UNIVERSITY OF OSLO
Department of Informatics

Traffic Characteristics and Queueing Theory: Implications and Applications to Web Server Systems

Master thesis

Jon Henrik Bjørnstad

May 22, 2006

Traffic Characteristics and Queueing Theory: Implications and Applications to Web Server Systems

Jon Henrik Bjørnstad

May 22, 2006

Abstract

Businesses rely increasingly on Internet services as the basis of their income. Downtime and poor performance of such services can therefore be directly translated into loss of revenue. In order to plan and design services sufficiently capable of meeting minimum Quality of Service (QoS) requirements and Service Level Agreements(SLA), an understanding of how network traffic and job service demand affect the system is necessary. Traditionally, arrival and service processes have been modelled as Poisson processes. However, research done over the years suggests that the assumption of Poisson traffic is fallible in many cases. This work considers performance of a web server under different traffic and service demand conditions. Moreover, we consider theoretical models of queues, response time formulas derived from this models and their validity for a web server system. We try to make a simple approach to a complex problem by modelling a web server as one simple queueing system. In addition, we investigate the phenomenon known as *self-similarity* which has been observed in web traffic inter-arrival processes. We have found indications that traffic with identical expectation values for inter-arrival and service time differing in distribution type affects the response time differently. Moreover, classical queueing models are found unsuited for doing capacity planning. Instead we suggest *"a worst case scenario"* approach in order for service providers to meet service level targets. Much of the previous work within these areas is of a highly mathematical and theoretical nature. We investigate from a more pragmatic viewpoint.

Preface

A paper titled *"On the Reliability of Service Level Estimators in the Data Centre"* based on the work done herein written in collaboration with my thesis supervisor, Professor Mark Burgess, has been submitted to the *17th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2006)* conference and currently awaits acceptance.

This thesis concludes my master's degree in Network and Systems Administration at the Oslo University College. Students enrolled in this programme finish of with a half year project and the master thesis therefore counts for 30 ECTS of my master's degree.

This project is one third of a three part, larger project about Quality of Service for web server systems initiated by Oslo University College. Whereas the other to projects look at QoS issues at WAN and LAN level, this project considers QoS problems at server level.

The project was motivated by the author's interest in traffic characteristics and its effect on server systems. This interest grew out of studying previous research on the matter. It was initially planned to look at the web server in detail (CPU, RAM, Disk etc.) to find bottlenecks for QoS. However, such a study would be very case specific and we wanted to be able to say something about web services in general. The work is therefore focused on widely used models and formulas for queueing systems to enable a general discussion of web servers.

Acknowledgments

There are many people to thank for making this thesis possible. First and foremost, I would like to thank my supervisor, Professor Mark Burgess. His undivided faith in this project and ability to bring out the best of analytical and practical skills in people has been invaluable in tough periods. This project would never have been the same without his help.

Secondly, I would like to thank my family for academic guidance, love and support, and for implicitly and explicitly showing their belief in me. A special thanks goes out to my grandmother whom I've haven't been able to visit as much as I wanted. I love you all very much.

Further, I would like to thank the whole staff at Oslo University College who through bachelor and master level has given me the best of guidance and tutoring. PhD students Geir Horn and Amund Kvalbein from the University of Oslo for input and sharing of their work. My fellow students, especially Sven I. Ulland, Gard Undheim, Espen Braastad and Ilir Bytyci, for sharing knowledge and being good friends.

A big thank you goes out to all my other friends for letting me do other things than study on occasion. I would also give special thanks to my band colleagues for letting me blow off some steam from time to time.

Last, but not least, I would like to thank my girlfriend, Heidi Vihovde Sandvig, for her undivided love and support throughout the whole masters programme. Her ability to inspire and encourage me has made this degree possible and I'm forever grateful for having her in my life.

May 2006

Jon Henrik Bjørnstad

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"I just had to take the hypertext idea and connect it to the DNS and TCP ideas and - ta-da! - the World Wide Web" - Tim Berners-Lee

When Tim Berners-Lee in 1989 introduced the research community to his great, new invention to ease information exchange, nobody could grasp the impact this would have on the world. What started as a simple way of sharing and updating information among researchers has now grown into a global venue for large scale trade, communications and information in the public domain. The World Wide Web has enabled whole new business models within areas such as banking, retail and marketing. Public services like health, education and other government services have also found their way into the Internet.

To meet the ever growing demands of an equally growing amount of users, companies and official institutions are investing considerable sums of money in IT infrastructure. Large investments are made in software development, hardware procurement and maintenance for data centers of possibly many thousand servers to make sure that systems are meeting performance demands at any time. Given this cost burden it is obviously desirable that no money is spent in vain, i.e. that no money is spent on making the IT infrastructure performing better than it absolutely has to do. This is where capacity planning and performance analysis comes into play. However, before the planning and analysis process can start one has to define metrics and establish their limits. This motivates the discussion on *Quality of Service (QoS)*.

1.1 QoS

Users of IT systems have high expectations regarding the general level at which a service is provided. Research has indicated that users are not willing to tolerate latency times greater than eight-ten seconds [1]. Further, the tolerance for latency decreases for subsequent page loads during a session. Disgruntled and irritated users will take there business elsewhere if they are not given the quality of service they expect. Also, the likelihood of those users returning to a poorly performing web site is decreased. For businesses that rely on Web services for making money this can be devastating.

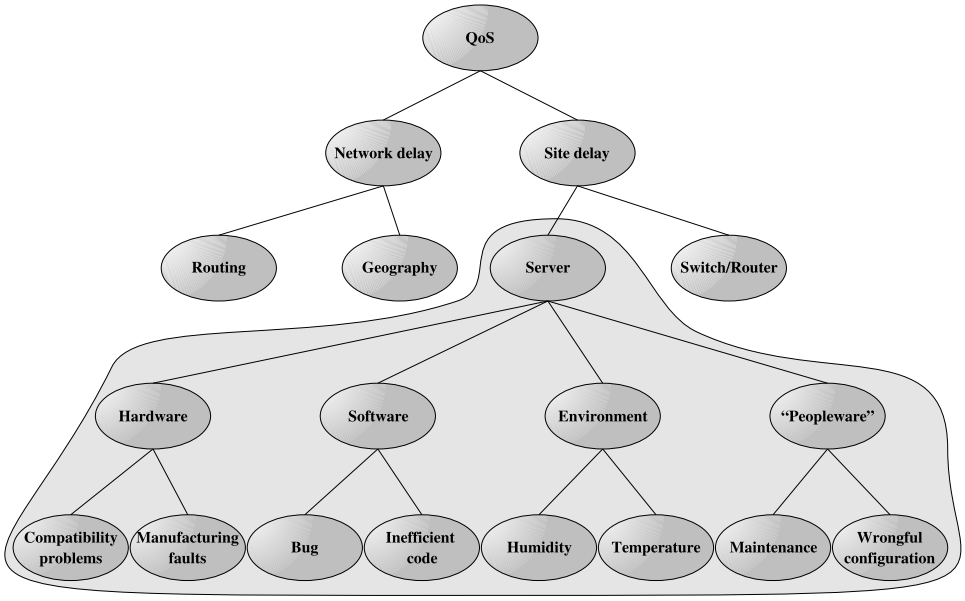


Figure 1.1: *Fault tree for poor Quality of Service. We look at the web server in specifics. All of the outlined factors have impact on web server performance. We attempt a consolidated view on the web server and consider it to be a black box, thus disregarding details that contribute to poor QoS.*

Keeping the latency and response times at minimum therefore has a strong economic incentive as it can be a directly related to revenue. There is no single definition of good quality of service. What is perceived by users as a good service relies on several different aspects. For instance, bad web site design can affect the feelings users have about the site. However, page design cannot easily be quantified and taken into account when considering QoS offered by a site. System engineers are therefore more interested in properties like latency and response time.

Degradation of latency and response times can be caused by different factors. A transaction in the client/server paradigm is usually an interplay between the following resources.

1. Client
2. Network
3. Server

Thus, causes for high response latency times can appear at any of these levels. For instance, non-caching web clients can be an important factor for performance degradation. Network congestion manifests itself through slow transaction of content. A

malfunctioning server will perhaps not be able to process a request fast enough.

1.2 Modelling network traffic

Proper capacity planning relies on the ability to characterize everything that may affect an offered service. For network services, the nature of traffic is at the very crux of the matter. The ability to effectively schedule service depends on the ability to predict demand. In pre-Internet time, the only traffic of any significance on networks was voice traffic. There was no need to worry about the effects of network layers as these didn't exist. Traffic, both inter-arrival and service rates, could easily be modelled as Poisson processes. Finding critical values for queueing was easy and permitted design of voice networks to meet any desired performance characteristics [2].

While traffic in line switched networks exhibits Poissonian properties, the assumption of likewise Poisson characteristics for packet switched traffic has in several cases been proven to be fallible. Traffic in packet switched networks exhibits a bursty nature; a property that is not aptly modelled as a Poisson process. Because of its attractive analytical qualities, Poisson processes are widely used in capacity planning and analysis. Using this approach can have non-negligible ramifications for the QoS offered at a site should traffic exhibit different statistical properties.

Many attempts have been done at describing the "true" nature of network traffic. In the late 80's and through the 90's, a considerable amount of literature on the subject was produced. The work done where more or less inconclusive, and in many cases could not give categorical answers. Part of the explanation is that network traffic is caused and affected by a large number of independent processes which renders it extremely hard to predict its characteristics. For instance, sociological phenomena, such as cultural or sporting events (e.g. the 1998 World Cup in France [3]), can have a considerable impact on usage of a particular web resource. Also, the characteristics of traffic depend on the services offered by a particular site which makes it difficult to conclude anything about web traffic in general.

1.3 Queueing systems

Web server systems are essentially queueing systems. Incoming requests are arriving and processed, and may have to spend some time in queue depending on the load at a particular time. Models that enable prediction of performance for simple queueing systems exist. Albeit, many of these are founded on the assumption of both exponentially distributed inter-arrival and service times (i.e. Poisson arrival processes). Operations during arrival and service processes different from Poisson can therefore not be expected to be well modelled by this simple theory.

Exact queueing theory for general arrival and service conditions is complex and not suited for queueing theory novices. However, fairly comprehensible approximate models exist that possibly are able to better predict system operations than simple Poisson models. If hard to understand, models are not used, but they are useless if they are unable to do fairly accurate prediction of a phenomenon.

1.4 Thesis outline

The work herein will study a single web server under varying service and traffic conditions with respect to the central QoS metric *response time*. Experimental results will be compared with simple queueing models in order to check their validity and applicability for performance prediction of web server systems.

In chapter 2 we will give some background information and have a look at research done in the field of traffic characterization, QoS and web server performance. Chapter 3 lists the objectives for this work while chapter 4 elaborates on the theory needed for analysis and execution of the experiments. Chapter 5 explains the experimental setup and chapter 6 contains the methodology for the experiments conducted herein. In chapter 7, the analysis of data collected from experiment is conducted. Finally, chapter 8 contains concluding remarks and thoughts for future work.

Chapter 2

Background

This chapter is aimed at giving the reader some background information of the technologies studied in this thesis. We review basic networking concepts and protocols, and the HTTP protocol in particular. This is followed by a brief overview of high volume service strategies. Some theory on statistics has also been included to provide comprehensibility of the section concerning previous research. The software used in conjunction with this project is then presented, and the chapter is concluded with a survey of related work and previous research within the studied fields.

2.1 Network and protocol concepts

This section will briefly review basic networking concepts and give background information on the HyperText Transfer Protocol(HTTP) which is the basis of World Wide Web communications.

2.1.1 The OSI model and the TCP/IP suite

Communication over a network is a complex interplay between protocols, hardware, software and operating systems. Because of the plethora of different devices from different manufacturers, it would be hard to achieve interaction of networking nodes without a standardized communication template. The *Open Systems Interconnection (OSI)* standard divides the basic communication functions required for devices to communicate into 7 layers [4]. The model encompasses functions from the lowest layer (layer 1, physical layer) to the highest (layer 7, application layer).

Physical layer, #1: This layer deals with the transfer of bits over a communication channel. It is concerned with system parameters such as voltage levels and signal durations, and mechanical aspects as socket type and number of pins.

Data link layer, #2: The data link layer is responsible for the transfer of frames. It inserts framing information to indicate the boundaries of a frame. In the case of LANs and Ethernet, it also includes medium access control procedures.

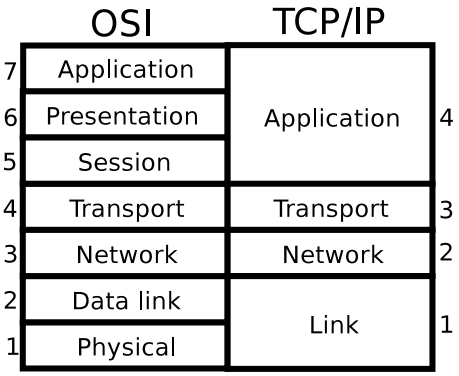


Figure 2.1: *The OSI and TCP/IP model compared.*

Network layer, #3: This layer provides for transfer of packets across a network. A key aspect of this layer is the addressing scheme used to identify end points of a communication.

Transport layer, #4: The transport layer is responsible for transfer of data from a process at the source to a process at the receiving end. Transport layer can either provide connection oriented, reliable transfer or the opposite, connectionless unreliable transfer.

Session layer, #5: This layer can be used to control the way data is exchanged. Some applications require half-duplex dialogs where the parties transmit data one at a time.

Presentation layer, #6: The presentation layer is intended to do conversion between differences in representations of data within end-point systems, i.e. MIME conversion and data compression.

Application layer, #7: This layer facilitates communication between applications and lower-layer network services. Through protocols at this stage applications negotiate formatting, security, synchronization and other requirements.

The OSI model is a totally generic model and does not concern itself with specific technologies or protocols.

Because of its widespread use, the TCP/IP model has to a large degree superseded the OSI model. Unlike the OSI model, the TCP/IP suite only operates with 4 layers. The application layer (Layer 4) encompasses the functionality of Layer 5-7 of the OSI model. Application protocols, like HTTP, belong in this category. Layer 3, the transport layer, common protocols for data transfer is TCP and UDP. In Layer 2 uses IP as addressing scheme, either in version 4 or version 6. In layer 1 the most prevalent protocol is Ethernet and is the counterpart to layers 1 and 2 of the OSI model(see figure 2.1).

2.1.2 Hypertext Transfer Protocol

Like any other protocol, the HTTP specification was not in its definite form when it was first introduced, and has undergone a few revisions to develop into its current specification. The first draft of the HTTP protocol, HTTP/0.9, was introduced in 1990 and was a simple protocol for raw data transfer across the Internet [5]. HTTP/1.0, specified in RFC 1945 [6] from 1996, had major improvements compared to the sparse functionality of HTTP/0.9. Perhaps the most important change was the support for Multipurpose Internet Mail Extensions (MIME) formatted messages which enabled transfer of any file over the HTTP protocol encoded in American Standard Code for Information Interchange (ASCII) text format.

HTTP/1.0 grew out of significant debating and experimentation, however lacked a formal specification. RFC 1945 is therefore only a document describing common usage of the protocol [7]. The lack of a formal specification lead to a vast amount of incorrectly or incompletely implemented applications claiming to be HTTP/1.0 compliant. In turn, this could lead to inability of communicating applications to determine the true capabilities of each other. With time HTTP/1.0 also proved to have other shortcomings. Amongst others, the effects of hierarchical proxies and caching, and the need for persistent connections and virtual hosts, were not sufficiently taken into consideration by the HTTP/1.0 specification. All of this necessitated a protocol version change.

HTTP/1.1, the third version specified in RFC 2616 from 1999, includes more stringent requirements than its predecessor in order to ensure reliable implementations of HTTP/1.1 features. The specification is about three times as long as the HTTP/1.0 specification, reflecting a significant increase in complexity. One of the most important changes made with this protocol revision was the default enabling of persistent connections. TCP connection setup can be resource intensive and the idea behind persistent connections, or `keep-alive`, is to provide pipelining of HTTP requests. In other words, once a TCP connection has been established between a client and a server, subsequent HTTP request can use the same TCP connection, provided that an associated timeout value is not exceeded. Although the specification of HTTP/1.1 is rather old in networking terms and dates back to 1999, the previous version 1.0 is still in wide use [1].

2.1.3 General HTTP operations

HTTP operations are based on a client-server model where a client requests a server for a resource hosted by the server. Resources are identified with Uniform Resource Identifiers (URI) which are globally unique[2]. They consist of the Fully Qualified Domain Name (FQDN) of a resource hosting server, and the path to a specific resource on that server. Typically, a resource is a HTML text file, but can be any file type sup-

[1] http://en.wikipedia.org/wiki/HTTP

[2] URIs are also known as WWW addresses, Universal Document Identifiers, Universal Resource Identifier, and the combination of Uniform Resource Locators (URL) and Names (URN).

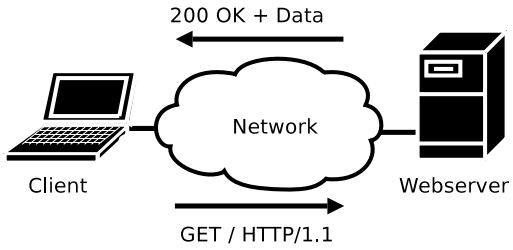


Figure 2.2: *An example of a successful HTTP request-response sequence.*

ported by MIME. Client requests are performed using various possible methods. Each method has different purposes and yields different results. The most frequently utilized methods are:

GET: This is by far the most used method for retrieval of a web resource. The `GET` simply means retrieve the information identified by the URI. The `GET` request can change to a "conditional GET" if the request header includes a special header field, i.e. `If-modified-since`. In this specific case, the resource will not be downloaded if it has not been changed since a certain time. This happens when a browser deploys caching to alleviate network load by not downloading resources already residing on the client.

POST: This method facilitates submission of user data to a given URI. The data can be of any kind, and is contained within the request message body. Document upload to a site is an example functionality facilitated by this request type.

HEAD: Asks for a response identical to GET. However, properly implemented Web server software should not answer with the message body, only with the HTTP header information. The HEAD method is useful in web application programming and testing because the information in a HEAD response is identical to the header information of a GET response. I.e., testing can be conducted without transmitting the requested resource in full.

The remaining, OPTIONS, PUT, DELETE, TRACE and CONNECT, are rarely implemented or allowed at server instances.

Responses to client requests are always coded with a status code consisting of a 3 digit number and a textual reason phrase. The status codes exists to guide the user agent in processing the response. HTTP status codes are divided into general classes based on the nature of the code.

1xx: Informational Request received, continuing process. An example of this is status code 101 which flags that the client should change the application protocol being used for the pending connection, e.g. an upgrade from HTTP/1.0 to HTTP/1.1.

2xx: Success This class of status codes indicates that the request was successfully received, understood and accepted. An example is status code 200 which is the indication of a successful completion of a request.

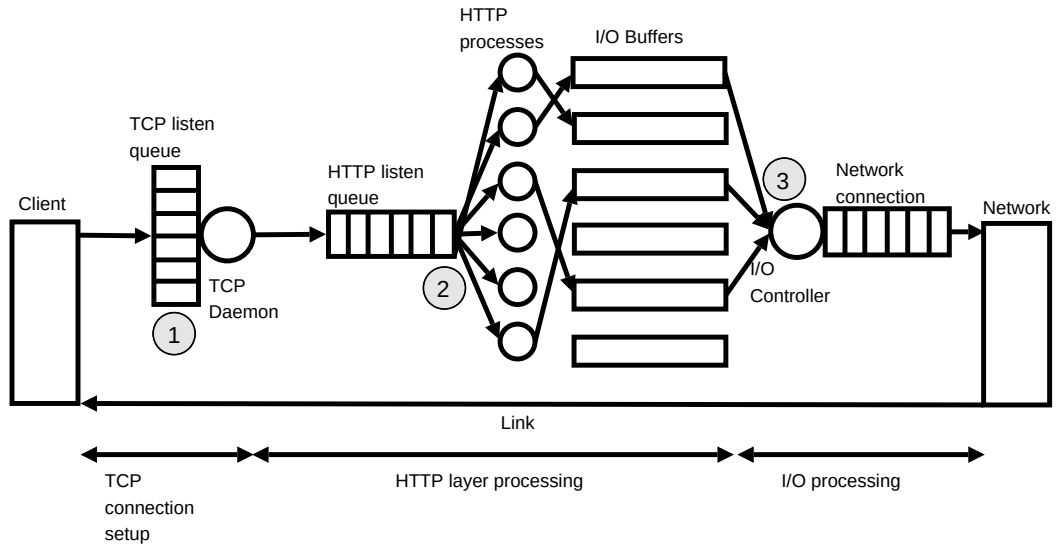


Figure 2.3: *Web server queueing model. Figure adapted from [8].*

3xx: Redirection Further action must be taken in order to complete the request. An example of this class of status codes is 304. This code is seen when a client is making a conditional GET and the document has not been modified.

4xx: Client Error The request contains bad syntax or cannot be fulfilled. An example of this class is code 404 which indicates that the requested URI was not found on the server.

5xx: Server Error The server failed to fulfill an apparently valid request. An example is code 500 which indicates that the server encountered an unexpected condition which prevented it from fulfilling the request.

For a full list of HTTP status codes the reader is referred to in the HTTP/1.1 specification[3].

2.2 Web services

The services offered through the web vary a lot from site to site. Also, the web server software responsible for delivering content may be different. However, the processing of requests generally follows the same procedure regardless of the distinctive site and software characteristics.

2.2.1 Web server dynamics

In most cases, URLs are requested trough a POST or a GET requests. Depending on whether or not the client and server is HTTP/1.1 enabled, a TCP connection is es-

[3] `http://www.faqs.org/rfcs/rfc2616.html`

tablished for each request. Under HTTP/1.0 operation without `keep-alive`, a new socket is established for each subsequent request, whereas HTTP/1.1 default operational mode allows several requests using the same TCP socket (often termed *request pipelining*). For the client to display a web page, it might need to initiate several subsequent HTTP transactions, but as pointed out, not necessarily several TCP connections. A web server process will typically listen for requests on a port in the $0 - 1023$ range[4]. Default port for a considerable majority of web server software is port 80, however it can be any vacant port number.

Processing a HTTP requests possibly leads to queueing at several levels of the web server. Web server operations can therefore be modelled as a series of queues [8, 9] as depicted by figure 2.3. Upon receiving a connection establishment request by a client (TCP-SYN packet), the server answers with a SYN-ACK packet and puts the request in the TCP listen queue provided there exists a vacant slot (stage 1 in figure 2.3). The client then finalizes the TCP connection setup by sending the final ACK packet of the three way handshake.

After the three way handshake is completed, the transaction request is ready to be transferred to the HTTP subsystem, which consists of and HTTP listen queue and one or more multi-threaded HTTP daemons [8]. The request has to wait in queue until the HTTP subsystem accepts it (stage 2 in figure 2.3) by employing an available HTTP thread. If there are no vacant threads, the request remains in the HTTP listen queue until a thread is ready to process it.

Depending on the nature of the requested resource, the thread will either fetch a file or execute server side code. Either way, the data that is to be transmitted back to the client as response is put into a network I/O buffer. If no such is available, the HTTP thread lingers and waits until a buffer is freed. The I/O buffers are emptied over a common network connection and the scheduling of network access for the buffers are done by an I/O controller according to some algorithm, e.g. round robin (stage 3 in figure 2.3). Once the response buffers are emptied, server side processing of the request is completed and the client has received the solicited resource.

2.2.2 Web service systems

Web service systems are extremely diverse. They range in scale from small personal web servers hosting one application, to large and complex web shops to even larger search engine systems. Most web sites today offer services that fall into one or more of the following categories [10].

[4]Port numbers 0 - 1023 are termed *Well known ports*. On Unix-derived operating systems, opening a port in this range to receive incoming connections requires root privileges.

- **Informational:** Online newspapers, product catalogs, manuals, online classified ads, white papers and books.
- **Interactive:** Registration forms and online games.
- **Transactional:** Electronic shopping, ordering goods and services, banking.
- **Workflow:** Online planning and scheduling systems, inventory management.
- **Collaborative environments:** Distributed authoring systems, collaborative design tools.
- **Online Communities:** Discussion groups, recommender systems, online marketplaces and auctions.
- **Web portals:** Electronic shopping malls, search engines and e-mail services.

The purpose of a certain site decides how a particular web application is implemented and what hardware is needed to support its operation. Due to all this diversity, a general discussion on web systems from a performance point of view is hard. There is no such thing as a typical or quintessential web site. Every system is different from each other and have different performance requirements.

Performance of web service systems

In analyzing performance of web services there are several topics that can be studied.

1. *Contents and application*. Performance can be improved by making contents smaller and application code more efficient.
2. *Server hardware and software*. Procure and configure hardware and software to meet the needs of a web service.
3. *Network bandwidth and infrastructure*. Employ several servers and do load balancing for performance enhancement.

Web performance exhibits enormous variations depending on multiple factors such as geographical location of clients and servers and the time of day. The diversity of web systems and the extreme variation of requests lead to no single, best practice for hardware and software procurements and performance tuning. However, the inherent bottlenecks are all the same. Depending on the service offered, these are more or less prominent.

Network capabilities: Network can be a bottleneck. Large scale systems are often connected with a high speed connection to the Internet. In these cases it is unlikely that the network is an important factor for poor performance. However, delays in connecting network between a client and a server can degrade server performance. Aggregation of delayed ACKs by many clients sitting on congested networks causes a job to linger in a system longer than it has to. This

may cause arriving jobs to be dropped because the processing queue is full, even though the server is under fairly light load [8].

CPU: CPU is fairly unaffected by sending plain HTML over the network. However, a dynamically generated web site can utilize the CPU heavily, especially if there is some kind of load intensive processing (like thumbnail processing).

RAM: RAM is a crucial component in server performance. If processing requirements exceeds the amount of RAM available, the server will start swapping to disk. Disk reads/writes are 50 to 100 times slower than reading from RAM[5], thus this will cause serious performance degradation.

Disk: Hard disk read/writes are slow, but unavoidable for web server systems. Delivering contents like multimedia or static text requires reading it from disk into memory. Database searches are also dependent on disk access. Improving disk speeds, can therefore significantly improve response time, though how much is dependent on the nature of the site.

Process Management: Server side scripts can be executed in three ways. For CGI scripts a process is created for each request and killed when the request is completed. FastCGIs are persistent processes to overcome the inefficiency of CGI spawn/kill procedure. Finally, there are server side scripts which are applications that run within the context of the Web server [11]. Spawning new processes is time consuming and thus the script execution paradigm of web server software is of importance for response time.

Connecting to Other Servers: Connecting to other servers, like an SQL server, can have non-negligible ramifications for response time. An overloaded SQL server processing multiple heavy database queries at once can cause the request to linger in the system, waiting for the SQL call to return.

2.3 High volume service strategies

As mentioned previously, characteristics of web service systems vary a lot depending on their purpose. Some are designed for simple chores and are not intended to serve a vast amount of requests, while others, like the search engine Google, are high volume and serves an more than 200 million requests per day (April 2003) [6]. No single server is able to process this load by itself. Several systems have to cooperate in order to meet the demands, i.e. a load balancing strategy becomes important.

