3. DNS-based load balancing, one of the main topics of this paper, is a popular way of distributing traffic amongst a set of Internet addresses

by returning a list of active addresses to the requesting client. These addresses can point to a set of servers or even a set of geographically separate sites.

4. Sites can connect to the net through several links, a practice known as *multihoming*. This enables both incoming and outgoing load balancing, in addition to the increased redundancy.

5. Dispatcher-based load balancing is used within a site to balance load between a set of real servers. Generally, the dispatcher assumes a virtual address for a service and receives requests which it then redirects to an appropriate server based on given criteria.

6. The real servers can again operate some form of balancing mechanism to decide whether to handle the request or redirect it to a more suitable server or site.

7. Content servers could access back-end servers – typically running databases and low-level services – in a load-balanced fashion.

8. Back-end servers could also incorporate balancing amongst themselves to avoid over-utilisation.

When talking about varying *levels* of load balancing, it is fair to identify the level to be proportional to the distance from the content served – long distance equals high level, and vice versa. In an informal manner, we can designate steps one through four in the figure as high-levels, and steps five through eight as low-levels of load balancing. This is discussed more in detail in the next chapter.

1.2.1.1 Extending Topologies

As mentioned earlier, it is generally not possible to exercise guarantees or even loose assumptions about QoS levels outside a provider's own network domain. Referring to the figure again, mid-size service operators would typically have control over their local network (steps five through eight), and possibly the DNS servers. The solution adopted by many operators in the industry is therefore to extend their domain. In stead of physically extending it by building new infrastructure, they negotiate contracts with other instances to handle traffic for them, creating a logical extension of operator authority. These contracts are the previously mentioned service levels agreements, SLAs. Business SLAs allow for a variable degree of flexibility in traffic engineering, and can prove to be exactly what is needed for the operation of a set of services. However, because of the complex nature of SLAs and their applications, this paper does not consider topology extension other than that of DNS and routing, which are discussed in the next chapter.

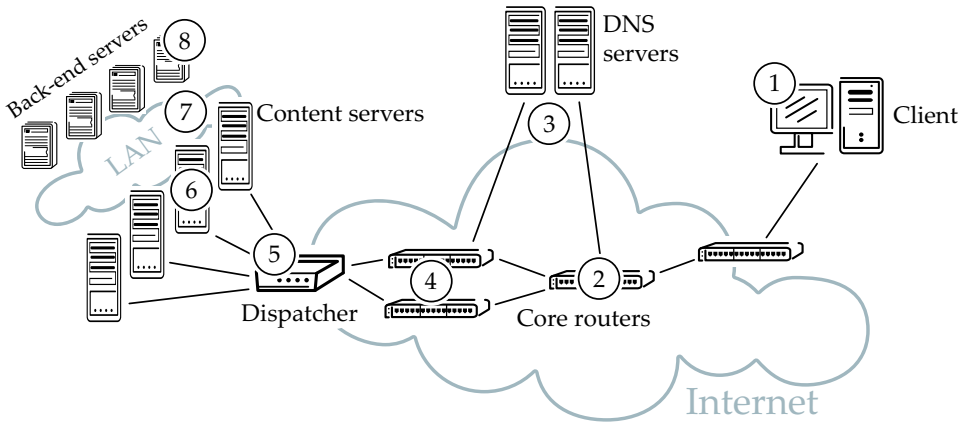


Figure 1.1: **Potential insertion points for load balancing.** *Load balancing can be introduced in several extents of a network path, for example at the core routing level (2) or for a cluster of content servers (5).*

1.3 Brief Note on Terminology

Now that we have introduced the topic, there are two mechanisms that can be identified: *load balancing* and *redundancy*. These are two distinct terms. Load balancing refers to the act of sharing an imposed service load between multiple processing servers. Redundancy in engineering is about allocating backup resources in case of failure in the operating resources. Depending on context, redundancy refers to having unused equipment waiting for failures to occur in the currently running equipment. However, when we talk about load balancing, we imply a form of active redundancy, with the result being a load-balanced service with the welcome side effect of high availability.

1.4 Paper Organisation

The remainder of the paper is organised as follows. Chapter 2 presents an introduction of background material and relevant technologies, with an accompanying survey of previous research in chapter 3. An overview of our experimental hypotheses is found in chapter 4. The experimental setup is detailed in chapter 5. Methodology and procedures are described in chapter 6, with the experimental results following in chapter 7. The paper is rounded off with a discussion and conclusions in chapter 8.

Chapter 2

Overview of Load Balancing

With the previous chapter focusing on why load balancing is desirable, the following pages outline and detail some background material and research in the field, and a selection of available implementation schemes. Interest in load balancing for computer servers and networks dates back at least two decades, with the introduction of various software hacks for the Berkeley Internet Name Domain server (BIND). [7] Since then, research has brought forward a vast array of load balancing techniques, some of which are relevant for this study.

One of the main issues to consider when implementing balancing is that of interoperability. Again, this has different meanings depending on scope. For example, it is generally a good idea to ensure client interoperability, so that new users need not modify their software or hardware to be able to communicate with a service. Specifically, this means that implementors should follow the guidelines and protocols described in the de facto Internet standards, e.g. IPv4 and HTTP version 1.0. Though it might be preferable to develop an entirely new set of IP standards that account for many of the problems faced by load balancing, it is unrealistic to expect wide adoption and replacement of fundamental Internet protocols in the short run. However, there is a middle ground between adherence to standards and developing new ones: Interoperability is ensured by having load balancing operate behind the scenes. Exactly what is done behind the scenes is up to each implementation, as long as the public experiences a consistent "play," i.e. standard protocol behaviour is assured. This freedom has led to some proprietary solutions, often implemented in specific commercial load-balancing hardware or software suites.

2.1 The Load Balancing Big Picture

Proper classification of load balancing methods all depends on the viewpoint or perspective needed for a particular application or problem. In the

case of a strict TCP/IP stack model, protocols like DNS and BGP would be put in the so-called application layer. This is also the case for, say, FTP and SMTP – the ubiquitous Internet protocols for file and mail transfer, respectively. However, there is a subtle yet important difference in wording that may appear confusing: It is commonplace to "load-balance" protocols like DNS, FTP and HTTP. This means that a load-balancing algorithm takes protocol type into account when distributing load, for example by sending all DNS packets to a specific server designed to handle them. On the other hand, there is the wording "load-balancing *using* DNS" or "DNS-based load balancing," which means that DNS itself is used as an integral component of the load-balancing mechanism, i.e. it plays the role of a middle-ware dispatcher because of its ability to provide a mapping between OSI layers. Using the same phrasing, it does not make as much sense to do load-balancing *using* FTP, for example. This distinction is important to maintain when we now look at some of the many models on the topic.

2.1.1 Cause Tree Perspective

To better understand where the different types of load balancing challenges fit into the big picture, we introduce the QoS cause tree in figure 2.1. Cause trees are used to approach a problem from the bottom up, detailing the properties of each individual branch and then linked together to form the complete system. In this example, the notion of QoS (i.e. primarily the response time of a query) is divided into a set of causes that all contribute to the perceived end result. Therefore, QoS is placed at the root of the tree, and all the delays are aggregated from the bottom elements up towards the root. Assuming that on a reasonable scale, as depicted by a layer in the figure, each single cause of uncertainty is independent, we are able to express the total perceived uncertainty in an end-user Quality-of-Service perspective:

$$\Delta t = \sqrt{(\Delta t_{\text{DNS}})^2 + \Delta t_{\text{routing}})^2 + \cdots}$$

For an intricate system like the Internet, this approach has unfortunate shortcomings in that uncertainty in DNS is itself dependent upon the underlying network, with layers upon layers of inter-dependencies. On the other hand, this approach does help in categorisation of system elements.

Each of the causes in the figure is possible to use as a basis for balancing. For example, it could be convenient to balance server load based on client proximity to the server, or based on server CPU load. More conventionally, sites consider a combination of delay-inducing causes when deploying load-balancing equipment.

It would be ideal to accurately calculate weights or probabilities for each cause in the tree. This would be of great help for service providers when designing their network. Of course, since different sites can experi-

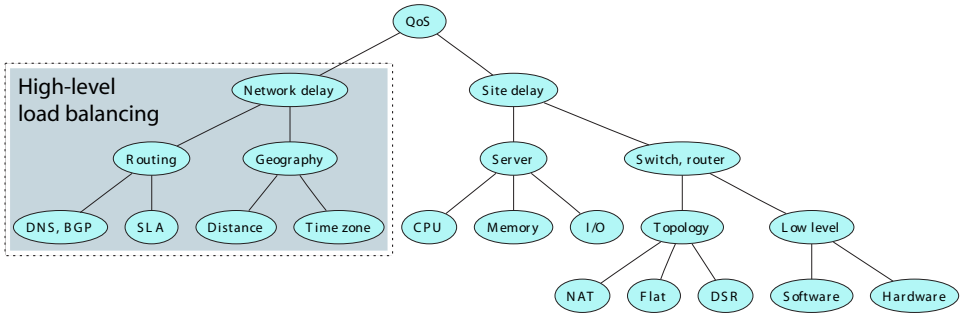


Figure 2.1: **Simplified QoS cause tree.** *Delays are aggregated from the cause branches towards the root, summing up to the total perceived response time. For example, a user browsing the web may observe a qualitative level of QoS aggregated from essentially* all *the entities shown in the tree. The highlighted area in the figure represents the high-level causes for degraded response times, and contains relevant keywords that will be discussed later in this chapter.*

ence widely varying forms of traffic, it makes sense to survey each site by itself. Developing detailed cause surveys like this lets network engineers quantify the most likely causes of site delay, and implement optimisations, much in the spirit of network and traffic engineering.

2.1.1.1 Levels of Scheduling

It is apparent that the difference between lower and higher levels of load balancing is directly related to granularity and scalability, both when it comes to data handling and for manageability. For example, the task scheduler in an operating system organises processes and threads so that they are all load-balanced and receive some amount of CPU time each. On this scale we deal with units of CPU cycles and microsecond deltas, which are not really relevant at all when we talk about server or network load balancing. However, if we zoom out several orders of magnitude and look at network topologies, we can typically operate with units of packets and bytes, and typical deltas of seconds to hours. This is where load balancing becomes more interesting and manageable in system administration. Towards the topmost levels are considerations such as geographic locality, routing mechanisms and service level agreements. These are interesting because they move away from details and allow us to get a handle on the big picture of large-scale network traffic and do load balancing of that very same traffic. This implies that we deal with trends and analysis of large amounts of data over very long time spans, typically hours and days, to months and even years. Again, the parallel to traffic engineering is apparent.

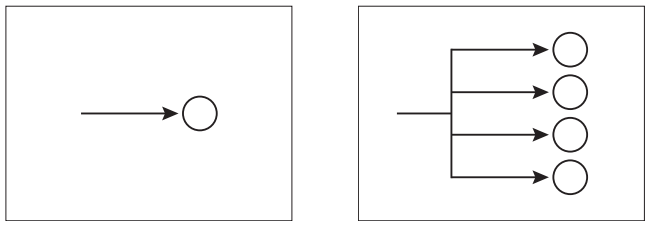
2.1.1.2 High-Level Scheduling

A consequence of taking the high-level approach to scheduling, is that the focus is shifted from detailed topologies to abstracted views of networks and traffic characteristics. This provides operators with an essential overview of the big picture, and allows them to design very powerful sets of tools to handle the networks and traffic, e.g. redirecting parts of the heavy network traffic to a less utilised site in a different part of the world. This is a primary reason for the importance of high-level thinking in network administration.

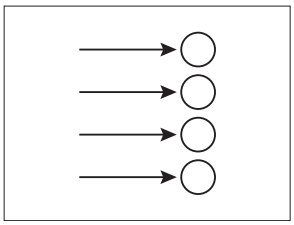
Two of the main tools that are used in high-level load balancing are DNS, the Domain Name System, and BGP, the Border Gateway Protocol. These are typically used to distribute load over geographically separate sites, between a site's multiple network links, or a combination. While DNS in itself was not designed particularly to handle the requirements of load balancing, it provides a limited set of options to facilitate simple balancing schemes. However, together with dynamical back-ends, DNS can be a well suited front-end to robust load balancing mechanisms. The routing protocol BGP, on the other hand, is designed to allow for relatively quick adaption to changes in network topology. In the realm of load balancing, it can be used to balance both incoming (IP anycast) and outgoing (BGP multihoming) connections. Details and inner workings of the protocols will be explained shortly, after introducing some of the basic modes of operation common to all balancers.

2.2 Modes of Operation

The basic idea behind a load balancer is naturally to balance an imposed load amongst a set of available service channels or servers, which in turn brings us to the topic of queues and queueing theory. [8, 9] In a basic queue system, we have a set of servers that process incoming quanta of a workload. That is, units of workload arrive as events and are placed into an incoming queue to await processing. When the unit is scheduled for processing, it is removed from the incoming queue and run through the server process. In such a system we can identify three central variables: The number of servers, the arrival process, and the service process. In standard $A/B/C$ Kendall queueing notation, A and B represent codes to describe the arrival and service processes, respectively, and C is the number of servers. For example, in a $M/M/1$ queue, we have one server and the arrival and service processes are both Markovian, which implies an exponential distribution for inter-arrival/service times. A principal goal of load balancing, then, is to introduce more service channels to increase the total processing rate of the system. In other words, load balancing implies a shift from



(a) $M/M/1$ (b) $M/M/k$ $k = 4$



(c) $(M/M/1)^n$ $n = 4$

Figure 2.2: **Brief overview of basic queueing models.** *These are some of the basic queueing models that are used to describe service systems, for example in computer networks. Figure (a) is a single service system, while figures (b) and (c) deploy multiple server processes. On a side-note, the two latter approaches may indeed have substantially different processing properties.*

$M/M/1$ to $M/M/k$-type of queues, see figure 2.2. The topic of queueing is discussed further in chapter 5.

Load balancers operate in many different ways, but there are similarities between them. For the sake of completeness, this section discusses some of the basic principles behind balancers and some of the low-level implementation details. The main focus remains on high-level load balancing schemes using DNS and multihoming.

2.2.1 Load Balancer Components

Any type of load balancer is built around a few fundamental components or concepts, regardless if it is implemented in hardware or software – see figure 2.3. First of all, a balancing device must be able to receive and send packets through some form of data-forwarding plane. Next, it must have an algorithm deciding how the load should be balanced between available nodes. These two components are basically sufficient for a load balancer to work. Depending on the requirements of network traffic and load distribution, they can have varying degrees of sophistication. A third component found in many balancers is a health-check mechanism that enables the load-balancing algorithm take into account server health, e.g. availability and load, when distributing traffic. Each of these three basic building

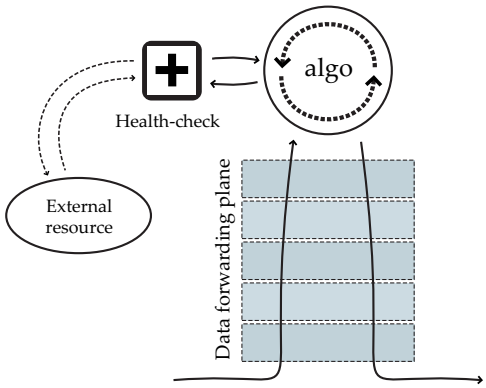


Figure 2.3: **Essential components in a load balancer.** *Requests flow from the network up through the data-forwarding plane. Based on variables fetched from internal and external resources (server health, response time, etc.), the requests are dispatched to an appropriate node, and sent down through the forwarding plane to the network.*

blocks is discussed in more detail below:

1. *Data input/output.* The data forwarding plane receives packets through one or more input ports (physical or logical), decapsulates the payload and sends it up the network stack to the load-balancing mechanism. Similarly, it encapsulates outgoing data coming from the balancer, and forwards it out through the correct port. Depending on sophistication, the forwarding plane has a varying degree of awareness about the TCP/IP stack. High-level balancers – often called *application switches* – could recognise all five layers of the TCP/IP model, and even beyond to application encapsulation.

   With commonplace network links reaching gigabit speeds and above, speed is of the essence for the forwarding plane. This requires very fast hardware and software, often justifying the price of such equipment. Also, depending on the balancing algorithm, there might be a need for large and fast memory caches to be able to handle stateful balancing, e.g. for persistent connections or trend analysis and adaption.

2. *Load-balancing algorithm.* This is the process that determines which packets go to which server or output port, and it is therefore tightly connected to the data-forwarding plane. How elaborate the balancing algorithm is, depends on what goals it should satisfy. Its complexity ranges from trivial to highly advanced. Examples of the former include per-connection *round-robin* scheduling, where connections are assigned to servers in an ordered, circulating fashion. A

Property	Unit
Availability	yes/no
Response time	msecs
Load	percentage
Proximity	msecs/hop-count
Least connected	conn. count
Time zone	time of day
Traffic characteristics	composite
User defined	custom

Table 2.1: **Typical metrics used in load-balancing algorithms.** *This is a selection of common metrics used in algorithms to determine which server should receive requests. Some of these variables are trivial (availability), while others require more sophisticated approaches (traffic characteristics).*

similarly trivial example is a random scheduling approach, where assignment is – not surprisingly – randomised to distribute connections uniformly amongst servers. Advanced algorithms consider volatile variables to more accurately calculate adaptive balancing schemes. These variables must sometimes be obtained from external resources – see below.

3. *(Optional) health-check system.* More advanced schemes make use of additional variables and insight into network and server state when calculating where to send connections. Many of these require the load balancer to query information from the servers or services to determine system health. Table 2.1 lists some of the commonly used variables and their units. Undoubtedly the most conventional and critical property used, is availability. By regularly sending probe packets to servers or even applications, the balancer can take faulty services into account and remove them from the active list. Other variables that help even out the load are:

   - *Response time* – determined by sending any kind of ping packet and calculating the average round-trip time.
   - *Load* – either on CPU, RAM, I/O, or a combination. The Unix load average is regularly used for this purpose.
   - *Proximity* – normally decided based on response time, router hop-count, IP lookup-tables, or a combination. Geographical location of IP address blocks is maintained by the ARIN registry, available to the public.
   - *Least connected* – the balancer keeps state information and can prioritise servers that have the least amount of concurrent connections.

- *Time zone* or *time of day* – these properties allow high-level balancers to redirect traffic to sites that are expected to have lower utilisation, e.g. redirecting day-time queries in Japan to idle sites in Europe.
- *Traffic characteristics* – the choice of algorithm could be influenced by the characteristics demonstrated by the traffic flow. The use and relevance of this issue is discussed towards the end of the chapter.
- *User defined* – these can take any form, and are only limited by the capabilities of the hardware and software available in the device.

Health-checking can be done in many ways. Common examples are link detection, ICMP/TCP/UDP pings, application probes (e.g. check if the web-server responds), SNMP queries, or custom script-based polls. If needed, the health checks can be performed on several levels in a network topology, in a tree-like aggregated structure. This lets back-end servers notify higher-level services and servers of their health, which might reflect a more accurate image of the site integrity. Also, it is worth noting that health-checks normally operate with very aggressive timeout values, so that non-responsive services are quickly taken out of the active list. For high-volume services, these values could lie in the magnitude of milliseconds. Lastly, health-check implementations should consider the difference between periodical and on-demand checking: Polling server status on-demand for each query could be very expensive in terms of total response time, and should be avoided if possible.

2.2.2 Lower Level Load-Balancing

Seeing how a complete load-balancing suite involves a combination of high, mid and low-level components, it is reasonable to include a short description of how a site can deploy a dispatcher-based load-balancing implementation to distribute load over a local area network. A dispatcher is simply another name for a load-balancing device, typically contained within rack-mountable casings. They generally have two or more network interfaces to allow for different network topologies. Depending on price and model, these dispatchers support a varying range of load-balancing options, discussed shortly. Apart from commercial solutions, dispatchers can also be manually built and developed using commodity hardware and software. As these devices can operate on a per-packet basis (not considering session persistence, etc.) and in close proximity to the real servers, they offer fine-grained control over traffic flow.

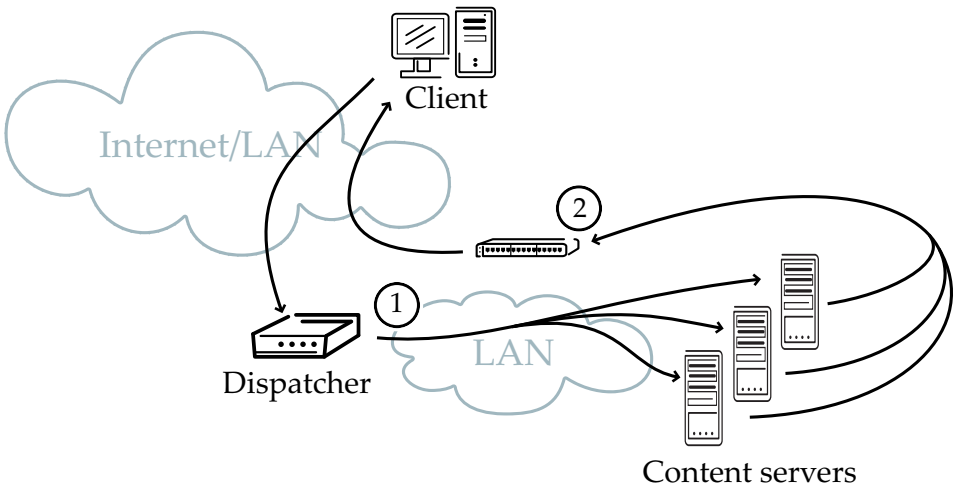


Figure 2.4: **Dispatcher-based topology setup.** *In this simple schematic, client requests reach the dispatcher which in turn decides where to redirect them. The content servers process any incoming requests and send the responses to the client, either back through the dispatcher (1) or through another link (2), off-loading the dispatcher.*

In everyday use, dispatcher-based systems are connected to a LAN, together with a set of back-end servers that are to share the load – see figure 2.4. The exact topology can vary slightly, with different implications for the data forwarding plane in the dispatcher. The challenge is to rewrite the packet headers and send them to the servers, all the while maintaining socket integrity with regards to the client. A TCP socket is identified by source and destination IP address, source and destination port numbers, and the protocol type. If a packet is received with any of these fields mangled, it is not considered part of the socket, and discarded. Some of the workarounds deployed by dispatchers include:

- *Network Address Translation* – The dispatcher assumes the actual server IP, and consequently receives all requests from clients. Both the dispatcher and the real servers are connected on a local network with its own private network block, e.g. 192.168.1.0/24. When the dispatcher receives a client request, it translates the destination address to a real server IP and sends the packet to that server. The server processes the request, and returns the response to the dispatcher, which in turn reverses the address translation and directs the response to the client. A drawback here is that the packets go through the dispatcher on the way out, imposing unnecessary load on it – case (1) in the figure.