2.3.1 Networking solutions

Load balancing is a strategy for distributing load amongst two or more web service systems to provide redundancy and high availability. Often one speaks of load bal-

[5] `http://phplens.com/lens/php-book/optimizing-debugging-php.php`
[6] `http://www.google.com/googlefriends/moreapr03.html`

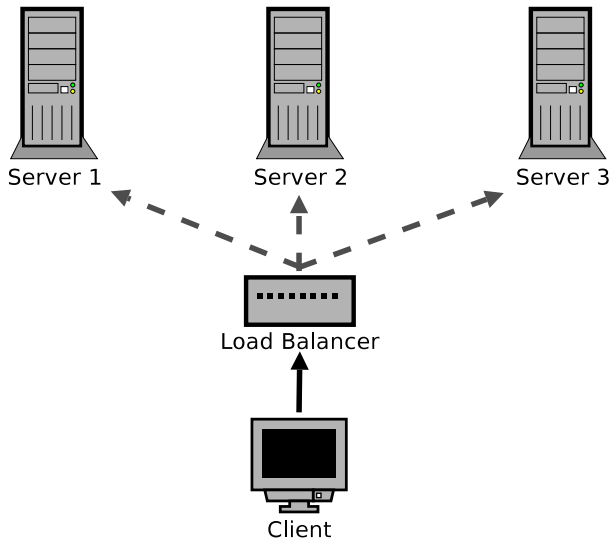


Figure 2.4: *Example of server load balancing using a load balancing device.*

ancing on two different levels - *Server Load Balancing*(SLB) on a Local Area Network (LAN) level and *Global Server Load Balancing*(GSLB) on a Wide Area Network(WAN) level.

Server Load Balancing

Server Load Balancing typically refers to strategies for distributing load in a LAN. Load balancers are routers or application layer switches that are able to distribute load to two or more server boxes or systems. The load balancer can in a TCP/IP scenario distribute load either based by IP or link layer address (usually Ethernet MAC) translation.

Load balancing based on IP address works by configuring the load balancer with a so-called *Virtual IP* interface which is the interface that is reachable for the outside world. *Destination Network Address Translation (DNAT)* is then used to forward the request to a server connected to the load balancer according to a scheduling algorithm (see figure 2.4). In TCP/IP, when a packet is sent to a destination IP, the resulting answer needs to be returned with this destination as source address. In other words, egress traffic has to be NATed on its way back; so-called *Source Network Address Translation*.

As mentioned, server load balancing is also possible to accomplish via Layer 2 redirecting. This is called *Direct Server Return(DSR)*. In DSR mode the load balancers use MAC address translation tables to redirect packets. If all the servers are configured with the same IP address, the packet is sent back with correct source IP and will be accepted by the transmission initiating party [12]. DSR eases the load on the balancer as NATing is avoided.

Scheduling can be done in many ways. Usually there is a possibility to do load balancing by iteratively choosing servers from a list - a so-called *Round Robin* schedul-

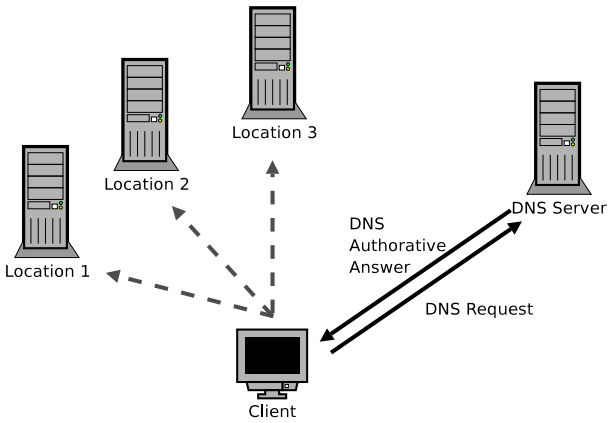


Figure 2.5: *DNS load balancing: The DNS query resolves to an IP e.g. according to a round robing algorithm which directs the client to different locations each time.*

ing algorithm. Some vendors have also implemented ways to check server load in real time. Many load balancers also have the ability to balance load based on application layer information (OSI layer 5-7). This is very useful in distribution HTTP traffic workload.

HTTP is a stateless protocol; it does not remember anything about previous requests. To mend this, web server applications have the ability to place cookies in client browsers in order to keep track of state upon subsequent requests. Usually this cookie is a session id for which the server holds affiliated data. For web applications, like web shops and Internet banks, this mechanism is crucial. However, session based web conversations are reliant on traffic being sent to the exact same server throughout the duration of the session, which obviously poses a challenge in doing load balancing by regular address translation. E.g., if the scheduling mechanism of the load balancer is simple round robin, there is no guarantee that subsequent requests belonging to the same session is processed by the same server each time. In order for load balancing to work with sessions, the balancer has to keep track of which session belongs to which server, i.e. *stick* requests belonging to a certain session to the session initiating server. In this way, the load balancer functions as an application layer switch. Sticky session operations are both resource intensive and has the caveat of uneven distribution of the load.

Global Server Load Balancing

Instead of doing load balancing locally, it is possible through Global Server Load Balancing techniques to send traffic to a whole different geographical location. The idea is to lessen the strain on one data center by globally distributing load to several data centers located elsewhere in the world.

GSLB can be facilitated by essentially two independent protocol systems - Domain Name System(DNS) and Border Gateway Protocol(BGP). DNS load balancing is perhaps the oldest load balancing discipline and existed long before conventional SLB

was a technology or a viable product [12]. Although DNS load balancing has been more or less superseded by LAN load balancers, some still employ it for spreading the workload by doing name serving from a pool of IP addresses. The Berkeley Internet Name Domain name server, which is a de facto standard for Unix like systems, is able to do DNS load balancing simply by mapping a domain name to several IPs. The following shows such a DNS record for an example domain with non-routeable IP addresses.

```
www.example.org   IN   A   192.168.0.1
                  IN   A   192.168.0.2
                  IN   A   192.168.0.3
```

The name server permutes the list in a round robin fashion and for each request puts the previous last entry first in the list. DNS has several caveats, one of them being caching. DNS servers usually cache authoritative answers for efficiency and will use the cached record for subsequent name resolve requests. A possible caveat of this is that the list of IP addresses might not reflect actual conditions, for instance if one of the servers has gone down.

Load balancing with BGP can occur when multiple routes to a location are announced with the same administrative distance and cost. Routers then have to decide which route they want to use, with the possible result that routers receiving the route announcements choose different routes.

An alternative load balancing strategy was used by Netscape in earlier days. For connections to their web site they had hard coded a list of IP addresses into the Netscape browser [13]. No DNS lookup was done for this site and the list was traversed in round robin fashion. However, this approach only did load balancing for traffic going to Netscapes site and is not feasible with todays vast amount of available browsers and web sites.

2.3.2 Server solutions

There are many server level solutions providing redundancy and high availability for high volume services. They differ in whether they are hardware or software implemented solutions.

Equipping servers with more than one processing unit is normal for web servers serving a high volume of requests. Several CPUs can be set up in parallel or there can be dual core CPUs with multithreading capabilities. A drawback using this approach is that multiple CPUs or multithreading CPUs share the remaining machine resources. An alternative approach would therefore be to make use of computer clustering technology. This is similar to doing network load balancing in that the load is distributed between several machines. However, in this scenario single processes are distributed, i.e. a single request might be served by several machines. A governor distributes the computing load for a request to many machines which complete their assigned part. When all the subtasks have been completed, the governor returns the response. Clustering requires special operating systems, web server and application software and is more complex than the regular network server load balancing scheme.

As previously mentioned, disk speed can be a serious inhibitor for high throughput operation. Organizing disks in arrays using RAID technology and do striping to several disks significantly speeds up disk I/O.

Web server software can have features to facilitate efficient delivery of contents. Apache's `mod_cache` module enables the server to cache responses in memory for fast retrieval of that content for subsequent requests of the same type. In new versions the `mod_cache` module even enables Apache to cache dynamic contents [14], e.g. output from server side applications.

A technology that has had its renaissance the last couple of years is *virtualization* which is a way of running many operating systems simultaneously on the same hardware. This allows sandboxing of services for security and reliability. Virtualization also brings attractive features to the scene for high volume services. To shuffle the load during peak hours, extra servers can be spawned on underutilized machines. It is even possible to migrate virtual machines across physical machines [15]. During low load periods, the virtual machines can be taken down so that hardware resources can be used for other tasks. There is a lot of buzz around virtualization at the moment and a vast amount of interesting research and development is done on the subject.

2.4 Traffic statistics - characterizing the load

Science is not about truths; science is probability management [16]. Therefore statistics are at the crux of any scientific study. Especially of interest to high volume serving systems is the expected inter arrival and service times for requests. Without these entities, capacity planning easily becomes a wild goose chase - like putting a finger in the air to find out which way the wind is blowing.

Expected service and arrival times are not the only quantities of interest for capacity planners. In that a stochastic process is defined as being a result of an underlying random process and arrival of requests at a site is said to occur at random, inter-arrival times have inherent deviations in value. This can also be the case for the service time requirements of requests. It is of great importance to know how large these deviations are. Variance σ^2, also called second moment of a data set, mean $\overline{x}$ being the first, is a measure of spread in a data set. Arrival processes on the Internet have proven to exhibit large deviations in arrival times and as such deviates from other common random processes where convergence can be seen over time.

2.4.1 Arrival processes

Traditionally, the arrival of stochastic events has been modelled as Poisson processes. These processes have attractive analytical properties and are easy to deal with. However, the assumptions that Poisson modelling is founded on are not necessarily valid for all random processes. Poisson modelling of a series of events implies the following [17].

1. The population mean $E(X_i)$ exists and is finite.
2. The population variance $var(X_i)$ exists and is finite.
3. The entities in a series, $X_i, \ldots, X_n$ are uncorrelated.

There are many examples where these initial assumptions are inappropriate. For instance, for some parameters the Pareto distribution does not have a well defined mean. Following from the definition of variance, without a mean value there is neither a well defined standard deviation. Also, the assumption that events are uncorrelated, i.e. that the arrival process is *memoryless*, has proven to be incorrect for many random events in packet networks.

One might then ask if it is appropriate to call processes with correlations to previous events random. Random processes are characterized by processes where the causal relationships are so complex that it would be infeasible to make account for them all. Therefore, a process can be termed random even though correlations between values are apparent.

Inter-arrival times in the Internet have shown to be insufficiently modelled by a Poisson process. Instead other statistical properties have been widely observed - *self-similarity, long range dependence, heavy tails and burstiness*.

Self similarity: Refers to a phenomenon that is exactly similar or have similarities over several scales(See section 2.4.2 for elaboration)

Long range dependence: Long range dependence refers to the degree of correlation of a stochastic process with itself and previous or posterior measured periods. The long range dependence of a stochastic process is found using the autocorrelation function. Self-similar processes are also long-range dependent [18].

Bursty traffic: Bursty traffic occurs when packets arrive in groupings or clusters, i.e. several short inter-arrival times followed by long idle periods. Bursty traffic *may* be long-range dependent [19].

Long or heavy-tailed packet arrivals or service requirements: Long tailed service or inter-arrival time distributions occur when a large portion of the probability mass is located in the tail of the *Probability Distribution Function (PDF)*. Intuitively, this means that the variation in values for such distributions is large.

Heavy-tailed behaviour will be revisited in section 4.1.2 in conjunction with the discussion on the Pareto distribution.

2.4.2 Self-similarity

Web traffic is significantly different from other types of network traffic as each type of service can have considerably different characteristics. Web traffic can exhibit bursty behaviour. ”Bursty” refers to the fact that data is transmitted randomly, with peak rates exceeding the average rates by factors of eight to ten. It has also been observed that web traffic is bursty across several time scales. This phenomenon can be be statistically described using the notion of self-similarity [11].

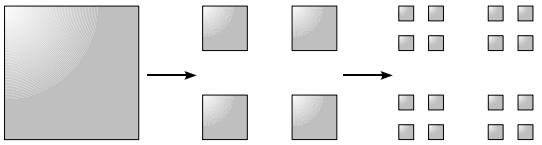


Figure 2.6: *2-D Cantor set. Created by starting with a solid or blank unit square, scaling to 1/3 its size, then placing four copies of the scaled square filling the same space as the full sized square. The figure shows a 3 level iteration. Figure adapted from [20].*

The study of self-similar phenomena dates back long before there was any operational packet switched networks. Research on self similarity and fractals were pioneered by Benoit B. Mandelbrot which studied the phenomenon in such diverse fields as fluid dynamics, economics and information theory.

Mandelbrot described phenomena where a certain property of an object, e.g. a natural image or a time series, is preserved with respect to scaling in space and/or time. If an object is self similar or fractal, its parts resemble the shape of the whole when magnified [18]. An example of a self-similar geometric object is the 2D Cantor set(see figure 2.6). The limiting object (i.e. $n =$ no. of iterations and $n \mapsto \infty$) of a cantor set has the property that if any of its corners are blown up suitably, then the shape of the zoomed in part is similar to the shape of the whole, i.e. it is self-similar.

For network traffic, the phenomenon manifests itself in a similar manner. Whereas there is exact resemblance for the deterministic fractals as the 2D cantor set represents, complete resemblance is not observed in self similar network events. Instead, one speaks of the level of similarity over different timescales.

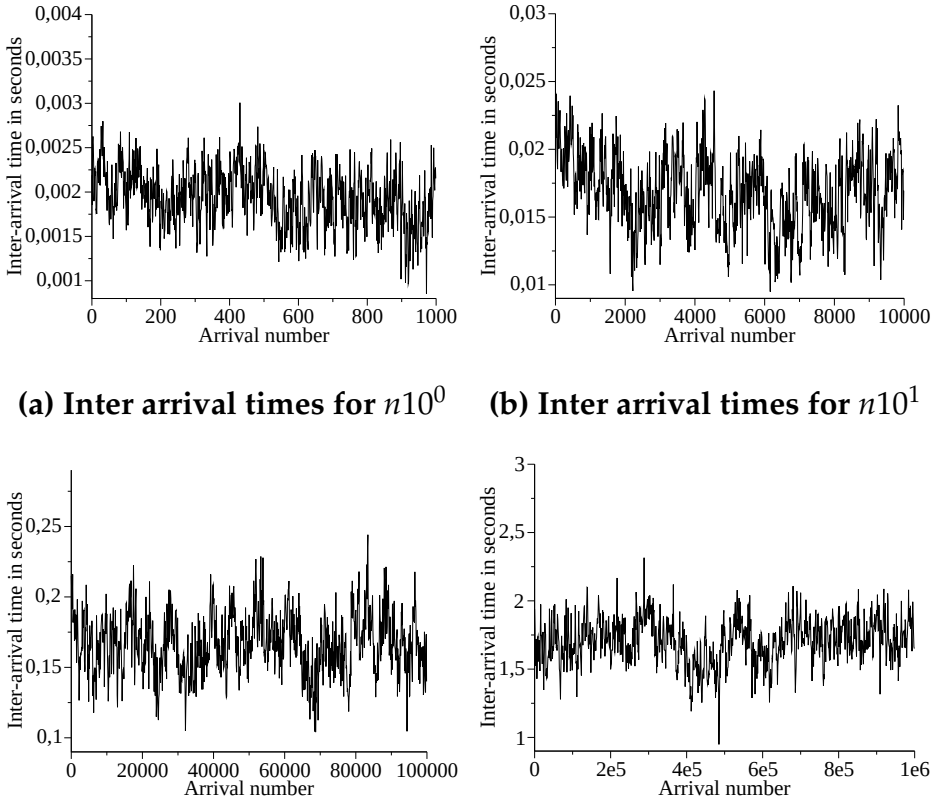
A common measure for approximating self similarity is the Hurst exponent[7] which is a dimensionless scalar. It typically takes on values in the range $0 \leq H \leq 1$.

- If $0.0 \leq H < 0.5$, the process is self similar with degree H and *Short Range Dependent*.
- If $H = 0.5$, the process is completely random, i.e. Poissonian distributed.
- If $0.5 < H \leq 1.0$, the process is self similar with degree H and *Long Range Dependent*.

I.e., as $H \mapsto 1$ or $H \mapsto 0$ the process gets more and more self similar. In network traffic one is generally concerned with the values ranging from 0.5 to 1.0. Self similar processes with $0.5 < H < 1.0$ have correlations with previous events, thus are referred to as *Long Range Dependent* (LRD). Poisson processes are so called memoryless and corresponds to $H = 0.5$.

Burst in network traffic causes troubles. First and foremost it causes transmission delay as packets need to wait in buffers. Secondly, if the buffers are completely full, we get packet loss. Loss of packets requires TCP retransmission, thus diminished transfer rate. In practice, buffers are overprovisioned in order to prevent losses. However, packet delays are perceived by users an unresponsive browser [21], thus poor QoS.

[7]Named after the British hydrologist H. E. Hurst.



(a) Inter arrival times for $n10^0$ **(b) Inter arrival times for $n10^1$**

(c) Inter arrival times for $n10^2$ **(d) Inter arrival times for $n10^3$**

Figure 2.7: *Graphs of a time series over different timescales. The graph shows inter-arrival time against its connection number. The series was made using the PERL module* `Math::Random::Brownian` *(see section 2.5.3) with a Hurst exponent of 0.9. The similarity is most prominent for (b) and (c) and more inconspicuous for (a) and (d). However, the 4 graphs clearly share equal properties.*

Whether or not a time series can be described in terms of LRD, heavy tailed distributions or memoryless has important implications for our ability to deal with it analytically. For a particular time series the variance, and even the mean, might not be well defined, which obviously complicates numerical analysis. Analysis plays an important part in capacity planning for a web site. Thus, taking shortcuts, automatically assuming that network processes can be treated as Possionian, can have undesirable consequences and lead to a poorly performing web service incapable of complying to service level agreements.

2.5 Software

As we intended to investigate server performance and queueing models under varying traffic conditions, a test suite capable of creating such conditions was needed. Several HTTP benchmark tools where considered. All of the assessed tools were surprisingly enough not able to meet our specific needs which necessitated implementation of a custom designed testbed.

2.5.1 Traffic generator

There was a need for a fast, light-weight HTTP able to perform experiments with arbitrary inter-request and service time distributions. The existing tools for benchmarking are often limited to making request with *deterministic* or *exponentially* distributed inter-request time. Examples of these are `httperf`, `ab` (which is a part apache utilities) and `hammerhead`. `S-Client`, used in [22], was only available for platforms other than Linux[8]. `SURGE` [23] is supposedly able to generate self-similar traffic and file retrieval based on Zipf's law [9]. However, there was no provision for alternating the service requirement for dynamically generated web pages. Maintenance of `SURGE` seems also to have been disrupted as the latest version 1.0 only has been tested under Linux kernel 2.0 with `gcc` compiler v2.7.2. It did not compile cleanly using the latest `gcc` version, and because correct operation neither could not be trusted on kernels under the 2.6 kernel tree, `SURGE` was abandoned. `Harpoon` and `D-ITG`[10] had promising functionality on paper, but proved to be awkward in use. `D-ITG` operation relies on both a `D-ITG` client and a `D-ITG` server, thus is unsuitable for testing of web servers running regular commodity web server software.

Because of the inability of the existing software to meet our needs, a custom designed test suite was implemented. This simply consisted of a small HTTP client emulator and a server side application for client request processing. The HTTP client

[8] Digital Unix and FreeBSD.
`http://www.cs.rice.edu/CS/Systems/Web-measurement/sources.html`

[9] Zipfs law refers in web context to the popularity of files. It states that if files are ordered from most popular to least popular, then the number of references, P, to a file tends to be inversely proportional to its rank,r: $P \propto r^{-1}$[23]

[10] Distributed Internet Traffic Generator

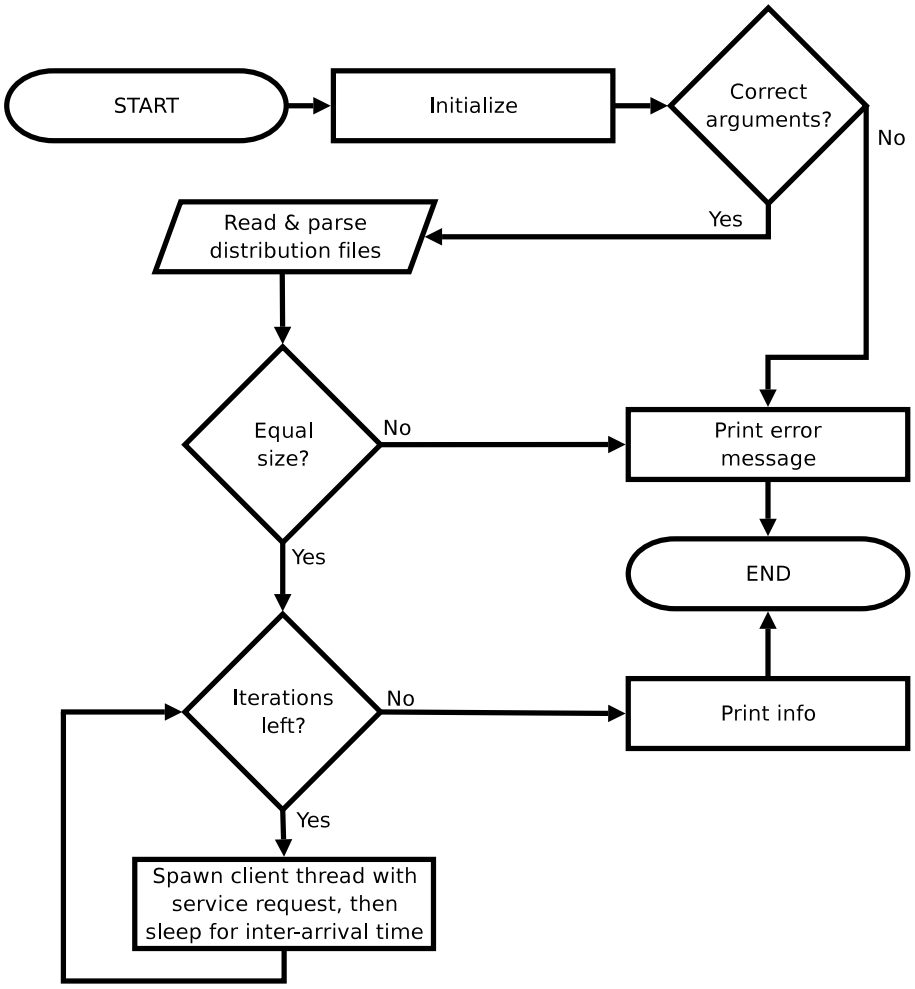


Figure 2.8: *A simple flowchart describing the structure of the small multithreaded client.*

emulator, simply called `client`, was written in C as C are known to be one of the fastest programming languages.

The client application starts of by reading two files respectively containing distributions of arrival-times and service demand. The distributions can be of any kind, the only requirement is that the number of entries matches. Each inter-arrival time/service demand pair constitutes one client request. HTTP requests are issued by using threads which are procedures that runs independently from its main program. Hence, there is no need for the client to wait for one HTTP request to finish before it can issue a new. This facilitates accurate timing resolution for inter-arrival times. The threading functionality is implemented using POSIX threading library `libpthread` included in the C development library `libc6-dev` under Linux. The client can run several threads in parallel and is therefore multithreaded.

Each thread is an emulation of a HTTP client and makes a GET request with GET-parameter called `"iter"` with an associated integer value. It is through the `"iter"` parameter that service requirement by the server can be varied. The request is hard coded into the application meaning that the particular request and its parameter cannot be changed, only the parameter value. Therefore the server side application has to be able to receive a GET parameter called `"iter"`. This calls for low application versatility, something that will be fixed in future versions.

After spawning of each thread the application sleeps for a given amount of time using the `select()`[11] system call. Using application sleep to induce desired inter-arrival times is a method also used in `S-Client`. When the client has iterated through the whole length of the distributions, it prints out information about how many threads it started and then exits. An application flowchart is shown in figure 2.8.

The client has four flags, 3 of them are compulsory: `-h` specifies the server subject to testing, `-s` is the service demand distribution and `-a` is the inter-arrival time distribution. The last, `-p`, is optional and if specified the client prints information about each request to standard output, containing the reply from the web server and the response time measured by the client.

It's important to note that full HTTP protocol functionality is not implemented into the client. It does not care about HTTP status codes or how to react upon them. Therefore, transfers have to be investigated in posterior to determine whether or not there have been successful transactions, e.g. by checking output from the client or traffic capture files. Because of the lack of HTTP functionality it will not be able to do pipelining of requests. In that respect, it will work similar to HTTP/1.0 without `keep-alive`. In addition, it is not able to fetch inline objects like images or multimedia. The client is thus only able to fetch on specific resource, which is sufficient for our experimental purposes. The source code for `client` is listed in A.1.1.

[11] It might seem strange that the application sleep periods are not provided by the `sleep()` or `usleep()` system calls. The reason for not using this is explained in section 7.5.

2.5.2 Server side application

Server side load generation was done using a simple PHP (Hypertext Preprocessor)[12] script. It parses the GET requests and extracts the value of the `"iter"` variable. The main part of the server side application is a `for`-loop that does nothing but iterates as many times as specified in the `"iter"` variable. A timestamp is recorded before and after the `for`-loop and the difference, i.e. the processing time, is sent back as response. By varying the `"iter"` variable the server will receive jobs of different service requirement. Apache makes use of PHP through loading a module on startup; the module `libapache2-mod-php4` in version 4.3.10 in our experimental setup. The source code for the PHP script is listed in section A.1.7.

2.5.3 Distribution generators

Neither the client nor the server side application is able to determine inter-arrival time and service demand for requests on their own. These have to be generated by external applications or scripts, and written to the files that the client takes as arguments. The method for generation of distribution files depended on the desired statistical properties and was done in two different ways.

GSL: The GNU Scientific Library is library for C and C++ programmers providing more than 1000 functions for various numerical operations [24]. Amongst these there are routines such as random number generators, special functions and least-squares fitting. The developers of GSL have implemented a frontend for the random number generator routines called `gsl-randist`. This small program is able to create random number sets conforming to various types of distributions, e.g. *Pareto* and *Exponential*. A custom made wrapper script in Practial Extraction and Report Language(PERL) enabled generation different distributions with equal first moments. The wrapper script is listed in section A.1.5.

Math::Random::Brownian: For generation of self-similar distributions, functionality provided by this PERL module was exploited. The module is able to generate *fractional Gaussian noise*(fGn) and it is cumulative sum *fractional Brownian motion*(fBm). In general, fGn processes are self-similar process with long range dependence [18]. The module is a frontend for C routines developed in conjunction with work done in [25]. As with GSL, the distribution generation by the module was controlled through wrapper PERL script and the source code is listed in section A.1.6.

2.5.4 Queue simulators

Simulators where needed to evaluate queueing models ability to predict response times. Several queueing simulators are available, e.g. the Microsoft Excel plugin `QtsPlus` and the `QtsPlus4Calc` extension to the OpenOffice Calc suite. However, it

[12]The acronym PHP actually stems from the earliest version of the program called "Personal Home Page Tools"

was unclear which queueing models they relied on and were therefore abandoned. Instead, the simulations based on formulas were facilitated through a script. The PERL script `q_sim.pl` is able to calculate expected response time for a queue under given traffic conditions and is based on well known formulas presented in section 4.2. Script source code is found in section A.1.4.