- *MAC-address translation* – In this mode, both the dispatcher and all

real servers each have a virtual interface set up with the service IP address. When the dispatcher receives a request, it rewrites the destination MAC address in the link-layer header, and sends the packet onto the real server network. One of the real servers processes the request, and sends the reply out through a *different* link, circumventing the dispatcher – case (2) in the figure.

- *IP-in-IP tunnelling* – Like MAC-address translation, this mode lets the real servers bypass the dispatcher when replying to requests. However, in stead of rewriting the link-layer header, the packet is encapsulated in a simple IP tunnel, and sent to one of the servers, which decapsulates the payload and processes the request. Because addresses are preserved, the response can be sent out through a different link – case (2) in the figure.

These are just some of the most commonly used low-level methods of load balancing. Other techniques are available, but any detailed mention of such falls outside the scope of this paper.

2.2.3 The Domain Name System

The Domain Name System (DNS) is the global name lookup service in the Internet. It is a distributed, hierarchical and redundant database running on thousands of servers worldwide, each responsible for one or more DNS zones. When a client issues a request for a URL, the browser will first try to resolve the hostname in the URL into an IP address, so that it knows where to send the request. This is where it is possible for a DNS server to influence the outcome of a query, effectively directing the client to a desired or lightly loaded site. However, the DNS approach to load balancing is not without challenges, as will be discussed shortly.

If we go more in detail on how DNS works, it is easier to understand the challenges. Consider figure 2.5, showing a step by step diagram of a DNS query for a site. The case is simple: A client in a normal end-user network has written in a URL in the browser, e.g. http://www.example.net/index.html, and presses enter. The following steps assume that the lookup is not previously cached anywhere before receiving the request.

1. First, the browser extracts the hostname www.example.net from the URL and runs it through a gethostbyname() system call.

2. The operating system redirects the request to the local resolver (typically ISP nameserver), asking for the *A-record* for www.example.net – an A-record is a standard hostname-to-IP mapping. The request to the resolver is *recursive*, which enables a flag meaning "I only want the final answer."

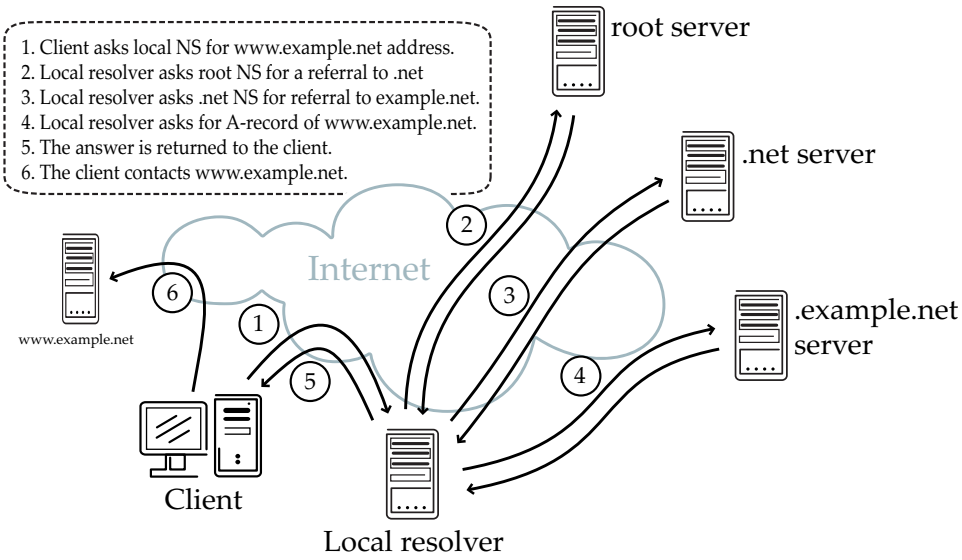


Figure 2.5: **Standard DNS lookup procedure.** *The client sends a recursive request to its configured local resolver (typically within the ISP network), which in turn carries out several iterative requests to fetch the final answer, and then sends it back to the client.*

3. The local resolver performs several *iterative* queries; iterative meaning "direct me to a better match." First it asks one of the thirteen root DNS servers. These know which servers control the top-level domains like com, org and net. The local resolver caches the response.

4. Further, the resolver asks one of the .net DNS servers for further directions to the example.net domain. Again, it caches the response.

5. Then when the iteration process reaches one of the example.net DNS servers, it will know the answer to the query, and reply with an IP address, e.g. 192.0.34.166. Yet again, the resolver caches the response.

6. The response is sent back to the client operating system, which also caches the response.

7. The IP address is returned to the browser, which in turn can contact the server and retrieve the content. In addition to the operating system caching the response, most popular browsers will do so as well.

Note how this rather simple example implies at least five levels of caching, not counting potential intermediate proxy nodes, e.g. http proxies.

To give an insight into how nameservers respond to queries, consider the listing below, showing what a client would receive (per May 2006)

when asking for the IP address of the cnn.com host. This particular output is from the 'dig' application, and shows three main sections: The question section echoes the request information; the answer section shows the matching records (if any); the authority section lists the authoritative nameservers for the domain.

```
1  ;; QUESTION SECTION:
2  ;cnn.com.                      IN      A
3
4  ;; ANSWER SECTION:
5  cnn.com.               300     IN      A       64.236.29.120
6  cnn.com.               300     IN      A       64.236.16.20
7  cnn.com.               300     IN      A       64.236.16.52
8  cnn.com.               300     IN      A       64.236.16.84
9  cnn.com.               300     IN      A       64.236.16.116
10 cnn.com.               300     IN      A       64.236.24.12
11 cnn.com.               300     IN      A       64.236.24.20
12 cnn.com.               300     IN      A       64.236.24.28
13
14 ;; AUTHORITY SECTION:
15 cnn.com.               600     IN      NS      twdns-04.ns.aol.com.
16 cnn.com.               600     IN      NS      twdns-01.ns.aol.com.
17 cnn.com.               600     IN      NS      twdns-02.ns.aol.com.
18 cnn.com.               600     IN      NS      twdns-03.ns.aol.com.
```

There are several interesting properties of the DNS response in the listing. First of all, it is clear that the nameservers return multiple A-records for the cnn.com hostname – this is known as a resource record set (RRset). Also, the addresses appear to be within the same provider network; the provider is verified to be America Online by querying the *whois* database for the IP addresses. Now, even if all addresses are within a very close range, this does not mean that they are geographically close to each other (for reasons which will be discussed in a later section about anycast routing). However, tracing the path to each IP from multiple locations around the world shows that they are most probably located in the same city, and maybe even the same data centre.

The authoritative nameservers for the cnn.com domain have more varied IP addresses (not shown here). Therefore it seems the nameservers are more geographically dispersed – closer inspection reveals that they are hosted in different operator networks. In other words, AOL cooperates with other operators to provide a redundant DNS service, a very common practice.

Going back to the answer section, it is obvious that some form of load-balancing scheme is running, though it is difficult to determine exactly what kind it is. Consecutive queries to one of the main DNS servers show that for each request, the A-record set is returned in a seemingly shuffled order. This could mean that the balancing mechanism relies on the DNS server responding with an address set in a given order, be it random or otherwise. Clients typically traverse the set sequentially, starting from the

top. That is, if the first address on the list does not work, the client tries the next one, and so on. This is however highly implementation-specific.

The numbers in the second column of the response show the time-to-live integer value (TTL) in seconds. This value governs how long the answer is cached in intermediate nodes, e.g. local resolvers. As long as the TTL is 0 or higher, queries will be answered from the cache in stead of being redirected to other servers. In the example, the A-records have a TTL of 5 minutes. Comparably, single host, low-traffic sites may operate with a TTL in the range of hours to days – informational documents recommend a value in the range of minutes when deploying DNS-based load balancing. [7] A low TTL ensures that DNS servers are queried often, and are hence given the possibility to influence the answer over relatively small time intervals. Low TTLs come at the cost of higher frequency of queries, which can add a considerable delay to the total page response time. This issue is discussed later.

In conclusion, we observe that load-balancing using DNS can adopt two basic mechanisms: First, delivering a resource record set in a given order; second, setting the TTL to a relatively low value. Unfortunately, these are by no means reliable. Considering the first point; no Internet standards or authoritative documentation require DNS implementations to preserve the order of resource record sets. Even if it could be considered a rule of thumb in the Internet community to leave any record set ordering intact, there is no reason to assume that all DNS software would follow such recommendations. As for TTL values, multiple levels of caching make it a challenge to predict and control the actual TTL observed by the end user. Caching is a wide topic, and will be discussed to some extent in section 2.2.3.2.

2.2.3.1 Example of Use

Basic DNS-based load balancing does not require any complex configuration to work. The following listing is an example taken from a normal BIND zone file with standard syntax, and it describes the fictional test.lan zone:

```
1  $TTL    60
2  test.lan. IN      SOA     ns.test.lan. hostmaster.test.lan. (
3                                 1         ; Serial
4                            604800         ; Refresh
5                             86400         ; Retry
6                           2419200         ; Expire
7                            604800 )       ; Negative Cache TTL
8  ;
9  test.lan. IN      NS      ns.test.lan.  ; NameServer record.
10 ns        IN      A       10.0.0.1      ; A-record for nameserver.
11 www       IN      A       10.0.0.13
12 www       IN      A       10.0.0.11
13 www       IN      A       10.0.0.14
14 www       IN      A       10.0.0.16
15 www       IN      A       10.0.0.15
16 www       IN      A       10.0.0.12
```

The first line defines the time-to-live value to use for all the records contained within the zonefile. Next, on lines 2 through 7, is the *start of authority* record, which describes behaviour of slave nameservers. The relevant records, however, are the six A-records that make up the RRset for the www.test.lan hostname. As a result, any lookup for the A-record for www.test.lan would return the entire set of addresses to the querying client. This leads us to the question of RRset ordering, i.e. in which order are the records returned to the client.

As previously mentioned, RRset ordering is not governed by any authoritative or recommended standards. Consequently, it is entirely up to the implementation of DNS software how to handle ordering of RRsets. BIND, major version 9, provides three basic methods of ordering, designated *fixed*, *cyclic* and *random*. The desired ordering can be specified in the BIND configuration file:

```
rr-set ordering {
    random;
};
```

The effects of the various ordering directives are further described in chapters 4 and 5.

2.2.3.2 Challenging Caches

Caching in the domain name system is a feature designed primarily to offload nameservers. Clients, iterative resolvers and other intermediate DNS-aware nodes cache DNS answers for the duration specified in the TTL value from the answering server. When kept in the cache, each individual TTL value is decreased by one every second. Inside this period, the caching node will consequently answer any cached request directly without contacting the authoritative servers. If the TTL expires, i.e. reaches a value less than zero, the cached data is cleared, and answers to any consecutive request must be fetched from another server to renew the data and TTL values.

It has already been outlined where the caching mechanisms are operating between a DNS server and an end-user. To clarify, consider figure 2.6 which shows many of the potential caching-points involved in a lookup. Now, if all the caches respect the original TTL value, the originating nameserver should have some control over expiry times and traffic flow and thus keep any inconsistencies at an acceptable level. In real life, however, this is not without challenges:

- If an originating nameserver answers with a randomised record set, there is no guarantee that the intermediate nameserver will shuffle the set for each query during the caching time.

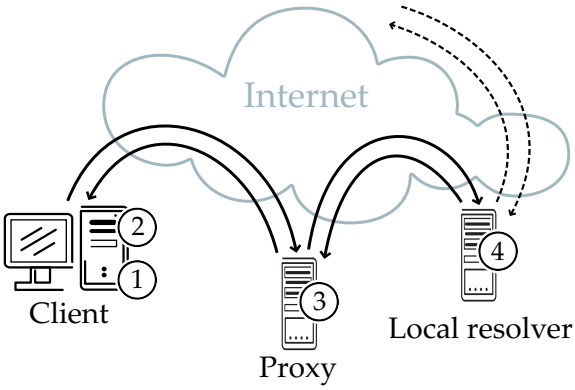


Figure 2.6: **Potential caching-points for DNS lookups.** *After performing a lookup, the DNS answer could be cached in the client's browser (1), operating system (2), zero or more proxies (3), and the client's configured local resolver (4).*

- Mis-configured caches could fail or ignore answers when the TTL is set to zero (meaning the answer should not be cached at all). Alternatively, low TTLs may be replaced by a minimum value set by the DNS software or administrator – possibly in the well-meaning interest of off-loading servers.

- Client software could be unaware of original TTL values. On the client-side, the operating system usually keeps a local DNS cache, comparable to intermediate nameservers. However, many Internet browsers keep an additional cache in memory. This cache is special, because it relies on information returned by operating system calls that may or may not return the TTL together with the answer, which is the case for the ubiquitous gethostbyname() call – it does not return TTL. Therefore, browsers that cache DNS answers use a preset TTL value which does not necessarily reflect the intended value from the originating nameserver.

Evidently, lowered cache times results in more fine-grained control over the load-balancing process. On the other hand, low TTLs will lead to an increased number of lookups. This is generally not a problem from the perspective of the nameserver, since there is little processing involved in answering DNS requests. However, it does add a considerable delay to the total response time when fetching a service object, e.g. a web-page. [10] This specific topic has received attention from researchers over the years, some of which is presented in the next chapter.

2.2.4 Routing-based Load Balancing

The following is a very brief summary of the operation of routing in the Internet: Simply put, routing in the Internet is the process of deciding upon a path through which packets are sent. These paths – or routes – are kept in routing tables at each router, and identify a destination prefix with an interface and meta-information about the route itself: Associated with each route is normally a next-hop address, an administrative distance-to-destination and a composite metric that is used to indicate route preference, usually made up of several values depending on the routing protocol. Therefore, if there exists several paths to a given destination, it is possible to balance the data flow between these paths to distribute load over available resources – the process previously referred to as traffic engineering.

The routing tables are populated with either static or dynamic routes, learned from manual input or from routing protocols, respectively. Dynamic routing uses auto-converging routing protocols to adapt to changes and failures, which is essential to the continuous operation of the Internet. Routing protocols are sorted into two categories: IGP and EGP, internal and external gateway protocols. The term 'gateway' refers to a router situated either inside (internal) or on the border of a network domain (external). Network providers operate routing domains – autonomous systems ASes – where they exchange routes and data internally and with other providers over the domain borders, which in turn build the foundation of practical end-to-end routing between hosts. In this respect, core routing and traffic engineering is restricted to a small set of operators – end-users have little or no influence on the seemingly transparent "Internet cloud."

2.2.4.1 IP Anycast

IP anycast is a fourth member in the IP communication scheme, preceded by unicast (one-to-one), broadcast (one-to-all) and multicast (one-to-many). Anycast is also a one-to-many scheme, but it differs in the sense that only one of the many available nodes receive the packet. In IPv4, this is made possible by announcing the same IP address block over BGP from several different locations, typically spread around the world, depending on use. The result is that requests for an address within the anycast address block are routed to the closest match by the core Internet routers, who decide routes based on the AS path. As a consequence, anycast is not well suited for connections that require any sense of state, since disrupting changes in the core infrastructure can reroute the packets to another node with the same anycast address. However, this is a desirable scenario for single DNS lookups, where there almost always is one request and one response. [11] Currently (May 2006), five of the thirteen root DNS servers are distributed

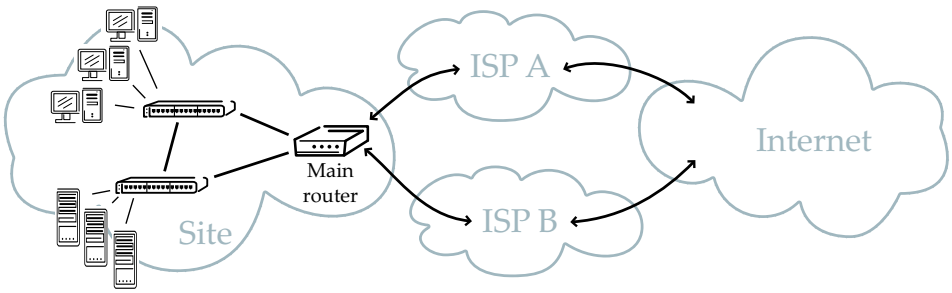


Figure 2.7: **Simplified schema of a multihomed site.** *A site can be connected to two or more ISPs to provide link redundancy and load-balancing facilities for its services. Normally, the site would communicate routing information to its neighbours using a protocol like BGP.*

using anycast. The F-root[1] service consists of 37 servers world-wide, all sharing the same IP address.

Some authors have tried to alleviate the statelessness of anycast communication by introducing minor protocol changes in host TCP/IP stacks. [12, 13] In short, they use anycast addressing in the initial connection handshake while simultaneously exchanging unicast addresses, either through new or existing protocol extensions, e.g. the IP source routing option. Engel et al conclude that, although their implementation requires host network stack modifications, their load balancing results clearly show that anycast is a serious candidate for Internet load distribution and service location. [12]

2.2.4.2 Site Multihoming

For a service provider, it is possible to connect and negotiate service level agreements with several ISPs to add redundancy and additional network resources to the service. One popular way of doing this is through *site multihoming* using BGP, the de facto routing protocol between autonomous systems in the Internet. Consider figure 2.7, where a site is connected to two separate ISPs, enabling redundant links and the possibility of load balancing – such a setup with k links is designated k-multihoming. Assuming the site communicates with the ISPs using BGP, there are two leading ways of implementing multihoming:

- For each link, the site announces separate address spaces. This would imply that traffic going to and from the address blocks would be statically routed over the links where they are announced. In case of link

[1] Currently, the root servers are designated A through M. See http://www.root-servers.org/ for a full list.

failure, the other links would still be available. However, client connections are not migrated over to the available address blocks, as it would not be accepted by the TCP or UDP protocols. A client would need to reconnect, for example by trying another address returned by the authoritative DNS server.

- Normally, a site would announce its address block(s) to all neighbours so that the routing mechanisms can automatically adapt to topology changes and failures. The degree of unavailability in case of link failure, depends on how quickly BGP is able to adapt and converge, but existing connections are maintained as long as they do not time out in the end hosts.

With a multihomed site, it is possible to balance both incoming and outgoing load. Incoming load balancing is typically done in the core ISP networks and is normally outside the authority of the site owners. Conversely, outgoing balancing is achieved by configuring the routing table of the site router, either with a simple route weight or more sophisticated algorithms.

In their paper, Akella et al provide a measurement-based analysis of multihoming. [14] They find that 2-multihoming improves performance by 25% or more for three of the four metropolitan areas considered. Further, they observe that there is little incremental improvement when moving past 4 providers. It is also concluded that the choice of upstream providers is crucial for performance.

2.2.5 Global Server Load Balancing

Somewhat related to load balancing using DNS and the core routing infrastructure, is advanced global server load balancing, GSLB. This relatively recent term surfaced around the turn of the decade and refers to the approach of deploying several sites at strategically favourable locations on a global scale – see figure 2.8. Together with site-local dispatcher balancing, GSLB sustains a very robust technique for service providers that have customers from all around the world.

Up until recently, GSLB has been implemented using traditional DNS-based load-balancing schemes. Indeed, DNS is still used as the main interface for the client when it requests a service address, mainly because of interoperability issues. The difference between advanced GSLB and normal DNS-based balancing lies in the back-end system, which determines what addresses to include in the DNS response to the client. Inner workings of the back-end system vary from vendor to vendor, but can make use of many of the methods already mentioned: BGP, where AS-path info can be propagated to the DNS servers for proximity calculation; NAT and tunnelling, to redirect requests to other sites (similar to server redirect in

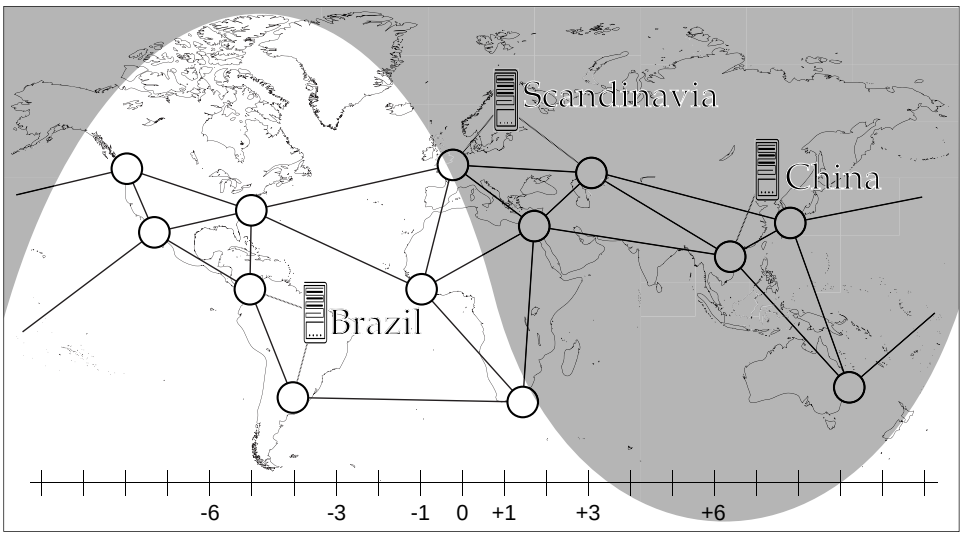


Figure 2.8: **Global server load balancing.** *By deploying sites at several locations in the world, a company can take advantage of routing, proximity, time zones, etc, to distribute load amongst separate sites. Because of the use of existing global infrastructure, this is also a highly redundant solution with the potential to thwart the effects of force majeure.*

dispatcher-based systems); DNS protocol modification, adding functionality like the SRV resource record to define multiple servers for a single domain [15]. More sophisticated approaches are also used, but they fall outside the scope of this paper.