Formulas for distributions, e.g. Exponential and Pareto, are based on a limit condition. In other words, a sample set of n values of a certain distribution type will conform to the distribution mean and standard deviation values as $n \mapsto \infty$. For the experiments conducted herein, the series of arrivals and service times are finite. A consequence of this, since the values of the sets are random variables, is that the statistical properties of the distributions might not manifest themselves. In order to test response time for a queue with finite number of requests, a simulator that does not rely on limiting formulas was needed. Therefore, a simulator, `simulate_distributions`, based on *hand simulation techniques* [26] and implemented in C, was constructed in order to replay the experiments done against the server. The application works similarly to the client. However, instead of making real HTTP requests, an algorithm calculates the response time value based on inter-arrival time and service demand. The results from the simulator can then be compared with experimental results to determine whether or not it is fair to consider a web server as one single queueing system with a *First Come First Served(FCFS)*[13] serving discipline. In addition, the formula values can be compared to that of the simulator to check the validity in a fully controlled environment. The founding theory for the simulator is presented in 4.2.2 and the source code for the application is listed in A.1.2.

2.5.5 Data collection

The versatile traffic capture tool `tcpdump` was used to capture traffic data between client and server for post-experiment analysis. `tcpdump` is based on the `libpcap` library and is available for most operating systems. Trace files of traffic is extremely useful for posterior analysis as every conceivable detail of client/server conversations are recorded. Versions of `tcpdump` and `libpcap` where respectively 3.9.1 and 0.8.3.

2.5.6 Data extraction

Since data collection was facilitated by `tcpdump`, an application capable of parsing and extracting the desired entities was needed. Specifically of interest were the TCP session duration times as these represents the response time for the system. A lot of searching was carried out to find existing software with these capabilities, however it turned out that very few capture file analysis applications provided the functionality needed. The application closest to fulfilling our needs was `tcptrace`. It is able to extract TCP session times, however was found to be awkward to use and also it lacked other required functionality. Instead, a PERL script,`doAnalysis.pl`, with the

[13] FCFS serving discipline is assumed by the queueing formulas. This is explained in section 4.2.

use of the modules `Net::Pcap` and the `NetPacket` decode/encode modules, was implemented and provided the functionality sought after.

`doAnalysis.pl` analyzes every packet and looks for start and end of TCP sessions by inspecting the TCP flags. In addition to extracting response time, functionality for inter-arrival distribution and service requirement extraction was also implemented in order to replay each experiment in a simulated environment. The script source code is listed in section A.1.3.

2.5.7 Other

Other software used in conjunction with this thesis is presented briefly in this section.

Visualization and plotting tools: `xmgrace` was used for plotting and visualization of data. The distribution self-similarity tool `SELFIS` was used for calculation of the Hurst exponent of the various data sets.

Web server software: The web server software used in the work herein was the industry standard `apache2` in version 2.0.54.

In addition to the mentioned applications and scripts, a plethora of small PERL and Bash scripts were implemented for various purposes. These are not elaborated on herein as they were less central for the thesis work.

2.6 Previous research

Contributions made through the years to the network traffic characterization and performance research fields are of an astonishing amount. This section will give an overview of some of the most important and prevalent work done within these fields.

2.6.1 Traffic characterization and access patterns

In the late 1980's and early 90's, astonishing discoveries where made at Bellcore labs regarding the nature of packetized traffic which necessitated a shift in our understanding of networks. Until then, it was believed that traffic in packet networks exhibited the same characteristics as in its circuit switched counterpart. The old telephone network, which has been around for over 100 years, is well understood and its design is a highly refined discipline. Knowledge collected from research and engineering activity has enabled construction of networks capable of providing any level of service [2]. Thus, it was natural to adapt the existing knowledge about networks to the new packetized network paradigm. However, by doing so networks are only modelled correctly in a limited number of cases as indicated by the discoveries at Bellcore and subsequent research. This section is aimed at giving an overview of the work done in efforts to characterize network traffic.

Self-similar and heavy-tailed network traffic

In 1994, Leland et al. [27] published what would prove to be a seminal paper elaborating on the findings at Bellcore Labs. In the period August 1989 to February 1992 they measured Ethernet traffic on different segments of the Bellcore network using custom designed tools. By capturing timestamps and header information they could analyze packet per time unit distributions and compare them to synthetic traffic from an appropriately chosen Poisson model. The analysis revealed that the traffic exhibited self similar characteristics; a phenomenon that is not well captured by a Poisson model. These revelations shook the well established ideas of network traffic being dominated by Poisson processes and gave birth to a vast amount of subsequent research on the subject.

Raatikainen [28] did investigations similar to [27] and measured the arrival times of Ethernet frames to a file server. However, the results of this study where more inconclusive. Ethernet frame arrivals did indeed prove to have self similar properties for some of the measurement periods. But for half of the periods there could not be tracked any evidence for or against self-similar characteristics.

The studies conducted in [28] and at Bellcore considered traffic at the Local Area Network(LAN) level. Paxson et al. [29] brought the idea further and investigated traffic characteristics in Wide Area Network(WAN) TCP traffic. They found that for application layer protocols like TELNET and FTP, connection arrivals were modelled quite aptly by Poisson. However, the nature of data transfer proved to be very different from a Poisson process. They found that data transfers exhibited a bursty nature consistent with long-range dependent behaviour.

Whereas arrivals user initiated sessions for traffic types described in [29] could well be modelled by a Poisson process, the picture is different for HTTP traffic. Web traffic has more structure to it than most other types of traffic. Web documents can contain a variety of inline objects such as images and multimedia. They can consist of frames and client side script. Different versions of HTTP (i.e. version 1.0 and 1.1) coexist and interact. Implementations of the TCP/IP stack might behave slightly different depending on the operating system. Users stopping in the middle of the transfer and users having multiple browser open at a time also has significance for the behavior of HTTP. In addition, servers and browsers from different vendors behave differently and have different parameter values. All this volatility leads to no single or quintessential template of a Web interaction [30]. Network traffic can therefore be expected to behave different for HTTP than for other application layer protocols.

Crovella et al. [31] took upon themselves to find probable causes for observed self-similarity in HTTP traffic. They traced the self-similarity along two threads. Firstly, an investigation of the size distribution of files available on the net was conducted. It was found that the distribution of file sizes where heavy-tailed. Because of the operational mode of TCP, transmission of large files results in low packet inter-arrival times as responses to requests are pushed out as fast as possible, i.e. there is a traffic burst. As noted in [20], the heavy-tailedness of certain network variables, such as file sizes, can be the root cause for observed self-similarity in network traffic. Crovella et al. also found from browser logs that user think times, i.e. the idle period between subsequent page requests, where heavy-tailed distributed. User think time was how-

Invariant	**Protocol level**	**Distribution**
Session arrivals	Session	Poisson
Session duration	Session	Pareto
Session size	Session	Pareto
WAN Traffic at TCP level	Transport	Self similar
TCP connections/Web session	Transport	Self similar
Inter-arrival time of packets	Network	Heavy-tailed
Inter-arrival time of packets generated by user at keyboard	Network	Pareto
Inter-arrival time of Ethernet frames	Network	Self-similar

Table 2.1: *Summary of characteristics found from research done in the late mid 90's. The table is adapted from [32].*

ever not decisive and they concluded that the observed self-similarity of inter-arrival times could mostly be attributed to distribution of file sizes.

Crovella et al. followed their line of reasoning in subsequent work [21], though broadening the perspective a bit. The chief concern here, like in [31], was whether or not the observed self similarity in network traffic could be attributed user requests or the distribution of available file sizes. In contrast to their previous work, they tried to take into account the caches that are present at various points in the network between a client and a server. Evidence was found suggesting that *client caching* substantiated further distribution of file sizes as the main cause for self similarity in network traffic. Caching has the effect of making the set of actually transmitted files distributionally similar to the set of available files. Hence, the set of transmitted files are relative insensitive to the particular requests made by users.

Mah [33] investigated the HTTP reply sizes by doing measurements on clients. The mean file sizes where much larger than the median file sizes which is consistent with distributions of reply sizes that are heavy-tailed. Hence, the findings were in accordance with [31, 21]. The HTTP request sizes were found to have a *bimodal* distribution as requests generally are either small GET requests for simple file retrieval, or POST requests with possibly lots of transmitted data through HTML forms.

The nature of a HTTP transaction depends on which version of the protocol is used. As described in section 2.1.2 HTTP versions 1.0 and 1.1 differs in the use of persistent connections. Prior to the release of the final HTTP/1.1 draft, a study was made to investigate the implications of connection persistence for traffic characteris-

tics [34]. In this work, an implementation of the preliminary HTTP/1.1 specification tested against the existing 1.0 version to uncover differences in traffic characteristics. Proof was found that persistent connection improved latency times to a certain degree. However, the traffic characteristics appeared to be the same.

In a fairly recent study of the Abilene network [14] there were also found evidences of heavy-tails in HTTP traffic [35]. The particular result of interest in this paper is the number of servers with which each client communicates in server-to-client and client-to-server connections, and the number of clients handled by each server. The values obtained in this study were all of a heavy-tailed nature and exhibited standard deviations two or three orders the magnitude larger than the mean values. Although these results seemingly investigated properties of web traffic fundamentally different, it just goes to show that the heavy tails of Web traffic systems are quite pervasive.

Although there seems to be evidence that predominantly indicates heavy tails and self-similarity in network traffic, [36] claims that the large-scale aggregation in the Internet core causes a shift in behavior towards Poisson process. This is substantiated by research done in [37]. There is evidence that Poisson models could still be applicable as the number of sources in backbone links increases, leading to large volumes of traffic multiplexing. Karagiannis also points out limitations and caveats in our ability to detect self-similarity. The prime methods for long range dependence detection are the Hurst estimators. As the name indicates, this only provides an estimate, and the methods for doing this estimation has shown to perform poorly in particular cases, and can also produce contradictory results.

Research on this matter is inconclusive, even though the majority of studies concludes with pervading self-similarity and/or heavy-tailedness in Web traffic. For the purpose of our work we only assume that network traffic *can* exhibit behavior different from a Poisson process. More specifically, we assume that the web session request process can exhibit self-similar and heavy-tailed properties.

Modelling network traffic

Doing research on the nature of packet traffic has proven to be extremely difficult. Because of the large number of factors involved it is extremely difficult to realistically recreate network traffic in a controlled testing environment. This obviously impedes scientific research. Network traffic is shaped by a plethora of influencing factors as it is a result of an interplay between protocols, software, humans and networking and computing infrastructure. Therefore, simulating conditions of the World Wide Web in a controlled experiment environment is a significant challenge. A significant amount of literature is all the less produced on the subject of traffic modelling and realistic web load generation.

A process which is known to be self similar is *fractional Brownian motion*(fBm). Such a process can be generated directly using the inward recursive *random midpoint displacement algorithm*. It subdivides intervals in the time interval $[0, T]$ and the value of a midpoint in an interval $[t_n, t_{n+1}]$ is constructed from the values of the interval

[14] Also known as *Internet2* which is a research network connecting, at that time, about 200 universities world wide

endpoints. As this algorithm is based on precomputed values for start and end times, it cannot work as a generator in real time [38].

Another way of generating fBm is to use the Hosking method. Here, the process values are drawn from a standard normal distribution whose first and second moments are computed based on quantities that depends on all previous values of these quantities [38].

In recent work [38] it was found that the best generator with respect to a desired value of the Hurst exponent is multiplexing of several heavy-tailed ON/OFF sources. The OFF periods corresponds to idle times and ON periods corresponds to transmission of traffic at peak rate. The length of the on and off periods is heavy-tailed distributed. It was found that for a number of such sources, preferably 100 or more, one could accurately generate traffic with desired level of self-similarity. In addition, as this process is not dependent on the precomputed or preset values, the generator works well for real time generation. This idea was explored in earlier works as well [39, 40].

Several other methods for traffic generation where considered in [38]. However, they were all found to be inferior to the method of multiplexing heavy-tailed ON/OFF sources.

2.6.2 Queueing performance and traffic characteristics

Client/server communication might require participation of many interconnecting nodes. Each of these nodes performs a service, whether it is routing, switching or processing of web requests. Hence, all of these systems can be analytically described and treated in terms of queueing theory.

Queueing performance is strongly affected by the nature of network traffic. For routers, bursty traffic fills up queueing buffers routers with packets waiting to be processed. Traffic bursts arriving at web server systems can cause request waiting queues to grow significantly. In general, it was noted in [18] that the presence of self-similarity with large values for the Hurst exponent in network traffic can have severe performance effects for networks with significant buffering. Under bursty traffic conditions transmission channels can be overloaded. The implications of this are that packets have to be buffered while waiting for a vacancy of the channels. This obviously degrades performance with respect to response time which consequently can lead to poor QoS.

Erramilli et al. [41] studied the effects of long range dependent traffic on queueing systems and found that long range dependence had measurable and practical impact on queueing systems. They also found that the nature of network traffic had significance for a number of packet traffic engineering problems, such as buffer sizing, admission control, and rate control. It was also noted that if traffic characteristics is ignored it can lead to overly optimistic performance predictions, which network resource allocation and capacity planning is based upon.

Other research studied queues with application to specific server operation. In [42] the operation of the omnipresent Apache web server software was modelled. Apache can operate with several processes running at once, several threads running once,

or with a combination. Therefore, modelling regular, one processor web system job scheduling as FCFS, an assumption a lot of simple queueing theory relies on, is not correct. Also pointed out in by this work was that neither service nor arrival processes is necessarily Poissonian. A sophisticated model taking general service times and number of processes/threads running in processor sharing fashion was therefore derived - the $M/G/1K*PS$[15] queue. The purpose of this model was to study the blocking probability of the web server, i.e. the rejection probability for requests given a certain HTTP listen queue size. Results from simulation with the model and experimental results with Poissonian arrivals and general service time distribution proved to be fairly congruent. This work further substantiates the necessity of a more elaborate model than the ones relying on assumptions of Poisson processes for web server systems.

2.6.3 Server performance modelling

To achieve a desired level of quality of service under any traffic conditions, extensive knowledge about how web server hardware and software interacts and affects each other is required. Knowing the expected nature of traffic and queueing system only tells us how we might expect a system to perform; knowledge about software and hardware interaction enables performance tuning.

Slothouber [9] derived a simple model for considering web server systems founded on the notion of serial queues. Several components interact during a web conversation and a request goes through different stages, with different queues at each stage. The response time is therefore an aggregate of the times spent at different levels within the web server.

Mei et al. [8] followed this line of reasoning in a subsequent paper. One of the focal points in their work was to investigate the effects of congestion in networks for response time and server blocking probability. High end web servers usually reside on a network with excess bandwidth to be able to shuffle load at peak rates. However, customers inherently have low rate network connections with possible asynchronous transfer mode, such as ADSL lines. Therefore, returning ACKs in from client to server can be severely delayed. In turn this causes the request to linger for a longer time in the system than would be the case if the connecting network was of high quality. Aggregation of such requests could eventually cause the TCP and HTTP listen queue to fill up, making the server refuse new connection requests even if it is subject to fairly light load.

2.6.4 Quality of Service

What is perceived as good quality of service is not objective, it depends on individuals and their expectations towards a service. An extensive case study investigating user-perceived quality[43] of service revealed that one can not infer an appropriate QoS solely based one single response time value. Human psychological mechanisms

[15] Queueing notation is explained in section 4.2

Perceived quality	**Non-inc. loading times**	**Inc. loading times**
High	$0 - 5s$	$0 - 39s$
Average	$5 - 11s$	$39 - 56s$
Low	$> 11s$	$> 56s$

Table 2.2: *Table showing page loading times and for incremental and non-incremental loading times. Results are adapted from [43].*

are highly relevant for the perceived level of service. The least inconspicuous element with regards to this is the design and layout of the web page as a soothing appearance generally gives the impression of a high quality product. A more surprising result of the study was that forbearance for transfer delays was also found to be related to mental constructs. Users tolerance for delays decreases for subsequent page requests during a single session. Thus, the QoS demands are not stationary, not even for a single user in a single web interaction. It was also indicated that the type of page loading had impact on how the quality was characterized. It was found that incremental page loads increased tolerance by a factor of more than 7. Incrementally loaded pages are partly loaded before the request is completed and therefore gives the illusion of better responsiveness than non-incremental page loading does. The findings clearly show that there is room for exploiting psychological mechanisms for providing a high level of user perceived QoS.

Chapter 3

Objectives

Inspired by previous work done in the area of traffic characterization and web server performance, we investigate several issues. Research has shown that inter-arrival times cannot be expected to conform to a simple distribution with simple properties in every case. Nor can service time requirements, as some requests might demand a significant server sojourn time. The first hypothesis that we investigate relates to this.

> **Hypothesis 1** *The average response time of a series with HTTP requests is affected by the distribution type and variance of inter-arrival time and service time requirement.*

Analytical treatment of queueing systems that do not conform to Poissonian models of random events is complex. However, research performed by queueing theorists has resulted in simple response time formulas for queues with general inter-arrival and service time characteristics. Given knowledge of the nature of service and inter-arrival times, such formulas could be helpful for site capacity planning. In some cases these formulas only yields approximations and we investigate the ability of such a formula to predict response time for a single web server system.

> **Hypothesis 2** *The simple formula 4.25 for calculating average response times of queues is well suited for predicting average response times of a single web server system.*

The basis for comparison of formula and experimental results is the assumption that web servers can reasonably well be modelled as a single queueing system with FCFS scheduling. This is clearly a simplification. A web server system is not only comprised of several queueing systems working in serial; it also serves request using a processor sharing (PS) discipline. A simulation of a FCFS queue using the same input as in the experiment might give answer to whether or not the simplification can be justified.

> **Hypothesis 3** *A web server, even though it is comprised by several queueing systems in serial, can be modelled as one, single queueing system with FCFS serving discipline.*

The ultimate goal of these studies is to arrive at some conclusion or rule of thumb for service providers with regards to capacity planning for SLA compliance. Such

agreements are often based on probabilities for delivering a service within a certain target. We believe that average value formulas are not suited as estimators for such agreements.

> **Hypothesis 4** *Average response time formulas, independent on their accuracy, are not suited for doing capacity evaluation in order to meet Service Level Agreements.*

Doing measurements of first and second moments of inter-arrival time at a site might lead to the conclusion that traffic exhibits Poisson behaviour. However, unaccounted for by these quantities is the ”dependence” of between arrivals, i.e. whether inter-arrival times of the same type are ”clustered” certain periods of time. We therefore set out to investigate response time when the system is subject to varying degree of self-similarity.

> **Hypothesis 5** *The average response time of a web server system is affected by the degree of self similarity in request inter-arrival. More specifically, higher self-similarity degree in input traffic results in higher average response time.*

Chapter 4

Theory

This chapter will give an overview of the theory needed to investigate our objectives. Some basic statistical theory followed by a discussion of distributions. This leads up to queueing theory employed in this study. The chapter is concluded with some remarks about how to estimate the self-similarity in distributions.

4.1 Statistics

Statistics is one of the most valuable tools for scientists trying to describe an observed phenomenon. The most central statistical quantities of a dataset are the mean, variance and the standard deviation.

4.1.1 Mean, variance and standard deviation

The most prominent quantity for any measurement series is the *expectation value* or *mean*. The mean of a dataset is defined as the sum of its values divided by its value count, i.e.

$$\overline{x} = E[x] = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.1}$$

The mean is an abstract value and might not even be amongst the dataset values. Its main purpose is as a quantity characterizing the measured data - a *typical* value for the dataset. It is often termed the *first moment* of a dataset.

The mean value only provides a coarse description of the dataset. Two measurement series can have the same mean value, but exhibit very different properties. Consider two sets of data, a set of N values all with value X, and a set with N values where half the values is 0 and the second half is $2X$. These datasets have the same mean, but completely different distribution of the dataset values. The second data set exhibits a spread in values while the first has none. Therefore we define the *second moment* of a set of data - the *variance*.

The variance is a measure for the spread in the measured data. It's defined as follows.

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\overline{x} - x_i)^2 \tag{4.2}$$

The variance is the sum of all the squared differences between the mean value and the measured values divided by the number of values minus one. The variance therefore tells us how much the measured values differ from the mean value, i.e. how *representative* the mean value is for the dataset.

From the variance we can also derive a third statistical quantity of common interest. The *standard deviation* is simply the square root of the variance.

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\overline{x} - x_i)^2} \tag{4.3}$$

For normally distributed dataset with mean $\overline{x}$ and standard deviation σ, 68% of the dataset values will be in the interval $[\overline{x} - \sigma, \overline{x} + \sigma]$.

4.1.2 Distributions

A usual way of characterizing random processes is by statistical distributions. Distributions are theoretical constructs that assist in the prediction and analysis of stochastic processes. Statistical distributions are central to queueing models as they describe the nature of the arrival and the service process. The herein studied statistical distributions and their properties are presented in the following.

Exponential

The exponential distribution is the simplest and most common distribution for stochastic processes. It has simple statistical properties and it is therefore normal to assume that stochastic processes have an exponential distribution of inter-arrival times. Exponential modelling of inter-arrival times has been successfully been done in areas like fault modelling. For the exponential distribution we have the following functional relationships [44].

- Probability Density Function: $f(t) = \lambda e^{-\lambda t}$
- The probability of a random variable T being larger than a number t: $P[T > t] = e^{-\lambda t}$
- Expectation value (mean): $E(T) = \frac{1}{\lambda}$
- Variance: $\sigma^2 = \frac{1}{\lambda^2}$

Arrival processes that have an exponential distribution of inter-arrival times are termed *Poisson* processes. The λ seen in the expressions above is the rate of the Poisson process.

The Poisson distribution is concerned about the discrete random occurrences over an interval, i.e. rates. The exponential distribution is continuous and describes the time between those random occurrences.

Pareto

Pareto is another widely used distribution for stochastic variables. It has been utilized in several research areas, e.g. in sociology to model the distribution of wealth [45]. For the Pareto distribution we have the following relationships [46].

- Probability Density Function: $f(t) = \frac{\alpha\beta^\alpha}{t^{\alpha+1}}$
- The probability of a random variable T being larger than a number t: $P[T > t] = \left(\frac{\beta}{t}\right)^\alpha$
- Expectation value (mean): $E(T) = \frac{\alpha\beta}{\alpha-1}$
- Variance: $\sigma^2 = \frac{\alpha\beta^2}{(\alpha-1)^2(\alpha-2)}$

α is called the *shape parameter* and defines the slope while β is called the *location parameter* and defines the lower bound for distribution values. As previously mentioned, many of the phenomena in networking can best be modelled using heavy-tailed distributions. Pareto distributions *can* be heavy-tailed, but do not have to be. A random variable has a heavy-tailed distribution if

$$P[T > t] \sim ct^{-\gamma},\ t \mapsto \infty \tag{4.4}$$

where $0 < \gamma < 2$ and c is a positive constant [18]. If we rewrite the expression for probability for Pareto distributions we see that the Pareto distribution is similar and for $0 < \alpha < 2$ it is heavy-tailed. We also note that for $0 < \alpha < 2$ there is no defined variance and for $0 < \alpha < 1$ the expectation does not exist.

4.1.3 Linear regression

In general, regression analysis is a method for modelling the relationship between a variable Y, often called response variable or dependent variable, and other variables $X_1, \ldots, X_n$, often called predictors or input variables. If the regression analysis involves more than one response variable it is called *multivariate regression*.

The simplest case of regression analysis is *linear regression* of a straight line. For this kind of regression one assumes a relationship between one dependant variable with one independent. In other words, there is a relationship on the form $y \propto x$. This can be expressed as follows.

$$y(x) = ax + b \tag{4.5}$$

where a is the slope and b is the intercept point with the y-axis.

Deriving this relationship from a dataset is done through the *method of least squares* [47]. For a data set of n samples from a $\{(X, Y)\}$ population[1] the following formulas give estimates of the slope and intercept point.

$$a = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{4.6}$$

$$b = \overline{y} - a\overline{x} \tag{4.7}$$

$\overline{x}$ and $\overline{y}$ are the means of x and y respectively and are calculated as explained in section 4.1.1. Linear regression will be employed later in this thesis to obtain system specific parameters.

4.2 Queueing theory

Queueing theory is important for computational analysis of any queueing systems. Queueing systems exist in a multitude of areas including traffic, retail, networking and IT systems. Queueing theory is the primary tool for understanding how well these queueing systems perform.

4.2.1 General concepts

A queue is characterized by as many as 6 properties and is denoted in Kendall notation $A/S/c/B/K/P$ [16].

A	The probability distribution for inter-arrival time.
S	The probability distribution for service time.
c	The number of parallel service channels.
B	The restriction on system capacity, i.e. the buffer size.
K	The maximum population size.
P	The policy or scheduling discipline if it exists.

Some queue notation only uses the three first symbols of the Kendall notation. If K,B and P is omitted it should be perceived as a queue where there is no limitations on queue capacity and population size ($K = B = \infty$), and the scheduling discipline is FCFS [48].

[1] Also called a *bivariate population* [47]

A and S	*c*	*B*	*K*	*P*
M - Exponential *D* - Deterministic E_k - Erlang H_k - Hyper-exponential *G* - General	$[1, \infty]$	$[1, \infty]$	$[1, \infty]$	FCFS - First Come First Served LCFS - Last Come First Served RSS - Random Selection for Service PR - Priority PS - Processor Sharing GD - General Discipline

Table 4.1: *Table of values for A/S/c/B/K/P queueing systems.*

The service and arrival times can be distributed in various ways and this affects the efficiency of the queue. Also of importance is the number of servers and the serving discipline. The buffer size, or queueing capacity, is decisive for whether or not a service request is able to enter the queue. In general, the population size is considered to be infinite and is therefore of little interest for analysis using queueing models. With respect to web server systems, this assumption is certainly appropriate as the possible number of requests is infinite. In table 4.1 a listing of some possible values for the queue entities is presented.

If a request for a service does not enter the queue upon arrival it is said to have *balked*. In a web server context this happens when the web server queueing buffer is full. If a request enters the queue, but after a while leaves it, the request is said to have *reneged* [48]. For web server requests this happens when a request is discontinued, e.g. due to impatient users deciding to leave the site.

Users are likely to renege or balk poorly performing queues, i.e. queues with a low perceived Quality of Service. Response time of a queue is a measurable closely related to QoS and is therefore a quantity of great interest when evaluating the queueing system performance for the purpose of capacity evaluation and planning.

The M/M/1 queue

The classical queue in queueing theory is the $M/M/1$ queue. Because of its computational and conceptual simplicity, this is the queueing system that novice queueing theorists usually first get acquainted with.

This queue has exponential service and inter-arrival times and consists of only one serving unit. Also, the last three symbols are omitted which implies infinite system capacity and population size, and FCFS serving discipline.