One interesting aspect of GSLB is that of time zone differences. Consider figure 2.8 again. It shows a coarse approximation of the global daytime/nighttime relationship. If, for example, an international service provider has deployed sites in Brazil, Italy and South Korea, it is reasonable to assume that local daytime customer activity is significantly higher than in the night. By exploiting these assumptions, the provider can modify the behaviour of the GSLB implementation to redirect users not solely to the closest server, but to under-utilised servers in other parts of the world. The result could be a much more stable service for customers, regardless of their location.

2.3 Related Technologies

This section is set aside for a short discussion of relevant technologies in the perspective of high-level load balancing in the Internet.

2.3.1 Content Delivery Networks

Balancing load on a global scale is closely related to the recently popularised *content delivery networks*, CDNs. [16] A CDN deals with redirecting the client to an appropriate content node, but also manages back-end content distribution amongst the nodes, which can be done in widely varying ways. A simple approach is to upload all content to each node, but this has obvious scaling issues. Intelligent back-ends could distribute the content in an adaptive fashion, e.g. based on geographical popularity, time of day, etc. The commercial Akamai CDN service deploys more than 12,000 servers in over 1,000 networks worldwide, and uses BGP as an integral part of the back-end system to determine network topologies. [17] Current trends in GSLB and CDN research tend towards real-time delivery of dynamic applications, as opposed to static content.

2.3.2 Multicast

Multicast facilitates one-to-many communication channels, where one data source reaches several receivers at the same time. Receivers notify their local routers of multicast group membership, and are thus subscribed to selected services, e.g. streaming audio. The clear advantage over traditional content serving is that the source need only allocate resources for one single stream, as it will be copied as many times as needed by intermediate routers along the path to the receivers. Following this practice, all clients would receive the exact same content at the approximate same time. This severely limits the applicability of multicast to real-time live media, e.g. streaming audio and video.

2.3.3 HTTP Redirect

HTTP redirects are part of the protocol specification, and are used to notify the client, and any intermediate HTTP caches, of alternate locations for the wanted content. [18] When a client asks for a web object, the server can respond with a message similar to the one below:

```
1 HTTP/1.1 302 Found
2 Cache-Control: private
3 Content-Length: 0
4 Content-Type: text/html
5 Location: http://www.example.net/alternate/location.html
6 Server: Microsoft-IIS/6.0
7 Date: Wed, 15 Mar 2006 11:58:37 GMT
```

The “302 Found” return code is part of the 3XX redirection codes and means that the requested resource has been temporarily moved to a different URI, indicated by the “Location” field on line 5. On line 2, the “Cache-Control” field tells intermediate caches that they should not cache this response – the response message is intended solely for a single user.

Reviewing the expression for independent uncertainties, it is clear that with a HTTP redirection scheme, a client would have to perform two (or more) HTTP requests to retrieve the content:

$$\Delta t = \sqrt{\cdots + 2 \times (\Delta t_{\mathrm{HTTP}})^2 + \cdots}$$

One HTTP request consists of a series of parts. First, a three-way TCP handshake is exchanged between the client and server. Then the client asks for the content and the server returns it. Finally, if disabling persistent connections, the TCP socket is closed. All in all, this procedure could result in at least 10 packet exchanges, 6 from client to server and 4 from server to client. The "click-to-view" delay can be considerable if this is done over a high-latency link. Consequently, it would be preferable to keep HTTP redirects to a bare minimum.

2.4 Traffic Characteristics

In the interest of applying proper traffic engineering practices in the Internet, it is essential to characterise traffic workload – the fundamental component in the third phase of the TE principles. [5] Understanding how traffic flows, lets us build and maintain a stable and scalable network infrastructure, but at the potential price of complexity. Over the years, there has been conducted a wide range of studies on Internet traffic modelling, [19] starting with the assumption that large-scale Internet traffic follows the mathematically simple Poisson process, that is, packet arrivals are independent and inter-arrival times follow the exponential distribution. From the mid-90s, research adopted long-range dependence and self-similarity as key characteristics for more accurate modelling of Internet traffic. Accurately describing Internet traffic characteristics is by no means an easy task, since the net is under continual change, expansion and renewal. Karagiannis et al conclude that traffic models must be regularly re-evaluated to verify their accuracy. [19]

A more in-depth discussion on traffic characteristics can be found in section 3.3, where the Pareto and Gaussian distributions are introduced as viable alternatives to the more complex counterparts.

Chapter 3

Previous Research

Several researchers have explored the limitations and possibilities of load balancing using DNS, and high-volume service design in general. The most relevant of this research is presented below.

3.1 DNS-based Load Balancing

In their very popular article on dynamic load balancing, Cardellini et al present a survey and empirical comparison of a selection of proposed and commercially available concepts of balancing, including constant and adaptive TTL schemes for DNS, dispatcher-based packet rewriting, and server-based mechanisms. [20] They divide the DNS-based approaches into constant and adaptive TTL algorithms, the former sub-categorised into stateless and stateful designs – see table 3.1. The results of their study show that both constant TTL with server and client state information, and the adaptive TTL scheme perform better than stateless round-robin approach. Specifically for their simulation, the adaptive algorithm keeps the probability of server overload under 1, whereas the constant algorithm has more than one overloaded server 20 percent of the time. Comparably, the basic round-robin scheme overloads at least one server more than 70 percent of the time. The simulation considers both an exponential and heavy-tailed distribution to represent client load, and a 5 percent DNS lookup rate per request.

Bryhni et al also investigate and compare a set of load-balancing implementations, with a focus on dispatcher-based systems. [21] The nameserver approach is discussed to some extent, and is included in the experiment in the form of round-robin DNS with TTLs of 1 and 24 hours. Despite the relatively high value, they observe that the approach performs better than expected. For a TTL of one hour, it is shown that connection loads (percentage of connections given to each server) range from 10.7 to 15.8 percent, on a cluster of eight servers. Correspondingly, a TTL of 24 hours yields a more

TTL	State info	Description
Constant	Stateless	Simple round-robin scheduling, with the drawbacks of distribution imbalance and unawareness of server capacity and availability.
Constant	Server	Polling or asynchronous alerts let the DNS server exclude highly utilised or unresponsive servers from the active server list.
Constant	Client	Base the response on client location or observed domain load, known as the *hidden load weight* index.
Constant	Server and client	A combination of server and client-state algorithms.
Adaptive	Multiple	Returns varying TTLs depending on the hidden load weight and heterogeneous server capacity.

Table 3.1: **A selection of DNS load-balancing approaches.** *These are the different approaches to DNS-based balancing considered in the paper by Cardellini et al. [20] The presented TTL algorithms are discussed and compared, showing that their own adaptive TTL design outperforms the DNS-based alternatives.*

uneven load per server, in the range of 8.1 to 18.5 percent. They argue that lower TTL values would increase DNS traffic, which is deemed to be "network overhead" as it does not carry user information. On a sidenote, their trace-driven simulation results show that a dispatcher-based design running a round-robin algorithm yields the best distribution of load and amongst the lowest response times observed for that particular scenario.

In another paper by Cardellini et al, they present a more thorough examination of web-server load balancing using DNS, and introduce HTTP redirection as a potential remedy for the otherwise coarse-grained nature of DNS. [22] They recognise the standard layered approach to load balancing, where a first-level site assignment is carried out using DNS – typically to determine which geographical location to send the request to, based on proximity. Next, a second-level assignment using some type of dispatcher device would assign the request to an appropriate server. However, because of the somewhat granular properties of DNS-based load balancing, an entire site could be overloaded before the first-level assignment algorithms would be able to adapt. With this motivation, they introduce a third-level assignment in the form of individual server HTTP redirection. This means that, in the event of poor high-level load distribution, each individual server is capable of replying with a standard HTTP redirect message, notifying the client of an alternative location of the content. This would introduce an inevitable delay that the authors seek to minimise by deploying *selection* algorithms that limit the use of redirects by selecting only a subset of requests for rescheduling. Further, a *location* policy determines which site a request should be redirected to. In summary, the results from

the study show that any redirection scheme would guarantee a maximum response time below 20 seconds – in this context, response time refers to a full page load, including all embedded objects. This is compared to a pure proximity-based DNS algorithm, which would only guarantee the same response times 80 percent of the time. It is shown that redirecting only a selection of requests reduces instability caused by bursts when redirecting *all* requests: basing the selection on request size and the location on least-loaded cluster, yields a more even load distribution than proximity or circular assignments.

An extensive study of the effectiveness of DNS-based load balancing by Shaikh et al, shows that lowered TTL values must be carefully chosen to balance page responsiveness against excessive latency observed by the client. [10] The authors recognise that, to allow a fine-grained and responsive DNS-based server selection scheme, the TTL should be set to zero or a very low value. On the other hand, this would lead to the client contacting the authoritative nameserver for each service request (for zero TTL), which naturally increases overall latency – up to two orders of magnitude, according to the paper. Also, there is the potential issue of the nameserver becoming a bottleneck, given the increase in DNS requests. One of the representative cases in the analysis shows that 25 percent of the lookups add more than 3 seconds of overhead for regular sites, and more than 650 ms for a set of popular sites. Further, the authors study the impact of embedded objects in web pages, which are often hosted on off-site servers. In these cases, lookup overhead grows to between 3 to 48 percent of the total page loading time.

Shaikh et al continue by pursuing the validity of client-to-nameserver proximity, a technique used in global server load balancing to deliver content from a server close to the client. The results show that the median cluster size of client-nameserver pairs is between 5 and 8. Cluster size refers to the supposed diameter (in router-hops) between a client and its configured nameserver. Also, more than 30 percent of the pairs are in 8-hop clusters. It is concluded that latency measurements to originating nameservers are typically a weak indicator of the actual client-to-server latency. This claim is supported by Bestavros et al [23] and Mao et al [24]. To combat this weakness, Shaikh et al suggest an extension to the DNS protocol. They introduce a new resource record designated *CA* – client address – a very simple record that only holds the client's actual IP address and a TTL of zero.

Other authors also explore the effectiveness of lowered TTL values. Jung et al [25], Teo et al[26], Park et al [27], Carter et al [28]. See the following summary for the most important points presented by these researchers.

3.2 Summary

- Round-robin DNS dispatching performs very poorly compared to alternative methods such as a purely dispatcher-based approach. [20]
- DNS caching for a TTL value of one hour introduces skewed load on a clustered web server. [21]
- Serving content on a proximal basis only (closest match) may increase system impact on response time. [22]
- Without caching, DNS lookup overhead can grow up to two orders of magnitude. Additionally, lookup overhead may grow nearly 50% as the number of embedded objects increases. [10]
- DNS-based server selection based on client proximity is often flawed, since clients typically reside 8 or more hops from their nameservers. [10]
- DNS TTL values must be carefully chosen to balance responsiveness against extra client latency. [10]
- Reducing A-record TTL values to a few hundred seconds has little adverse effect on cache hit rates. [25]
- Sharing a recursive caching nameserver between more than 10–20 clients has little benefit. [25]
- Widespread use of low TTL values (few minutes) should not degrade DNS cache hit-rates. [25]
- When measuring the mean response times for completed lookups (MRTc) for a set of sites, the minimum MRTc is 0.95 seconds and the maximum is 2.31 seconds. [29]
- Most clients and local recursors respect the authoritative DNS TTL values, but some do not; Up to 47% of web clients and 14% of local recursors ignore the auth TTL, setting it up to two hours higher than desired. [30]
- Mean cost of DNS lookups per web object is 7.1 ms, while mean DNS cost per page is 529.7 ms. In other words, DNS overhead adds 12.2% to the entire page loading time, and 5.09% overhead when using parallel and persistent connections. [31]
- Huitema et al observe a very poor performance in DNS, where 29% of standard lookups took more than 2 seconds to complete. Their alternative control sites reported similar results of 24% to 32% of lookups taking longer than 2 seconds. [32]

3.3 Traffic Characteristics

Network traffic is often said to exhibit a certain characteristic. This can include properties such as packet size and packet inter-arrival times on the network layer, or inter-arrival times, duration, and size of sessions carried by protocols on higher layers. [33] These various characteristics are often described using probability distributions that approximate the traffic properties actually observed. Some of the most popular distributions share the feature of being computationally simple, which means that they are easy to work with, while still providing sufficiently accurate results. Some examples are the common Gaussian, Zipf and Pareto distributions.

When it comes to wide-area network traffic, there is no consensus on which model best suits a given characteristic, though a recurring theme is that experimental data shows signs of long-range dependence, self-similarity and bursty processes. The problem, however, is to properly model such behaviour. [34] Long-range dependence or long-tailed distributions refer to statistically significant correlations over a wide range of time scales; a self-similar process is exactly or approximately similar to a part of itself, i.e. its behaviour appears the same on any spatial or temporal scale.

The most reasonable conclusion to draw from previous research on the topic of modelling WAN traffic, is that there is no definite answer to the problem. When it comes to Internet traffic, the complexity of the system, and its inherent habit of continual change and development, forces researchers to constantly re-evaluate the models that at least *try* to reflect the observed features of the network. [19]

3.3.1 Pareto and Gaussian Distributions

In our context, we are mainly interested in the response time observed when requesting application data such as HTTP payload. Although modeling this type of data has been a topic of generally inconclusive debate for several years, a recent paper recognises the use of a generalised Pareto distribution as a suitable model for long-tailed network traffic, but extends the idea by advocating the generalised Pareto mixture distribution (GPMD) – that is, a weighted finite mixture of a Pareto and, say, a Gaussian distribution – as a suitable model that more accurately reflects the diversity in traffic at various time scales, inhomogeneous nodes, data sources and different protocols. [35]

The Pareto distribution is a power law probability distribution that exhibits a typical skewed, long-tailed nature. [36] It was originally developed to model wealth distribution amongst humans, but has later found use in other areas, including computer networking, where it can be adopted to model file size or inter-event time distributions, to name a few. A Pareto distribution is characterised by the shape parameter α and the location pa-

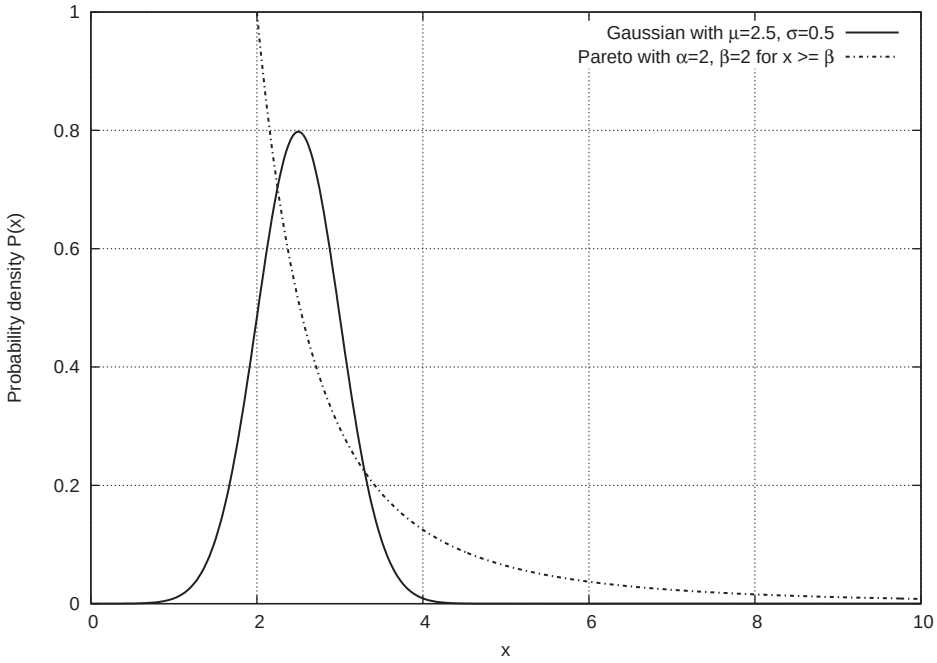


Figure 3.1: **Examples of Pareto and Gaussian distributions.** *These are plots of the respective probability density functions. Both are skewed to the right according to their respective location parameters.*

rameter β. Its probability density function is given by

$$f(x|\alpha,\beta) = \alpha\frac{\beta^\alpha}{x^{\beta+1}} \qquad x \geq \beta$$

The Gaussian or normal distribution is perhaps the most well-known statistical distribution, mainly due to its adaptability: Normality is an inherent feature of many natural phenomena, both biological, physical and social. [37] The Gaussian probability density function is described by the shape parameter σ and the location parameter μ, and is given by

$$f(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \quad e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

An example of both of these distributions is shown in figure 3.1, where the Gaussian distribution is defined by $\mu = 2.5$ and $\sigma = 0.5$. For the Pareto distribution, we have $\alpha = \beta = 2$. Typical for the Gaussian plot is that events are distributed symmetrically around the mean μ, to form the characteristic bell curve. In other words, events are more likely to occur around the mean, and the probability diminishes as we move to either side. The Pareto plot

shows that events are more likely to occur near the location parameter, and follow a power law distribution that forms a long tail for increasing values of x. For the specific Pareto function in the plot, the mean is calculated to

$$\overline{x} = \frac{k\beta}{k-1} \quad \text{for}\, k > 1 = \frac{2 \cdot 2}{2-1} = 4$$

Now, what would happen if we make a compound distribution by mixing these two together to form a Pareto mixture distribution? The Pareto distribution can be used to model the tail of a process. That is, at some point in the main distribution, say at $x = 3$ in the figure, we mix in the Pareto distribution to form an extended tail. The result of such mixing is discussed in the next chapter, along with the introduction of NetEm.

Parts of the theoretical work on GPMD have been implemented in the acclaimed NIST Net network emulation package. [38] The software package offers some pre-built distributions, such as Gaussian, Pareto, and a mixture of the two. It also lets users define their own distribution tables that can be incorporated into the network emulation engine. Research on how NIST Net compares to real-world network traffic has shown that a weighted mixture composed of a 25% Gaussian and 75% Pareto distribution yields a good affinity with (to?) observed data. A more detailed view on the mixture distribution follows shortly in the discussion on NetEm, which has borrowed the GPMD implementation from NIST Net.

3.4 Aim of this Study

With the use of virtual network emulation, we plan to incorporate some of the observed properties of DNS into the network and hosts. Specifically, we add relatively expensive DNS overhead (hundreds of milliseconds) to the emulated wide-area network. With the input of TTL values, we can observe the end result and make qualified observations and conclusions about the impact they have on overall performance. Our hypotheses on the topics are formulated and presented in the next chapter.

Chapter 4

Hypotheses

The following are hypotheses set forth based on the background and previous research surveys in the preceding chapters.

Hypothesis 1

DNS server performance: *Authoritative-only and recursive DNS server performance (requests per second) easily surpasses 4,000 queries per second on a 1.6GHz Intel architecture, using either BIND or the less common PowerDNS software.* The 4,000 qps boundary is derived from the daily peak measurement from the K root server (2,800 qps approx),[1] plus the supposed effect of a heavy flash crowd.

Hypothesis 2

Resource record set ordering: *When returning resource record sets from an authoritative source, the ordering of that set must be considered volatile the instant it reaches a DNS-aware node in the network path, be it intermediate nameserver caches, local recursors or stub resolvers. In other words, the order of RRsets should always be considered volatile, and not a dependable basis for load balancing control.*

Hypothesis 3

BIND scheduling entropy: *Given a RRset of any composition, the entropy of built-in scheduling algorithms in BIND – fixed, cyclic and random – perform as advertised, i.e.* fixed *scheduling does not modify the RRset ordering, and could lead to over-utilisation of a single server;* cyclic *delivers perfectly uniform distributions amongst the available servers; and* random *offers a sufficiently randomised*

[1] See http://k.root-servers.org/stats/linx/index.html for additional information and statistics for the K London mirror.

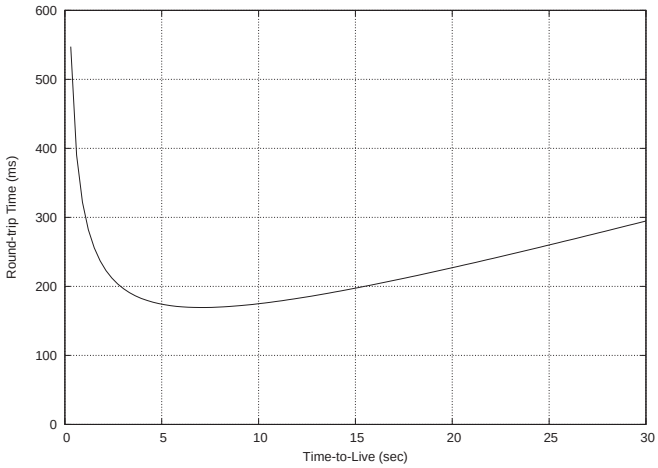


Figure 4.1: **Supposed effect of TTL on RTT.** *In this imaginary plot of TTL vs round-trip time, a low TTL would suggest a high number of relatively time-intensive authoritative DNS lookups, while a higher TTL would mean lower granularity of the balancing mechanism. The TTL and round-trip times are strictly fictitious.*

distribution to avoid over-utilisation of homogeneous servers. BIND is chosen because of its dominant position in a global deployment perspective. [39]

Hypothesis 4

Effects of TTL choice: *The choice of TTL values for a given zone must be carefully considered, seeing how research observations of DNS lookups range from 500 to 2,300 ms.* A TTL in the magnitude of seconds leads to an overweight of expensive DNS lookups. On the other hand, a TTL of minutes to hours has adverse effects on the ability to control the load balancing properties of the system.

The supposed effect of TTL vs scheduler efficiency is shown in figure 4.1. Based on the hypotheses, we can ask if there is an optimal choice of TTL, and in that case, what input values does it depend on?

Chapter 5

Experimental Design

In this chapter we discuss the theory and practice of the experimental setup, which will be used to carry out the measurements later. The setup is designed around the hypotheses, allowing us to test their validity.

5.1 System Modelling

A system – be it something simple like a light switch, or a more esoteric one, such as the entire Internet – can be thought of as a set of independent yet interrelated components that together represent a unified entity. [40] Each component can again be thought of as a stand-alone system with its own sub-components and properties. In the context of computer science, especially in association with local or wide-area networking, any given system is frequently referred to as a *system of systems*. This is particularly obvious when considering the Internet, the largest computer network in the world: Any given node, be it a core infrastructure router, a server of any kind, or an end-user computer, is a complex system in and of itself.

Approaching systems and networks in this manner introduces the concept of *scale*, or *scope*, which can be related to the discussion on levels of load-balancing in chapter 2; i.e. when defining a scale, its magnitude would not only affect the dimensions of space, but also that of time. In other words, when measuring huge systems like wide-area networks in the Internet, it is reasonable to maintain a comparatively large time scale. This greatly simplifies the models involved.

5.1.1 Queues and Delays

To understand a networked system, primarily its behavioural characteristics and observed quirks, we can set out to model it according to an appropriate scale. In the case of a wide-area network, we can use a model partially derived from the QoS cause tree discussed earlier, where the overall

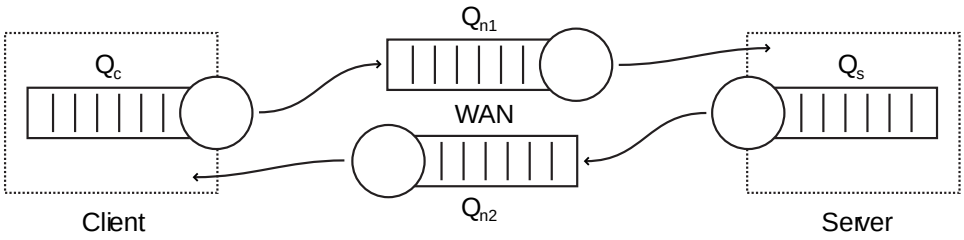


Figure 5.1: **Simplified Wide-Area Network Model.** *This is a very simplified view of wide-area communication between a client and server. The client sends requests through its outgoing queue Q_c, through the virtual network queue Q_{n1} to the server. Responses from the server follow a similar path on the way back.*

quality reflects the perceived round-trip time or delay of a request, measured in milliseconds. We then recognise the two main categories of delay-inducing causes: site and network delay. These are essentially queues with certain properties, such as arrival rate, service rate, buffer size, etc. E.g., in their 2001 paper on performance modelling, van der Mei et al suggest a queue-based system for modelling web servers. [41] Their approach is focused on the phases of connection handling, along with HTTP, I/O and network processing. The model is fairly comprehensive, as it considers details of TCP connection setup, host TCP/IP stack properties, listen queues, HTTP server threads, and behaviour of I/O buffers. Evidently, the model considers a system consisting of one web server (though it can undoubtedly be extended) at a relatively high level of detail.

To maintain applicability and simplicity in our consideration of wide-area networks, however, a less detailed model is needed. A suggested model is shown in figure 5.1. We identify two nodes in the system: a client and a server. These two nodes each have their own outgoing packet queues, Q_c and Q_s respectively. Additionally, the network connection between the nodes can be approximately represented by a bidirectional queue, or rather two separate queues – one for each direction of data flow, designated Q_{n1} and Q_{n2}.

When assessing this model, it is apparent that it does not depict a real system to any significant degree; The server and client have no incoming queues, no processing queues, and the virtual network queues are considerably simplified. This is intentional and serves to keep the system straightforward. However, the experimental implementations use a slightly different approach, where the virtual network queues are unified with the outgoing host queues. With such a setup, it is very easy to emulate the inhomogeneous nature of WAN systems, where total round-trip time is affected by network propagation/proximity, site delay, site and server processing power, etc. Details are discussed further when reviewing the experimental

setups.

5.2 Hardware Considerations

Ideally, an examination of wide-area Internet traffic would make use of actual infrastructure, software and hardware. However, because of the potentially costly and time-consuming work associated with assembling equipment, locations and approvals, this idea was abandoned in favour of a network environment using virtual machines with emulated WAN behaviour.

An alternative approach to WAN studies would be to adopt off-line trace-driven analysis, where existing data dumps are obtained from sources with access to backbone routers. Two institutions that provide this kind of packet traces are CAIDA[1] and NLANR.[2] This kind of analysis is indeed useful for certain applications, but because the traces are anonymized (i.e. all addresses are removed and replaced by non-routable ones, such as 10.0.0.0/8), it is impossible to determine node proximity and route paths of individual packets. Also, the trace is of course observed from one single vantage point, and therefore does not provide any insight into actual end-to-end latency properties.

5.2.1 Virtual Network with Xen

The test environment is implemented in a virtual network using Xen, a virtual machine monitor for the x86 architecture. [42] A total of 10 virtual machines (VMs) run on top of a high-end, consumer-grade server, see figure 5.2. Compared to other virtualisation engines, Xen comes close to native performance, and clearly outperforms alternatives like VMware and User-Mode Linux. This makes Xen a suitable choice for virtual networks, especially because of its performance isolation features, which means that individual VMs operate without being adversely affected by the load of others. When running the experiments, the web servers will sustain a moderate to high load. Performance isolation will ensure that the client machines are not influenced significantly by the servers, and can run the load-generating applications and measurement tools unaffected.

The host machine is a dual AMD Opteron 242 1.6GHz system, with 2GB RAM and a Samsung SP0411N ATA 40GB disk drive. Each virtual machine is equipped with its own disk image that is mapped to a virtual disk drive within the VM, and a limited amount of RAM: A 500 MB disk drive and

[1] CAIDA Anonymized OC48 Data available at http://www.caida.org/analysis/measurement/oc48_data_request.xml

[2] The National Laboratory for Applied Network Research publish extensive traces at http://pma.nlanr.net/

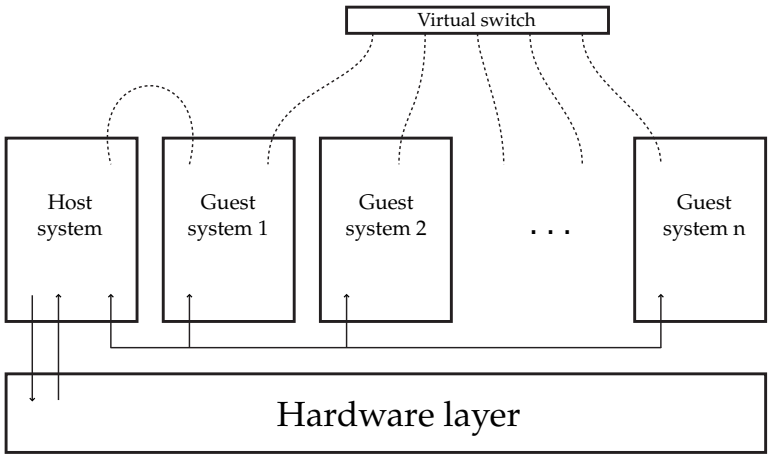


Figure 5.2: **Xen – conceptual design.** *The host system acts as a bridge between the actual hardware and the guest operating systems. Dashed lines represent virtual network connections.*

64 MB RAM for each host, except the DNS server, which has 128 MB RAM. Configuration of the system was carried out using the tool MLN, an easy to use front-end to Xen and User-Mode Linux. [43] The relevant configuration files can be found in appendix A.1.

Performance of the host system's disk drive is reasonably high, yielding a buffered disk read rate of 52.4 ± 0.3 MB/sec (15 samples), reported by the *hdparm* tool. However, when running the same test with 30 samples on a virtual machine, the results were quite different: For each consecutive run, the reported buffered disk read speed steadily increased from 33 MB/sec up to around 132 MB/sec, where it flattened out after 28 iterations. This is clearly indicative of caching behaviour in Xen that appears to undermine the cache flushing mechanism that hdparm uses before each run. Any vigorous analysis of these numbers is irrelevant, however, since disk I/O performance on the virtual machines lies outside the scope of our examination. That said, the numbers show that disk throughput can not be considered a bottleneck in the setup.

CPU and RAM resources are more central, and play the main part in determining VM performance. Since the virtual machines essentially share access to the real hardware, the host system should not only have sufficient CPU time and RAM for all machines; it should also be able to handle a high rate of hardware interrupts, that is, requests to and from the hardware devices to ask for CPU time. An interrupt triggers a context switch in the CPU, and by monitoring the context switch counter on the host system, it is possible to determine whether the sheer rate of switches represents a bottleneck in the experiments.

When running a network performance test, by fetching a 250 MB file over HTTP 15 times, the network throughput was measured to 433.6 ± 1.6 Mbps, or 54.2 ± 0.2 MB/sec. This appears to be similar to the disk performance measured earlier. On the host system, the context switching rate reported by *vmstat* peaked at around 18600 cs/sec, with a corresponding CPU utilisation of 40 percent. For the disk measurement, we found a peak rate of approximately 4400 cs/sec, with a CPU load of 4 percent, but 40 percent waiting time – evidently waiting for disk I/O. When trying a flood ping between two virtual machines, the host system observed a sustained rate of 12000 cs/sec and 15 percent CPU load. Comparably, an idle system generates around 20 cs/sec.

A qualitative conclusion to draw from these somewhat informal measurements, is that the performance of both the host system and virtual machines is far greater than needed for disk and network throughput. The requirements on CPU and RAM resources are somewhat harder to determine and will be tested later. However, it is safe to say that the current system forms a solid basis for the simulations.

5.2.2 Emulating WANs – NetEm and IProute

Originally inspired by the need for a simulation framework for protocol development, NetEm has made its way to the mainstream Linux kernel tree. [44] Built together with the existing mechanisms for QoS and differentiated services available in the kernel, it provides users with a very flexible kernel-level implementation of WAN emulation, including delay, packet loss, duplication and reordering, rate control and non-FIFO queueing. One of the main highlights of NetEm is the protocol independence, which allows the use of unmodified user-space networking software.

NetEm is built into the feature-rich Linux QoS skeleton, which offers a wide array of possibilities for queueing, queue manipulation and packet classification. Options and control over these features are handled by the iproute2 user-space software, which ships with a toolset for controlling all aspects of networking in Linux; everything from link speed and interface addressing, to routing and traffic control. With iproute2, the traffic control options are managed through the *tc* command line program. For example, a command like

```
# tc qdisc add dev eth0 root netem delay 200ms 50ms 25%
```

would set the eth0 qdisc up with a delay of 200 ± 50 ms, with a 25 percent correlation factor, which means that the next random element depends 25 percent on the previous one. If not specified, the elements are generated according to a uniform random distribution. The *qdisc* keyword is short for *queueing discipline*, and is used interchangeably with the word *scheduler*.

A timer limitation in the kernel has an adverse effect on the operation of NetEm. The kernel setting for timer frequency can take three values: 100, 250 and 1000Hz. By default, the 2.6 kernel series use a 100Hz timer. A consequence is that NetEm can only deal with a time granularity of 10 ms, i.e. a packet can only be delayed in steps of 10 ms. Fortunately, setting the frequency to 1000Hz reduces the granularity to 1 ms, which is sufficient for our purpose. The host system and all virtual machines run a 2.6 series kernel with 1000Hz timer frequency.

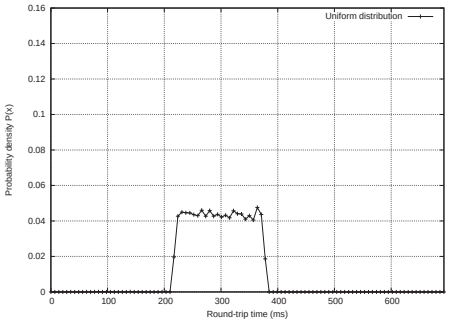
5.2.2.1 Delay Distributions

With NetEm, it is possible to specify the delay distribution it should follow, with parameters for mean and standard deviation. Four different types are bundled with the software package: The default uniform random, and table-based distributions for normal, Pareto and a normal/Pareto mixture. To assess the different possibilities, a small experiment was conducted where a client machine was set up to use a scheduler with 300 ± 80 ms latency. A 25 percent correlation factor was used for the uniform random distribution – the factor has no effect on the table-based distributions. Data was gathered by sending 17700 ping requests to the DNS node. The command lines for the tc program were as follows:

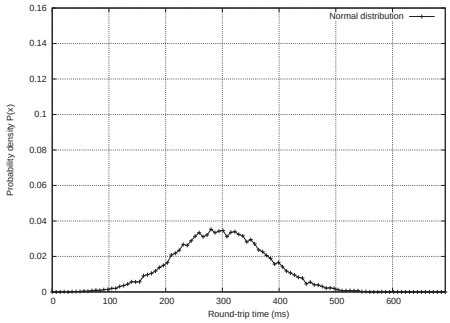
```
(a)# tc qdisc add dev eth0 root netem delay 300ms 80ms 25%
(b)# tc qdisc add dev eth0 root netem delay 300ms 80ms normal
(c)# tc qdisc add dev eth0 root netem delay 300ms 80ms pareto
(d)# tc qdisc add dev eth0 root netem delay 300ms 80ms paretonormal
```

Normalised results from the tests are shown in figure 5.3. Plots (a) through (c) are shaped according to the parameters specified on the command line. The normal/Pareto mixture is shown in plot (d). The shape can be identified as normal distribution up to around $x = 300$ ms, where the Pareto form is incorporated to model the elongated tail. Note how the distribution does not form a *heavy* tail, since it appears to have a well-defined mean around 300 ms.

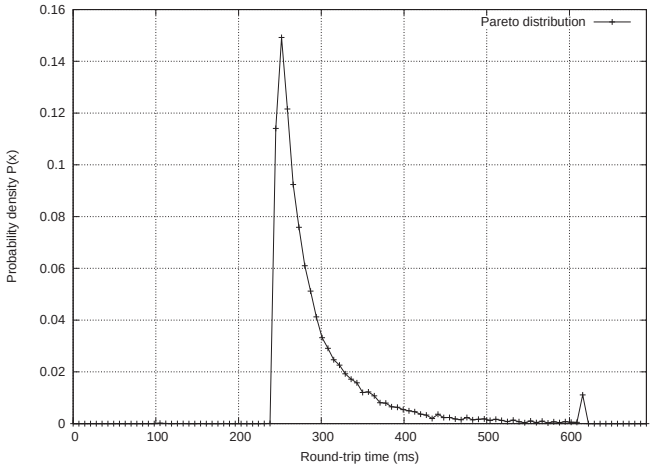
Note the small spike present around $x = 600$ ms in plots (c) and (d). These seem to be caused by the Pareto generator, seeing how there is no similar spike in the Gaussian plot. The spike is located at approximately two times the mean of 300 ms, which led us to believe that ARP requests were to blame; Every now and then, hosts refresh their ARP cache by asking for the physical address of a local IP address. These requests are also delayed through the scheduler, and may cause a delay for ping packets that are sent upon ARP cache expiry. Close examination of the packet traces showed that this was not the case. Even though ARP traffic was present, it did not appear to cause any additional delay, and there was no packet loss in the entire trace.



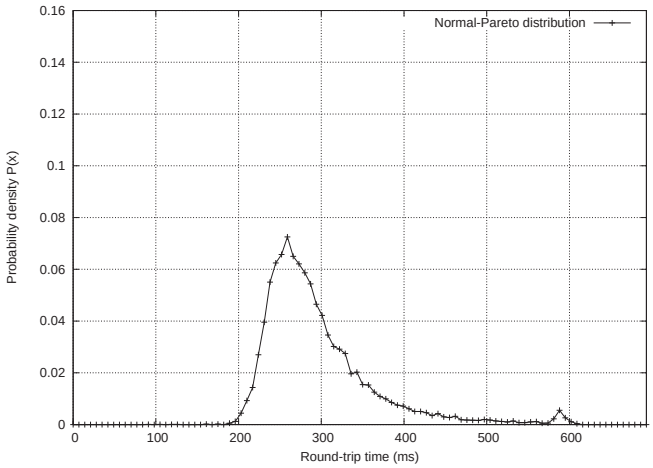
(a) Uniform random distribution



(b) Normal distribution



(c) Pareto distribution



(d) Pareto-Normal distribution

Figure 5.3: **NetEm delay distributions.** *These are the four basic delay distributions offered by the NetEm software. The plots are normalised experimental data, based on approximately 17700 ping round-trip times for each distribution, grouped into 7 ms deltas.*

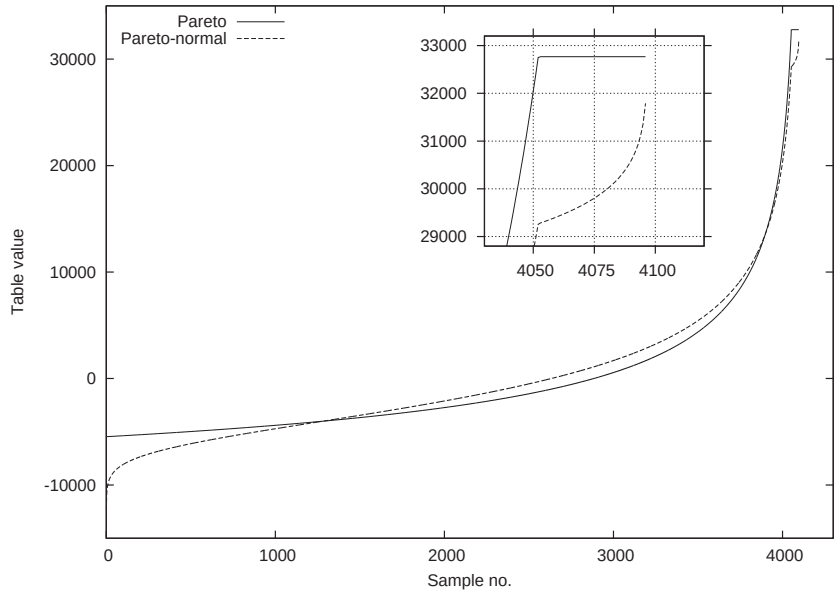


Figure 5.4: **Pareto Generator Phenomenon.** *This is a plot of the inverse cumulative Pareto and Pareto-Normal tables used by NetEm. 4096 samples are generated, and we can see that the Pareto table overflows at sample 4053, flattening out abruptly. This also affects the Pareto-Normal distribution, where we see a marked indentation in the same area.*

Further analysis led to the source of the problem: The NetEm Pareto generator produces signed 16 bit integers within a scaled inverse of the cumulative distribution function – see figure 5.4. Towards the end of the process, the generated numbers tend to overflow the data type, so the author of the software has decided to maintain the max value for all overflows, that is, every number above $2^{15} - 1 = 32,767$ is set to $32,767$. This increases the occurrence and therefore also the probability for this maximum. Despite this slight anomaly, the Pareto generator was left unchanged throughout the experiments.

5.3 Software

To summarise the hardware setup, we now have a logical network set up in a virtual environment. The hosts within the network are interconnected according to the diagram in figure 5.5. This section details the installed software on the machines.

The selection of software is used to help carry out the experiments and measurements. All of these are standard tools available for free, some with minor modifications to better suit the task. Some background information

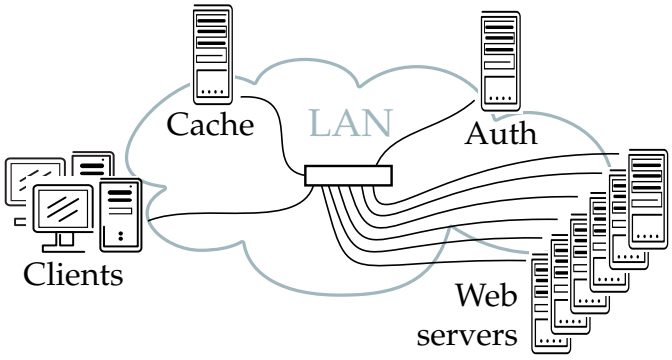


Figure 5.5: **Logical Network Setup.** *This diagram shows the logical network setup as it is running on the host machine. The nodes in the network communicate through a virtual switch.*

and usage notes of each tool are discussed in the following sections.

5.3.1 Operating System Details

All hosts in the network, including the host system, run Ubuntu Linux version 5.10. The kernel running on all systems is a Xen-patched 2.6.16-rc5 release, modified slightly to enable the 1000 Hz timer frequency. Xen version 3.0.2 is used throughout. Some kernel TCP parameters are modified on the clients and web servers. These are accessible through the kernel interface, located in /proc/sys/net/ipv4/:

tcp_fin_timeout: 10 – maximum duration in seconds for a socket to remain in the FIN-WAIT-2 state, which indicates that the local side has initiated shutdown of the socket and is waiting for the other side to acknowledge. Default is 60.

tcp_max_tw_buckets: 20 – maximum number of sockets in the TIME-WAIT state. Excessive sockets are removed. A value of 20 is far lower than the default of 180,000, but was chosen because of the risk of running out of sockets when testing with high query rates.

tcp_max_orphans: 20 – maximum number of sockets that are not connected to any user file handle. This helps remove stale sockets upon restart of the HTTP server, which is done between each measurement session. Default is 4096.

These settings help avoid socket congestion on the hosts. Since the client machine is essentially emulating a large number of clients, it is easily filled up with half-open sockets, which block the creation of new sockets. It also

helps to quickly remove stale sockets on the servers, in case of abrupt termination of traffic generator programs or otherwise severe packet loss.

In addition to these modifications, there is the potential problem of connection tracking table overflow. This table is part of the *netfilter* framework, and maintains a list of active connections. When simulating such a high rate of requests, and not closing the sockets properly, the table continues to increase in size until it hits a maximum defined in *ip_conntrack_max* (default is 8704). In stead of increasing this size and waiting for connections to time out for every session, we chose to disable tracking of packets. Iptables can do this by modifying the *raw* table of netfilter. The raw table is traversed before any other table, and is normally used to include configuration-like changes like these:

```
# iptables -t raw -A PREROUTING -j NOTRACK
# iptables -t raw -A OUTPUT -j NOTRACK
```

5.3.2 DNS Server – BIND

BIND, the Berkeley Internet Name Domain server software, is considered the de facto software for serving both authoritative data and for recursive/-caching lookup services. It is developed by the Internet Systems Consortium, a non-profit corporation aimed at supporting fundamental Internet infrastructure through software development, protocols and operations. [45] In a RIPE presentation dated May 2004, BIND was claimed to occupy around 86 percent of all surveyed DNS servers. [39]

The current version of BIND, major version 9, supports a vast set of configuration parameters, which gives an administrator a high degree of control over most features of the two main parts of the system, i.e. the *named* daemon and the resolver library. Simply put, the daemon manages the server-side of the software, answering queries as defined in the *named* configuration; the resolver library is a client library used by the server process to obtain and exchange information through communication with other DNS servers. The daemon and resolver library are configurable through a vast array of options and settings. However, in the experiments put forth in this paper, BIND is configured almost exclusively with default options; the options that purposefully deviate from the defaults are clearly indicated as such and described below.