For this queue it is possible to derive simple computational methods to investigate its performance under different conditions. Consider the state diagram for an $M/M/1$ queue depicted in figure 4.1. The number of jobs in the system, given by the

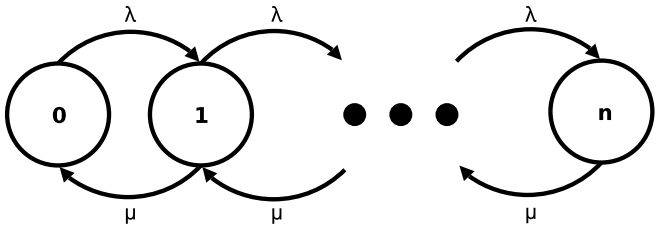


Figure 4.1: *State transition diagram for a queueing system. Each state is represented as a circle. The numbers inside the circles denotes which state the system is in, i.e. how many jobs that currently resides in it. Transitions are denoted by arrows with rates λ and μ.*

numbers within the circles, at a given time is a result of the rate at which jobs arrive (λ) and are processed (μ). A reasonable assumption about such systems is that the flow of transitions going into a state n must be equal to the flow of transitions going out of that state. This assumption is the *flow equilibrium equation*. This assumption allows us to derive functional relationships between the incoming/outgoing rates and probabilities of the system being in a state [11].

$$
\begin{aligned}
p_{n-1}\lambda &= p_n\mu && (4.8)\\
p_n &= \rho p_{n-1} && (4.9)\\
p_n &= \rho^n p_0 && (4.10)
\end{aligned}
$$

The quantity ρ is called the *traffic intensity* and is defined as $\frac{\lambda}{\mu}$. Equation 4.10 allows us to express the expected number of tasks in a system with a given ρ [16].

$$
\begin{aligned}
E[n] &= \sum_{n=0}^{\infty} np_n && (4.11)\\
&= \sum_{n=0}^{\infty} n(1-\rho)\rho^n && (4.12)\\
&= \sum_{n=0}^{\infty} n\rho^n - \sum_{n=0}^{\infty} n\rho^{n+1} && (4.13)
\end{aligned}
$$

A well known "trick" when dealing with series like these is to relabel the n counter. When doing this for the second term of 4.13 we arrive at a geometric series for which we know the exact formula. Setting $n \mapsto n+1$ leads to following derivation.

$$
\begin{aligned}
E[n] &= \sum_{n=0}^{\infty} n\rho^n E[n] - \sum_{n=1}^{\infty}(n-1)\rho^n && (4.14)\\
&= \sum_{n=1}^{\infty}\rho^n && (4.15)\\
&= \frac{\rho}{1-\rho} && (4.16)
\end{aligned}
$$

The summation in 4.15 is on the form $\sum_{n=1}^{\infty} a^n$ for which the exact solution is $\frac{a}{1-a}$ if $0 \leq a < 1$ [49]. We assume that ρ is positive and less than 1. It won't have any meaning talking about negative traffic intensity. Also, if the arrival rate is equal or larger than the service rate, the queue will grow to an infinite length, thus the expected number of jobs in the system cannot be defined.

Little's law provides us with a relationship between expected number of jobs in system $E[n]$ and the response time R. It states that the average response time is the expected number of jobs in system divided by the rate of arrivals, i.e. $R = \frac{E[n]}{\lambda}$ [16]. Using this we can arrive at a formula for the response time - the quantity of interest herein.

$$E[T] = \frac{1}{\mu(1-\rho)} = \frac{1}{\mu-\lambda}$$

λ is the arrival rate, i.e. number of arrivals per time unit, μ is the service rate, i.e. the number of request being processed per time unit.

The assumption that these formulas rely on is that the arrival and service processes are memoryless with exponentially distributed arrival and service time [11]. Alas, inter-arrival times cannot be expected to conform to an exponential distribution in any case, as discussed earlier in section 2.6, nor can the service process. Hence, modelling a server system with these underlying assumptions can cause inaccurate prediction of performance.

The M/G/1 queue

To enable discussion of queues with service times distribution different from exponential, we need to define the $M/G/1$ queue. An $M/G/1$ has exponential arrival time distribution and a general service time distribution. For QoS assessment the quantity of interest is the expected response time $E[T]$, i.e. the sojourn time of a request in the queueing system. The sojourn time is a sum of two parts; the expected queueing time, or waiting time before entering the serving unit, $E[W]$, and the service time itself for the request $E[S]$ [50].

$$E[T] = E[W] + E[S] \quad (4.17)$$

Further, the expected number of jobs in a system $E[N]$ is the sum of expected number of jobs in queue $E\left[N_q\right]$ and in the server $E[N_s]$

$$E[N] = \left[N_q\right] + E[N_s] \tag{4.18}$$

The *utilization* of a queue, ρ, is the fraction of the time that the server is busy, i.e. $\rho = P[N_s > 0]$. Another way of expressing this is

$$\rho = E[N_s] \tag{4.19}$$

For the expected number of jobs in a queueing system with general service time, Pollaczek and Khinchin[2] derived the following formula known as the *P-K mean value formula* [50].

$$E[N] = \rho + \frac{\rho^2(1 + C_S^2)}{2(1 - \rho)} \tag{4.20}$$

where $E[N]$ is the expected number of jobs in a system, ρ is the traffic intensity and C_S^2 is the *coefficient of variation* and is defined as following $C_S^2 = \frac{\sigma_S^2}{(E[S])^2}$ where $E[S]$ is the expected service time and σ_S^2 is the variance for service time.

By combining equation 4.19, 4.18 and 4.20 we arrive at the following expression for expected number of jobs in a queue.

$$E\left[N_q\right] = \frac{\rho^2(1 + C_S^2)}{2(1 - \rho)} \tag{4.21}$$

Little's theorem provides us a way of deriving at the expected waiting time in queue $E[W]$. Little's theorem states that the expected number of jobs in queue is the product of the arrival rate and the expected waiting time in queue, i.e. $E\left[N_q\right] = \lambda E[W]$. Combining this with equation 4.21 gives an expression for $E[W]$.

$$E[W] = \frac{1}{\lambda}\frac{\rho^2(1 + C_S^2)}{2(1 - \rho)} = \frac{\rho E[S](1 + C_S^2)}{2(1 - \rho)} \tag{4.22}$$

This is possible because traffic intensity, ρ, is defined as follows: $\rho = \lambda E[S]$.

We can now arrive at a relationship for response time for a single server queue with a Poissonian renewal process and a general service time distribution from equation 4.22 and 4.17.

$$E[T] = E[S] + \frac{\rho E[S](1 + C_S^2)}{2(1 - \rho)} \tag{4.23}$$

This formula enables us to compare the response times of any single server systems on the form $M/G/1$.

[2] In literature also called Khintchine[51]

The G/G/1 queue

A queue with general inter-arrival and service time distribution is denoted as $G/G/1$. Analytic treatment of such queues has proven to be very difficult. No easily usable exact result exists for these queues due to the difficulty in determining the transition probabilities between queueing states [52](see derivation of formula for the $M/M/1$ in 4.2.1). However, several approximations regarding queueing length and average waiting time have been proposed. The caveat of these approximations is that they only represent upper bounds. In addition, they are generally not valid for low traffic intensities. Hence, they are called *heavy traffic approximations* and are valid for $\rho \mapsto 1$[52]. One of these are mentioned in [53] and has the following form.

$$E[W] \approx \frac{\rho E[S](C_s^2 + C_a^2)}{2(1-\rho)} \tag{4.24}$$

As the total system time is a combination of both queueing, or waiting, time and service time, the upper bound for response time becomes:

$$E[T] \approx \frac{\rho E[S](C_s^2 + C_a^2)}{2(1-\rho)} + E[S] \tag{4.25}$$

For exponentially distributed inter-arrival time it is fairly easy to see that we get the response time formula based on the Pollaczek-Khinchin equation (see 4.23). For such inter-arrival times, $C_a^2 = 1$ as $\sigma_a^2 = (E[A])^2$. This follows from standard deviation and mean of exponential distributions. On this $M/G/1$ form the formula 4.25 is therefore exact.

Moreover, with an exponentially distributed service time, this formula is equal to the formula for $M/M/1$ queues (equation 4.2.1). Thus, for all queue constellations with exponentially distributed inter-arrival times, formula 4.25 is said to be exact.

Another way of doing heavy traffic approximation is presented in [51] and is derived from work done in [54]. It is similar to the one presented here and yields approximately the same results, thus omitted here. It's worth noting an important limitation before employing these formulas. Generally, a queue on the form $G/G/1$ means that the inter-arrival and service times are *independent* random variables. Some prefers to emphasize this by denoting the queues $GI/GI/1$. The implication of the independence restriction is that these queueing models are not valid for distributions with long range dependence. We will therefore not explore the validity of these expressions for self-similar traffic.

Processor sharing discipline

Computer systems are pervaded with multiplexing mechanisms, such as time sharing, of resources and it is therefore inaccurate to assume a FCFS service discipline for web servers. In time sharing systems, a job is assigned a portion of service time, δt, even if the job that arrived before them is not completely processed. Jobs are switched

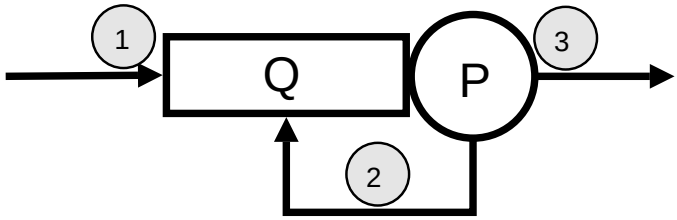


Figure 4.2: *A model of the G/G/1/PS queue. The job enters the queue in* 1*. Upon ending the assigned time in the processor, the job gets switched back into queue in* 2*. When the job is finally done after possibly many visits at the processor, it leaves the system in* 3*. The figure was inspired by [42].*

between processing and queueing mode until they are finished and leave the server system as depicted in figure 4.2. If no priorities are affiliated with the jobs this service scheduling is plain round robin. Further, if we have exponentially distributed service and arrival times with one serving unit we obtain the $M/M/1/PS$ queue (PS for Processor Sharing).

What might be surprising is that the expected service time is for a $M/M/1$ independent of whether the serving discipline is PS or FCFS [54]. However, it becomes quite apparent when considering the following formula for response time of a request with service requirement x in a PS system.

$$T(x) = \frac{x}{1-\rho} \tag{4.26}$$

By definition the average of x, $\overline{x} \equiv E[S]$. Further, we know that $E[S] = \frac{1}{\mu}$. The response time for $\overline{x}$, i.e. the expected response time, is therefore.

$$T(\overline{x}) = E[T] = \frac{\overline{x}}{1-\rho} = \frac{E[S]}{1-\rho} = \frac{1}{\mu(1-\rho)} = \frac{1}{\mu-\lambda} \tag{4.27}$$

Intuitively, this makes sense. Round robin processor sharing hinders starvation of short jobs waiting for long ones to finish. However, the gain for the short jobs is the pain for the long ones, meaning that they have to pay with longer system sojourn time. For infinitely large datasets, average response time from a Processor Sharing system will therefore converge to the FCFS mean value.

What is even more surprising and shown by Sakata et al. [55] was that this convergence was completely independent of the service time distribution. They studied the $M/G/1$ queue and found that expected system response time were only dependent on the mean service time $\overline{x}$. Thus, for a $M/G/1$ it has no effect on mean response time whether the service discipline is FCFS or round robin. In turn, this means that the $M/G/1$ mean value formula should also be valid for the $M/G/1$ queues with processor sharing discipline. We were not able to find comprehensible literature on how the PS serving discipline affects the heavy traffic approximations for $G/M/1$ and $G/G/1$ queues.

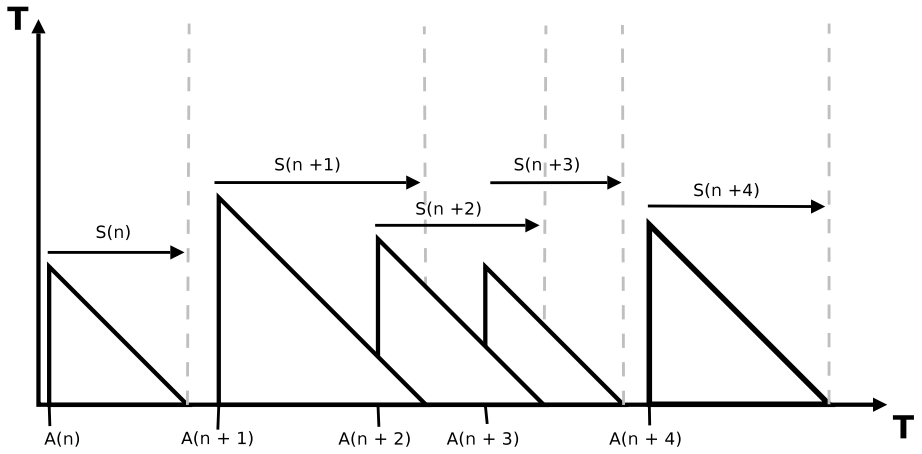


Figure 4.3: *Inventory model for a $G/G/1$ queue. A job n arrives at A_n ($A(n)$ in the figure) with a certain service requirement S_n ($S(n)$ in the figure).*

4.2.2 Hand simulation and an inventory queueing model

Herein we describe how to simulate a single server queue by *hand simulation* explained in terms of an inventory model. Consider a general queue consisting of one processing unit (i.e. a $G/G/1$) and a FCFS serving discipline. Further, consider a job with a service demand t as an amount that is decreased with time t. This is depicted in figure 4.3 as isosceles triangles. Service demands of the jobs decrease linearly with slope -1 due to time being denomination for both x- and y-axis. For such a system several functional relations can be derived.

- The inter-arrival time between job $n-1$ and n is I_n.
- The arrival time A_n for job n is equal to the sum $\sum_{i=1}^{n} I_i$. If the start of the period is marked by the first arrival of a job, the arrival time for job n is equal to $\sum_{i=2}^{n} I_i$.
- The time T_n job n starts processing is the maximum of arrival time $A(n)$ and the departure time D_{n-1} for the previous job, $\max(A_n, D_{n-1})$.
- The departure time D_n for job n is equal to the time the job starts to get processed T_n plus the job service demand S_n.
- The *response time* R_n for job n is equal to the difference between job departure time D_n and job arrival time A_n

The time a job enters the processing state can never be less than the arrival time of the job. However, if there is queueing, the time at which a job gets processed is determined by the departure time of the previous job. This is depicted in figure 4.4 where the departure times have been corrected for queueing due to jobs ahead in the system.

This simple algorithm can be utilized to calculate individual and average response time for a queueing system with one server, either with precomputed inter-arrival and service times or real time generation of these. We implement this theory into a queue

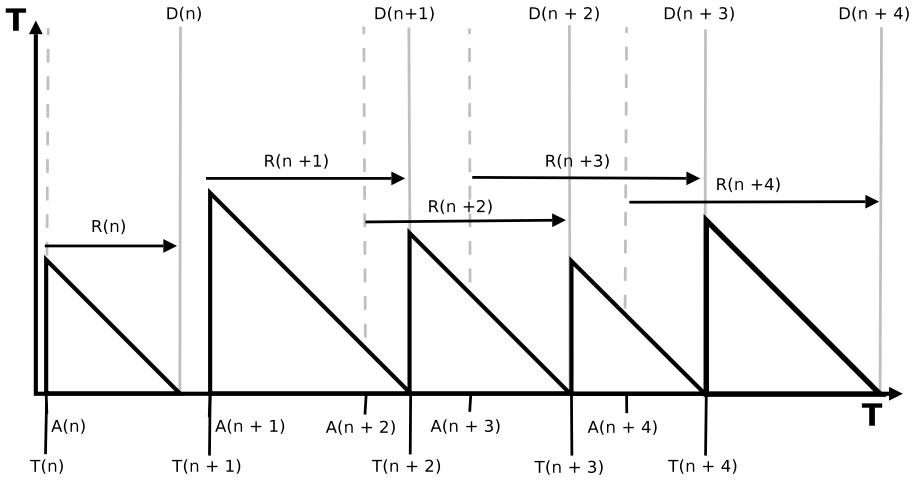


Figure 4.4: *Inventory model for a $G/G/1$ with response times. The difference between departure time D_n ($D(n)$ in the figure) and arrival time A_n is the response time of a request.*

simulator to investigate the consequences of modelling a web server as a FCFS system, as the previously mentioned queueing formulas assumes.

4.3 Hurst estimators

The Hurst estimator is a measure of self similarity, i.e. the degree to which one can relate an event to previous observations. It is named after the British hydrologist H. E. Hurst who in the 1950s studied and modelled the seasonal fluctuations of the flow of the river Nile [38, 17]. His investigations of the amount of water flowing into a particular reservoir could not sufficiently be described by a Poisson random process. There seemed to be a correlation between subsequent events, thus the process could not be a memoryless Poisson process.

There are a number of ways to estimate the Hurst exponent which vary in complexity. What is important to have in mind is that these merely estimate the degree of long range dependency. In many cases the different methods yield non-congruent results, and literature exists that reveal weakness of performance in specific cases [36]. However, for the fairly low-end discussion of web traffic characteristics presented herein, the methods that exist are considered to be appropriate for indicating differences in *degree* of self-similarity in distributions.

Methods for estimation of the Hurst exponent can roughly be split into two groups; those who work in the time domain and those that operate in the frequency domain [56]. The mathematics of these methods vary in complexity, thus it's beyond the scope of this thesis to elaborate on all. However, the mathematical approaches for a few of the simplest methods are briefly presented below.

Absolute value method

A number of estimators for Hurst exponents is based on what is called an *aggregated series*. This is similar to local averaging where a time series of N is split into k series of m entities. For a block i, the mean of the m values represents the value of i. This can be expressed as follows [56]

$$X_k^{(m)} = \frac{1}{m} \sum_{i=(k-1)m+1}^{km} X_i,\ k = 1, 2, \ldots \frac{N}{m} \tag{4.28}$$

In the absolute moments method, a log-log plot is taken of the aggregation level m and the mean of absolute values, $\overline{X_k^{(m)}}$, for the differences between aggregations, $X_k^{(m)}$, and the distribution mean, $\overline{X}$. $\overline{X_k^{(m)}}$ is defined as follows:

$$\overline{X_k^{(m)}} = \frac{1}{N/m} \sum_{k=1}^{N/m} \left| X_k^{(m)} - \overline{X} \right| \tag{4.29}$$

For several aggregation levels this should result in a straight line with slope $H - 1$ where H is the Hurst exponent [56].

Aggregated variance method

The variance method is similar to the previous, but instead of looking at first moments, it considers variances of aggregated series[25].

$$Var\left(X_k^{(m)}\right) = \frac{1}{N/m} \sum_{k=1}^{N/m} \left(X_k^{(m)} - \overline{X}\right)^2 \tag{4.30}$$

A log-log plot of the variance for different aggregation levels should result in a straight line where the slope is is $2H - 2$. Both the *Aggregated variance* and the *Absolute value* method are estimators that work in the time domain.

Others

Time domain estimators in general investigates the power-law relationship between a specific statistic of the time series and an aggregation block size m [56]. Others that belong in this group are *R/S analysis* and *Variance of residuals*. *Periodogram method*, *Whittle estimator* and *Abry-Veitch* are estimators that work in the frequency or wavelet domain. These are all rather complex and will not be elaborated on further. They will however all be employed in the analysis section for estimating the Hurst exponents of self-similar distributions. The reason for this is that each method exhibits different ability to estimate the Hurst exponent depending on the method used for distribution generation. For distributions of fractional Gaussian noise, which are used in the experiments herein, it has been indicated in literature that the best methods for Hurst

exponent estimation are Whittle or Periodogram [36]. For henceforth calculations of the Hurst exponent, the Java application `SELFIS` [56] is used which is freely available[3].

[3] `http://www.cs.ucr.edu/~tkarag/Selfis/Selfis.html`

Chapter 5

Experimental setup

Herein we present the simple experimental setup enabling investigation of our objectives.

5.1 Hardware and OS

The selection of hardware had to be carefully planned. Web traffic in real production systems is an accumulation of many client requests. As there was a limited number of clients to our disposal, our greatest concern was the ability to generate enough traffic. The original plan was to do the experiments using an IBM Bladecenter as a web server system. This is a high end system and generating enough load to stress it is therefore not trivial. One might need several fairly powerful machines acting as clients, something that obviously poses a practical challenge. Instead, we chose to scale down the web server system using an older desktop computer as test server. This way we only needed *one* powerful client machine for traffic load generation. The equipment used in this experiment is outlined below.

Client: Dell Dimension 5100 Desktop (3.4-GHz Intel Pentium 4, 3,0 GB RAM, 250 GB hard drive) running Ubuntu 5.10 GNU Linux ”Breezy Badger”, Linux kernel 2.6.12.

Server: Compaq Presario 4400 Desktop (1.1-GHz Intel Celeron, 256 MB RAM, 20 GB hard drive) running Debian GNU Linux Stable, Linux kernel 2.6.8.

Cable: 1 Enhanced CAT 5 UTP Crossed Ethernet patch.

The testing equipment was set up in a simple topology by connecting the client and server with the crossed Ethernet cable. This facilitated a closed environment and hindered interference by network traffic other the HTTP conversations between the client and the server. Both computers were equipped with a 100 MBit Ethernet card and operates in full duplex mode. Transferred data between client and server is at a minimum, thus the network is therefore not considered to affect the experiments significantly. The topology is depicted in figure 5.1.

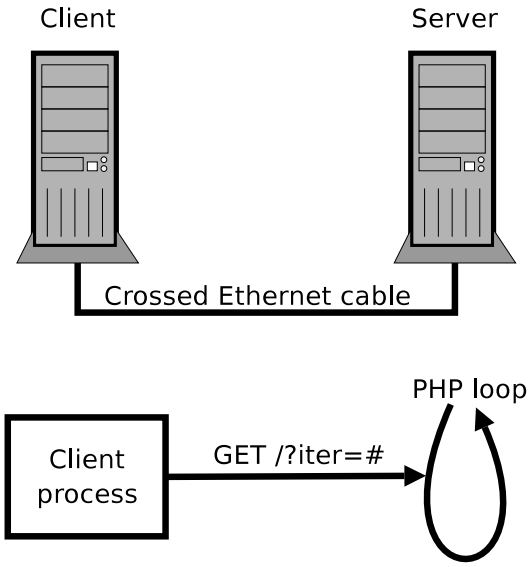


Figure 5.1: *Experiment topology. The client is connected to the server with a crossed cable. A HTTP client sends request with a workload parameter* `iter` *to the server to create different scenarios.*

During the experiments system processes other than the ones participating in the experiment where limited to an absolute minimum. Unforeseen events and interrupts from such processes can disturb the measurement processes and server operation, and thus cause unequal conditions for the experiments. The tests were done using machines that had been in daily use and had lots of services installed and running. In order to avoid uninstalling all these services, the default machine runlevel was changed from the default (which is level 2 in Debian and Ubuntu systems) to a modified runlevel 3 which made `init` only start the most basic services.

Chapter 6

Methodology

This chapter concerns the experimental approach used in collecting results.

6.1 Determination of system specific parameters

When doing evaluations of web server QoS level a prominent quantity is *response time*. Response time is the time from the user has sent the request until he gets the complete answer and is the aggregation of delay imposed by the different mechanisms that are part of client/server communications. Response time for a request x can therefore be decomposed as follows:

$$T_R(x) = T_N(x) + T_D(x) + T_Q(x) + T_{SP}(x) \tag{6.1}$$

where $T_R(x)$ is the response time for request x, $T_N(x)$ is the total delay imposed by the network for request x, $T_D(x)$ is the total minimum processing delay for connection setup etc. imposed by client and server, $T_Q(x)$ is the total queueing time for request x and $T_{SP}(x)$ is the service time requirement.

For the purpose of the experiments conducted herein we will consider a client/server system as *one* queueing system. Even though this is a somewhat simplified view, considering every subsystem that affects server response time can soon become unwieldy. We therefore make an abstraction and the loss of granularity provides us manageability. In this respect, the response time of the client/server system is equal to the queueing response time.

If there is no queueing, the total response time simply becomes the sum of service time and minimum delay imposed by the system.

$$\begin{aligned} T_R(x) &= T_D(x) + T_{SP}(x) + T_Q(x) + T_N(x) && (6.2)\\ &= T_D(x) + T_{SP}(x) + 0 + T_N(x) && (6.3)\\ &= T_{SP}(x) + T_D(x) + T_N(x) && (6.4) \end{aligned}$$

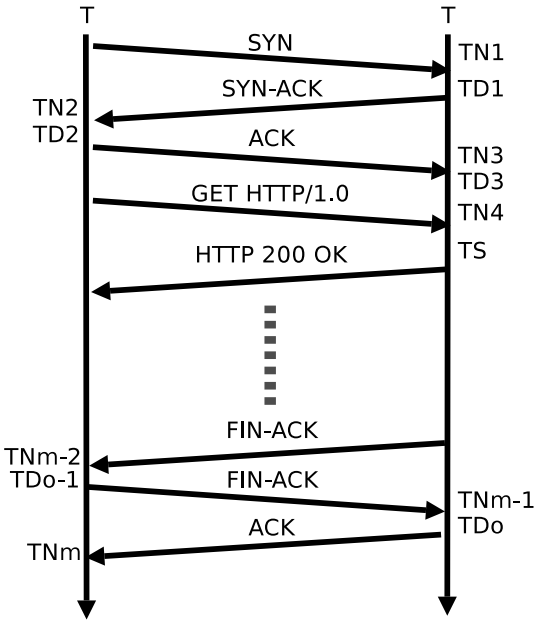


Figure 6.1: *Schematic chart of an HTTP conversation without queueing. The chart tries to depict network time (TN), delay time(TD) and processing time of the HTTP subsystem (TS). Network time and delay time is contributed to at various points in the conversation, thus their total value is a sum of all these small contributions.*

$T_{SP}(x)$ is the processing time for the HTTP subsystem and is varied to generate different service demands, i.e. it can be expressed as $T_{SP}(x) = N(x) * t_{iter}$ where $N(x)$ is the number of iterations and t_{iter} is the average time per iteration. The conversion from service time to number of iterations allows server side script to simulate workload with a certain time requirement. This leads us further to the following relationship for the response time given no queueing.

$$T_R(x) = N(x)t_{iter} + T_D(x) + T_N(x) \tag{6.5}$$

If the assumption is made that both $T_N(x)$ and $T_D(x)$ is independent of x, i.e. constant, the sum of these two values constitute the total *minimum delay imposed by the system* T_{ND}. The following expression describes response time under non-queueing conditions.

$$T_R(x) = N(x)t_{iter} + T_{ND} \tag{6.6}$$

Clearly, this is a formula on the form $ax + b$ and can be determined through experiments and subsequent linear regression analysis. If experiments are conducted with *deterministic* request inter-arrival and service demands and no queueing, the response time can be plotted as a straight line and a functional relationship can be found by employing linear regression techniques. The values for a and b can then be fed into distribution generators to produce a desired workload for experiments.

A \ **S**	**Exp**	**P,** $\alpha = 2.01$	**P,** $\alpha = 2.2$	**P,** $\alpha = 2.4$	**P,** $\alpha = 2.6$
Exponential	$M/M/1$	$M/G/1$	$M/G/1$	$M/G/1$	$M/G/1$
Pareto, $\alpha = 2.01$	$G/M/1$	$G/G/1$	$G/G/1$	$G/G/1$	$G/G/1$
Pareto, $\alpha = 2.2$	$G/M/1$	$G/G/1$	$G/G/1$	$G/G/1$	$G/G/1$
Pareto, $\alpha = 2.4$	$G/M/1$	$G/G/1$	$G/G/1$	$G/G/1$	$G/G/1$
Pareto, $\alpha = 2.6$	$G/M/1$	$G/G/1$	$G/G/1$	$G/G/1$	$G/G/1$

Table 6.1: *Test matrix for the exponential and Pareto distributed arrival and service times. Expectation values for each inter-arrival time and each service demand is the same. The table cells contain the queueing system for each case.*

6.2 Traffic and service generation

Different scenarios are created by text files passed to the HTTP traffic generator `client` containing inter-arrival times and service requirements of differing distribution types. Through carefully selecting parameters when generating these files, an arbitrary level of traffic intensity ρ can be set. There are many combinations of expected service and inter-arrival times that yield the same level of intensity. Therefore, either the expected value for service or arrival times need to be decided on.

There is a limit to the granularity of which the client machine can produce traffic. Due to the CPU limitations, the inter-arrival time can't be arbitrarily small. Hence, the inter-arrival time cannot be arbitrarily small in order to achieve desired traffic intensity.

6.2.1 Pareto and Exponential queues

The purpose of this section is to test the applicability of simple formulas for a web server system. Through varying both service requirement and inter-arrival time characteristics, different queueing scenarios are created. Specifically studied herein are exponentially and Pareto distributed service requirements and inter-arrival times.