When configuring the BIND system for a DNS zone, it is necessary to alter two sets of files: The zone files and the daemon options. A zone file describes the properties and data of a zone, e.g. the test.lan zone. An example configuration for that zone is shown below, with comments explaining the records. This zone is the one that will be used in the experiments, with the TTL value as a variable.

```
$TTL    30
@       IN      SOA     ns.test.lan. hostmaster.test.lan. (
                                1         ; Serial
                           604800         ; Refresh
                            86400         ; Retry
                          2419200         ; Expire
                           604800 )       ; Negative Cache TTL
;
@       IN      NS      ns.test.lan.
@       IN      A       10.0.0.1  ; test.lan
ns      IN      A       10.0.0.1  ; ns.test.lan
www     IN      A       10.0.0.13 ; This is an RRset consisting
www     IN      A       10.0.0.11 ; of six entries, pointing
www     IN      A       10.0.0.14 ; www.test.lan to all of the
www     IN      A       10.0.0.12 ; actual web servers.
www     IN      A       10.0.0.15 ;
www     IN      A       10.0.0.16 ;
```

As shown, the hostname www.test.lan is set to resolve to the list of IP addresses occupied by the web servers, i.e. 10.0.0.11 through 10.0.0.16. Note how the IPs for www.test.lan are not listed sequentially. This is done purposefully to reveal any list sorting done by the servers.

As mentioned before, most of the BIND options are set to default values. Two of them are of special interest for us: *recursion* and *rrset-order*. An example of use is shown below, taken from the named.conf.options file:

```
options {
  recursion no;
  rrset-order {
    order random;
  };
};
```

This disables recursion, which means that the server refuses to answer any query with the *recursion desired* flag set. Recursion should be allowed on the caching server, however. The *order* directive is given the random parameter, which results in the server delivering RRsets in a randomised fashion. Other parameters are *fixed* and *cyclic*.

5.3.3 DNS Server – PowerDNS

An alternative to BIND is PowerDNS, an authoritative-only DNS server with a very flexible back-end. [46] Despite its relatively low deployment of 1.22 percent in 2004 [39], it appears to be an alternative with growth potential and an influential customer base, at least according to the PowerDNS creators.

Our interest in PowerDNS stems from the fact that it is highly flexible, and lets users serve zones from database back-ends in addition to traditional zone files. This is done by populating a database of choice with the zone information, and then specifying the SQL query needed to fetch the data in the configuration file. An example of a query from a populated

database is shown below. Note that it contains the same information as the static zone file (though the SOA content is truncated):

```
# select id,name,type,content,ttl from records;

 id |     name     | type |         content             |  ttl
----+--------------+------+-----------------------------+-------
  1 | test.lan     | SOA  | ns.test.lan hostmaster@tes | 86400
  2 | test.lan     | NS   | ns.test.lan                | 86400
  9 | ns.test.lan  | A    | 10.0.0.1                   | 86400
  3 | www.test.lan | A    | 10.0.0.11                  |    30
  4 | www.test.lan | A    | 10.0.0.12                  |    30
  5 | www.test.lan | A    | 10.0.0.13                  |    30
  6 | www.test.lan | A    | 10.0.0.14                  |    30
  7 | www.test.lan | A    | 10.0.0.15                  |    30
  8 | www.test.lan | A    | 10.0.0.16                  |    30
```

By appending the "ORDER BY random() LIMIT 1" expression to the SQL query, only one random record is returned. One of the experiments concerns the effects of this approach, compared to the BIND way of returning all records in a randomised or cyclic fashion. Note how it is only the A-records for the web server cluster that have been given a low TTL. This should be the case for real-life production use of BIND as well, but the BIND example uses a single TTL on all records to keep the listing simple.

5.3.4 Apache Webserver

Every webserver in the network run identical installations of Apache version 2.0.54-5ubuntu4. They are all configured the same, with most of the default values unchanged. These are the exceptions:

Timeout 10 – a ten second timeout for stale connections. Default is 300.

KeepAlive Off – disable keep-alive connections. These are HTTP 1.1-style connections that lets a client fetch several objects through one connection. Since we want to emulate many clients, this is disabled. Default is on.

ServerName www.test.lan – this forces the server name to www.test.lan instead of the local hostname.

Additionally, all servers were prepared with a web root directory containing a single PHP script. It accepts one GET-parameter "iter" which tells the script how many times to loop through a simple for-loop that does nothing. The execution time is printed to the client upon completion. For example, a client can request the script through the URL http://www.test.lan/index.php?iter=200000. An iteration count of 100,000 corresponds to 0.1 seconds of CPU processing on an idle system. See appendix B for a full script listing.

Name	Description
QueryPerf	Bundled with BIND, this is a tool for testing raw DNS server performance rate, measured in queries per second.
Flood	This tool is part of the Apache project framework, and is designed to allow very flexible configuration of the query generation. It is based on a concept of query farms, and lets users configure dynamic requests.
HTTPerf	Originally a project from HP research, HTTPerf is a powerful tool for web server stress testing.

Table 5.1: **Traffic Generator Tools.** *These are the tools used for testing both DNS and HTTP performance properties throughout the experiments.*

5.3.5 Traffic Generator Tools

Throughout the testing, we mainly use three different traffic generators; one for DNS and two for HTTP. For an overview, see table 5.1. Each tool is discussed in detail in the following paragraphs.

5.3.6 QueryPerf – DNS testing

Along with the BIND software distribution, there are several contributed programs and libraries. Amongst them is *queryperf*, a simple nameserver query performance tool. It is designed to straightforwardly stress-test both authoritative and caching nameservers. To use queryperf, the user supplies a list of name,type pairs, e.g. "www.test.lan, A" and queryperf runs through the list, gathering measurement data. The list is only traversed once, which means that if we want to test with 10,000 queries, the list must contain 10,000 entries of "www.test.lan, A". Version 1.1.1.2.2.5.4.3 is used, dated June 2004.

5.3.7 Flood – HTTP testing

Flood is developed within the Apache Server Project, and is a tool for profile-driven HTTP testing. [47] It is profile-driven in the sense that the user specifies a set of options in a flexible configuration language. After specifying the profile properties, the program is started and generates running statistics to the console, including timestamps with relative times for opening sockets, sending requests, and receiving the reply.

Configuring flood consists of defining a profile with a set of URIs that should be tested; a test methodology that includes socket type to use and URI traversal scheme (round-robin); a *farmer* is set up to generate either a certain number of hits, or a session time-limit within which hits are generated. Finally, a *farm* defines how the farmers are used, i.e. total number of farmers to run, how many to initialise in parallel, and a relative delay for starting subsequent farmers until the total number is reached.

The version 1.0 source code of flood is modified so that for each HTTP query, it runs a hostname lookup. This is in contrast to default behaviour, where IP addresses are cached throughout the session. The reason for this cache override is to try to simulate not one client, but several. Doing a hostname lookup for each requests mimics a crowd of clients within a single domain, where they all have configured the Cache server as their primary resolver.

5.3.8 HTTPerf – HTTP testing

HTTPerf is versatile HTTP benchmarking tool, designed to fit both micro and macro-scale performance testing. [48] Invocation of httperf is done by supplying it with a flexible set of command line parameters, e.g. target server, URI to request, request rate, total number of connections, requests per connection, etc. Since httperf is a HTTP benchmarking tool, it uses the normal mechanism of requesting the address of the host on initialisation only. Hence, it is not directly suited to emulate a large number of independent clients requesting the IP address before connecting to the web server. For this reason, the source code of httperf version 0.8 was modified to do a hostname lookup before each request, along with some minor changes to extend the internal hostname cache. A limitation of the modified code is that the program only fetches the topmost record if an RRset is returned; the other records are not tried, even in the case of failure.

Below is a usage example of httperf. The *hog* option forces the use of a full source port range, which is otherwise limited to ports 1024 through 5000. *client I/N* specifies that this host is the Ith of a set of N clients. When specifying *period*, the argument mean delay (in seconds) between each query; the 'e' indicates that the inter-query time should be exponentially distributed. Both *buffer* options specify TCP socket properties, which can be tuned to balance between client resource use and query rate performance. Finally, the *num-conns* specifies the total number of queries to make, and *num-calls* specifies the number of requests per connection (HTTP 1.1 style).

```
httperf \
  --hog \
  --client=0/1 \
  --server=<server> \
  --port=80 \
  --uri=/index.php \
  --period=e0.05 \
  --send-buffer=4096 \
  --recv-buffer=16384 \
  --num-conns=1000 \
  --num-calls=1
```

5.3.9 Clock Synchronisation – NTP

To obtain correlated results across separate hosts, it is imperative to maintain synchronised clocks. We solve this by running NTP servers on all nodes in the network. The DNS server is connected to the Internet, and queries ntp.ubuntulinux.org for time data. All other virtual machines query the DNS server to set the time. NTP version 4.2.0a+stable-8ubuntu2 is used throughout.

5.3.10 Measuring Load

When running some of the experiments, it is desirable to measure the load on the systems. The standard UNIX load average facility (easily accessible through /proc/uptime on Linux) is too coarse-grained for our purpose, with a minimum 1 minute average reading. Instead, we install and use the *atsar* tool, which allows a reporting interval of minimum one second. It supports reporting of a multitude of variables, but we are primarily interested in CPU usage. By running atsar on all servers, together with synchronised clocks, we can correlate the readings between all servers and observe CPU load over time.

5.3.11 Analysis Tools

Several small scripts and tools are used in the analysis of the gathered measurements. These are scripts to calculate simple means, standard deviations, query times, regular and cumulative distribution frequencies, simple moving averages, etc. See appendix B for full listings.

Chapter 6

Methodology

In this chapter, we lay out the experimental procedures in detail, building upon the tools and practices described previously. The experiments are designed to determine the validity of the hypotheses set forth in chapter 4.

6.1 DNS Server Performance

To establish a common ground for the experiments, it is helpful to determine the approximate rate of service one can expect from typical authoritative and caching nameservers, i.e. queries per time. This also helps verifying whether the DNS service itself is a potential bottleneck in the setup.

The aforementioned tool queryperf is used exclusively in this experiment. It will simply flood a given nameserver list with requests to establish an understanding of how many queries per second the nameservers are able to handle on their own. A text file containing a list of 10,000 queries is constructed on the form below, and is run ten times sequentially for each server:

```
www.test.lan A
www.test.lan A
www.test.lan A
[...]
```

Both the authoritative and caching servers are tested individually. The authoritative server handles the zone described in section 5.3.2, and it will be served by both BIND and PowerDNS. The TTL is set to a relatively large value – five hours – to avoid any potentially time-intensive refresh upon expiry when testing the recursive server performance. Regarding options, both the authoritative and caching server run on defaults, except the auth-only which has the *recursive: no;* option set. Both BIND servers use default values for RRset ordering, so RRsets are returned in a cyclic fashion.

When measuring PowerDNS performance, the TTL is also set to five hours. Additionally, PowerDNS caches responses from the back-end to

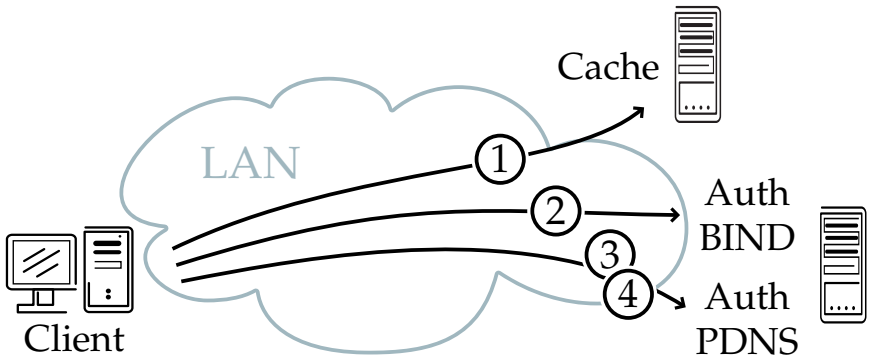


Figure 6.1: **Experiment Setup – DNS Performance.** *The client runs* $10 \times 10,000$ *queries against the caching server (1), and then against the auth-only server, first running BIND (2), then PowerDNS with caching (3), and finally PowerDNS with no back-end caching (4).*

keep latency as low as possible. The configuration directive *cache-ttl* determines how long to cache back-end responses. Note that this value is not associated with the resource record TTL found in the zone data. We shall test both with a back-end cache TTL of 0 and five hours.

In total, we are conducting four measurements, each consisting of ten times 10,000 requests. NetEm will not be used in this setup, as we are testing raw server performance. This helps isolate the variables in the experiment. The setup is outlined in figure 6.1.

6.2 RRset Ordering Implications

For sites that seek to rely on RRset ordering for their load balancing implementation, it is essential that the RRset order given by the authoritative server remains intact all the way to the requesting client. However, as discussed in the background chapter, RRset ordering is never guaranteed to be preserved. Because of its status in global name serving, only BIND will be considered in this experiment.

This experiment is very simple, yet decisive: By testing RRset ordering directives on the authoritative and caching nameserver, it is possible to determine the effect of them from the client's perspective – see figure 6.2 for experiment configuration. This is done by combining the available settings of ordering – *fixed*, *cyclic* and *random* – on both servers, and then observing the result of a query on the client. With three options on two servers we are left with nine cases in total. Since the BIND zone file is set up with IP addresses for the www.test.lan RRset in a shuffled order, we are able to determine if there is any internal sorting done when loading the zone.

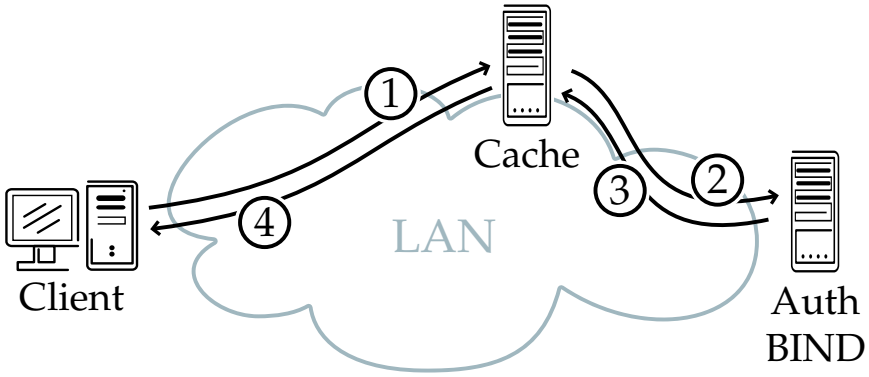


Figure 6.2: **Experiment Setup – RRset Ordering.** *The client runs queries its local cache for the name www.test.lan, which is forwarded to the authoritative server. Depending on the settings for RRset ordering on the two servers, the client observes varying results.*

Because of its widespread use, only BIND is considered in this experiment. It would indeed be interesting to examine a larger array of implementations, but that would not contribute significantly to our study.

6.3 BIND Scheduling Entropy

Despite the arguments against DNS-based load balancing, it is still in wide use. Therefore, as a complement to the research done by Bryhni et al, the following experiment is designed to determine the exact properties of the BIND RRset handling and ordering, which is essentially what determines the balancing of requests amongst a set of servers. In other words, by varying the *rrset-order* parameter and the TTL, and then generating a large amount of HTTP queries, we can check web server access logs to find out how the requests have been distributed amongst the servers.

The authoritative nameserver for the test.lan zone is set up as described in the previous zone configuration, with a relatively high TTL of five hours to avoid any source of error attributed to expensive auth-lookups – a TTL of this magnitude will let the experiment run continuously from start to end without the need for any additional lookup for authoritative data. All BIND options on the authoritative nameserver are set to defaults, i.e. RRset ordering is cyclic. The caching nameserver is configured to use a varied rrset-ordering parameter. Details of the procedure are listed below:

1. Set up the authoritative-only server to return the www.test.lan RRset with a TTL of five hours. RRset ordering is cyclic.

2. Set up the caching server to use an RRset ordering of random.

3. On the client machine, run *dig A www.test.lan* to load the answer into the caching nameserver. Dig is a small DNS client.

4. Run a series of HTTP requests towards the www.test.lan hostname. HTTPerf was used to generate the requests. The command below was run 50 consecutive times to generate a total of 50,000 requests per test:

```
httperf \
  --hog \
  --client=0/1 \
  --server=www.test.lan \
  --port=80 \
  --uri=/index.php \
  --period=e0.05 \
  --send-buffer=4096 \
  --recv-buffer=16384 \
  --num-conns=1000 \
  --num-calls=1
```

5. Record the number of requests received by each server by checking the Apache access logs. Clear the access logs.

6. Modify the caching server to use an RRset ordering of cyclic. Repeat steps 3 through 5.

7. Modify the authoritative server to use a TTL of 3 seconds. Repeat steps 3 through 5.

6.4 Impact of TTL on Response-time

There does not appear to be any single consensus on the choice of TTL values for a set of A records in a zone. This stems from the already controversial use of DNS as a tool for load balancing. In this part of the experiment, we aim to examine some the behavioural mechanisms behind the controversy, and observe how the choice of TTL affects response times.

6.4.1 Static Back-end

In the background chapter, there was a brief discussion on the impact of TTL values on the total response time observed by the client. The rationale is that a low TTL leads to an increased amount of relatively expensive DNS lookups, while a low TTL results in poor balancer granularity, and thus a higher probability of over-utilisation per server. This supposed relationship was presented in the hypotheses chapter, and can be reviewed here in figure 6.3.

To test the influence of TTL on response-time, we use the setup shown in figure 6.4:

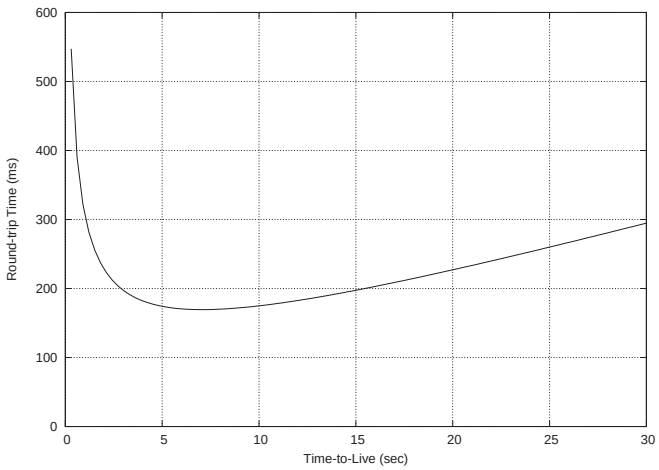


Figure 6.3: **Supposed effect of TTL on RTT.** *In this imaginary plot of TTL vs round-trip time, a low TTL would suggest a high number of relatively time-intensive authoritative DNS lookups, while a higher TTL would mean lower granularity of the balancing mechanism. The TTL and round-trip times are strictly fictitious.*

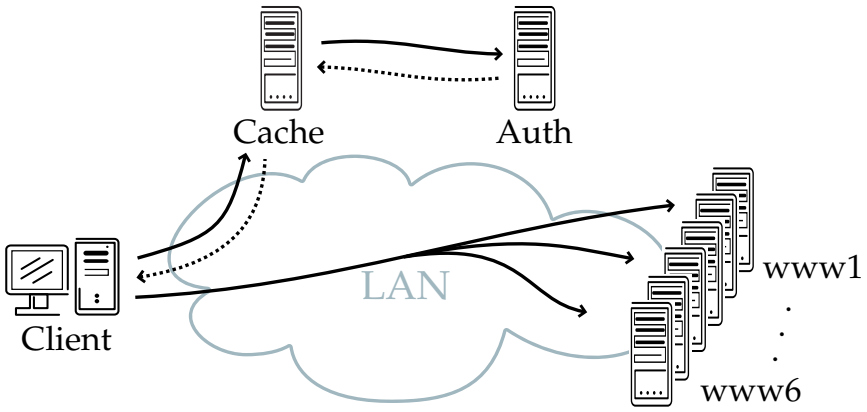


Figure 6.4: **Experiment setup, RTT vs TTL.** *The client runs the flood tool configured to access the www.test.lan hostname. DNS requests are sent to the local cache, which fetches data from the authoritative server. All nodes except the client are set up with a delaying scheduler.*

Node	Avg. delay	St. dev
Auth DNS	300 ms	50 ms
Cache	10 ms	3 ms
wwwX	20 ms	5 ms

Table 6.1: **Flood static – NetEm settings.** *NetEm settings for the various nodes, describing average delay and standard deviation. These values are used when running flood against a static BIND setup.*

- One client generates requests using the modified flood software. The configuration file for flood is listed in appendix A.2.1. It is set up to generate a total of 2,000 requests: Two requests per thread, 1,000 threads per session, start with two threads, and spawn a new one every second. This results in an approximate sustained request rate of 4 req/sec.
- NetEm delay settings for all nodes are described in table 6.1.
- The local caching nameserver is configured to use the default RRset ordering of cyclic.
- The authoritative nameserver also uses the default cyclic RRset ordering, and will be configured to use an initial TTL of 0, which is then increased by one for every 10 runs.

6.4.2 Dynamic Back-end

When exchanging the static back-end provided by BIND with a dynamic one using PowerDNS, we can exercise very flexible control over the DNS responses. PowerDNS is set up to use the PostgreSQL back-end with a simple database schema for the zone, see section 5.3.3. The measurements are conducted from the client, which first sets the TTL value and then runs a flood session. This process is repeated for all the input TTL values, e.g. 0 to 100. To set the TTL remotely, a simple SQL UPDATE command is sent over ssh with a command such as

```
1 ssh 10.0.0.1 "su -c \"psql -c \\\"UPDATE records SET ttl=${TTL}
2 WHERE name='www.test.lan';\\\" -d pdnstest\"
3 postgres"
```

The excessive escaping of double quotes is needed to pass the query correctly to the remote side of the ssh session.

What we would like to examine in this experiment, is the effect of returning only one record from an RRset of active servers. This is actually the current implementation of the www.amazon.com host (May 2006). If

a client asks for the A record of that name, it will be given one IP address only with a TTL of 600 seconds, or 10 minutes. Repeated queries towards one of the authoritative nameservers for that host returns either of two IP addresses. It is unclear what scheduling scheme lies behind the answer, but it may reflect current server load. Since the test was only conducted from one location, it may also be that the two answers observed are geographically close to the test location. Either way, this approach has potentially severe limitations.