Using results of test runs of formula queueing simulator, `q_sim.pl`, we determined the values for the Pareto α-parameter. An equality point for $M/M/1$ and $G/G/1$ queues was observed for $\alpha \approx 2.5$ for both service and inter-arrival distribution. For larger values of the α, the Pareto queues performed better. Also, formulas relies on a well defined variance, i.e. $\alpha > 2$ and a no distribution heavy-tail. The α-parameter was therefore determined to vary between 2.01 with high variance to low variance for 2.6. All in all, 25 individual experiments were to be conducted, distributed as shown in table 6.1.

Distribution generation

Distributions where generated using the `gsl-randist` front end to distribution generating functions in the GNU Scientific Library. In order to reveal differences in response time due to distribution type differences, expectation values must be kept con-

stant. To facilitate this, a wrapper script for `gsl-randist` written in PERL, `exponential-ParetoMaker.pl`, was constructed. Through mean value formulas for exponential and Pareto distributions the script determines which parameters to feed `gsl-randist`. Upon generation completion the resulting files where put in a directory structure according to types, i.e. whether it was an inter-arrival time or service requirement type of file. Due to time consuming nature of the experiments, execution of the 25 individual tests was automated by a shell script named `runDistroBatch.sh`. It reads the contents of the distribution directories, creates a directory for each experiment, initiates the measurement procedures and starts the client for each composition of inter-arrival and service time. Upon experiment completion a traffic capture file and a file containing output from the client is left in the directory.

We note that we do not test the system under heavy-tailed traffic as the theoretical formula 4.25 for calculating response time is reliant on a well defined variance. Therefore, the Pareto distributions must be constrained to $\alpha > 2.01$ which do not qualify as heavy-tailed distributions.

6.2.2 Note on assumptions

Several assumptions have been made in conjunction with the experiment described above. In order to apply the previously stated formulas, the web server system must be assumed to have the following properties.

- $B = \infty$, i.e. queueing capacity of the web server at any level is unlimited. This is clearly an over simplification. Since we are considering the web server to be one queueing system, it is the queue for TCP connections that sets the buffer size. For Linux systems, this is defined in the `tcp_max_syn_backlog` under the `/proc` pseudo-filesystem. It was set to 1024 on the server and it is believed that this limit is not exceeded. However, should requests be rejected due to a full buffer, it will manifest itself through presence of TCP RST packets in the traffic captures.

- $K = \infty$, i.e. the population size is infinite. This is not true as the input files represent finite sets of requests. The experiments must therefore be regarded as a snapshot of an infinite process.

- $c = 1$, i.e. there is only one processing or server unit. For single web service systems with only one server, like the one studied herein, this assumption holds since there is only one CPU.

- $P = \textit{First Come First Served}$, i.e. serving discipline is FCFS. Obviously this is wrong. Computer systems multiplex resources and CPU time is given to each process according to a processor sharing discipline. All the same, queueing models relying on FCFS are to a large degree used to describe server systems. Response time values from experiments and simulations might reveal the consequences of this assumption.

A \ **S**	**Exp**	**P,** $\alpha = 2.01$	**P,** $\alpha = 2.2$	**P,** $\alpha = 2.4$	**P,** $\alpha = 2.6$
$H = 0.5$	-	-	-	-	-
$H = 0.6$	-	-	-	-	-
$H = 0.7$	-	-	-	-	-
$H = 0.8$	-	-	-	-	-

Table 6.2: *The test matrix of distributions with differing degree of arrival-time self-similarity and exponential and different Pareto service time distributions.*

6.2.3 Self-similar traffic, Pareto and Exponential service

This section looks at the effects of self-similar traffic. In particular, we want to investigate how response time is affected when the web server is subject to traffic of varying degree of self-similarity. The experiment is similar to the previous, however the arrival processes are all self-similar with alternating values for the Hurst exponent. Like before, we aim to keep the expectation values for inter-arrival time and service requirement equal for each distribution. Moreover, we try to keep expectation values equal to the previous tests in order to perform comparison of results. We replicate the test matrix from before, replacing arrival time distributions with self-similar(table 6.2). Note that the type of queue is not listed in the matrix. As previously mentioned in section 4.2.1, formulas for queue response time rely on independent inter-arrival and service time. Self-similar traffic is long range dependent, thus these formulas are not valid for self-similar arrival processes. Therefore, experimental results will *not* be compared with formula results. However, they will be compared with results from the hand simulation tool `simulate_distributions`.

Distribution generation

Generation of the self-similar distributions, more specifically *fractional Gaussian noise(fGn)*, was facilitated by the PERL module `Math::Random::Brownian` incorporated in a script named `selfSimDistroMaker.pl`. The module is able to generate fGn in many ways. We chose to use the Hosking method as this is said to be exact [25]. Each of the distributions was generated using the same parameters as before, i.e. with the same expected inter-arrival time.

The set of numbers returned from the module is a mixture of negative and positive double values. Obviously, no inter-arrival time is negative, so the wrapper PERL script checks which number is the smallest negative number and adds the absolute value of this to every generated number. In other words, the distribution values are moved up the vertical axis by an amount equal to the largest negative value.

Also, there is no facility for defining desired expectation value using `Math::Random::Brownian`. To get around this the PERL script sums up all the entities and calculates mean. All the values are in turn adjusted according to a desired expectation value by taking multiplication of every dataset entry by a *expected value - distribution mean* ratio.

It is conceivable that these operations might disturb the statistical properties of the distribution. Therefore, all the generated distributions were analyzed and checked for conservation of the self-similar properties using the `SELFIS` tool.

6.3 Data collection

The collection of data facilitated the experimental analysis. Data was only collected at the client to avoid measurements affecting HTTP request processing.

Each session of HTTP requests based on a service and arrival time distribution was captured. As the self-made test suite does not have any capturing or statistical abilities, traffic captures were done using `tcpdump` which was described in section 2.5.5. The following command initiated the captures.

```
# tcpdump -i eth0 -w <file> 'port 80 and host <server>'
```

This command tells `tcpdump` to capture traffic on interface `eth0`, write the capture til file `<file>` and only capture traffic going to or from port 80 where the host with IP or FQDN `<server>` is involved. Traffic captures are extremely useful and versatile for analysis of network transmissions. Amongst other, the captures contain timestamps for each of the packets which makes it possible to track a TCP session for extraction of session duration. Boundaries for a TCP conversation are set by the initial SYN and the final ACK response to a FIN-ACK. The time interval between these packets is the TCP session duration which we define as the response time for an HTTP request.

The small client application used for traffic generation has a flag that makes it print the server response to standard output. In the experiment, this functionality was exploited to dump the PHP processing times and HTTP session time recorded by the client to file. Although the traffic dumps should suffice, the logs from the client could be useful if unexpected behavior is experienced. The command initiating the client for the experiments is outlined below.

```
# ./client -h <server> -a <a-time> -s <s-time> -p >> \
  <log.txt>
```

The `-p` flag enables printing of HTTP responses and measured response time by the client application to standard output which in turn is continuously appended to log file `log.txt`. However, the client does not use `mutex` to lock shared resources amongst the threads. A thread waiting for a resource to become unlocked might affect the timing and mutexes ware therefore avoided. Standard output is a shared pipe and the absence of resource locking leads to the mangled output from time to time and loss of response time values. It was a necessity for this program to work that it made no use of `mutex` locks; therefore the response times gathered by the client threads are only used as supporting data for the response times extracted from traffic captures.

Initially, traffic dumps were also done on the server in order to have supporting data. However, running `tcpdump` at the server proved in preliminary tests to interfere greatly with the operation of the web server. Calibration runs, i.e. the experimental

tests for determination of system specific parameters described in 6.1, with `tcpdump` running at both client and server gave extremely large deviations in response times. Thus, parameter estimation for the linear relationship in 6.1 becomes dominated by uncertainty. We therefore chose to abandon server side data collection.

6.3.1 Extraction of data

Extraction of data was done solely on basis of the traffic capture files and provided several quantities of interest. However, to be sure that data from successful transactions was collected, "sanity" checks of the dump file had to be done. In other words, the capture files had to be checked that all initiated HTTP requests were successfully completed by looking for occurrences of RST packets and response status messages other than 200 OK. In addition, if the number of 200 OK response messages did not amount up to the number of requests made, something had gone wrong.

Every complete TCP session was tracked and the time between the initial SYN and the finalizing ACK for a session represents the response time for the system. We are not only interested in the average response time and its standard deviation; we are also interested in the response time distribution. For each experiment, the individual response time were therefore collected in a file.

In addition to extracting response times, the actual inter-arrival time distribution were of interest. For each complete connection the inter-arrival time was therefore recorded. Also, the service requirement associated with each request was extracted from the dump files. The purpose of this was to compare the actual traffic conditions with the input given to the client. All of the data extraction was facilitated by the PERL script `doAnalysis.pl` previously described (section 2.5.6).

6.3.2 Note about service time generation

The service time for each request was generated by varying the number of iterations for the server side PHP script. An alternative to this spin delay is to have the server side application sleep for a specified period of time given in a GET variable. However, this does not create a realistic scenario as web requests usually need some kind of processing. Using a `for`-loop for generating workload creates actual workload for the server and therefore results in a better simulation of a real production environment.

6.4 Sources of error

Programming errors and bugs: The author is not a C, PERL or a Bash programming virtuoso and is therefore humble when it comes to the possibility of programming errors in the self-made applications and scripts. Most of the experiments and collection of data relies on this software and should programming errors exist, then we are dealing with a *systematic errors*.

Disruptive system processes: Even though care was taken in selecting services that run on the testing machines, the possibility of unforeseen system processes hog-

ging system resources *cannot* be excluded. These events are due to stochastic events out of our control and therefore are a source for *random error*.

Networking conditions: The overhead time is considered to be constant. Therefore, the networking conditions are expected to remain the same throughout the experiments. The amount of data being transmitted between the client and server is minimal so congestion is not expected to occur. However, should the network for some reason become saturated, the response time will increase. These are events that cannot be predicted and thus we are dealing with a *random error*.

6.4.1 Notes on experiment

It's important to point out that we do not make claims about the characteristics of web traffic in general. The investigations carried out herein merely looks at web server operations subject to different types of workload with the same distribution means. The motivation for using Pareto and self-similar processes stems from previous research proving that such distributions pervade packet switched network communications.

Chapter 7

Results

In this chapter, we present the experimental results and analyze them to find answers to the formerly stated hypothesis.

7.1 Note on mean value accuracy

It is considered as poor practice to state results with a greater accuracy than the standard error allows. The results in this thesis will appear strange to some experimental scientists in other fields, where data are not characterized mainly by their uncertainty. Often one estimates this error by the standard deviation. This is a problem for large values of σ as the assumption of Gaussian distributed values might not be appropriate. Queue response times are a result of two possibly highly variable random processes, and the response time distribution is inherently prone to large deviations. Stating results with lower accuracy than done herein would give no foundation for comparison as many of the results would be equal. For the purpose of the work done herein we have therefore chosen to state average values with a greater accuracy than the standard deviation allows.

7.2 Determination of system specific parameters

In order to generate an arbitrary level of traffic intensity, ρ, a mapping from iterations to processing time is needed. This relationship is system specific and it must therefore be experimentally determined. Resulting parameters are put into distribution generators for generating distributions representing desired level of workload.

As stated in section 6.1, we claim a linear relationship between number of iterations and service time. Through experiments and response time measurement with *deterministic* distributions and no queueing, it is possible to derive this relationship. By *deterministic* we mean that every request has the same service demand and inter-arrival time.

In order to deduce the linear relationship, experiments with gradually increasing service demand was conducted. Traffic data from 10 sessions, with service demands

Iterations	**Mean response time**, $\overline{T}$	Std. dev., σ
10000	0.0205s	0.0008s
20000	0.039s	0.001s
30000	0.057s	0.002s
40000	0.075s	0.002s
50000	0.093s	0.003s
60000	0.112s	0.003s
70000	0.130s	0.003s
80000	0.149s	0.004s
90000	0.167s	0.004s
100000	0.190s	0.005s

Table 7.1: *Response times from sessions with deterministic service requirements and inter-arrival times. Times are given in seconds.*

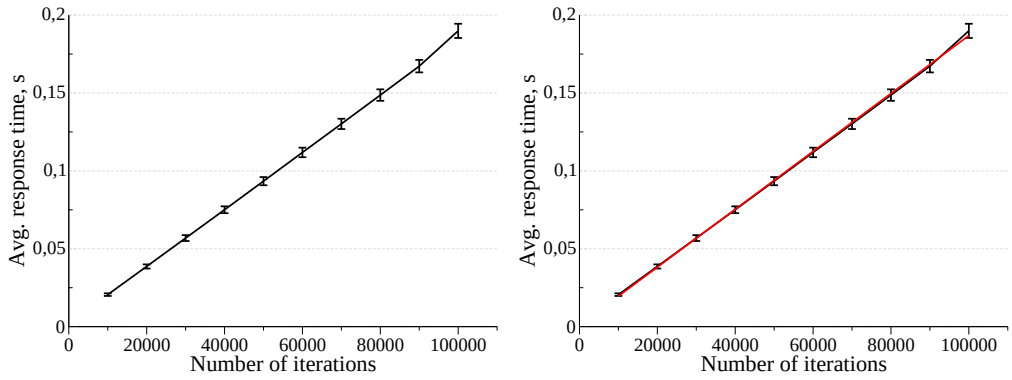
ranging from 10000 to 100000 iterations, were captured to files which where subsequently fed to the `doAnalysis.pl` script for extraction of response time. The resulting values are shown in table 7.1.

In order to avoid queueing at the server, the arrival time had to be adapted to the largest value of the service demand, i.e. 100000 iterations. If inter-arrival time of requests were higher than the maximum time for request processing, then, in theory, there will be no queueing. The URL of the load generating PHP script was therefore loaded in a browser several times to reveal the service time requirement, and it was found to be in the magnitude of $0.180s - 0.20s$ for 100000 iterations. In addition, simple studies of the overhead time (i.e. delay imposed by system) showed that this were in the magnitude of $0.001s - 0.002s$. An inter-arrival time of $0.3s$ should therefore provide an ample timing buffer for queueing not to occur.

The values were plotted using the tool `xmgrace` and the plots can be observed in figure 7.1. Apart from a small break in the curve between 90000 and 100000 iterations, it is quite apparent that the response time for sessions without queueing, has a linear relationship with the number of iterations. It shows a, more or less, perfect straight line and through linear regression we can derive the system specific quantities time per iteration and minimum delay imposed by system. Using regression functionality in `xmgrace` provides us with the following formula.

$$T_R(n) = 1.86 \times 10^{-6} n + 9.94 \times 10^{-4} \tag{7.1}$$

The slope of the curve represents the time requirement per iteration, and the constant represents the minimum delay. These parameters are fed into the distribution generator script to provide the desired expectation value for service time.



(a) Plot w/o regression line **(b) Plot w/ regression line**

Figure 7.1: *Regression plots. (a) Shows the data from table 7.1 without regression line. (b) Shows the plot together with the regression line in red that specifies the linear relationship.*

7.3 Queueing formulas

This section will investigate into the formulas presented in section 4.2 and their validity for web server systems under varying traffic and service conditions. Results from theoretical formulas will be compared with simulation and experimental results derived using equal parameters. Expected service time and traffic utilization is set to be the same for each experiment and simulation run. The properties that are varied are the inter-arrival and service time distribution type and the variance affiliated with these. In turn this leads to overall equal traffic intensity for all the experiments. The parameters are set as follows:

$$\rho = 0.95 \tag{7.2}$$
$$E[S] = 0.09s \tag{7.3}$$

Expected inter-arrival time is computed from these values for both simulation and distribution generation in the following manner.

$$E[A] = \frac{E[S]}{\rho} \approx 0.0947s \tag{7.4}$$

This follows from the relationship $\rho = \frac{E[S]}{E[A]}$. Distributions of service demand and arrival times to be tested were exponential and various versions of the Pareto distribution. Formula simulations, experiment and hand simulations followed the test matrix as depicted by table 6.1 and the distribution sizes were 10^5 entries.

A \ S	**Exp**	**P,** $\alpha = 2.01$	**P,** $\alpha = 2.2$	**P,** $\alpha = 2.4$	**P,** $\alpha = 2.6$
Exp.	1.80	43.48	2.89	1.84	1.49
Par., $\alpha = 2.01$	43.48	85.16	44.57	43.52	43.18
Par., $\alpha = 2.2$	2.89	44.57	3.98	2.92	2.58
Par., $\alpha = 2.4$	1.84	43.52	2.92	1.87	1.53
Par., $\alpha = 2.6$	1.49	43.18	2.58	1.53	1.19

Table 7.2: *Response time found by simulator based on equation 4.25. Response times are given in seconds.*

7.3.1 Theoretical results

The theoretical results were obtained through using the custom made queue simulation script `q_sim.pl`. This simple script is founded on the functional expression 4.25 and calculates the expected queueing time based on expectation values for arrival and service times and their variance. The values for service and traffic intensity are hard coded into the script before its execution. The results of the simulation are shown in table 7.2.

Apparent from the results, the formula does not distinguish between service and inter-arrival time distribution; the table is symmetric along the diagonal. Even though all values have been calculated from the exact same expectation values, there is a significant difference between smallest and largest value. As the α parameter (the shape parameter) of the Pareto input distributions approaches 2, the distributions gets increasingly heavy-tailed. In other words, the distribution variance increases which results in the expected values for service and inter-arrival time being less "typical" for their respective distributions. According to the formula, this should manifest itself through relatively large response times. For instance, a queueing system subject to Pareto distributed inter-arrival and service time with $\alpha = 2.01$ are expected to produce response times in the magnitude of *1000* times larger than the expected *service* requirement.

7.3.2 Experimental results

Experimental results were derived using the previously described methodology. As the distributions were limited and generated by random number generators, it was conceivable that their mean value deviated from the desired values. To be sure that we were able to produce the desired traffic level, the mean value of each generated distribution was checked before the execution of the experiments and found to be in fair accordance with the desired values.

Table 7.3 lists the experimental results for average response time. Apparent from the table is an all over discrepancy from values calculated by the formula. In addition, the standard deviations are high; in some cases several times larger than the mean value, which is consistent with a heavy-tailed distribution. Moreover, the tendency of higher response time as service time variance increases is not present. For expo-

A \ S	**Exp**	P, $\alpha = 2.01$	P, $\alpha = 2.2$	P, $\alpha = 2.4$	P, $\alpha = 2.6$
Exp.	1.74 ± 4.39	1.09 ± 4.42	1.32 ± 4.83	1.38 ± 4.45	1.25 ± 3.48
Par.,$\alpha = 2.01$	2.18 ± 5.14	1.50 ± 5.60	1.33 ± 4.81	1.40 ± 4.32	1.49 ± 3.98
Par.,$\alpha = 2.2$	1.81 ± 4.59	1.03 ± 4.64	1.08 ± 4.17	1.07 ± 3.64	1.08 ± 3.49
Par.,$\alpha = 2.4$	1.54 ± 4.19	0.86 ± 4.11	0.84 ± 3.63	0.88 ± 3.27	0.85 ± 2.97
Par.,$\alpha = 2.6$	1.34 ± 4.05	0.72 ± 3.98	0.74 ± 3.52	0.78 ± 3.07	0.76 ± 2.81

Table 7.3: *Experimental results for exponential and Pareto distributions. Values are given in seconds. We note that response times cannot be negative as suggested here by the standard deviations. For henceforth discussions, standard deviations are therefore only meant to describe the variability in response time values.*

nential *service* times there seem to be some increase in response time for an increase in inter-arrival time variance. However, for exponential *inter-arrival* time, the increase in service time variance results in a decrease in average response time. For almost all of the inter-arrival time distributions we see a decrease of response time as service time variance increases. Explanations for this behaviour are discussed in section 7.3.4.

7.3.3 Hand simulation results

A \ S	**Exp**	P, $\alpha = 2.01$	P, $\alpha = 2.2$	P, $\alpha = 2.4$	P, $\alpha = 2.6$
Exp.	1.80 ± 1.76	2.87 ± 3.81	2.14 ± 2.59	1.73 ± 1.93	1.48 ± 1.55
Par.,$\alpha = 2.01$	2.31 ± 2.15	3.66 ± 4.27	2.74 ± 3.01	2.21 ± 2.30	1.88 ± 1.88
Par.,$\alpha = 2.2$	1.82 ± 1.79	3.07 ± 3.91	2.19 ± 2.63	1.71 ± 1.93	1.42 ± 1.52
Par.,$\alpha = 2.4$	1.55 ± 1.58	2.75 ± 3.69	1.91 ± 2.41	1.44 ± 1.71	1.17 ± 1.30
Par.,$\alpha = 2.6$	1.40 ± 1.45	2.56 ± 3.55	1.74 ± 2.28	1.29 ± 1.57	1.02 ± 1.17

Table 7.4: *Hand simulation results for the original inter-arrival time and service requirement distributions introduced to the system. Values are given in seconds.*

As earlier pointed out, the formulas assume that the serving unit operates by FCFS discipline. Web server systems have a more advanced scheduling mechanism using processor sharing and priorities. However, processor sharing and FCFS queues have the same expected response time for exponentially distributed inter-arrival times according to theory. In order to check the validity of the formulas for batch processing(FCFS) under the given traffic conditions, the generated inter-arrival time and service requirement distributions were fed to the `simulate_distributions` simulator. Table 7.4 summarizes the results of the simulations.

As with the experimental values, values from simulation deviates significantly from what is predicted by the formulas. Since the simulator works as a FCFS queueing system, and the formulas are for such batch processing systems, the deviations are

quite surprising. However, it is possible to track similar *trends*. For the largest distribution variances, i.e. for Pareto distributions and $\alpha = 2.01$, the highest response time is obtained. Also, for the distributions with the lowest expected variance, Pareto with $\alpha = 2.6$, we get the lowest response time. Worth noting is that we get exactly correct value for $M/M/1$ queues; an indication for correct implementation of the simulator because of the $M/M/1$ models theoretically proven accuracy.

For the rest of the queues with exponentially distributed inter-arrival times there are rather large deviations except for $\alpha = 2.6$. As the formula for $M/G/1$(the formula derived from Pollaczek-Khinchin, see section 4.2.1) queues with FCFS serving algorithm is precise, this can be indicative of unequal parameters being passed to the different systems. We discuss the probable causes for the discrepancies in formula, experimental and simulation values in the following.

7.3.4 Analysis

Herein we try to identify the probable causes for the deviations in average response time values that are seen between formula, experimental and simulated results.

Original distribution limitations

In order to explain the deviations between values obtained using formulas and values obtained by simulation using input distribution files, further investigation of the statistical properties of the distribution files is necessary. As previously mentioned, mean value of each distribution was checked before the experimental runs to make sure that they were in accordance with desired expectation values. What was not checked, however, was the distribution variance. As the formula is heavily dependent on the coefficient of variance, different distribution variance can partly explain the discrepancies between formula and simulation values. Table 7.5 lists the values for mean and standard deviations for the distributions and values derived by using first and second moment formulas for the distributions. Apparently, the distribution variances do not cohere with standard deviations derived from formulas, especially for distributions with high expected standard deviations.

The formulas for exponential and Pareto distribution are based on limit sizes. It is expected that mean and standard deviation of a set of random variables will converge to formula values as the number of data points goes to infinity. Since distributions herein are limited (100000 data points), it is conceivable that distribution sizes are too small for convergence to happen. Table 7.5 indicates that high variance distributions would need a larger set of data points in order to track limiting values as the largest gap between formula and distribution values is for $\alpha = 2.01$.

Distribution conservation

Due to the inaccuracy inherent in computer system timing, it's probable that the inter-arrival time distribution is not perfectly conserved. The reasons for this can be several. Unsolicited hogging of system resources during execution might suspend the client application for some time with a resulting skew of inter-arrival times. In addition,

Arrivals	**Distribution**	**Formula**
Exponential	0.0946 ± 0.0941	0.0947 ± 0.0947
Pareto, $\alpha = 2.01$	0.0942 ± 0.197	0.0947 ± 0.6682
Pareto, $\alpha = 2.2$	0.0943 ± 0.132	0.0947 ± 0.1428
Pareto, $\alpha = 2.4$	0.0943 ± 0.0961	0.0947 ± 0.0967
Pareto, $\alpha = 2.6$	0.0944 ± 0.0757	0.0947 ± 0.0759
Service	**Distribution**	**Formula**
Exponential	48046 ± 48520	47884 ± 47884
Pareto, $\alpha = 2.01$	48060 ± 77362	47884 ± 337746
Pareto, $\alpha = 2.2$	48032 ± 58000	47884 ± 72187
Pareto, $\alpha = 2.4$	48008 ± 45661	47884 ± 48871
Pareto, $\alpha = 2.6$	47991 ± 37636	47884 ± 38338

Table 7.5: *Mean and standard deviation for distributions and formula. The arrival times are given in seconds whilst the service requirement is given in iterations.*

A \ S	**Exp**	P, $\alpha = 2.01$	P, $\alpha = 2.2$	P, $\alpha = 2.4$	P, $\alpha = 2.6$
Exp.	1.62 ± 1.60	2.67 ± 3.65	1.98 ± 2.44	1.58 ± 1.79	1.34 ± 1.43
Par.,$\alpha = 2.01$	2.05 ± 1.95	3.30 ± 4.05	2.44 ± 2.82	1.96 ± 2.13	1.67 ± 1.72
Par.,$\alpha = 2.2$	1.63 ± 1.64	2.79 ± 3.72	1.99 ± 2.47	1.54 ± 1.78	1.27 ± 1.38
Par.,$\alpha = 2.4$	1.40 ± 1.44	2.51 ± 3.52	1.73 ± 2.26	1.30 ± 1.58	1.05 ± 1.19
Par.,$\alpha = 2.6$	1.27 ± 1.32	2.34 ± 3.39	1.59 ± 2.14	1.16 ± 1.45	0.92 ± 1.07

Table 7.6: *Simulation results for measured values of inter-arrival time and service requirement. Values are given in seconds.*

there are limits to the timing resolution of the operating system. Although efforts were made to circumvent problems associated with `usleep()` system call(see section 7.5), the client still proved to have some difficulties in exactly reproducing input inter-arrival times. This tendency was most noticeable for sub 10*ms* inter-arrival times.

Functionality in the `doAnalysis.pl` script therefore facilitated extraction of TCP SYN packets inter-arrival times in order to extract the real traffic conditions for the experiments. These distributions were in turn simulated, again using the hand simulation application to check whether this would yield response times coherent with experimental results. The results of these simulations are shown in table 7.6 and bar-charts comparing simulated and experimental results can be seen in figures 7.2.

In order to now compare the different results, the formula results must also be revised to take into account the actual values of the experiments. Table 7.7 lists the values obtained using *real* values for mean and standard deviation together with the corresponding traffic intensities for each experiment.

Although the obtained values have now been significantly improved, there are still considerable discrepancies for certain queueing constellations. Most prominent

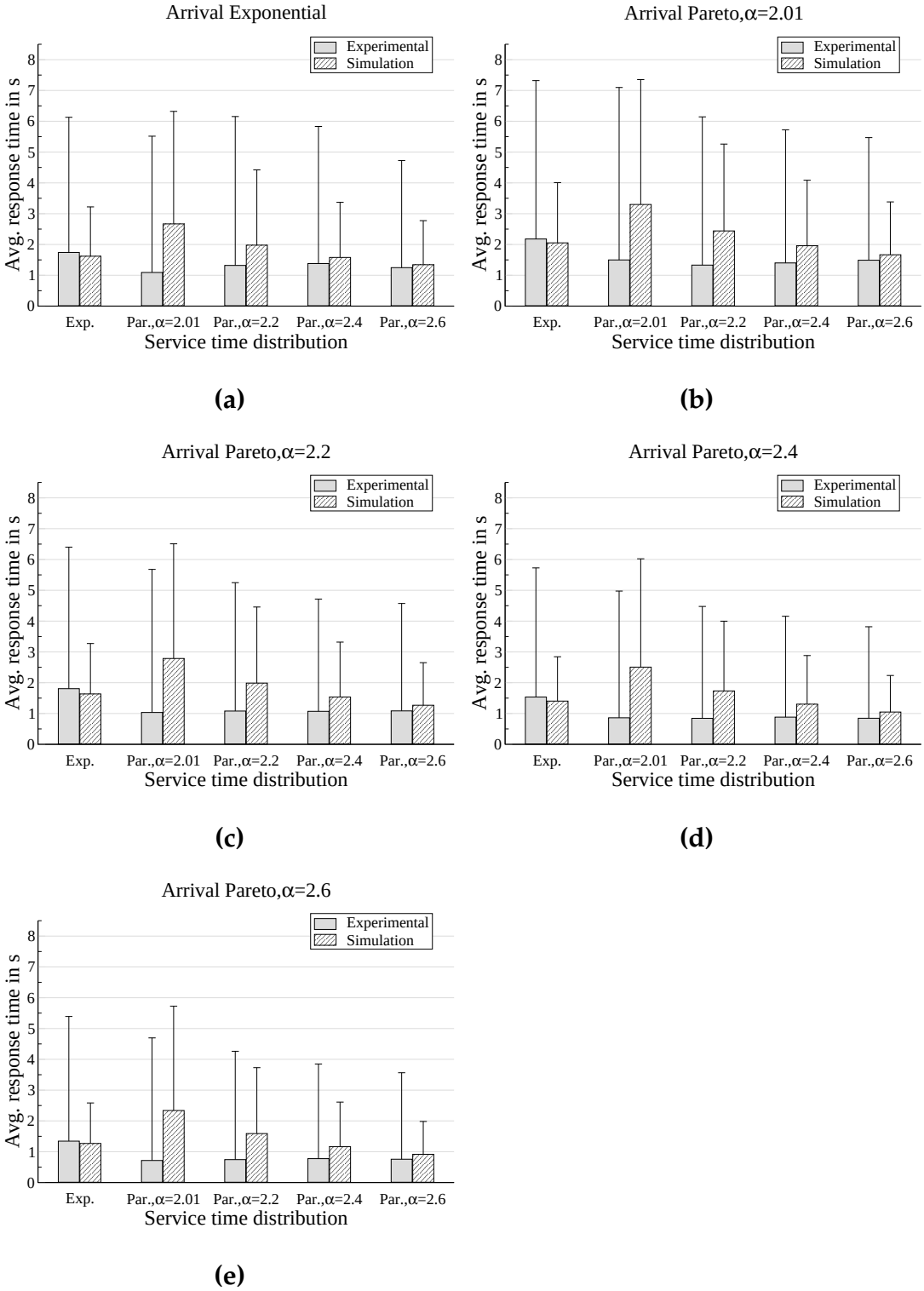


Figure 7.2: *Mean and standard deviation for experimental and hand simulation results.* ***(a)*** *Exponential inter-arrival,* ***(b)*** *Pareto inter-arrival w/* $\alpha = 2.01$ ***(c)*** *Pareto inter-arrival w/* $\alpha = 2.2$,***(d)*** *Pareto inter-arrival w/* $\alpha = 2.4$, ***(e)*** *Pareto inter-arrival w/* $\alpha = 2.6$.

A\S	**Exp.**	**P,**$\alpha = 2.01$	**P,**$\alpha = 2.2$	**P,** $\alpha = 2.4$	**P,**$\alpha = 2.6$
Exp.	1.78 ($\rho = 0.950$)	3.11 ($\rho = 0.950$)	2.14 ($\rho = 0.949$)	1.66 ($\rho = 0.949$)	1.41 ($\rho = 0.949$)
Par.,$\alpha = 2.01$	5.09 ($\rho = 0.954$)	6.58 ($\rho = 0.955$)	5.45 ($\rho = 0.954$)	4.90 ($\rho = 0.953$)	4.60 ($\rho = 0.953$)
Par.,$\alpha = 2.2$	2.78 ($\rho = 0.953$)	4.21 ($\rho = 0.953$)	3.15 ($\rho = 0.953$)	2.63 ($\rho = 0.952$)	2.35 ($\rho = 0.952$)
Par.,$\alpha = 2.4$	1.91 ($\rho = 0.952$)	3.31 ($\rho = 0.952$)	2.28 ($\rho = 0.952$)	1.78 ($\rho = 0.951$)	1.52 ($\rho = 0.951$)
Par.,$\alpha = 2.6$	1.53 ($\rho = 0.951$)	2.91 ($\rho = 0.952$)	1.90 ($\rho = 0.951$)	1.41 ($\rho = 0.951$)	1.16 ($\rho = 0.950$)

Table 7.7: *Formula results in seconds for the measured traffic conditions. Also included is the traffic intensity, ρ for each of the queueing constellations.*

are the values for the $M/G/1$ queues for which theoretical proof has been given for formula accuracy. Also, we know that for such queues, whether or not the serving discipline is processor sharing or FCFS has been proven to have no effect on average response time.

Another noticeable trend is the low response time for experimental results for larger service distribution variance. The highest response times of the hand simulations are obtained with Pareto distributed service time with $\alpha = 2.01$. Corresponding experimental values are amongst the lowest response times. Additionally, the experimental values have fairly large standard deviations, which is indicative of distribution heavy tails and lack of typical values for the response times. These phenomena are likely to be related to the difference in serving discipline as explained in the following sections.

Response time distribution variance

The discussion so far has been focused on providing a fair basis of comparison by adjusting simulation and formula input to the real traffic conditions created in the experiments. We can now concentrate on analyzing differences due to system properties.

The experimental results listed previously are dominated by a large data set spread. Compared to results from simulation, the variance can be many magnitudes larger for experiments. The likely reason for the difference in behavior is the scheduling discipline. Unlike the FCFS discipline assumed by formulas and present in the hand simulation system, computers in general process jobs using processor sharing and priorities. For a system with exponential processing time it was noted in [54] that response time variance is potentially much greater in a processor sharing systems using round robin, than in a batch (FCFS) processing system. The reason is fairly simple

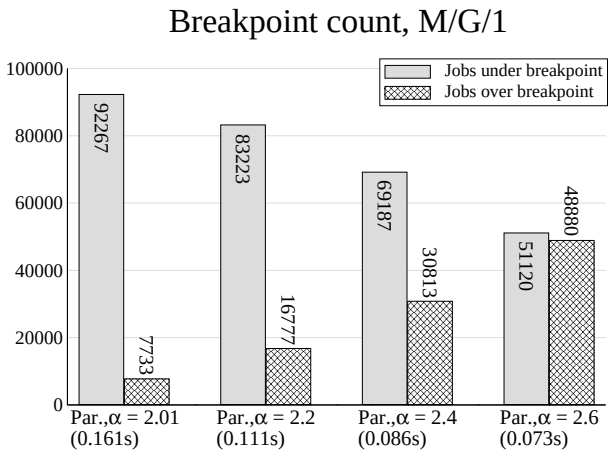


Figure 7.3: *Figure summarizing count of service requirements falling under and over breakpoint values derived using formula from [54]. The breakpoint values are listed in parenthesis and given in seconds.*

and quite intuitive. A job in a system with *exponentially* distributed service times and inter-arrival times ($M/M/1$ queue) whose service requirement is of average size will, on average, have the same response time independent on whether or not it is a processor sharing or a FCFS system. However, jobs with less than average service time will be better treated in a processor sharing system than with FCFS serving discipline. Vice versa, jobs with greater than average requirement will receive poorer treatment.

For $M/G/1$ systems the breakpoint between better and worse processing is for jobs with service time requirement x with value $x = \overline{x^2}/2\overline{x}$ [54]. For the queue constellations with exponentially distributed arrival time we can thus find the breakpoint and investigate how many of the service requirements fall under and over this. The result of this procedure is depicted by figure 7.3. The figure clearly shows and predominance of jobs falling under the breakpoint. It should be noted that the dataset sizes might be too sparse in order for the breakpoint formula to be valid. However, if this approach is appropriate in this case, the vast amount of jobs under the breakpoint could explain the overall decrease of response time going from simulated to experimental $M/G/1$ queues. It is believed that the same phenomenon partly can explain the differences seen in the rest of the queueing $G/G/1$ constellations. Alas, we were not able to find comprehensible breakpoint formulas for these cases.

Thus, processor sharing possibly causes the span of response time values to be much larger than in a FCFS system, which in turn manifests itself through higher response time distribution variance. It should therefore be possible to track differences in degree of response time distribution heavy-tailedness. In section A.2.1 the probability distributions of response time from simulations and experiments are plotted to reveal differences in tail behaviour. Although the distributions are not directly comparable due to mean value differences, the *shape* of the probability distributions can be compared. Compared to the simulated FCFS, the plots indicates an increase in heavy-tailedness in *every* case for the experimental results. It can therefore be argued that the

response time of the web server system does not have a well defined mean and therefore lacks a typical value. Dealing with mean response time for capacity planning and evaluation might therefore lead to wrong conclusions on system efficiency.

Also, the large variance and the long tail in experimental values are indicative that FCFS modelling of web servers, as the queueing models rely on, does not capture crucial characteristics of the response time.

Response time distribution mean

The response time variability can also explain why there are such large deviations in mean response time values between web server and simulated systems. As previously noted, datasets with large variance must potentially be much larger than low variance datasets before convergence of average value can be seen. Because queueing systems are affected by stochastic input, the output of the system will be random as well [57]. Therefore, in order for averages to converge one would possibly need datasets approaching infinity. A data set size of 100000 is fairly low when dealing with random processes, for many highly variable random processes too small to observe distribution convergence. It can be hard to determine the necessary dataset size in each case. Experiments might need to run for a significantly longer period than FCFS systems because of the high variance output associated with processor sharing.

Arrivals vs. Service characteristics

From the barcharts 7.2 one can see a clear trend. Service time characteristics seem to have less impact on the experimental response times than the inter-arrival time characteristics. As inter-arrival time distribution becomes more and more variable for exponential service time, the experiments and simulations show that response time increases. Also, the average response time for simulated and experimental $G/M/1$ queues are fairly coherent. There are slightly higher values for the experiments as could be expected since this is not a protected environment as the simulations are. However, for the queues with Pareto service requirement there is a clear discrepancy between simulated and experiments. The queueing formula 4.25 makes no distinction between arrival and service process characteristics which, according to our results, is obviously not accurate for web server systems. However, the hand simulation results show similar trends as the formula. The reason why we see differences in simulated and experimental values for $G/G/1$ queues and $M/G/1$ queues can be explained in terms of Pareto distribution characteristics and serving discipline

The versions of the Pareto distribution studied herein are defined by two parameters. The degree of heavy-tailedness is defined by the α parameter, also called the *shape parameter*. The other parameter, β is called the *location parameter* and defines the lower bound on distribution values. Due to the form of the Pareto distribution formula and the fact that service and arrival times are kept constant, the β variable varied as the shape parameter is varied. The distribution variance is therefore a result of occasional spiking values. In order to hinder a skew in expectation value, most of the service requirements are below the distribution mean for Pareto distributions to even out the spikes. For inter-arrival times, most of the jobs are arriving at a faster rate than the

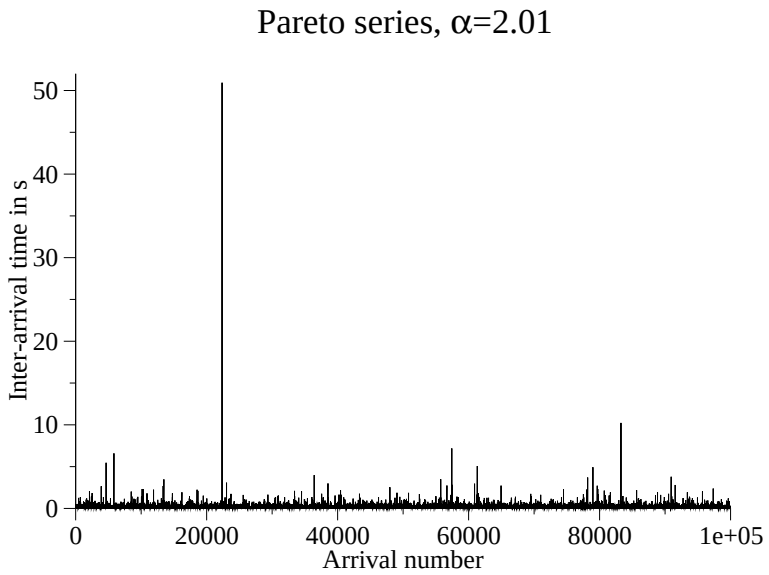


Figure 7.4: *Plot of inter-arrival time Pareto distribution for* $\alpha = 2.01$ *and expectation value* ≈ 0.095*. We clearly see the distribution spikes.*

mean with occasionally large sleep time, while the service requirement is generally lower than the service requirement mean. In a FCFS system, jobs arriving after a spiking service time request experience queue starvation, however this is not the case for a Processor Sharing system. Short jobs get processed even if there is currently a job with large service requirement residing in the server. Thus, for Pareto processes distributed processes, our studies show that increase in inter-arrival time variance has a degrading effect on performance, while a larger span in service time values does not necessarily affect response time adversely.

Traffic intensity level

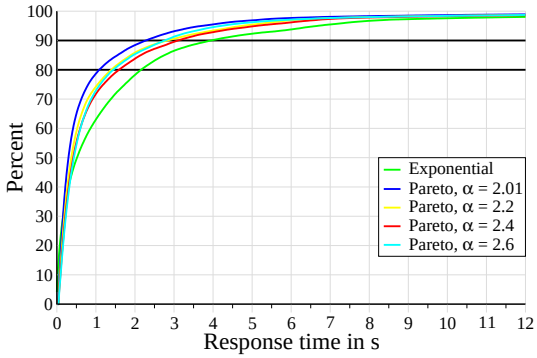
As previously explained in section 4.2.1, the formulas studied are heavy traffic approximations and is said to be valid for $\rho \mapsto 1$. Even though the traffic intensity for experiments and simulations were set as high as 0.95, it might still not qualify as heavy traffic. A higher traffic intensity level might therefore be more appropriate for studies of the validity of heavy traffic formulas for web server systems.

Significance for SLA's

We have so far attempted to give reasonable explanations to the differences in observed values for response time from experiments, simulations and theoretical formulas. Yet to be addressed is the significance of the observed phenomena for service providers.

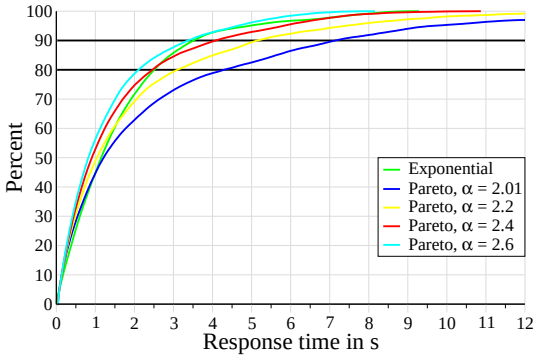
With the level of variance experienced from experimental results it is nearly impossible to make any formal agreements regarding the average service level. Because both arrival and service processes are stochastic processes and the effect of processor

Exp. CDF - Arrivals, Exponential



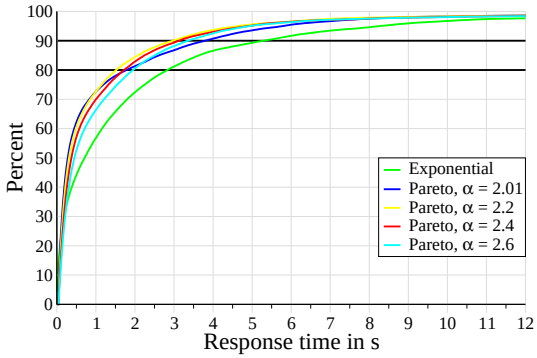
(a_1)

Sim. CDF - Arrivals, Exponential



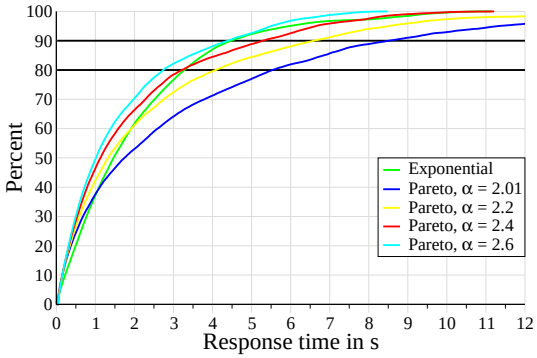
(a_2)

Exp. CDF - Arrivals, Pareto, α = 2.01



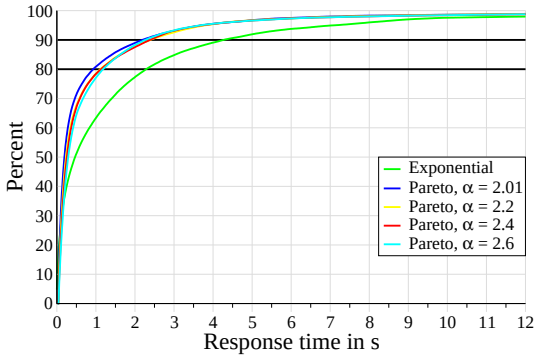
(b_1)

Sim. CDF - Arrivals, Pareto, α = 2.01



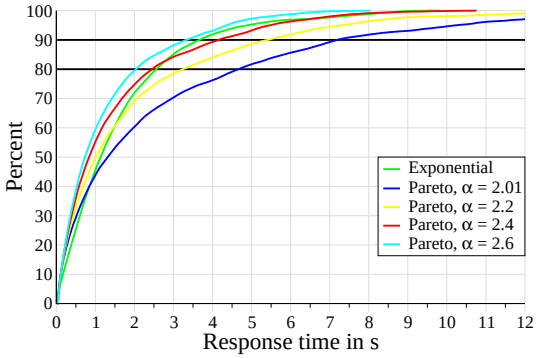
(b_2)

Exp. CDF - Arrivals, Pareto, α = 2.2



(c_1)

Sim. CDF - Arrivals, Pareto, α = 2.2



(c_2)

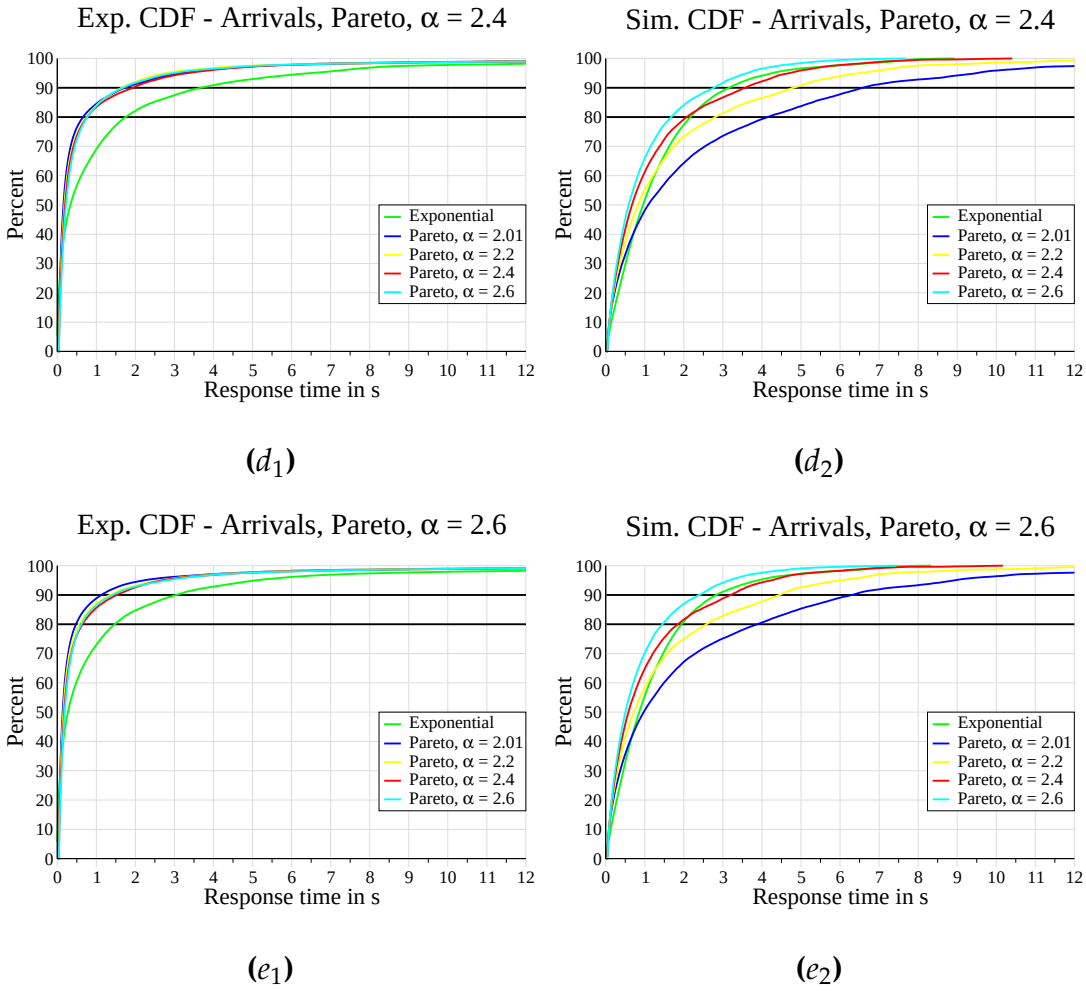


Figure 7.5: *Cumulative Distribution functions for response times obtained through experiments and simulations. The straight, black lines depict SLA limits for 80% and 90%. (a_1) Experimental, exponential inter-arrival, (a_2) Hand simulation, exponential inter-arrival, (b_1) Experimental, Pareto inter-arrival w/ $\alpha = 2.01$, (b_2) Hand simulation, Pareto inter-arrival w/ $\alpha = 2.01$, (c_1) Experimental, Pareto inter-arrival w/ $\alpha = 2.2$, (c_2) Hand simulation, Pareto inter-arrival w/ $\alpha = 2.2$, (d_1) Experimental, Pareto inter-arrival w/ $\alpha = 2.4$, (d_2) Hand simulation, Pareto inter-arrival w/ $\alpha = 2.4$, (e_1) Experimental, Pareto inter-arrival w/ $\alpha = 2.6$, (e_2) Hand simulation, Pareto inter-arrival w/ $\alpha = 2.6$.*

sharing has on response time tails there are an intrinsic lack of typical values. Instead, it might be more fruitful to consider the likelihood of being within a certain agreement. For instance, one could consider 80% or 90% compliance of a service level target. Depending on the nature of traffic the differences between these two levels can be quite significant. This becomes quite apparent if we plot the *Cumulative Distribution Functions(CDF)* for the experimental and simulated response time distributions (see figures 7.5). Because of the large deviations of values, going from 80% to 90% is not trivial from a service providers point of view. The CDF plots show that going from 80% to 90% in Service Level confidence means, more or less, a double of the service level target.

We also note that the worst scenario is when we have high variance inter-arrival time distribution and a regular exponential service requirement. This is an interesting result with regards to capacity planning. If capacity planning is based on assumptions that inter-arrival times are highly variable and the service process has exponentially distributed service times, there will always be an overprovisioning for our experimental setup. In other words, we will always be able to meet SLA agreements. For FCFS modelling of web servers, SLA compliance in any case is given by assumptions of high variance in both inter-arrival and service time. Moreover, capacity planning based on the latter will result in overprovisioning also for the web server. Indeed, considering the web server to be a FCFS system will enable compliance of SLA's and might therefore be a good strategy for performance evaluation to meet SLAs even though this can possibly result in significant excess capacity. However, as becomes apparent later, this presupposes the absence of inter-arrival self-similarity.

Because of the high variance in queue response time and the lack of a typical value, mean values are hardly useful as performance metric. Thus, average response time formulas, even if they were accurate, are not specifically applicable for specifying service level targets for web server systems.

7.4 Self-similar traffic

Herein we describe and explain the results of experiments with self-similar inter-arrival times and general service requirement. As mentioned previously, the formulas tested in the previous section are only valid for independent input. For long range dependent distributions no simple formula for calculating the expected response time exists. The objective of this section is therefore only to investigate performance differences through simulations and experiments.

Like before, the measured inter-arrival times where extracted in order to check whether the statistical properties of the distributions where conserved. More specifically, the inter-arrival distributions were checked for self-similarity using the `SELFIS` tool. The results can be viewed in section A.3.1 and shows that self similarity was well preserved by the client.

It was desirable with inter-arrival distribution variance equal to that of the exponential inter-arrival distribution from previous tests. A closer look at the standard deviations of the inter-arrival distributions showed that $\sigma \approx 0.02s$ for all the distributions. Although the generating function has possibility for defining distribution

A \ S	**Exp**	P, $\alpha = 2.01$	P, $\alpha = 2.2$	P, $\alpha = 2.4$	P, $\alpha = 2.6$
$H = 0.5$	0.98 ± 3.42	0.55 ± 3.32	0.62 ± 4.11	0.57 ± 3.01	0.55 ± 3.44
$H = 0.6$	1.06 ± 3.97	0.63 ± 4.39	0.63 ± 3.67	0.62 ± 3.12	0.60 ± 3.10
$H = 0.7$	1.45 ± 4.35	0.70 ± 4.09	0.81 ± 3.89	0.80 ± 3.59	0.70 ± 2.90
$H = 0.8$	2.70 ± 5.69	1.57 ± 5.66	1.63 ± 5.33	1.59 ± 4.59	1.76 ± 4.56

Table 7.8: *Mean response time and standard deviations derived from experiments using inter-arrival time distributions with varying degree of self-similarity. Response times are given in seconds.*

variance and took a variance equal to the exponential inter-arrival time distribution as parameter, the resulting distribution had a much lower spread in data. Adjusting the variance parameter to higher values gave the same results, and we believe that the cause for this is a bug in the `Math::Random::Brownian` module or its underlying C function calls. A direct comparison with exponentially distributed traffic is therefore not possible.

In the following we present results from experiments and simulations, succeeded by an analysis of these results.

7.4.1 Experimental results

We first present the results of the experimental runs. As before the, response times were extracted from traffic captures and the resulting dataset was subject to calculation of mean and standard deviation. Table 7.8 summarizes the results. Again, the response times are dominated by extremely high variance. The response time increases in accordance with the level of inter-arrival self-similarity. However, the increase is not linear. In other words, increasing self-similarity degree with 0.1 has different effect depending on where you are at the self-similarity scale. For instance, going from $H = 0.7$ to $H = 0.8$ affects response time more adversely than going from $H = 0.6$ to $H = 0.7$.