If a service provider deploys a solution much like the Amazon example above, it can effectively introduce a single point of failure between clients within a domain and the Amazon webserver. Once a client within a domain has asked for the IP address of www.amazon.com, the recursing server of that domain caches the reply and answers local clients from cache only. One scenario, then, is that a client reaches the main web page of amazon.com, upon which a fault occurs with the server after a few seconds of browsing. The client would be unable to continue communication with the Amazon system, because the client has lost contact to the only known point of reference it has to the server. If the nameservers would return several IP addresses, the client could move on to try the alternatives. Another scenario is a flash crowd effect from one domain. A somewhat far-fetched example could be an ISP announcing a review of a new book at their customer pages. Given enough interested clients, they have the aggregated momentum to overload the single server returned by their cache. With a TTL of 10 minutes, there is plenty of time to flood the service before an alternate IP is returned.

In our experiment, we would like to investigate how this phenomenon behaves. We use the flood tool with a similar request rate as with the static setup, i.e. 4 requests per second – see appendix A.2.2 for the flood configuration file. However, we add relatively CPU-intensive HTTP requests, using an iteration count of 400,000, which would require around 0.4 seconds of processing time. This would clearly overload one single server, as requests are coming in faster than they can be processed. For six servers, however, this represents a low to mild workload. Each flood session is repeated four times.

While conducting the experiments, we gather data dumps on the client and atsar CPU utilisation on the servers. This lets us analyse both lookup times (DNS and HTTP) and server load. Eventually, we are interested in determining if there exists an optimal or equilibrium state where a certain TTL yields desirable outcomes compared to other values. If we observe evidence that this or these sorts of state exists, under what conditions are they valid?

The initial settings for NetEm were identical to those used in the previous BIND setup. In short: DNS server 300 ± 50 ms; cache 10 ± 3 ms; web servers 20 ± 5 ms.

Chapter 7

Experiment Results

This section presents the results obtained from the experiments described in the previous chapter.

7.1 DNS Server Performance

To help identify the bottlenecks in the simulated system, we measured the performance of the DNS servers used in the main experiments. Using the tool queryperf, which floods the given server list with pre-loaded requests, we obtained the results depicted in figure 7.1. As shown, the caching server (BIND) yields $11,123 \pm 58$ requests per second, and the authoritative-only server (BIND) reaches $7,541 \pm 18$ req/sec. When using PowerDNS with a five hour back-end cache, the result is $10,850 \pm 24$ req/sec. On a side note, analysis of an network dump conducted after the actual experiment showed that each query generated 245 bytes (58 bytes request, 187 bytes reply) of network-layer data, i.e. excluding link-layer headers. For a request rate of 10,000 req/sec, this corresponds to a network utilisation of 19.6Mbps or 2.3MB/sec. Round-trip times for the requests were in the magnitude of 0.15 ms/req for the first three cases, and around 1.11 ms/req for the fourth case, where we use PowerDNS with no back-end caching. Conversely, simple pings involve around 0.14 ms/req.

The performance difference between the caching and auth server is clear; the recursor performs nearly 150% better than the auth-only server. This distinction could be attributed to internal processing asymmetry. However, the definite cause remains unclear. The theory of asymmetric code paths follows from the logic that the recursor is nothing more than a pure cache, and thus serves all requests with minimal overhead directly from RAM. In our scenario, this is also the case for the authoritative server; it loads the zone file into RAM on startup, and serves all queries from there. In other words, both servers operate with a very fast RAM back-end, and any differences must therefore be attributed to internal processing charac-

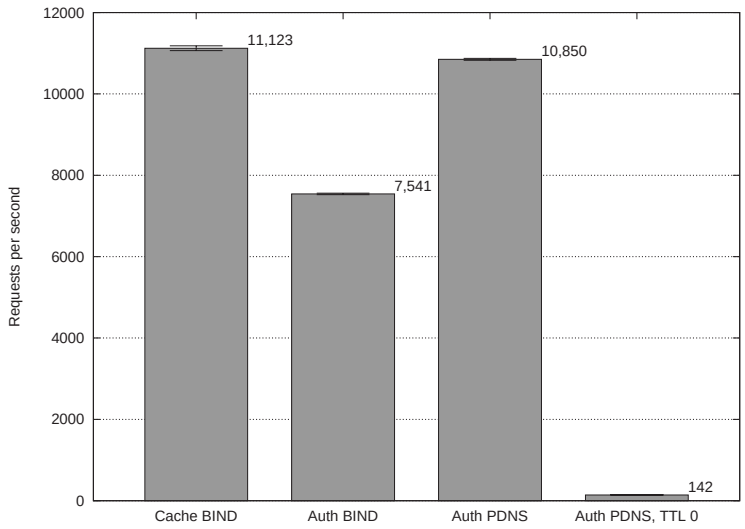


Figure 7.1: **DNS server performance.** *This bar chart shows the difference in performance between the caching/recursive and authoritative-only nameservers, using both BIND and PowerDNS.*

teristics.

Measured auth and recursive server load with atsar during the query runs showed a sustained CPU intensity of approximately 60 percent, 40 of which were used for user-space processing and around 20 for software interrupt control. On the host system, we observed a sustained rate of 15,500 interrupts per second and 7 percent total CPU load. Therefore we can deduce that it is not CPU resources that act as a bottleneck, but rather the network scheduling mechanisms that are overloaded with interrupt management. This is a consequence of the very high rate and small packet sizes used by DNS lookups.

There was some difficulty running the experiment using PowerDNS without back-end caching. Queryperf never completed a single run, and a smaller list of queries was generated and tested instead. Even with a list of five entries, only three queries were completed and two lost. Evidently, there was some problems with the back-end – either the back-end itself (PostgreSQL) or the back-end driver supplied with PowerDNS. An alternative querying mechanism was tried, consisting of a simple shell script that ran ten times 10,000 invocations of *dig* towards the server in serial. This resulted in an average query rate of 142 ± 1 req/sec, markedly lower than the alternatives. However, running the same test towards PowerDNS with five hour caching gave 157 ± 1 req/sec. In other words, the serial approach has severe limitations that do not scale anywhere near parallel execution. Closer examination of the problem did not lead to any conclu-

Case	Auth	Caching	Client-observed result
1.	fixed	fixed	Fixed and sorted.
2.	cyclic	fixed	Fixed and sorted. Cache observes cyclic responses from Auth.
3.	random	fixed	Fixed and sorted. Cache observes random responses from Auth.
4.	fixed	cyclic	Sorted cyclic. Order is reset when TTL expires.
5.	cyclic	cyclic	Sorted cyclic. Order is reset when TTL expires.
6.	random	cyclic	Sorted cyclic. Order is reset when TTL expires.
7.	fixed	random	Random.
8.	cyclic	random	Random.
9.	random	random	Random.

Table 7.1: **RRset ordering implications.** *The configuration setting for RRset ordering has impact on the eventual RRset observed by the client, i.e. the last name-server to handle the set appears to decide the final order.*

sion beyond the observed difficulties with parallel execution. 142 req/sec is more than enough to complete the experiment on PowerDNS with zero back-end caching.

To put the remaining numbers in perspective, consider the RIPE NCC-operated "K" root-server cluster. Per May 2006, the London mirror node, the most popular of the cluster, observes a maximum query rate slightly above 2,800 requests per second in peak hours. That said, root servers are strictly authoritative and refuse to answer requests with the *recursion desired* flag set. In addition, many of them run the NSD DNS software, which is optimised for authoritative-only operation.

7.2 BIND RRset Ordering

The experimental setup with a client, cache and authoritative server, was sufficient to get an overview of how resource record sets are handled on the way from the source to the destination. The nine different cases are shown in table 7.1. From the results, it is apparent that it is the last DNS-aware node in the chain that decides final ordering of the RRset. An immediate effect of this is that the directives governing RRset ordering are never final and should not be used as a basis for weighted load balancing. That is, it would be ill-advised to base a complex load-balancing scheme on the simple balancing mechanisms available in BIND. "Complex" would refer to any load-balancing implementation that relies on more involved techniques than round-robin or random dispatching to an actively maintained server list.

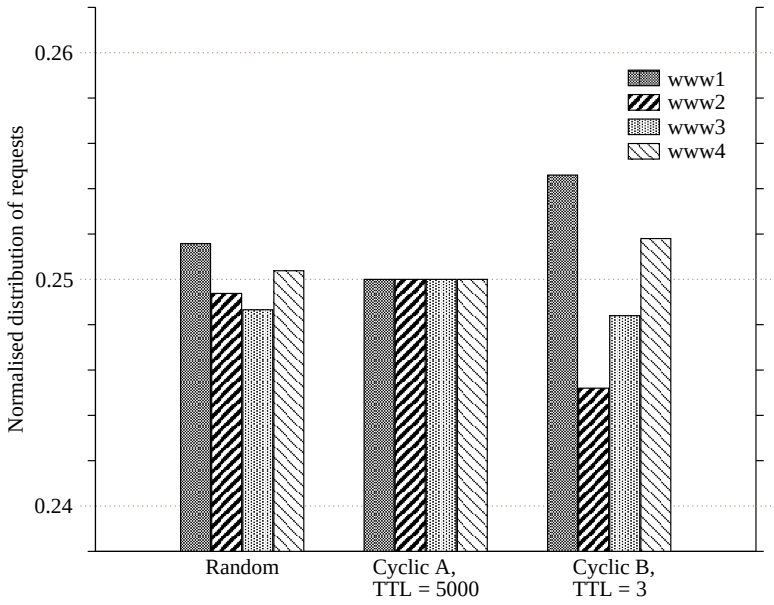


Figure 7.2: **Normalised request distributions.** *This chart shows the normalised distributions of requests amongst the servers 1 through 4, for different RRset ordering schemes; random, cyclic A with TTL 5000, and cyclic B with TTL 3. The Y-axis has been rescaled to better observe the differences in the ordering schemes.*

7.3 BIND Scheduling Entropy

The results of this experiment were partially as expected, but also somewhat surprising. First, the fixed and random scheduling did not exhibit any curiosities; neither did the cyclic scheduling when using a high TTL. The fixed ordering did indeed only return fixed and sorted answers, which result in close to exclusive use of the first server in the RRset. The probability of over-utilisation is strongly dependant upon client behaviour, since it is solely up to the client how to handle the answer from the DNS server; some may try the first entry in the set and then give up, while others may traverse the list, possibly even in shuffled order.

As for the random ordering; close inspection of the BIND source code reveals that the entropy of the random scheduling is based on a built-in ARCFOUR aka RC4 pseudo-random number generator, with the possibility of fall back to the system random pool – `/dev/random` on Linux systems.

When lowering the TTL to 3 seconds for cyclic scheduling, however, the result was somewhat unique. Normalised results for the three tests are shown in figure 7.2 – normalised numerical data can be found in table 7.2. Only four web servers were used in this experiment.

Server	Random	Cyclic A	Cyclic B	Approx.
www1	0.2516	0.2500	0.2546	0.2559
www2	0.2494	0.2500	0.2452	0.2441
www3	0.2487	0.2500	0.2484	0.2480
www4	0.2504	0.2500	0.2518	0.2520

Table 7.2: **Normalised request distributions.** *These numbers show the normalised request distributions per server for three RRset ordering schemes; random, cyclic A with TTL 5000, and cyclic B with TTL 3. For example, when using a cyclic ordering with a high TTL, the distribution is perfectly uniform, giving each server a fourth of the load. The approximations are derived from the synthesised formula described in the text, and show a relatively accurate match with the data observed in cyclic B.*

As shown, the random distribution worked well, giving a near uniform result for each server involved. Since all requests were dispatched from the client before the TTL of 5000 seconds could run out, the result of the cyclic distribution fit perfectly with the expected outcome: All HTTP requests were distributed flawlessly amongst the servers, i.e. 12,500 requests per server. Now, when lowering the TTL to three seconds, the result in the graph indicates that there is some systematic behaviour present. The relationship between www1 and www4 is exactly the same as that between www4 and www3, and www3 and www2. That is, we observe a "staircase distribution" amongst the servers.

It was not easy to determine the cause for this staircase phenomenon observed for cyclic distributions with low TTL values. Eventually, we connected the findings from the previous experiment on RRset ordering with the results in this test. As shown earlier, RRset ordering for cyclic scheduling is reset every time the TTL expires. The effect of this is easily described by an example:

- A client requests the A-record for a hostname, and its local recursor fetches the answer from the authoritative server.
- If we designate the returned addresses 'A' through 'D', the client observes the order to be ABCD.
- Upon the next request to the recursor – before TTL expiry – the order would have cycled one step to the right: DABC. On the next lookup it would be CDAB, then BCDA, etc.
- When the TTL expires, the order is reset to ABCD, no matter where in the cycle the previous response was.

- Because of this, the "popularity" of the servers would be in the order A, D, C, B.

To explain this further, consider the following: In the experiment, HTTPerf reported a rate of 20.9 req/sec. For a TTL of three seconds, this amounts to 62.7 req/TTL. Since we have four servers, we can only distribute 60 of these requests evenly amongst them within one TTL period. What happens to the remaining 2.7 requests? The remaining fraction of requests in this case corresponds to 33.5 ms, and result in a skewed start of the TTL for every TTL duration. This is because after 62 requests, the client would wait 33.5 ms before the TTL expires. Because of the request rate, the client would wait 14.3 ms before making the next request to the server. The result of this rather involved logic, is that the first 60 requests would be evenly distributed amongst the servers, and – because of the cycling – the remaining fraction of requests would be primarily assigned to server A, then D, C, and finally B.

We developed the following formula for determining the distribution of requests between a set of servers when using a BIND-style cyclic scheduling mechanism. Here, W_n is the unnormalised weight (popularity) of a node n. Additionally, we have query rate R, time-to-live T, and the number of servers N.

$$W_n = \frac{\lfloor \frac{RT}{N} \rfloor}{RT - (RT \bmod N)} + \frac{\frac{1}{N}\left(N - \left\{ \begin{array}{ll} n, & n \in [2, N] \\ (N+1), & n = 1 \end{array} \right\} + 1\right)}{RT - (RT \bmod N) + 1}$$

Though the formula looks rather arcane, it represents a relatively simple calculation. The first term of the formula represents the distribution of the whole number of requests, while the second term represents the distribution of the remainder. That is, the first term expresses the probability of serving requests within the evenly divisible number of total requests in the TTL period:

$$\frac{\lfloor \frac{\text{requests per TTL}}{\text{number of servers}} \rfloor}{\text{number of evenly divisible requests}}$$

The second term expresses a similar relationship, but for the remaining fraction of requests after the evenly distributed bulk. The conditional brace is introduced to accept the cyclic behaviour of BIND, where the end result is that server popularity is inversely proportional to the server number, except for the first server.

To verify the analysis, the input parameters from the Cyclic B test were run through the formula. The normalised result, shown under the "approx" column in table 7.2, is very close to the observed data. Deviations are probably due mainly to uncertainty in measurement reporting from the HTTPerf tool.

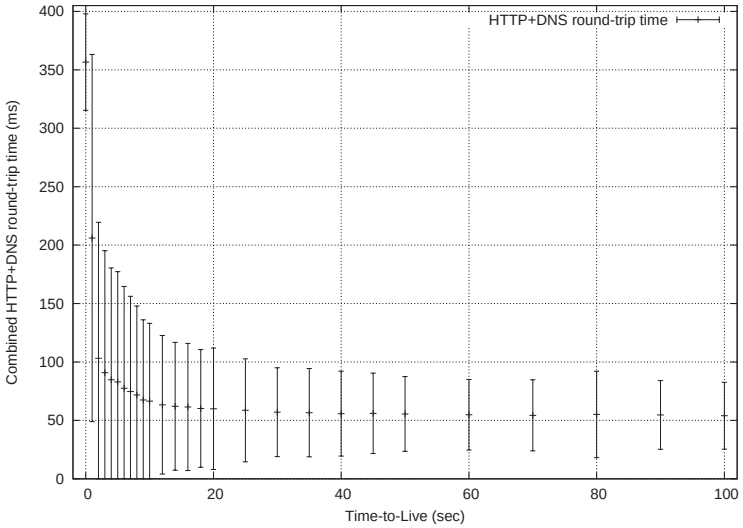


Figure 7.3: **Flood static – Total page load time.** *This plot shows the total page load time, DNS lookup + HTTP request, observed by running the flood tool against the shared web server address. Note the relatively low deviation for TTL 0.*

What are the implications of these results? Considering that they show the effect measured on one client only, it would be misleading to suggest that any distribution skew would be discernible for a larger number of clients within one domain, let alone for many clients in several domains. In those circumstances, assuming a reasonable popularity of the domain served by the authoritative server, clients would observe a near uniform shuffle of the returned RRset. Also, the bar chart in figure 7.2 has been rescaled to better distinguish the otherwise insignificant differences.

7.4 Impact of TTL on Response-time

Below are the results that indicate how the TTL value affects response time in wide-area networks.

7.4.1 Static Back-end

The results from running the flood tool with the initial event farmer resulted in the plot in figure 7.3. Round-trip times in the plot are fetched from the flood output, which therefore include both the DNS lookup and the HTTP request. What we observe is that with TTL 0, the round-trip time is about 350 ms with a relatively low standard deviation compared to the following samples. This is because a TTL value of zero forces the

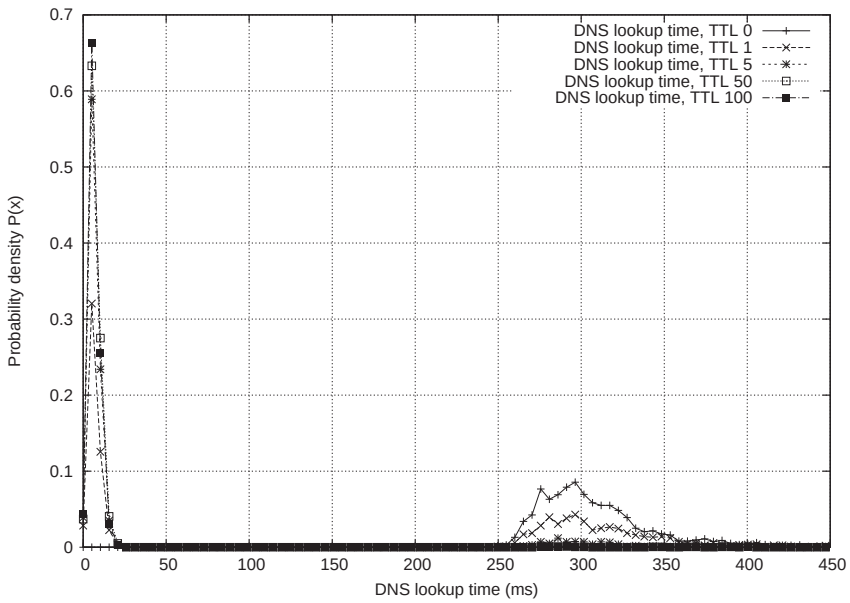


Figure 7.4: **Flood static – DNS lookup times.** *This is a probability density plot for the round-trip times observed in DNS lookups for a varying TTL. We clearly see the two peaks at* $x = 20$ *and* $x = 300$*.*

caching server to fetch the DNS query from the authoritative server every time, i.e. the answer is never cached; the variation we see is attributed to the configured uncertainty in the network emulation engine. As the TTL increases beyond 0, the cache hit-rate grows correspondingly, and uncertainty diminishes.

Because the HTTP queries were very light and need only a fraction of CPU time to complete, there is no adverse effect of server over-utilisation. This is not the case in the following experiment on dynamical back-ends, however.

Further analysis of the behaviour of the measurements is twofold. First we examine the properties of DNS lookups as part of the total page loading time. The plot in figure 7.4 shows the relationship between lookup time and TTL. It is clear that all requests are either bound to a short lookup time, i.e. when querying the cache; or a longer lookup time when TTL has expired and the authoritative server must be contacted. See figure 7.5 for a more detailed view. This is a clear bimodal behaviour, where the TTL is an input parameter that determines the skew towards a low or high cache hit-rate.

With a TTL of 0, we see that there are no lookups in the range of 0 to 30 ms. This is expected, since a zero TTL disables caching. All of the DNS lookups for TTL 0 reside in the range of 200 to 500 ms, clearly shown in the

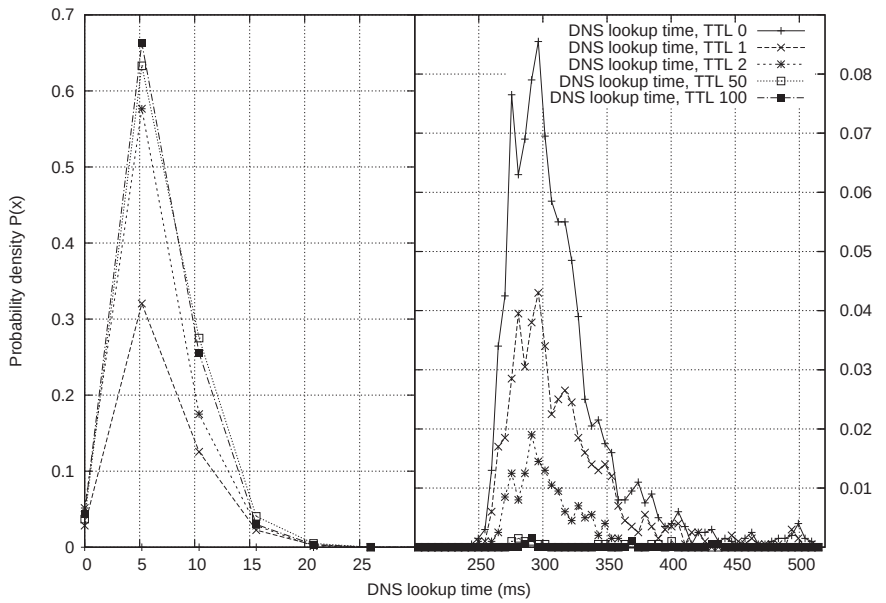


Figure 7.5: **Flood static – DNS lookup times (scaled).** *This figure is a scaled version of the previous plot. Scaling is individual for both peaks, to better observe their properties. Note the characteristic dip around $x = 280$ for TTL 0, 1 and 2; its cause remains unclear.*

plots. As we increase the TTL, the number of low-latency lookups grows, while the expensive authoritative lookups diminish. This happens very quickly as the TTL increases, with a near exponential proportional.

A somewhat strange phenomenon can be observed in the detailed plot. At approximately $x = 280$ ms, there is a marked dip in the number of occurrences. The reason for this dip remains unclear. Since it cannot be identified in the generated NetEm distributions, it might be a valid assumption that the network emulation is not the cause for the anomaly.

As a secondary part of the data analysis, the HTTP round-trip times appear to follow a distribution that bears direct resemblance to the Pareto-Normal distribution in NetEm. The plot is shown in figure 7.6. This is expected, as they are intimately related. We can also observe the small spike in the tail of the distribution, which is attributed to the flaw in the Pareto-Normal generator engine – see section 5.2.2.1.

7.4.2 Dynamic Back-end

When exchanging the static BIND back-end with a highly flexible and dynamic PowerDNS setup, an administrator is given a very powerful tool that can aid in manageability and performance.

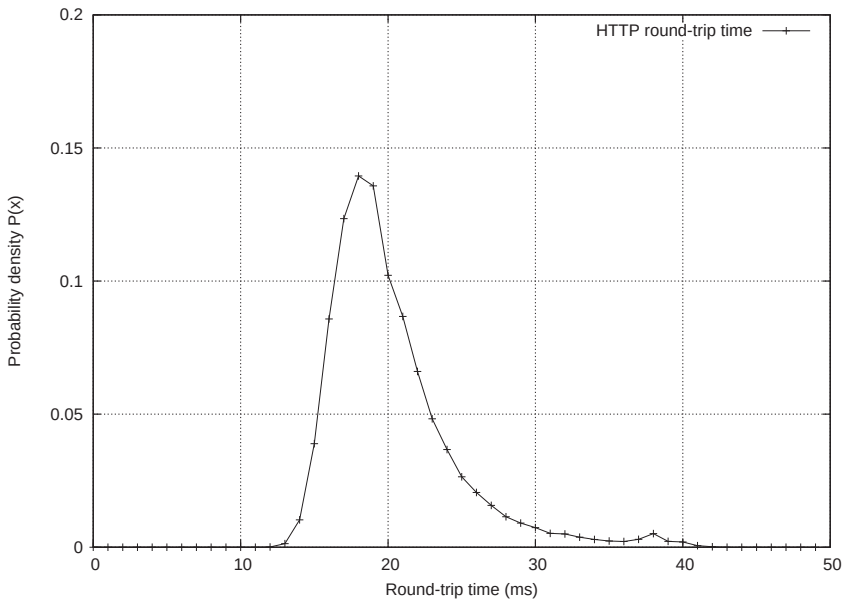


Figure 7.6: **Flood static – HTTP round-trip times.** *This is a probability density plot for the 54,000 HTTP requests. The Pareto-Normal distribution is easily recognisable. Also note the small spike around* $x = 38$*.*

In our first trial, we set up the network configuration as described in the methodology section. This setup was intended to cause overload of a single server, and the results we present here indeed show that the request rate together with load per request was more than enough to cause problems. When analysing the network dump files gathered from the client, we observed a rise in both round-trip time and deviation as the TTL was increased. The plot in figure 7.7 is derived from the network dumps and shows this relationship. What we observe is a steady and almost perfectly linear increase in HTTP response time corresponding to the input TTL value.

For TTL 0, we observe a relatively low response time with comparably low standard deviation. Since we use a zero TTL, all DNS lookups are directed to the authoritative source, which returns a random entry from the set of six addresses, i.e. a one in six chance for returning any given address. Since we get a new address for each HTTP request, the scheduling granularity is very fine; all servers participate equally in answering the heavy requests.

Incidentally, the measurements for TTL 1 are almost exactly the same as for TTL 0. A cause for this could be that the request rate from the client is less than one per second, which would result in a zero cache hit-rate. This is not the case, however: The flood configuration is set up to generate

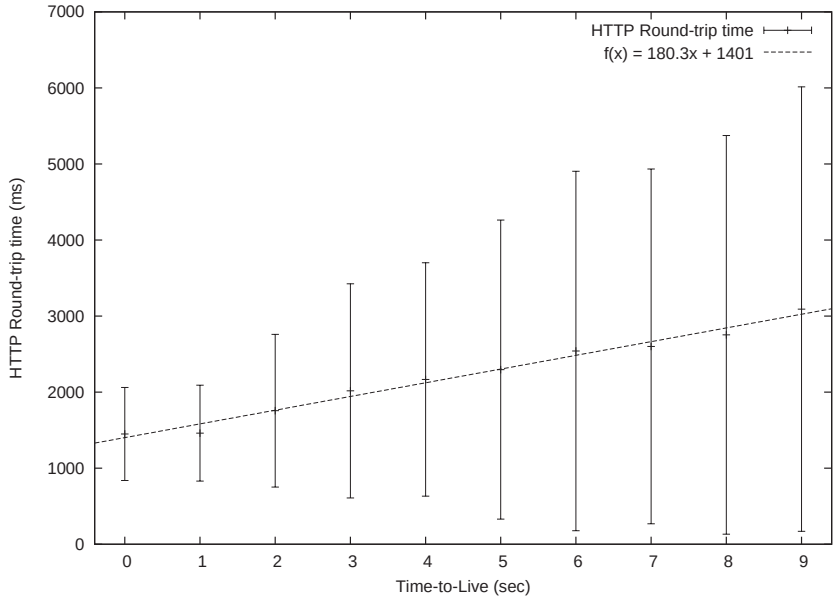


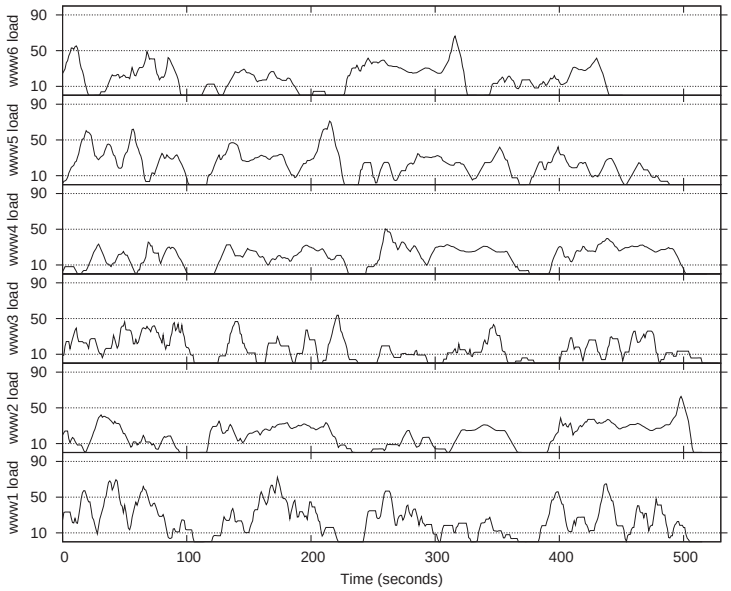
Figure 7.7: **Flood dynamic – HTTP response time.** *This plot shows how HTTP response time behaves when the client only receives one server address at any given time. It leads to server overload as the TTL increases.*

approximately four requests per second. Further analysis would be needed to decide the cause of this anomaly, but it is not conducted here.

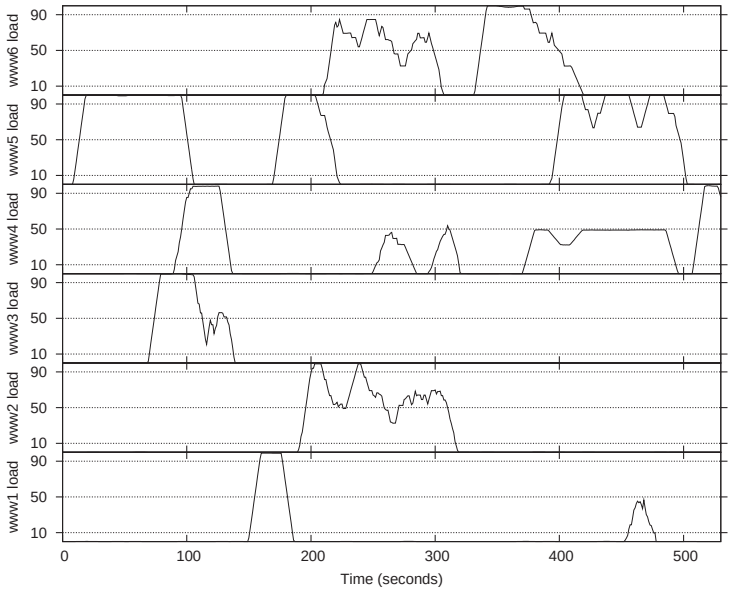
Now, as the TTL increases linearly, so does the response time *and* the uncertainty. This indicates thrashing behaviour on the servers. That is, the server processing queues build up quickly, and the servers also spend more and more resources on managing processes rather than on the processing itself. Simply put, each server is temporarily overloaded for a duration directly related to the TTL value. We can easily observe this by analysing the load imposed on the servers for the different TTL scenarios.

Consider figure 7.8, where we see a time series plot of the load for each server for TTL 0 (a) and 20 (b). The measurements were gathered at a resolution of one second, but were smoothed out using a simple moving average with a window of ten seconds. For a zero TTL, it is apparent that no server reaches 100% CPU utilisation. The load is rather unstable, but is generally kept well below 50%. On a side note, the plot for TTL 0 clearly shows the four different flood sessions, starting around 0, 110, 220 and 360 seconds, respectively.

When we increase the TTL to 20, the outcome is markedly different. There are several long periods of a 100% sustained load and some irregular patterns where the incoming request flow has stopped, and the server is struggling with its already long queue of requests. Evidently, the probabil-



(a) Server load with TTL 0



(b) Server load with TTL 20

Figure 7.8: **Web server utilisation with varying TTL.** *The figures show the observed load on all six web servers during the testing, for TTLs 0 and 20.*

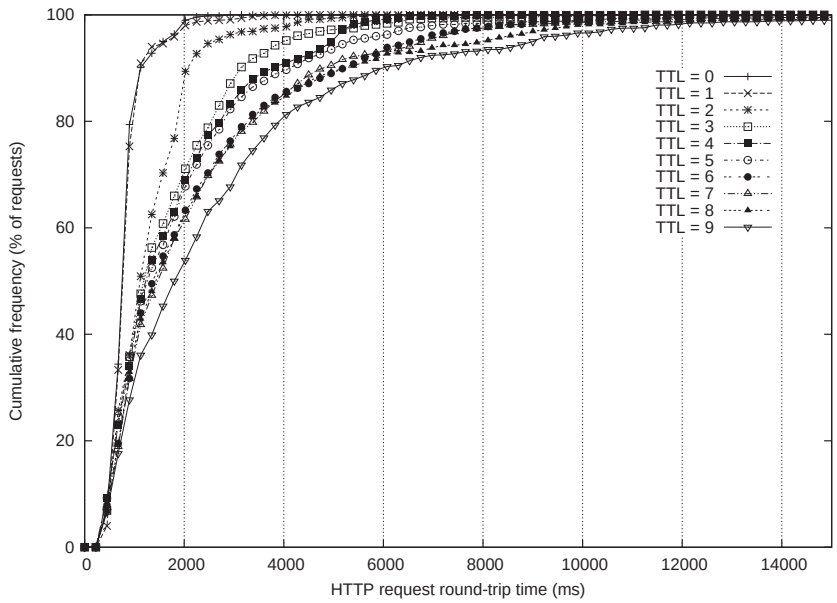


Figure 7.9: **Flood dynamic – Cumulative HTTP response time.** *This graph shows the cumulative frequency distribution of response times for HTTP queries. A higher TTL leads to a higher degree of delayed responses, because of higher possibility of over-utilisation.*

ity of overutilisation increases proportionally with the TTL in this case. A service operator would be interested in keeping the probability of overload as low as possible, and could analyse data flows to determine a suitable TTL to achieve this goal.

Another interesting point to look at is the HTTP response times as plotted in figure 7.9, using the cumulative frequency distribution. Using this approach, we can observe that for TTL 0, most requests – nearly 90% – are within the range of 0 to 1,000 ms per request. Conversely for TTL 9, around 60% of requests are within a range of 0 to 2,000 ms/req. Similarly, a TTL of 9 seconds also results in approximately 90% of requests lie within the 0 to 6,000 ms/req range. Results of this kind is what forms the basis of service level agreements, where providers guarantee some levels of service for customers; e.g. a level can define minimum delays with a given probability.

7.4.3 Dynamic Back-end Equilibrium

With the interest of further exploring the properties of the dynamic back-end DNS server, we started looking for a compromise between the two results previously discussed, i.e. the case set forth in hypothesis 4. The reasoning was that a low TTL would lead to many expensive DNS lookups,

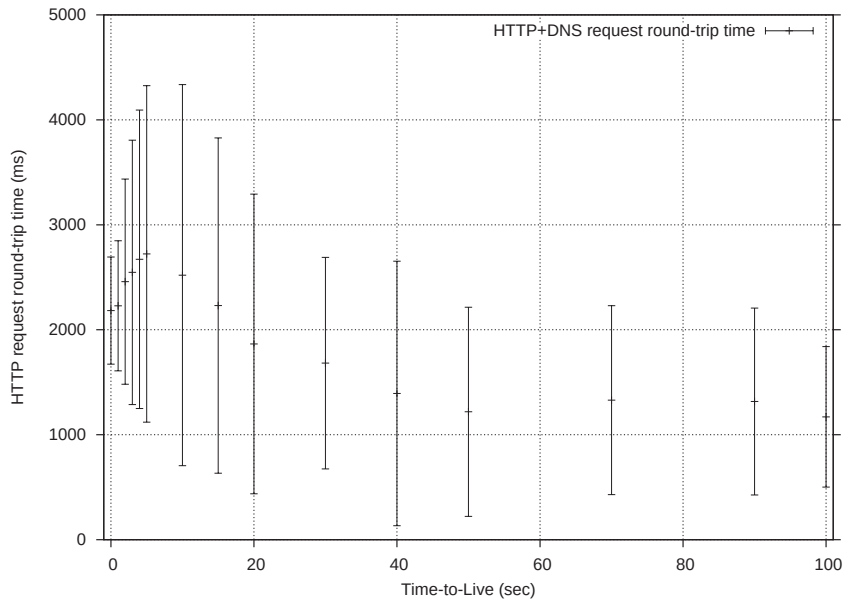


Figure 7.10: **Flood dynamic – TTL equilibrium.** *Plot of total page loading time (DNS + HTTP) for varying TTL. We observe an initial increase followed by a decrease in response times, caused by the combination of TTL and request generator characteristics.*

while a high TTL would lead to over-utilisation of servers.

What we did was to run the flood tool several times, tweaking the values for iteration-count and request rate. To find the desired equilibrium took longer time than expected because the results never seemed to stabilise as intended; either the response time was proportional with the TTL, or it was inversely proportional – like in the previous experiments. Finally it was realised that with the experimental setup and methodology used, the result would apparently never turn out as initially hypothesised. The closest we got to an equilibrium was when using an iteration count of 900,000, one request per thread, 100 threads, start-count 1 and spawn one thread per second. In other words, an approximate request rate of 1 req/sec and a processing rate close to 0.92 seconds. These settings suggest that the servers should be able to cope with the load, but the result shown in figure 7.10 was a bit surprising.

What we see in the figure is a plot of total DNS + HTTP lookup time for a varying TTL. The plot is derived from data gathered from the flood traffic generator. NetEm is set up with a very expensive DNS lookup, $1,000 \pm 100$ milliseconds, while the cached lookups are still 10 ± 3 ms.

The somewhat counter-intuitive result is an artifact of the chosen scenario, where there is almost an equilibrium between the cost of a DNS

lookup and the cost of a HTTP request. For TTL 0, the client performs an expensive auth DNS lookup for every request, and thus the result is found to be around 2,000 ms, with a relatively small standard deviation. As the TTL increases, some DNS lookups are served from the cache while others are fetched from the auth server. Therefore the deviation is very high up to TTL 15, where it starts to decrease.

As for the peculiar elevation with a max around TTL 5, this is an artifact of the traffic generator: The very first generated requests are sent out with similar timing intervals. This behaviour is not mirrored by the processing system (DNS and web servers), and is instead causing additional delays (up to six seconds for TTL 5). Consider the scenario: The first request spawns an auth request, which takes a very long time; the answer is returned to the client almost simultaneously with the next request being generated. This second request will hit the cache and be returned almost instantly. As a result, both these two first HTTP requests are sent to the web server address nearly simultaneously, where they are processed in parallel and cause build-up in the processing queue.

In the figure, we can see that after the first bulk of requests, the generator and processing systems slide into a mutual rhythm for packet generation and processing, with the result being a smoother and lower total round-trip time. The somewhat abrupt decrease in standard deviation for TTL is caused by the length of the flood session; i.e. the TTL is longer than the test itself, so there is only one authoritative lookup in the very beginning. With the strong overweight of cached lookups, the observed deviation is comparably small.

Chapter 8

Conclusions and Discussion

Throughout this paper, we have examined the use of the Domain Name System as a mechanism for load balancing. Insight into available literature on the topic has shown that DNS lookups are an often neglected, yet substantial part of total page response time – not only because of network traversal delay, but because of the recursive nature of queries. The use of caching offers a partial remedy to this challenge of mitigating the frequency of time-intensive authoritative lookups, but introduces problems of its own. Caching time for a given resource record is governed by the time-to-live parameter set at the authoritative nameserver for that record. A TTL of a few seconds suggests that auth lookups are frequent, and the cache hit-rate is low. A TTL of hours to days would mean a considerably higher cache hit-rate, but forfeits the ability of the authoritative nameserver to influence the information within the much longer period until TTL expiry.

Further, we have investigated aspects of popular DNS implementations and their practice of answering of requests, both for caching and authoritative modes. Based on a varying set of input parameters, our goal has been to determine the degree of uncertainty in meeting QoS demands, especially that of round-trip times.

8.1 Review of Hypotheses

Further we shall review our hypotheses in more detail and inquire into their validity.

8.1.1 DNS Performance

In view of the suggested hypothesis, that both an auth-only and caching server are able to handle in excess of 4,000 queries per second, our results both support and partially refute this claim. We see that, for a fully

functional system, both the BIND and PowerDNS implementations perform well beyond the suggested lower boundary. On the other hand, the PowerDNS setup without back-end caching is far below the same boundary. More specific hypotheses and experiments are needed to figure out the cause of this behaviour, though these fall outside the scope of the thesis.

For a mid-range server like the one used in our experiments, an authoritative service rate between 7,500 and 11,000 requests per second is quite enough to manage a legitimate load, even for flash crowds. The issue of security should not be ignored, however. Attacks in the form of denial of service are known to happen, and should be accounted for when designing a highly available system. While it is unlikely that a single source can launch an attack and successfully deny access to one or more DNS servers, a properly coordinated distributed attack has far greater impact. For example, if an adversary were to be capable of generating a sustained rate of around 50,000 DNS requests per second, it is unlikely that a normal server would be able to cope. Also, such rates would congest a 100Mbps network link, potentially adding another bottleneck.

8.1.2 RRset Ordering

The findings in this experiment support the initial hypothesis that RRset ordering must always be considered volatile. In our results we considered the BIND implementation and its various options for handling RRsets, i.e. fixed, random and cyclic scheduling. Because the initial setup of the RRset was purposefully shuffled, we were able to determine that the list is indeed sorted before answering a client, except when using random scheduling – not surprisingly.

The results also concluded that it is the last DNS-aware node in the chain that determines the ultimate ordering of the RRset – not counting the client, which can also alter the ordering independently. This confirms the volatile nature of handling zone data once it has left the authoritative source.

Our experiment considered only the BIND implementation because of its dominant position in deployment. It is the opinion of this author that further studies to determine the extent of this behaviour need not be pursued, be it in other implementations or the Internet in general. The reasoning behind this claim is based on the conclusion that RRset ordering is always volatile. Therefore it is never advisable to base an elaborate load balancing scheme on the assumption that the authoritative answer is conserved. That is, it is mistake to map server preference – be it based on load, proximity or otherwise – onto the order of the RRset, with the belief that clients most likely try the topmost entry in the list first.

8.1.3 Scheduling Entropy

Results from the experiment on scheduling entropy confirmed the expected behaviour of BIND, except from the phenomenon observed for cyclic scheduling with a low TTL. If the original hypothesis had been a little more well thought through, it would perhaps be apparent that a caching server never maintains the state of a cyclic scheduler upon TTL expiry. That is, it does not remember where it left off in the cycle when the TTL expires. The "staircase distribution" was a striking observation, and proved an interesting challenge to reproduce mathematically.

8.1.4 Effect of TTL

By far the most intriguing experiment was on the effect of TTL values on round-trip times. Our initial hypothesis can be considered vague at best. This was intentional, as we did not have material to build more specific presumptions on. As the experiments were carried out, however, a multitude of questions arose that would need some clarification.

It is clear that the TTL value and request rates are closely related when observing the round-trip time in WANs. The choice of TTL determines the cache hit-rate for a given request rate. Consequently, the TTL should be chosen based on an analysis of existing or expected site traffic. The end result could be either a static TTL that suits the current traffic trends to a sufficient degree, or an adaptive TTL scheme that alters the TTL based on a continuous re-evaluation of traffic characteristics. Such values can also be incorporated into service level agreements.

When analysing the results from the dynamic back-end experiment, the limitations of the Amazon approach were obvious. Returning only one address leads to a temporary single point of failure and an increased probability of individual server overutilisation. If a service provider has a list of ten active servers or sites, it would be a much better idea to return a smaller *subset* of this list rather than a single entry. This eliminates the single point of failure and helps mitigate the probability of server overload.

8.2 On DNS as a Balancing Mechanism

DNS is a ubiquitous and integral part of the Internet today, and will most probably retain this role for years to come. As a protocol it has served well for over 20 years, but is it the right tool for load balancing? Clearly it is not. On the other hand we must consider the wide deployment of DNS, which renders it nearly the only viable *interface* for load balancing on a global scale (IPv4 anycast is another). However, when coupled with a solid support of a dynamical database back-end (based on a database with state information gathered from the active servers, maybe proximity information from

ARIN's IP-to-country mappings, time zone info, etc.), DNS can indeed be a very powerful tool in the global balancing challenge.

8.3 Future Work

There are several interesting aspects of the work on QoS and global load balancing, that could benefit from further studies. An extension of the analysis in this paper would be to approach the topic from a trace-driven point of view, as opposed to simulating a WAN environment. Trace-driven research bases its measurements on real-life situations by inserting traffic monitors at strategic locations in a network, e.g. at an ISP main router or an Internet exchange point. This could also help in the development of a sort of deployment plan or recipe for global services, to help determine numbers and values for servers, cache times, etc.