7.4.2 Hand simulation results

As experienced in experimental results presented in section 7.3, a skew of request inter-arrival time is imposed by the system. Therefore, we do not care to simulate using the original distributions as this does not comprise valid foundation for comparison. Instead we simulate using the inter-arrival time and service requirement extracted by the `doAnalysis.pl` script. The results of these simulations are presented in table 7.9. Generally, there are lower variance in the response time distributions as could be expected from the previous experiments. The results for exponential service time are fairly in accordance with experimental results. However, changing the service distribution to high variance Pareto results in differences, again as could be expected. The self-similar traffic seems to have more or less the same effect on both systems and the reason for the response time behaviour is elaborated on in the following.

A \ S	**Exp**	P, $\alpha = 2.01$	P, $\alpha = 2.2$	P, $\alpha = 2.4$	P, $\alpha = 2.6$
$H = 0.5$	0.93 ± 0.94	1.90 ± 2.96	1.17 ± 1.74	0.77 ± 1.08	0.56 ± 0.73
$H = 0.6$	0.96 ± 0.99	1.96 ± 2.95	1.24 ± 1.76	0.84 ± 1.14	0.63 ± 0.81
$H = 0.7$	1.36 ± 1.70	2.20 ± 3.07	1.46 ± 2.00	1.09 ± 1.47	0.88 ± 1.17
$H = 0.8$	2.53 ± 2.95	2.98 ± 3.54	2.47 ± 3.01	2.28 ± 2.91	2.19 ± 2.98

Table 7.9: *Simulation results using the measured inter-arrival times for self-similar input traffic. Response times are given in seconds.*

7.4.3 Analysis

Many of the phenomena previously elaborated on are also applicable for explaining the results seen in this section. Processor sharing discipline causes the differences in variance between simulation and experimental results. Pareto distribution characteristics together with difference in serving discipline can explain the difference in mean response time between experimental and simulation results.

What is new in this experiment is that the variance is approximately equal for every *inter-arrival distribution*. Thus, the differences in response time have to be caused by other distribution properties. Differences in response time are mainly caused by the long range dependency of the inter-arrival times.

Clustered inter-arrival

Self-similarity in a networking context means clustered or bursty arrivals of requests. In other words, requests arrive in groups with similar inter-arrival times. The degree to which one can observe this phenomenon is determined by the Hurst exponent. Thus, for larger values of the Hurst exponent, a larger degree of clustering can be observed. As there is no performance gain for future jobs associated with idle periods, what is decisive for average response time is the periods of queueing. I.e., it is the periods where jobs arrive faster than the system can process them that cause diminished performance. The duration of these periods are associated with the Hurst exponent. For high values of the exponent, the periods of queue build up becomes on average longer, causing the sojourn time for the queued jobs to aggregate. To substantiate this statement, *local averaging* (as briefly described in section 4.3) of inter-arrival time distributions for the experiments is employed and the resulting plots are showed in 7.7. When doing local averaging values are averaged over an interval of a specific size. For distributions with low level of clustering, the variances between the averaging points are low, i.e. the averaging intervals will be sparsely spread around the mean value. For higher levels an increase in variance between the averaging points can be seen. The original distributions have the same mean and variance, so this behaviour must be due to values of one type being clustered together with other values of the same type. Low value for an averaging point means that a considerable majority of the interval values is below the mean value. For averaging points of high value most of the interval values are above the mean value. Thus, there is a clustering of inter-

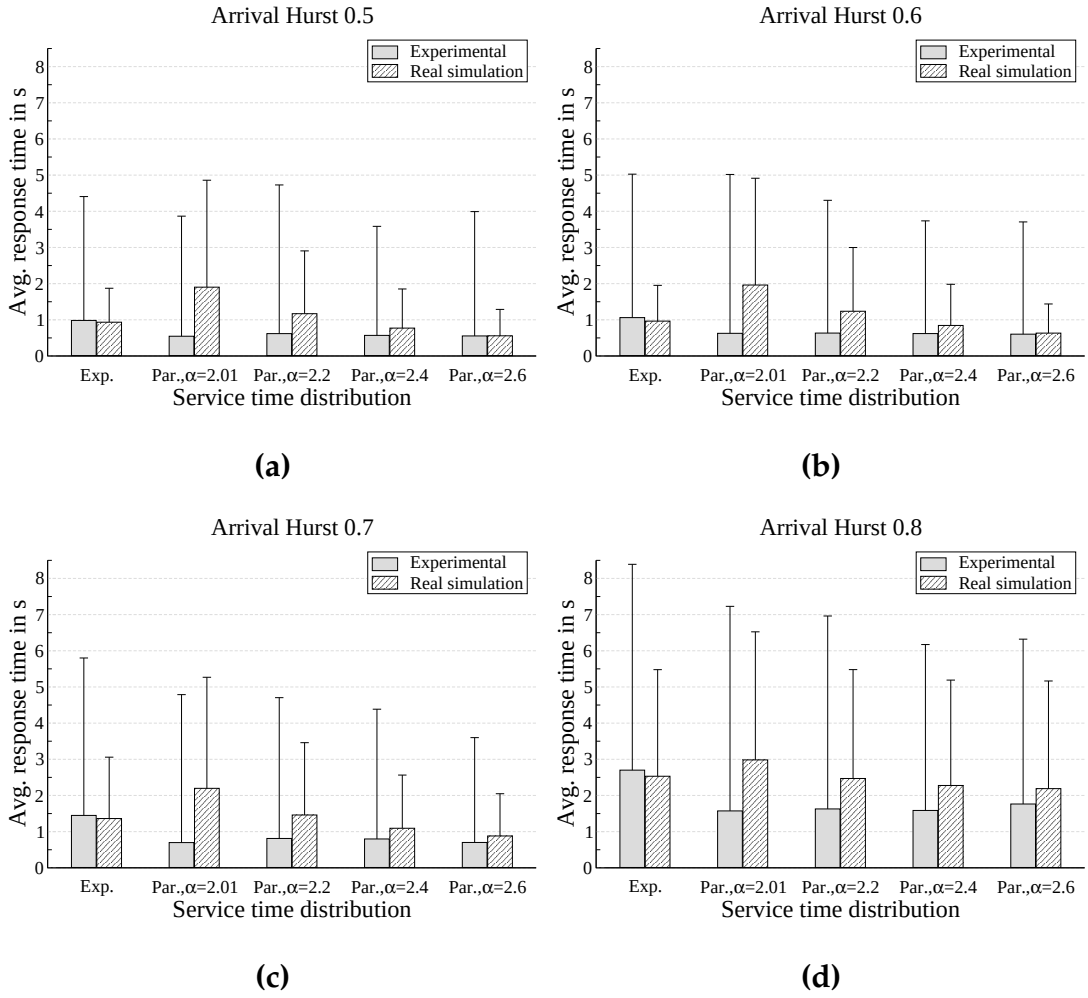


Figure 7.6: *Barcharts showing mean response time and associated standard deviation.* ***(a)*** *Hurst exponent 0.5,* ***(b)*** *Hurst exponent 0.6,* ***(c)*** *Hurst exponent 0.7****(d)*** *Hurst exponent 0.8.*

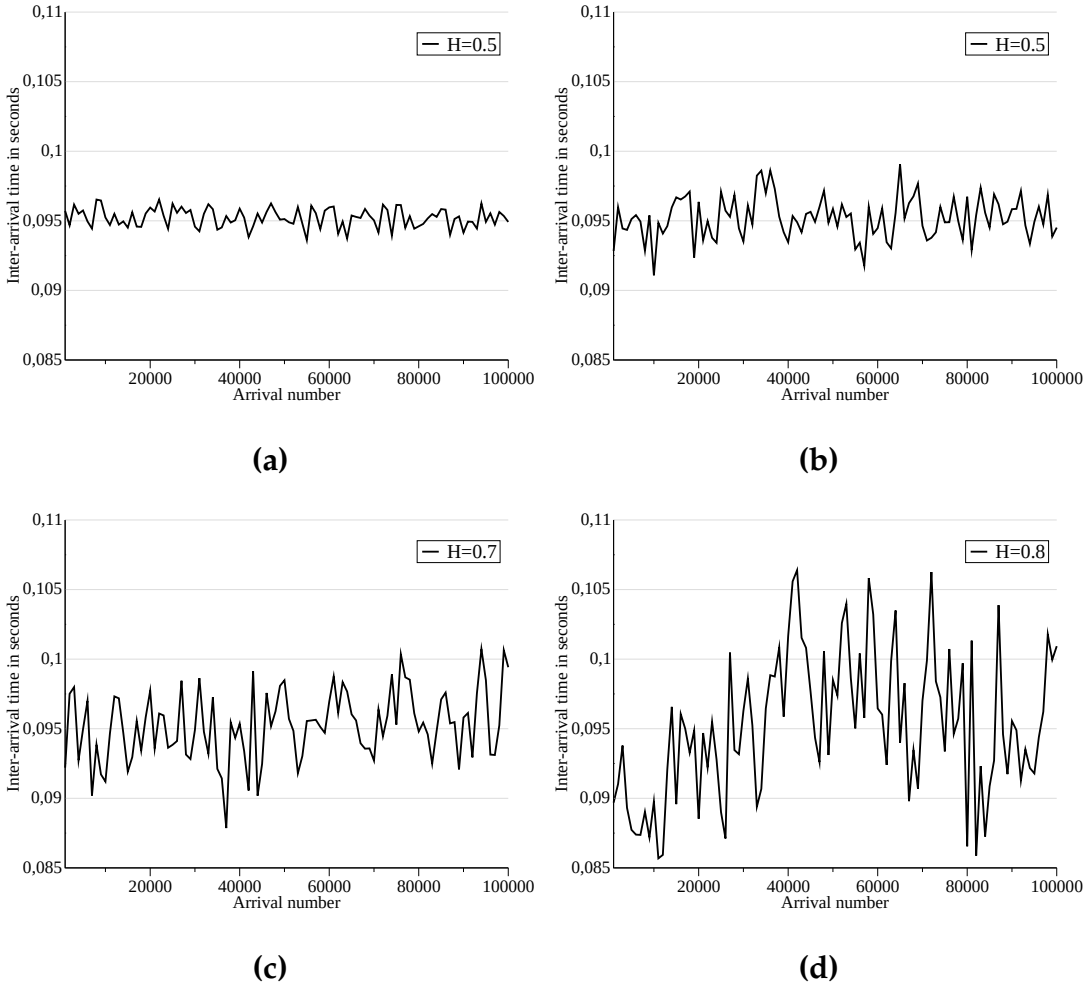


Figure 7.7: *Local averaging of self-similar input traffic with aggregation level of* 1000*. There are clearly higher variances in the averaging points as we move towards higher degree of self-similarity.* ***(a)*** *Hurst exponent 0.5,* ***(b)*** *Hurst exponent 0.6,* ***(c)*** *Hurst exponent 0.7****(d)*** *Hurst exponent 0.8.*

arrival times with low and high values. The clusters with low arrival intervals causes queueing to occur and response time to aggregate. Therefore, two arrival processes with similar expectation value and variance, but different levels of self-similarity, results in different response times. In doing worst case scenario modelling as previously suggested, self-similar traffic characteristics also have to be taken into account.

We note that since the standard deviations in inter-arrival distributions are relatively small ($\approx 0.02s$), the difference between low inter-arrival time and high is relatively low. It can therefore be argued that the system haven't been subject to bursty traffic self-similar traffic since bursty traffic are characterized by peak rates several orders of magnitude the average rate. A higher distribution variance might have enabled studies under bursty traffic conditions. As mentioned, changing the distribution variance was not possible with our distribution generation method.

7.5 Experimental difficulties and problems

During the course of planning and conducting the experiments, several mishaps, errors and caveats were experienced. Herein we briefly review these and how they were solved.

One of the classical mistakes when dealing with random number generation is consistently seeding the number generator with the same seed. Early experiments suffered from this mistake and therefore yielded very low results for average response time. The reason for this is simple. If the same seed is used for subsequent initiations of a random number generator, the shape of the resulting distributions will be similar. Even if the expected service requirement and inter-arrival time was different in value, an equal seed results in equal shapes for the resulting distributions. In a queueing sense this has the effect of low or zero queue build up due to the fact that short inter-arrival times are preceded by jobs with low service requirement. In the case of equal distribution types for service and inter-arrival they will follow each other exactly with a skew according to the difference in expected values as specified by the traffic intensity. I.e., the results for response time of an experiment with expected service time $0.09s$ will also be $0.09s$ due to the fact that the inter-arrival time always is slightly higher than the service time requirement of the preceding job. This caveat was fixed by modifying the distribution generating script so that each generation read out the interrupt counter from `/proc/interrupts` and used this as seed.

Another problem that arose was the timing granularity and accuracy of the HTTP traffic generating client. At first, the client was implemented using the `usleep()` system call for facilitation of sleep between each thread spawning. However, `usleep()` only *tries* to suspend the application in question for the specified time; it does not guarantee a call back upon sleep time expiry. A more robust and accurate timing functionality thus had to be implemented. Using similar approach as the web benchmarking tool `httperf` [58] resulted in some timing accuracy improvement. `httperf` exploits the system call `select()` for highly accurate timing. The `select()` call waits for a number of file descriptors to change status for a specified period of time [59]. Although this method provided better timing granularity and control, the client did not achieve accuracy comparable to that of `httperf`, especially for sub $10ms$ sleep times.

The `httperf` code is complex and conceptually very different from our custom designed client. Thorough investigation of the `httperf` code is therefore necessary in order to understand the cause for these differences, something that time didn't allow.

More problems were to occur. For certain experimental runs a premature termination by the client application was observed. This was especially a problem for experiments with self-similar input traffic with high level of self similarity ($H \geq 0.8$). After doing some investigation and reasoning, the likely cause for this misbehaviour was found to be limits imposed by the operating systems. By following performance tuning advices[1] the misbehaviour was reduced. However, for really large values of the Hurst exponent, the problem still persisted. Initially it was planned to do experiments with Hurst exponent as high as 0.9. However, this had to be abandoned as we couldn't fix the cause for abrupt termination by the client.

Other minor problems also occurred, like sheer programming errors and misconfiguration of participating machines. All in all, the experiments, containing 45 part experiments lasting 3 hours each, were run between $5 - 6$ times before successful and error free completion.

[1]Performance tuning advices were available at this site the following site:
`http://wwwx.cs.unc.edu/~sparkst/howto/network_tuning.php`

Chapter 8

Conclusions

"In any sufficiently complex system, no one person can know how everything works" - George's law

Web server operation is complex. It is an interplay between hardware, software and protocols, and tracing causes for behavioral patterns is not trivial. Adding to the complexity are the awkward characteristics observed for network traffic. This thesis has attempted to make simple approaches to complex problems in order to investigate if simple tools can predict response time and be useful for design of service level agreements. In addition, studies on the effect of varying statistical properties for arrival and service processes on response time have been conducted.

In the following we summarize our results with regards to the objectives previously listed. We cannot make categorical conclusions as the response time variance is relatively high which causes the observed differences to be statistically insignificant. We therefore concentrate on the trends and subsequent discussion should be seen in the light of this.

The question asked was whether or not response time was affected by varying the statistical properties apart from the mean value of the arrival and service process. Our results support this hypothesis. However, varying traffic characteristics did not have the effect predicted by queueing formulas. According to theory, increased distribution variance should give a larger average response time. As we were able to observe this phenomenon in the $G/M/1$ case, increase of *service time* variance proved to have the opposite effect. Our experimental results consistently showed a decrease in average response time average for moving from exponential to Pareto distributed service times. We note that our datasets of 100000 data points might be too sparse in order for convergence to occur. However, if infinitely many data points are needed in order for the formulas to be valid, they have lost their practical value. The formulas did show more promise for the simulated FCFS queues as they were able to capture trends, but even here the formula overestimated the response time mean value.

As the formulas are heavy traffic approximations and said to be valid as the traffic intensity reaches 1, the traffic intensity level which where chosen herein might not be sufficiently high. What is strange, however, is the discrepancies between formula,

simulated and experimental $M/G/1$ queues as the formula in these cases have been theoretically proven to be exact with regards to the state transition model. Again, the distribution sizes might prevent average value convergence. In order to evaluate the formula for these queues, a larger set of data would be beneficial. However, formulas that have these limits on area of application are not of any practical value for providers of web services.

In this work the effects of arrival process self-similarity has also been studied. These studies where not targeted to do comparison with theoretical values as the response time formulas relies on independent service and arrival processes. For self-similar traffic, each arrival possibly has a large correlation with previous events and thus are not independent. The results of our experiments indicate that the degree of self-similarity significantly affects server response time. Especially, as the level of self-similarity increases, the effect seems to aggregate. It was originally planned to compare the results of these studies with experimental results for $M/M/1$ queues. But since we were not able to generate self-similar distributions with an equal level of dataset variance, this had to be abandoned. The results of the self-similarity experiments indicates that not only are the first two moments of the arrival and service process decisive for response time; the history or clustering of short and long inter-arrivals is also of great importance.

We also stated as one of the hypotheses that web server systems can accurately be modelled as a single FCFS queue. Comparison of results from simulation of a single FCFS queueing system and experiments does not substantiate this. For exponentially distributed inter-arrival and service time, mean response time from simulation are not that different from that obtained through experiments. However, as the variance of service time increases, the gap between experimental and simulation results is growing. Again we note that the picture could be different if larger datasets were available for analysis.

What was also found through our studies was the increase in response time variability when going from simulated to experimental results. The tail behaviour of response time distribution is largely different from the FCFS queue. Response time tail behaviour is a focal point for many SLAs [60]. In considering usual service level targets we found that highly variable inter-arrival time and exponentially distributed service times comprised the worst case scenario. Using this as basis for capacity planning will enable a service provider to be in compliance with service level targets for all the traffic types studied herein. Moreover, assuming highly variable service and inter-arrival times and FCFS scheduling will create an even worse scenario in terms service level target value. Thus, it will provide a buffer against unforeseen traffic fluctuations. Such an approach will inevitably cause overprovisioning, but the payoff is that service providers will generally be able to meet SLAs a significant larger portion of the time.

The phenomena studied herein are complex and would possibly require a significant amount of research in order to be fully understood. However, it is important that efforts are made in understanding network traffic characteristics and its interplay with queueing systems in order to equip us with appropriate capacity planning techniques. Future work will hopefully get us closer to this goal.

8.1 Future work

Through working with this thesis many ideas for future work have been forged. Alas, sheer time limitations have made us unable to pursue them at this time.

First and foremost, experiments with processor sharing simulators might provide knowledge of how appropriate it is to model web servers as single queues with PS scheduling algorithm. Computers do not only operate with a simple round robin scheduling algorithm; the jobs also have priorities. In addition, web server software might impose their own priorities, for instance giving lingering jobs lower priority to avoid starvation of shorter jobs. Also, the web server has a number of queues working in serial. For the time being, we do not have sufficient knowledge to say anything about how this affects response time.

We neither had the time nor means for an in-depth analysis of self-similar traffic characteristics and its effect on web server systems. We were not able to adjust the distribution variance which made investigations of high variable self-similar traffic impossible. It was also noted in [38] that the most appropriate way for generating self-similar traffic was through many, preferably over 100, independent ON/OFF processes with the on and off periods following a heavy-tailed distribution. Functionality for such ON/OFF processes could be implemented into the client constructed in conjunction with this thesis and make the client distributed to achieve the number of recommended sources. Such studies could possibly be conducted using a virtual machine environment.

Network traffic is subject to daily, hourly, weekly and seasonal fluctuations. I.e., the peak periods for certain periods of the day a certain day of the week for a certain time of year are not necessarily the same. A way of doing real time adaption to the load demand would therefore be greatly beneficial. If a method for predicting the response time for each request as it arrives was available, virtualization could be used as a way of doing real time resource allocation in order to meet QoS requirements. Prediction of the request sojourn time could be based on the number of jobs present in the system upon arrival and an approximation of service requirement for the job. In this way, spare capacity of other machines could be utilized to minimize response time.

For a site not required to meet strict SLAs, the psychological mechanisms of humans could be exploited. As indicated by [43], the difference in perceived quality for incremental and non-incremental loading of web pages is quite significant. Which loading method a request results in could be decided by the current load at the web server. Moreover, one could exploit that users generally becomes less patient during a session. In a controlled way, assigning less processing time to requests at the beginning of a session, keeping most resources busy with handling requests of users in the middle or the end of sessions might raise the general perception of the Quality of Service received at the site. The timing scheme could take into account values as listed in table 2.2.

This thesis would have greatly benefited from a narrower scope. In our enthusiasm and willingness to learn, possibly too many aspects of the studied fields were included. Also, the thesis objectives changed during the course of the project period,

and we ended up with different project that was originally planned. Because of timing limitations, this subsequently hindered a thorough, in-depth study of every experimental case. The learning curve has been steep and we have been introduced to vast amount of previously unknown theory and technologies. The development of the custom designed test suite was time consuming, especially due to lack of C programming experience. Also, difficulties were met on several occasions during the experiment period which required us to start over. Given more time we would possibly have come further in our studies of the web server. However, it is our hope that the work here has contributed to a heightened awareness for the studied matters and also uncovered some of the caveats with such studies. There seems to be no end to the proliferation of Internet services and it is therefore of the utmost importance that we have tools that enable accurate and appropriate capacity planning for Internet services.

Bibliography

[1] V. Cardellini, E. Casalicchio, and M. Colajanni. A performance study of distributed architectures for the quality of web services. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, volume 9, page 9019, Washington, DC, USA, 2001. IEEE Computer Society.

[2] T. B. Fowler. A short tutorial on fractals and internet traffic. *The Telecommunications Review*, pages 1–14, 1999.

[3] Martin Arlitt and Tai Jin. Workload characterization study of the 1998 world cup web site. *IEEE NETWORK*, 14(3):30–37, May-June 2000.

[4] Alberto Leon-Garcia and Indra Widjaja. *Communication Networks: Fundamental concepts and Key architectures*. McGraw-Hill, 2 edition, 2004.

[5] R. Fielding, J. Gettys, J. Mogul, H.Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, IETF, June 1999.

[6] R. Fielding, H.Frystyk, and T. Berners-Lee. Hypertext transfer protocol – http/1.0. RFC 1945, IETF, May 1996.

[7] Balachander Krishnamurphy, Jeffrey C. Mogul, and David M. Kristol. Key differences between http/1.0 and http/1.1. *Comput. Networks*, 31(11-16):1737–1751, 1999.

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

[9] L.P. Slothouber. A model of web server performance. In *The 5th International World Wide Web Conference, Paris, France*, 1996.

[10] Athula Ginige and San Murugesan. Guest editors' introduction: Web engineering: An introduction. *IEEE MultiMedia*, 8(1):14–18, 2001.

[11] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[12] Tony Bourke. *Server load balancing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

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

[14] The Apache Software Foundation. *Apache2 Documentation*.

[15] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[16] Mark Burgess. *Analytical Network and System Administration: Managing Human-Computer Systems*. John Wiley & Sons, Ltd., 2004.

[17] Jan Beran. *Statistics for Long-Memory Processes*. Chapman & Hall/CRC, October 1994.

[18] Kihong Park and Walter Willinger, editors. *Self-Similar Network Traffic and Performance Evaluation*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[19] Martin J. Fischer and Carl M. Harris. A method for analyzing congestion in pareto and related queues. *Telecommunications Review*, pages 1–16, 1999.

[20] 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.

[21] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the world wide web. pages 3–25, 1998.

[22] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[23] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[24] Free Software Foundation. *GSL - GNU Scientific Library*.

[25] Ton Dieker. Simulation of fractional brownian motion. Master's thesis, University of Twente, 2002.

[26] H. G. Perros. *Computer Simulation Techniques: The definitive introduction*. Available for free download from http://www.csc.ncsu.edu/faculty/perros/books.html, 2004.

[27] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, 1994.

[28] K. Raatikainen. Symptoms of self-similarity in measured arrival process of ethernet packets to a file server, 1994.

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

[30] Hyoung-Kee Choi and John O. Limb. A behavioral model of web traffic. In *ICNP '99: Proceedings of the Seventh Annual International Conference on Network Protocols*, page 327, Washington, DC, USA, 1999. IEEE Computer Society.

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

[32] Martin J. Fischer and Thomas B. Fowler. Fractals, heavy tails and the internet. *Sigma*, pages 11–16, 2001.

[33] Bruce A. Mah. An empirical model of http network traffic. In *INFOCOM '97: Proceedings of the INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution*, page 592, Washington, DC, USA, 1997. IEEE Computer Society.

[34] Kenneth J. Christensen and Nandini J. Javagal. Prediction of future world wide web traffic characteristics for capacity planning. *Int. J. Netw. Manag.*, 7(5):264–276, 1997.

[35] Mark Meiss, Filippo Menczer, and Alessandro Vespignani. On the lack of typical behavior in the global web traffic network. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 510–518, New York, NY, USA, 2005. ACM Press.

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

[37] Jin Cao, William S. Cleveland, Dong Lin, and Don X. Sun. On the nonstationarity of internet traffic. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 102–112, New York, NY, USA, 2001. ACM Press.

[38] G. Horn, A. Kvalbein, J. Blomskøld, and E. Nilsen. An empirical comparison of generators for self similar simulated traffic. *Submitted to Elsevier Performance Evaluation*, 2005.

[39] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Trans. Netw.*, 5(1):71–86, 1997.

[40] S. Deng. Empirical model of www document arrivals at access link, 1996.

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

[42] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an m/g/1/k*ps queue. In *Telecommunications, 2003. ICT 2003. 10th International Conference on*, volume 2, pages 1501–1506, Feb/March 2003.

[43] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. *Comput. Networks*, 33(1-6):1–16, 2000.

[44] Eric W. Weisstein. Exponential distribution. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ExponentialDistribution.html.

[45] H. O. A. Wold and P. Whittle. A model explaining the pareto distribution of wealth. *Econometrica, Journal of the Econometric Society*, 25(4):591–595, October 1957.

[46] Eric W. Weisstein. Pareto distribution. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ParetoDistribution.html.

[47] Franklin A. Graybill and Hariharan K. Iyer. *Regression analysis: concepts and applications*. Duxbury Press, 1994.

[48] Donald Gross and Carl M. Harris. *Fundamentals of queueing theory*. John Wiley & Sons, Inc., New York, NY, USA, 3 edition, 1998.

[49] Eric W. Weisstein. Geometric series. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GeometricSeries.html.

[50] Randolph Nelson. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[51] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[52] Richard C. Larson and Amedeo R. Odoni. *Urban Operations Research*. Prentice-Hall, 1981. Online book: http://web.mit.edu/urban_or_book/www/book/.

[53] David L. Jagerman, Baris Balcõglu, Tayfur Altiok, and Benjamin Melamed. Mean waiting time approximations in the g/g/1 queue. *Queueing Syst. Theory Appl.*, 46(3-4):481–506, 2004.

[54] Leonard Kleinrock. *Queueing Systems: Computer Applications*, volume 2. John Wiley & Sons, Inc., 1976.

[55] M. Sakata, S. Noguchi, and J. Oizumi. An analysis of the m/g/1 queue under round-robin scheduling. *Operations Research*, 19(2):371–385, March - April 1971.

[56] Thomas Karagiannis, Michalis Faloutsos, and Mart Molle. A user-friendly self-similarity analysis tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):81–93, 2003.

[57] David W. Kelton, Randall P. Sadowski, and Deborah A. Sadowski. *Simulation with Arena*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[58] David Mosberger and Tai Jin. httperf: a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[59] *SELECT(2),Linux Programmer's Manual*.

[60] Bruno Abrahao, Virgilio Almeida, Jussara Almeida, Alex Zhang, Dirk Beyer, and Fereydoon Safai. Self-adaptive sla-driven capacity management for internet services. In *10th IEEE/IFIP Network Operations and Management Symposium*, pages 557–568, 2006.