We would however like to encourage spending time on exploring robust solutions and innovations that can solve the challenges we have discussed, rather than devoting more focus on problems in the current situation. This is not to say that quantifying problems in existing infrastructure is insignificant – it is not. Our argument is merely based on the already plentiful literature available on the limitations of current standards.

Some novel ideas have been introduced over the years, and are discussed in short below.

8.3.1 Alternative Protocols

Replacing or renewing the DNS protocol is a common proposal in discussions about its inferior qualities, but it is far from trivial to carry through. Not only does it require clever development; deployment is maybe a greater challenge. A parallel to this situation is the needed, yet ever-postponed global adoption of IPv6, which has been standardised for nearly ten years.

Some extensions to the existing DNS protocol have been suggested, most noticeably the SRV and NAPTR/DDDS supplements. From the RFC describing the SRV record: "The SRV RR allows administrators to use several servers for a single domain, to move services from host to host with little fuss, and to designate some hosts as primary servers for a service and others as backups." [15] The Naming Authority Pointer (NAPTR) together with the Dynamic Delegation Discovery System (DDDS) offers a flexible set of tools for answering lookups, where entries can be matched using regular expressions, and are given weights to indicate preference. An interesting project would be to measure and quantify the abilities of these protocols when it comes to scheduling granularity and flexibility.

The logic behind either of the SRV or DDDS approaches does not advertise speediness, as they only address the *flexibility* of answers, e.g. being

able to do fine-grained control over the load balancing mechanism. The challenge of improving DNS also encompasses the QoS demands from end-users, that is, keeping the response time low. This suggests a shift towards development of advanced routing and caching schemes. Routing is an immense topic in itself and is left off here before going in any detail.

8.3.2 Proactive Caching

DNS caching mechanisms are passive, or reactive, by definition, that is, an answer is fetched and cached only as a reaction upon a request. As we have seen, this can lead to increased page loading times. To mitigate the expensive repeated lookups upon TTL expiry, it is possible to introduce a proactive caching scheme, where the caching nameserver (or another resolver) actively refreshes the cached data just before it expires. Assuming that the caching nameserver is very close to the client, this can help eliminate the repeat lookups that are common to low TTL times. A trivial proactive caching scheme could for example refresh the cached data for $n \times TTL$ periods if the data has not been asked for. The n could be very large, given the small space needed for a DNS answer. Large amounts of RAM and data compression can increase the caching capacity. If the average size of a compressed answer is 150 bytes, a server with 2GiB RAM would be able to store well over 14 million answers.

Some authors have explored the effectiveness of proactive caching in nameservers, and have found it to be effective for certain conditions, i.e. subsequent lookups occurring within a given multiplier of the TTL. [49] Further work is needed to properly quantify and evaluate these prototypes, which could qualify them for standardisation. Again, an obvious barrier is the question of deployment.

Proactive caching can also be implemented in client applications. Imagine a web browser that, upon loading a page, performs parallel DNS lookups for all the links it observes. This has the potential of lowering the waiting time for subsequent clicks, but with the possible side effect of overloading servers. Development and evaluation of a prototype for this type of active caching would be a very interesting project for future work. Deployment for this type of backwards-compatible client application is markedly more straightforward than replacing universal software infrastructure.

Bibliography

[1] Miniwatts Marketing Group. Internet world stats. Technical report, Internet World Stats, 2005.

[2] Google Inc. Financial data, first quarter, 2006. http://investor.google.com/fin_data.html [Online; accessed 9-May-2006].

[3] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into Web server design. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):1–16, 2000.

[4] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for Traffic Engineering Over MPLS. RFC 2702 (Informational), September 1999.

[5] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and Principles of Internet Traffic Engineering. RFC 3272 (Informational), May 2002.

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

[7] T. Brisco. DNS Support for Load Balancing. RFC 1794 (Informational), April 1995.

[8] Wikipedia. Queueing theory — wikipedia, the free encyclopedia, 2006. [Online; accessed 29-April-2006].

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

[10] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the effectiveness of DNS-based server selection. In *Proc. of IEEE INFOCOM 2001*, Anchorage, AK 2001.

[11] Sandeep Sarat, Vasileios Pappas, and Andreas Terzis. On the use of anycast in DNS. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 394–395, New York, NY, USA, 2005. ACM Press.

[12] R. Engel, V. Peris, E. Basturk, V. Peris, and D. Saha. Using IP anycast for load distribution and server location. In *3rd IEEE Globecom Global Internet Mini-Conference*, November 1998.

[13] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. RFC 1546 (Informational), November 1993.

[14] Aditya Akella, Bruce Maggs, Srinivasan Seshan, Anees Shaikh, and Ramesh Sitaraman. A measurement-based analysis of multihoming. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–364, New York, NY, USA, 2003. ACM Press.

[15] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782 (Proposed Standard), February 2000.

[16] A. Vakali and G. Pallis. Content delivery networks: status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003.

[17] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(7):50–58, 2002.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[19] Thomas Karagiannis, Mart Molle, and Michali Faloutsos. Long-range dependence: Ten years of internet traffic modeling. *IEEE Internet Computing*, 8(5):57–64, 2004.

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

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

[22] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Geographic load balancing for scalable distributed web systems. In *MASCOTS*, pages 20–27, 2000.

[23] A. Bestavros and S. Mehrotra. DNS-based internet client clustering and characterization. In *WWC-4. 2001 IEEE International Workshop on Workload Characterization*, pages 159–168. IEEE Computer Society, 2001.

[24] Zhuoqing Morley Mao, Charles D. Cranor, Fred Douglis, Michael Rabinovich, Oliver Spatscheck, and Jia Wang. A precise and efficient evaluation of the proximity between web clients and their local DNS

servers. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 229–242, Berkeley, CA, USA, 2002. USENIX Association.

[25] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Transactions in Networking*, 10(5):589–603, 2002.

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

[27] 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.

[28] Robert L. Carter and Mark E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *INFOCOM '97: Proceedings of the INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution*, page 1014, Washington, DC, USA, 1997. IEEE Computer Society.

[29] Richard Liston, Sridhar Srinivasan, and Ellen Zegura. Diversity in DNS performance measures. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 19–31, New York, NY, USA, 2002. ACM Press.

[30] Jeffrey Pang, Aditya Akella, Anees Shaikh, Balachander Krishnamurthy, and Srinivasan Seshan. On the responsiveness of DNS-based network control. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 21–26, New York, NY, USA, 2004. ACM Press.

[31] Leeann Bent and Geoffrey M. Voelker. Whole page performance. Technical report, La Jolla, CA, USA, 2002.

[32] C. Huitema and S. Weerahandi. Internet measurements: The rising tide and the DNS snag. In *Proceedings of the 13th ITC Specialist Seminar Internet Traffic Measurement and Modeling, Monterey, CA, Sept. 2000*.

[33] MJ Fischer and TB Fowler. Fractals, heavy tails and the internet. Technical report, Mitretek Technology Summaries, 2001.

[34] Matthias Grossglauser and Jean-Chrysostome Bolot. On the relevance of long-range dependence in network traffic. *IEEE/ACM Transactions on Networking*, 7(5):629–640, 1999.

[35] B. W. Rust. Fitting nature's basic functions part iv: The variable projection algorithm. *IEEE/AIP Computing in Science and Engineering*, 5(2):74–49, March 2003.

[36] Wikipedia. Pareto distribution — wikipedia, the free encyclopedia, 2006. [Online; accessed 6-May-2006].

[37] Wikipedia. Normal distribution — wikipedia, the free encyclopedia, 2006. [Online; accessed 6-May-2006].

[38] Mark Carson and Darrin Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[39] Peter Koch. DNS software survey, 2004. Presented at the RIPE 48 Meeting.

[40] Wikipedia. System — wikipedia, the free encyclopedia, 2006. [Online; accessed 4-May-2006].

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

[42] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[43] Kyrre M. Begnum and John Sechrest. The MLN project homepage, 2006. http://mln.sourceforge.net/ [Online; accessed 7-May-2006].

[44] Stephen Hemminger. Network emulation with netem. In *LCA national Linux conference '05*, 2005.

[45] The Internet Systems Consortium. The isc homepage. http://www.isc.org/[Online;accessed6-May-2006].

[46] Albert Hubert. PowerDNS nameserver. http://www.powerdns.com/ [Online; accessed 7-May-2006].

[47] Justin Erenkrantz et al. Flood – a profile-driven http load tester, 2006. http://httpd.apache.org/test/flood/ [Online; accessed 8-May-2006].

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

[49] Edith Cohen and Haim Kaplan. Proactive caching of dns records: Addressing a performance bottleneck. In *SAINT '01: Proceedings of the 2001 Symposium on Applications and the Internet (SAINT 2001)*, page 85, Washington, DC, USA, 2001. IEEE Computer Society.

Appendix A

Configuration Files

A.1 MLN/Xen Configuration

```
global {
        project dnslb
}

switch main {
        boot_order 1
}

superclass hosts {
        xen
        term screen
        template ubuntu-server-V0.1.ext3
        size 550M
        memory 64M
        nameserver 10.0.0.1
}

host dns {
        xen
        boot_order 10
        term screen
        template ubuntu-server-V0.1.ext3
        size     500M
        memory 128M
        startup {
                echo 1 > /proc/sys/net/ipv4/ip_forward
        }

        modules {
                ip_table_nat
                ip_conntrack
                ip_tables
```

```
33                      ipt_MASQUERADE
34          }
35
36          network eth0 {
37                      address dhcp
38          }
39          network eth1 {
40                      switch main
41                      address 10.0.0.1
42                      netmask 255.255.255.0
43                      broadcast 10.0.0.255
44          }
45  }
46
47  host www1 {
48          superclass hosts
49          boot_order 21
50          network eth0 {
51                      switch main
52                      address 10.0.0.11
53                      netmask 255.255.255.0
54                      broadcast 10.0.0.255
55                      gateway 10.0.0.1
56          }
57  }
58
59  host www2 {
60          superclass hosts
61          boot_order 21
62          network eth0 {
63                      switch main
64                      address 10.0.0.12
65                      netmask 255.255.255.0
66                      broadcast 10.0.0.255
67                      gateway 10.0.0.1
68          }
69  }
70
71  host www3 {
72          superclass hosts
73          boot_order 21
74          network eth0 {
75                      switch main
76                      address 10.0.0.13
77                      netmask 255.255.255.0
78                      broadcast 10.0.0.255
79                      gateway 10.0.0.1
80          }
81  }
```

```
82
83 host www4 {
84         superclass hosts
85         boot_order 21
86         network eth0 {
87                 switch main
88                 address 10.0.0.14
89                 netmask 255.255.255.0
90                 broadcast 10.0.0.255
91                 gateway 10.0.0.1
92         }
93 }
94
95 host www5 {
96         superclass hosts
97         boot_order 21
98         network eth0 {
99                 switch main
100                address 10.0.0.15
101                netmask 255.255.255.0
102                broadcast 10.0.0.255
103                gateway 10.0.0.1
104        }
105 }
106
107 host www6 {
108        superclass hosts
109        boot_order 21
110        network eth0 {
111                switch main
112                address 10.0.0.16
113                netmask 255.255.255.0
114                broadcast 10.0.0.255
115                gateway 10.0.0.1
116        }
117 }
118
119 host client1 {
120        superclass hosts
121        boot_order 50
122        network eth0 {
123                switch main
124                address 10.0.0.21
125                netmask 255.255.255.0
126                broadcast 10.0.0.255
127                gateway 10.0.0.1
128        }
129 }
130
```

```
131 host client2 {
132         superclass hosts
133         boot_order 50
134         network eth0 {
135                 switch main
136                 address 10.0.0.22
137                 netmask 255.255.255.0
138                 broadcast 10.0.0.255
139                 gateway 10.0.0.1
140         }
141 }
142
143 host cache {
144         superclass hosts
145         boot_order 50
146         network eth0 {
147                 switch main
148                 address 10.0.0.50
149                 netmask 255.255.255.0
150                 broadcast 10.0.0.255
151                 gateway 10.0.0.1
152         }
153 }
```

A.2 Flood Configuration

A.2.1 Flood static

```
1  <?xml version="1.0"?>
2  <!DOCTYPE flood SYSTEM "flood.dtd">
3
4  <flood configversion="1">
5    <!-- A urllist describes which hosts and which methods we
6    want to hit. -->
7    <urllist>
8      <name>Test Hosts</name>
9      <description>A bunch of hosts we want to hit</description>
10     <url method="GET">
11             http://www.test.lan/index.php?iter=2000
12     </url>
13   </urllist>
14
15   <!-- The profile describes how we will hit the urllists.
16   Round robin runs all of the URLs in the urllist in order
17   once. -->
18   <profile>
19     <name>RoundRobinProfile</name>
20     <description>Round Robin Configuration</description>
21
```

```
22      <useurllist>Test Hosts</useurllist>
23
24      <!-- Specifies that we will use round_robin profile
25      logic -->
26      <profiletype>round_robin</profiletype>
27      <!-- Specifies that we will use generic socket logic -->
28      <socket>generic</socket>
29      <!-- Specifies that we will use verify_200 for response
30      verification -->
31      <verify_resp>verify_200</verify_resp>
32      <!-- Specifies that we will use the "easy" report
33      generation -->
34      <report>relative_times</report>
35
36    </profile>
37
38    <!-- A farmer runs one profile a certain number of times. -->
39    <farmer>
40      <name>Joe</name>
41      <!-- run the Joe farmer for two iterations -->
42      <count>2</count>
43      <!-- Joe uses this profile -->
44      <useprofile>RoundRobinProfile</useprofile>
45    </farmer>
46
47    <!-- A farm contains a bunch of farmers - each farmer is a
48    thread. -->
49    <farm>
50      <name>Bingo</name>
51      <usefarmer count="1000" startcount="2"
52              startdelay="1">Joe</usefarmer>
53    </farm>
54
55    <!-- Set the seed to a known value so we can reproduce the
56    same tests -->
57    <seed>23</seed>
58  </flood>
```

A.2.2 Flood dynamic

```
1  <?xml version="1.0"?>
2  <!DOCTYPE flood SYSTEM "flood.dtd">
3  <flood configversion="1">
4    <urllist>
5      <name>Test Hosts</name>
6      <description>A bunch of hosts we want to hit</description>
7      <url method="GET">
8              http://www.test.lan/index.php?iter=400000
9      </url>
10   </urllist>
11
12   <profile>
```

```
13      <name>RoundRobinProfile</name>
14      <description>Round Robin Configuration</description>
15
16      <useurllist>Test Hosts</useurllist>
17
18      <profiletype>round_robin</profiletype>
19      <socket>generic</socket>
20      <verify_resp>verify_200</verify_resp>
21      <report>relative_times</report>
22
23    </profile>
24
25    <farmer>
26      <name>Joe</name>
27      <count>2</count>
28      <useprofile>RoundRobinProfile</useprofile>
29    </farmer>
30
31    <farm>
32      <name>Bingo</name>
33      <usefarmer count="200" startcount="2"
34                startdelay="1">Joe</usefarmer>
35    </farm>
36
37    <seed>23</seed>
38  </flood>
```

Appendix B

Scripts

B.1 Analysis Scripts

```
#!/usr/bin/python

# avgstdev.py
# Process a file line by line, extracting the number in the
# given field. Print the average and standard deviation to
# stdout.
#
# Author: Sven I. Ulland
#

import sys
import string
import stats

if len(sys.argv) != 3:
        print "Usage: %s <infile> <fieldno>" % sys.argv[0]
        sys.exit(1)

myfile = open(sys.argv[1])
lines = myfile.readlines()

myvals = []

for line in lines:
        myvals.append(float(line.split(" ")
                [int(sys.argv[2])]))

print stats.mean(myvals), stats.stdev(myvals)
```

```
#!/usr/bin/python

# cumul.py
# Calculate and print the cumulative frequency distribution
# of a list of time intervals on the form
#
#      "HH:MM:SS.ssssss HH:MM:SS.ssssss"
#
# Where the first field is start time and the second is stop
# time.  Input parameters are input file, minimum and
# maximum boundary, and the number of steps (granularity of
# the output).
#
# Author: Sven I. Ulland
#

import sys
import string
import stats
import datetime

if len(sys.argv) != 5:
        print "Usage: %s <infile> <min> <max> <steps>" %
                        sys.argv[0]
        sys.exit(1)

myfile = open(sys.argv[1])
lines = myfile.readlines()

mydeltas = []

for line in lines:
        times = line.split(" ")
        time1h = times[0].split(":")[0]
        time1m = times[0].split(":")[1]
        time1s = times[0].split(":")[2].split(".")[0]
        time1n = times[0].split(".")[1]
        time2h = times[1].split(":")[0]
        time2m = times[1].split(":")[1]
        time2s = times[1].split(":")[2].split(".")[0]
        time2n = times[1].split(".")[1]

        stamp1 = datetime.datetime(2006, 4, 4, int(time1h),
                int(time1m), int(time1s), int(time1n))
        stamp2 = datetime.datetime(2006, 4, 4, int(time2h),
                int(time2m), int(time2s), int(time2n))
        delta = stamp2 - stamp1
        mydeltas.append((delta.seconds * 1000000)
                        + delta.microseconds)
```

```
50
51 mydeltas.sort()
52
53 min = int(sys.argv[2])
54 max = int(sys.argv[3])
55 steps = int(sys.argv[4])
56
57 interval = (max - min) / steps
58 freqs = [0] * (steps + 1)
59
60 cumul = 0
61 i = 0
62 x = 0.0
63 while x < max:
64         for val in mydeltas:
65                 if val >= x and val < (x + interval):
66                         freqs[i] = freqs[i] + 1
67                         cumul = cumul + 1
68
69         print "%g\t" %x,
70         print freqs[i], cumul
71         x = x + interval
72         i = i + 1
```

```
1 #!/usr/bin/python
2
3 # delta.py
4 # Calculate and print the average and standard deviation of
5 # time intervals, given in an input file on the form
6 #
7 #     "HH:MM:SS.ssssss HH:MM:SS.ssssss"
8 #
9 # Where the first field is start time and the second is stop
10 # time.
11 #
12 # Author: Sven I. Ulland
13 #
14
15 import sys
16 import string
17 import stats
18 import datetime
19
20 if len(sys.argv) != 2:
21         print "Usage: %s <infile>" % sys.argv[0]
22         sys.exit(1)
23
24 myfile = open(sys.argv[1])
25 lines = myfile.readlines()
```

```
26
27 mydeltas = []
28
29 for line in lines:
30         times = line.split(" ")
31         time1h = times[0].split(":")[0]
32         time1m = times[0].split(":")[1]
33         time1s = times[0].split(":")[2].split(".")[0]
34         time1n = times[0].split(".")[1]
35         time2h = times[1].split(":")[0]
36         time2m = times[1].split(":")[1]
37         time2s = times[1].split(":")[2].split(".")[0]
38         time2n = times[1].split(".")[1]
39
40         stamp1 = datetime.datetime(2006, 4, 4, int(time1h),
41                 int(time1m), int(time1s), int(time1n))
42         stamp2 = datetime.datetime(2006, 4, 4, int(time2h),
43                 int(time2m), int(time2s), int(time2n))
44         delta = stamp2 - stamp1
45         mydeltas.append((delta.seconds * 1000000)
46                         + delta.microseconds)
47
48 mydeltas.sort()
49
50 print len(mydeltas), mydeltas[0], stats.mean(mydeltas),
51 print mydeltas[len(mydeltas)-1], stats.stdev(mydeltas)
```

```
1 #!/usr/bin/python
2
3 # portorder.py
4 # Reorder an input file based on source and destination
5 # ports. The input lines are on the form below, derived from
6 # a processed pcap file:
7 #
8 #      "HH:MM:SS.ssssss srcport dstport"
9 #
10 # The result is written to the specified output file
11 #
12 # Author: Sven I. Ulland
13 #
14
15 import sys
16 import string
17 import re
18
19 if len(sys.argv) != 3:
20         print "Usage: %s <infile> <outfile>" % sys.argv[0]
21         sys.exit(1)
22
```

```
myfile = open(sys.argv[1])
lines = myfile.readlines()

p = re.compile(r"\d{2}:\d{2}:\d{2}\.\d{6} (\d+) (\d+)")

i = 0
while i < len(lines):
        m = p.match(lines[i])
        sport = m.group(1)
        if int(sport) != 80:

                j = 0
                while j < len(lines):
                        m = p.match(lines[j])
                        dport = m.group(2)
                        if dport == sport:
                                print "Swapping lines..."
                                lines[i+1], lines[j] =
                                        lines[j], lines[i+1]
                                break
                        j += 1

        i += 1

output = open(sys.argv[2],"w")
output.writelines(lines)
```

```
#!/usr/bin/python

# movingavg.py
# This script calculates a right-skewed simple moving
# average for a set of input data. It accepts an input file
# with lines on the form
#
#     "<relative seconds> <value>"
#
# The script assumes the time field to be linearly
# increasing. Input parameters is the input file and the
# size of the moving average window.
#
# Author: Sven I. Ulland
#

import sys
import string
import stats

def usage():
        print "Usage: %s <infile> <step>" % sys.argv[0]
```

```
23          sys.exit(1)
24
25  def sma(s, n):
26          myavg = []
27          for i in range(len(s)):
28                  data = s[i:i+n]
29                  accum = 0.0
30                  for j in range(len(data)):
31                          accum += data[j][1]
32
33                  myavg.append([s[i][0], accum / n])
34
35          return myavg
36
37  if len(sys.argv) != 3:
38          usage()
39
40  file = open(sys.argv[1])
41  lines = file.readlines()
42
43  readvals = []
44
45  for line in lines:
46          timestamp = int(line.split(" ")[0])
47          value = int(line.split(" ")[1])
48          readvals.append([timestamp,value])
49
50  avgseries = sma(readvals, int(sys.argv[2]))
51
52  for val in avgseries:
53          print val[0], val[1]
```

B.2 Web Server Scripts

```
<?php
        /* load.php (also used as index.php)
         *
         * Accepts the input parameter 'iter' which governs
         * the number of iterations to do in an empty for-
         * loop.
         */

        $start = microtime(TRUE);

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

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