Appendix A

A.1 Source code

A.1.1 HTTP client

```
/*****************************************************/
/*                                                   */
/* File: client.c                                    */
/*                                                   */
/* Created: spring 2006                              */
/*                                                   */
/* Author: Jon Henrik Bjoernstad                     */
/*                                                   */
/* Revision: 0.01                                    */
/*                                                   */
/* Compile:   gcc-3.4 -o client client.c -lpthread   */
/*                                                   */
/* Usage: ./client -a <a-file> -s <s-file> -h <server>  */
/*        [-p]                                       */
/*                                                   */
/* Description: A small multithreaded HTTP client that  */
/* generates web workload based on inter-arrival     */
/* and service requirement distributions.            */
/*****************************************************/
#include <stdio.h>
#include <sys/time.h>
#include <errno.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
```

```
30 #include <unistd.h>
31
32 #define closesocket(s) close(s)
33 #define SEC_TO_USEC 1000000
34 #define CFGST_INVALID_SOCKET  (-1)
35 /* http used to be requested on port 80 */
36 #define HTTP_PORT 80
37
38 struct dvector {
39   double *vals;     /* array of doubles */
40   size_t numVals;   /* number of values in vals */
41   size_t maxVals;   /* size of current vals alloc */
42 };
43
44 struct conn_info {
45   struct hostent *host_info;
46   int serv_time;
47 };
48
49 /* Need this for strtod() to work*/
50 extern int select();
51 extern double strtod();
52
53 int print = 0;
54
55 /* Using select to induce sleep times
56 static int rselect(double time)
57 {
58   struct timeval tv;
59   tv.tv_sec  = (long) time;
60   tv.tv_usec = (long)((time - tv.tv_sec)* 1e6);
61   int rc;
62   do {
63     rc = select( 0, NULL, NULL, NULL, &tv);
64   } while ( rc == -1 && errno == EINTR );
65   return rc;
66 }
67
68 /* Method for reading contents of file into a custom
69 datastructure */
70 struct dvector read_file_contents(char *filename)
71 {
72   struct dvector times = {NULL, 0, 0};
73   char buff[1024];
74   FILE * fin = fopen(filename, "r");
```

```
75    if (!fin)
76    {
77      perror("Unable to open input file");
78      exit(-1);
79    }
80    printf("Opening %s\n",filename);
81    /* Read each line, one double value per line */
82    while (fgets(buff, sizeof(buff), fin))
83    {
84      /* Parse a double value from the line */
85      char *endp;
86      double num = strtod(buff, &endp);
87      /* if we got a value, add it to the vector */
88      if (endp != buff)
89      {
90              if (times.maxVals <= times.numVals)
91        {
92          /* If we ran out of slots in
93          the current alloc, grow it*/
94          times.vals = (double *)realloc(times.vals,
95                          (times.maxVals += 100)*sizeof(double));
96          if (times.vals == NULL)
97          {
98            perror("Unable to allocate memory");
99            exit(-2);
100         }
101       }
102       /* Add our value to the vector */
103       times.vals[times.numVals++] = num;
104     }
105   }
106   fclose(fin);
107   return times;
108 }
109
110 /* Method invoked as pthread, does a HTTP request
111 and reads the answer */
112 void *get_page(void* host_ent)
113 {
114   char buffer[8192];
115   int count, sock;
116   struct sockaddr_in server;
117   struct hostent *host_info;
118   struct timeval starttime, endtime;
119   double duration;
```

```
120   struct timezone tz;
121   struct conn_info *co_info;
122
123   /* Structure containing connection info */
124   co_info = host_ent;
125         host_info = co_info->host_info;
126   memset( &server, 0, sizeof (server));
127
128   /* Initialize socket */
129   sock = socket( PF_INET, SOCK_STREAM, 0);
130   server.sin_family = AF_INET;
131   server.sin_port = htons( HTTP_PORT);
132   memcpy( (char *)&server.sin_addr,
133           host_info->h_addr, host_info->h_length);
134
135   if (sock < 0)
136   {
137     perror( "failed to create socket");
138     exit(1);
139   }
140   gettimeofday(&starttime,&tz);
141   if ( connect( sock, (struct sockaddr*)&server,
142                 sizeof(server)) < 0)
143   {
144     /* Can't connect */
145     printf("can't connect to server");
146   }
147
148   /* Create and send the http GET request */
149   int serv_time = co_info->serv_time;
150   sprintf( buffer, "GET /?iter=%d HTTP/1.0\n\n", serv_time);
151   send( sock, buffer, strlen( buffer), 0);
152
153   /* Get the answer from server and put it out to stdout */
154   do {
155     count = recv( sock, buffer, sizeof(buffer), 0);
156     if(print == 1)
157     {
158       write( 1, buffer, count);
159     }
160   }
161   while (count > 0);
162   gettimeofday(&endtime,&tz);
163   /* Close the connection to the server */
164   closesocket(sock);
```

```
165   double e_usec =
166           ((double)endtime.tv_usec/(double)SEC_TO_USEC);
167   double s_usec =
168           ((double)starttime.tv_usec/(double)SEC_TO_USEC);
169   duration = e_usec - s_usec;
170   duration += (double)endtime.tv_sec -(double)starttime.tv_sec;
171   if(print == 1)
172   {
173     /* Print measured response time */
174     printf("\n\nResponse time: %g\n", duration);
175   }
176 }
177
178 /***************** MAIN ********************/
179 int main( int argc, char **argv)
180 {
181   int i;
182   /* Read buffer */
183   char buff[1024];
184   char *s_addr = NULL;
185   char *arrival_timefile = NULL;
186   char *service_timefile = NULL;
187   int index;
188   int c;
189   opterr = 0;
190
191   /* Read command line arguments */
192   while ((c = getopt (argc, argv, "h:a:s:p")) != -1)
193   switch (c)
194   {
195     case 'h':
196       s_addr = optarg;
197       break;
198     case 'a':
199       arrival_timefile = optarg;
200       break;
201     case 's':
202       service_timefile = optarg;
203       break;
204     case 'p':
205       print = 1;
206       break;
207     case '?':
208       if (isprint (optopt))
209         fprintf (stderr, "Unknown option `-%c'.\n", optopt);
```

```
210       else
211           fprintf (stderr,
212                "Unknown option character '\\x%x'.\n", optopt);
213       return 1;
214     default:
215       abort ();
216   }
217
218   printf( "Server: %s\tArrival-file: %s"
219           "\tService-file: %s\tPrint: %d\n",
220           s_addr, arrival_timefile, service_timefile, print);
221
222   for (index = optind; index < argc; index++)
223     printf ("Non-option argument %s\n", argv[index]);
224
225   /* Check number of command line arguments */
226   if (  s_addr == NULL || arrival_timefile == NULL ||
227         service_timefile == NULL)
228   {
229     fprintf(stderr,
230     "Usage: %s -h <host> -a <arrival-distribution-file>"
231     "-s <service-distribution-file> [-p]\n", argv[0]);
232     exit(1);
233   }
234   /* Read contents of file into custom data structures */
235   struct dvector arrivals =
236             read_file_contents(arrival_timefile);
237   struct dvector service =
238             read_file_contents(service_timefile);
239
240   if(service.numVals != arrivals.numVals)
241   {
242     printf(
243         "Error: Uneven number of arrival and servicetimes.\n");
244     printf("No. arrivals: %d\tNo. servicetimes:%d\n",
245       arrivals.numVals, service.numVals);
246     exit(1);
247   }
248
249   /* Array holding http client threads*/
250   pthread_t callThd[arrivals.numVals];
251   struct hostent *host_info;
252   struct sockaddr_in server;
253   int status;
254   pthread_attr_t attr;
```

```
255
256   printf("Threads: %d\nServer: %s\n",
257           arrivals.numVals, s_addr);
258
259   host_info = gethostbyname(s_addr);
260   if (NULL == host_info)
261   {
262     fprintf( stderr, "unknown server: %s\n", s_addr);
263     exit(1);
264   }
265
266   pthread_attr_init(&attr);
267   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
268
269   malloc((arrivals.numVals*sizeof(pthread_t)));
270         unsigned long sle = 0;
271
272   struct conn_info connect_inf;
273   connect_inf.host_info = host_info;
274   int serv_time;
275   for(i=0;i<arrivals.numVals;i++)
276   {
277     /* Start the thread */
278     connect_inf.serv_time = (int) service.vals[i];
279     pthread_create( &callThd[i], &attr, get_page,
280                     (void*) &connect_inf);
281     rselect(arrivals.vals[i]);
282   }
283
284   printf("Started %d threads\n", arrivals.numVals);
285   pthread_attr_destroy(&attr);
286   pthread_exit(NULL);
287   return 0;
288 }
```

A.1.2 Queue simulator

```
1 /*****************************************************/
2 /*                                                   */
3 /* File: simulate_distributions.c                    */
4 /*                                                   */
5 /* Created: spring 2006                              */
6 /*                                                   */
7 /* Author: Jon Henrik Bjoernstad                     */
```

```
8  /*                                                                  */
9  /* Revision: 0.01                                                   */
10 /*                                                                  */
11 /* Compile:    gcc-3.4 -o simulate_distributions \                  */
12 /*               -lm simulate_distributions.c                       */
13 /*                                                                  */
14 /* Usage: ./simulate_distributions -a <a-file> -s \                 */
15 /*         <s-file> [-o <output-file>] [-p]                         */
16 /*                                                                  */
17 /* Description: A queueing system simulator used to                 */
18 /* replay experiments in a controlled environment. Works            */
19 /* as a First Come First Served queue                               */
20 /*******************************************************************/
21
22 #include <stdio.h>
23 #include <stdlib.h>
24 #include <string.h>
25 #include <unistd.h>
26 #include <math.h>
27
28 /* Need this for strtod() to work*/
29 extern double strtod();
30
31 struct dvector {
32   double *vals;     /* array of doubles */
33   size_t numVals;   /* number of values in vals */
34   size_t maxVals;   /* size of current vals alloc */
35 };
36
37 /* Method for reading contents of a file into custom
38 datastructure */
39 struct dvector read_file_contents(char *filename)
40 {
41   struct dvector times = {NULL, 0, 0};
42   char buff[1024];
43   FILE * fin = fopen(filename, "r");
44   if (!fin)
45   {
46     perror("Unable to open input file");
47     exit(-1);
48   }
49   /* Read each line, one double value per line */
50   while (fgets(buff, sizeof(buff), fin))
51   {
52     /* Parse a double value from the line */
```

```
53      char *endp;
54      double num = strtod(buff, &endp);
55      /* If we got a value, add it to the vector */
56      if (endp != buff)
57      {
58        if (times.maxVals <= times.numVals)
59        {
60          /* If we ran out of slots in the current
61          alloc, grow it */
62          times.vals = (double *)realloc(times.vals,
63              (times.maxVals += 100)*sizeof(double));
64
65          if (times.vals == NULL)
66          {
67            perror("Unable to allocate memory");
68            exit(-2);
69          }
70        }
71        /* Add our value to the vector */
72        times.vals[times.numVals++] = num;
73      }
74    }
75    fclose(fin);
76    return times;
77  }
78
79  /***************** MAIN ********************/
80  int main( int argc, char **argv)
81  {
82    /* Time for service starting current job. */
83    double          timeStartService = 0,
84    /* Departure time for current job. */
85    departureTime = 0,
86    /* "Real time" for arrivals. */
87    arrivalTime = 0,
88    arrTimeItem = 0,
89    servTimeItem = 0,
90    responseTimeItem = 0,
91    responseTimeSum = 0,
92    sumArrivals = 0,
93    sumService = 0,
94    avgRespTime = 0;
95    char *arrival_timefile = NULL;
96    char *service_timefile = NULL;
97    char *outputfile = NULL;
```

```
98  int index, c;
99  int print = 0;
100
101 opterr = 0;
102
103 /* Read command line arguments */
104 while ((c = getopt (argc, argv, "a:s:o:p")) != -1)
105 switch (c)
106 {
107   case 'a':
108     arrival_timefile = optarg;
109     break;
110   case 's':
111     service_timefile = optarg;
112     break;
113   case 'p':
114     print = 1;
115     break;
116   case 'o':
117     outputfile = optarg;
118     break;
119   case '?':
120     if (isprint (optopt))
121       fprintf (stderr, "Unknown option `-%c'.\n", optopt);
122     else
123       fprintf (stderr,
124         "Unknown option character `\\x%x'.\n", optopt);
125     return 1;
126   default:
127     abort ();
128 }
129 printf("Arrival-file: %s\tService-file: %s\n",
130             arrival_timefile, service_timefile);
131
132 for (index = optind; index < argc; index++)
133   printf ("Non-option argument %s\n", argv[index]);
134
135 /* Check number of command line arguments */
136 if (arrival_timefile == NULL || service_timefile == NULL)
137 {
138   fprintf(stderr, "Usage: %s -a <arrival-distribution-file>"
139     "-s <service-distribution-file>"
140     "[-o <responsetime-file>][-p]", argv[0]);
141     exit(1);
142 }
```

```
143
144   /* Read contents of input files into data structures */
145   struct dvector arrivals =
146           read_file_contents(arrival_timefile);
147   struct dvector service =
148           read_file_contents(service_timefile);
149
150   if(service.numVals != arrivals.numVals)
151   {
152     printf(
153       "Error: Uneven number of arrival and servicetimes.\n");
154     printf("No. arrivals: %d\tNo. servicetimes:%d\n",
155             arrivals.numVals, service.numVals);
156     exit(1);
157   }
158
159   /* First job, the queue is empty */
160   servTimeItem = (service.vals[0]*1.8588e-6) + 0.00099379;
161
162   /* Departure time for the first job is equal to the service
163   demand */
164
165   departureTime = servTimeItem;
166   double resp_Times[arrivals.numVals];
167   resp_Times[0] = servTimeItem;
168   responseTimeSum += servTimeItem;
169   sumService += servTimeItem;
170
171   int i = 1;
172   for (i = 1; i < arrivals.numVals; i++)
173   {
174     /*Works in the same way as the client.
175     The first inter-arrival time is the inter arrival
176     time between job 1 and job 2*/
177     arrTimeItem = arrivals.vals[i - 1];
178     servTimeItem = (service.vals[i] * 1.8588e-6) + 0.00099379;
179
180     sumArrivals += arrTimeItem;
181     sumService += servTimeItem;
182
183     /* Aggregate arrival time */
184     arrivalTime += arrTimeItem;
185
186     /*The time for the job to start service is
187     maximum of departureTime for previous job
```

```
188     and arrivalTime for current job*/
189     timeStartService = (arrivalTime < departureTime)?
190               departureTime : arrivalTime;
191
192     /*Departure time for current job is the time it
193     starts to be service plus its service demand*/
194     departureTime = timeStartService + servTimeItem;
195
196     /*Response time for current job is departure time
197     minus arrival time.*/
198     responseTimeItem = departureTime - arrivalTime;
199     resp_Times[i] = responseTimeItem;
200
201     if(print == 1)
202       printf("%g\n", responseTimeItem);
203
204     /*Summing response times for calculation of
205     average value.*/
206     responseTimeSum += responseTimeItem;
207     avgRespTime = responseTimeSum/i;
208   }
209
210   /*calculate the standard deviation*/
211   double sum_variance = 0;
212   double difference;
213
214   /* Output file containing response times */
215   FILE *ofp;
216   char obuff[1024];
217   if(outputfile != NULL)
218     ofp = fopen(outputfile, "w");
219
220   for (i=0; i < arrivals.numVals; i++)
221         {
222     difference = avgRespTime - resp_Times[i];
223     sum_variance += pow(difference, 2.0);
224     if(outputfile != NULL)
225       fprintf(ofp, "%g\n",resp_Times[i]);
226   }
227   if(outputfile != NULL)
228     fclose(ofp);
229
230   double variance = sum_variance/
231               ((double)arrivals.numVals-1.0);
232
```

```
233   /* Print summary */
234   printf("Final average response time: %g\n", avgRespTime);
235   printf("Standard deviation: %g\n", sqrt(variance));
236   printf("Number of requests: %d\n", i);
237   printf("Traffic intensity - rho: %g\n",
238         (sumService/sumArrivals));
239   return 0;
240 }
```

A.1.3 Traffic analysis script

```
1  : # *-*-perl-*-*
2  eval 'exec perl -w -S  $0 ${1+"$@"}'
3      if 0;
4  =pod
5  doAnalysis.pl: A script for extracting TCP session durations,
6  interarrival times and service requirements files.
7  Mean and standard deviation along with traffic intensity
8  is printed to stdout.
9
10 Usage: doAnalysis.pl <pcap-file>
11
12 Author: Jon Henrik Bjrnstad, spring 2006
13 =cut
14 #Import modules
15 use Net::Pcap;
16 use NetPacket::TCP;
17 use NetPacket::IP;
18 use NetPacket::Ethernet;
19 use Data::Dumper;
20 use strict;
21
22 # Check number of arguments
23 if($#ARGV != 0) {
24   print "Usage: $0 <pcap-file>\n";
25   exit(1);
26 }
27
28 my $error;
29 my $start_sec;
30 my $start_usec;
31 my %sessions = ();
32 my %complete = ();
33 my $pcap = Net::Pcap::open_offline($ARGV[0], \$error);
```

```
34 my $con_nr = 1;
35
36 # Variable for microsecond-second conversion
37 my $USEC_TO_SEC = 1000000;
38
39 # Exit if unable to open file
40 unless(defined $pcap) {
41   die "Something went terribly wrong:".$error."\n";
42 }
43
44 # Loop trough the entire file contents, for each packet
45 # call the doAnalysis subroutine with no parameters.
46 Net::Pcap::loop($pcap, -1, \&doAnalysis, '');
47
48 # Close pcap-file
49 Net::Pcap::close($pcap);
50
51 my $count = 0;
52
53 my $serv_sum = 0;
54 my $resptime_sum = 0;
55
56 # Put in new hash with connection number as key
57 foreach my $k (keys %sessions) {
58   # Check for all required entries.
59   if (
60     defined($sessions{$k}{'starttime'}) &&
61     defined($sessions{$k}{'con_nr'})    &&
62     $sessions{$k}{'resp_time'} > -1     &&
63     $sessions{$k}{'service_req'} > -1   &&
64     $sessions{$k}{'finack'} > -1        &&
65     $sessions{$k}{'status_code'} == 200 &&
66     $sessions{$k}{'RST'} == -1          &&
67     $sessions{$k}{'done'} > -1) {
68
69     %{$complete{$sessions{$k}{'con_nr'}}} =
70       %{$sessions{$k}};
71     $count++;
72
73     $resptime_sum += $sessions{$k}{'resp_time'};
74     $serv_sum += $sessions{$k}{'service_req'};
75     } else {
76     # Print connection info for incomplete connection.
77     print "Incomplete connection:\n";
78     print Dumper(\%{$sessions{$k}});
```

```
79    }
80  }
81
82  # Get path to dump file for file writing.
83  my @my_base_name= split('/',$ARGV[0]);
84  my $fn_base = "";
85
86  for(my $i = 0; $i < $#my_base_name; $i++) {
87    $fn_base .= $my_base_name[$i]."/";
88  }
89
90  open(I_ARR, ">".$fn_base."doAnalysis_interarrival.txt");
91  open(SERV, ">".$fn_base."doAnalysis_servicereq.txt");
92  open(RESP, ">".$fn_base."doAnalysis_responsetimes.txt");
93  my $prev_conn = 0;
94
95  # Variable for inter-arrival time extraction
96  my $prev_arrival = 0;
97
98  my $mean_serv = $serv_sum/$count;
99  my $mean_resp = $resptime_sum/$count;
100
101 my $var_inter_arr = 0;
102 my $int_arr_sum = 0;
103 my $var_serv = 0;
104 my $var_resp = 0;
105 foreach my $k2 ( sort {$a <=> $b} keys %complete) {
106   if($k2 < $prev_conn) {
107     print "Something wrong\n";
108   } else {
109     my $arr =
110       $complete{$k2}{'starttime'} - $prev_arrival;
111
112     #there is no interarrival between request nr 0 an
113     # request 1
114     if($arr > 0) {
115       print I_ARR "".$arr."\n";
116       $int_arr_sum += $arr;
117     }
118     $var_serv +=
119     ($complete{$k2}{'service_req'} - $mean_serv)**2;
120
121     print SERV "".$complete{$k2}{'service_req'}."\n";
122
123     $var_resp +=
```

```
124     ($complete{$k2}{'resp_time'} - $mean_resp)**2;
125     print RESP "".$complete{$k2}{'resp_time'}."\n";
126     $prev_arrival = $complete{$k2}{'starttime'};
127   }
128 }
129 #pad arrival file for later run with simulator
130 print I_ARR "0.0\n";
131 close(I_ARR);
132 close(SERV);
133 close(RESP);
134
135 my $mean_inter_arr = $int_arr_sum/($count -1);
136
137 open(I_ARR2,"".$fn_base."doAnalysis_interarrival.txt");
138
139 while (my $val = <I_ARR2>) {
140   $var_inter_arr += ($val - $mean_inter_arr)**2;
141 }
142 close(I_ARR2);
143
144 #Calculate rho based on values found by linear regression
145 my $rho = (($mean_serv * 0.0000018588) + 0.00099379)
146             / $mean_inter_arr;
147 $var_inter_arr = sqrt(($var_inter_arr/($count - 2)));
148 $var_serv = sqrt(($var_serv/($count - 1)));
149 $var_resp = sqrt(($var_resp/($count - 1)));
150 open(SUMMARY, ">".$fn_base."doAnalysis_summary.txt");
151 print SUMMARY "Mean inter-arrival time:  ".
152     $mean_inter_arr."\tStd: ".$var_inter_arr."\n";
153 print SUMMARY "Mean service requirement:".
154     $mean_serv."\tStd: ".$var_serv."\n";
155 print SUMMARY "Mean response time:".
156     $mean_resp."\tStd: ".$var_resp."\n";
157 print SUMMARY "Traffic intensity:".$rho."\n";
158 close(SUMMARY);
159
160 sub doAnalysis
161 {
162   my($user_data, $header, $packet) = @_;
163
164   if( !defined($start_sec) && !defined($start_usec)) {
165     $start_sec = $header->{tv_sec};
166     $start_usec = ($header->{tv_usec})/$USEC_TO_SEC;
167   }
168
```

```
169  my $ether_data = NetPacket::Ethernet::strip($packet);
170  my $ip_packet = NetPacket::IP->decode($ether_data);
171
172  #is it TCP?
173  if($ip_packet->{proto} == 6) {
174    my $tcp_segment =
175      NetPacket::TCP->decode($ip_packet->{data});
176
177    #initial SYN?
178    if($tcp_segment->{flags} == 2) {
179      my $s_sec =  $header->{tv_sec} - $start_sec;
180      my $s_usec = ($header->{tv_usec}/$USEC_TO_SEC)
181          - $start_usec;
182      my $time_start = $s_sec + $s_usec;
183      my $key = $tcp_segment->{src_port};
184
185      if (defined($sessions{$key}) &&
186        ($sessions{$key}{'resp_time'} == -1 ||
187        $sessions{$key}{'service_req'} == -1 ||
188        $sessions{$key}{'status_code'} == -1 ||
189        $sessions{$key}{'finack'} == -1 ||
190        $sessions{$key}{'done'} == -1))
191                        {
192        print "Unfinished connection attempt...\n";
193      }else{
194        #avoid port number wrap around problems
195        while ( defined($sessions{$key})) {
196          $key += 100000
197        }
198      }
199
200      #Initiate connection
201      %{$sessions{$key}} =
202          (
203           'starttime'   => $time_start,
204           'con_nr'      => $con_nr,
205           'resp_time'   => -1,
206           'service_req' => -1,
207           'status_code'   => -1,
208           'RST'         => -1,
209           'finack'      => -1,
210           'done'                => -1);
211      $con_nr++;
212    }
213    #extract servicerequirement from string on the form
```

```
214     #"GET /?iter=(service_req) HTTP/1.0
215     elsif($tcp_segment->{data} =~ /.*GET\s\/\?iter=(\d*).*/) {
216       my $key2 = $tcp_segment->{src_port};
217
218       #If service requirement already set, we have
219       #portnumber wraparound.
220       while (    defined($sessions{$key2}) &&
221         !($sessions{$key2}{'service_req'} < 0) ) {
222         $key2 += 100000;
223       }
224       if( defined($sessions{$key2})) {
225         $sessions{$key2}{'service_req'} = $1;
226       }
227     }
228     # Check for 200 OK
229     elsif($tcp_segment->{data} =~ /HTTP\/\d+.\d+\s(\d+)/) {
230       my $key5 = $tcp_segment->{dest_port};
231       while ( defined($sessions{$key5}) &&
232         !($sessions{$key5}{'status_code'} <0)) {
233         $key5 += 100000;
234       }
235       if(defined($sessions{$key5})){
236         $sessions{$key5}{'status_code'} = $1;
237       }
238     }
239     # Is the RST flag set?
240     elsif( ($tcp_segment->{flags} & 4) == 4) {
241       my $key6 =
242         ($tcp_segment->{src_port} == 80) ?
243         $tcp_segment->{dest_port} :
244         $tcp_segment->{src_port};
245       while(defined($sessions{$key6}) &&
246         $sessions{$key6}{done} == 1) {
247         $key6 += 100000;
248       }
249       if (defined($sessions{$key6})) {
250         $sessions{$key6}{'RST'} = 1;
251       }
252     }
253     # is it FINACK?
254     elsif($tcp_segment->{flags} == 17) {
255       # if the src_port is 80, then the server has
256       # sent the finack.
257       my $key3 =
258         ($tcp_segment->{src_port} == 80) ?
```

```
259           $tcp_segment->{dest_port} :
260           $tcp_segment->{src_port};
261         while ( defined($sessions{$key3}) &&
262           !($sessions{$key3}{'finack'} < 0) ) {
263           $key3 += 100000;
264         }
265         if(defined($sessions{$key3})){
266             $sessions{$key3}{'finack'} = 1;
267             $sessions{$key3}{'done'} = 1;
268         }
269       }
270       #Get response time from last ack
271       elsif($tcp_segment->{flags} == 16) {
272         my $key4 =
273           ($tcp_segment->{src_port} == 80) ?
274           $tcp_segment->{dest_port} :
275           $tcp_segment->{src_port};
276         while (    defined($sessions{$key4}) &&
277           !($sessions{$key4}{'resp_time'} < 0) ) {
278           $key4 += 100000;
279         }
280
281         if(        defined($sessions{$key4}) &&
282           $sessions{$key4}{'done'} == 1) {
283           #getcurrent elapsed time
284           my $time =
285           ($header->{tv_sec} - $start_sec)+
286           (($header->{tv_usec}/$USEC_TO_SEC) -
287           $start_usec);
288
289           my $resp_time =
290           $time - $sessions{$key4}{'starttime'};
291           $sessions{$key4}{'resp_time'} =
292             $resp_time;
293         }
294       }
295     }
296 }
```

A.1.4 Formula queueing simulator

```
1 eval 'exec perl -w -S  $0 ${1+"$@"}'
2     if 0;
3 =pod
```

```
4  q_sim.pl: Simulator based on approximate formulas for
5  response time of G/G/1 queues.
6
7  Usage: ./q_sim.pl
8  =cut
9
10 use strict;
11 my $RHO = 0.95;
12 my $ES =0.09;
13 my $EA =$ES/$RHO;
14
15 # Variance of expected service time, equal to the variance of
16 # exponential distributions
17 my $VEA_EXP= $EA**2;
18 my $VES_EXP= $ES**2;
19
20 my $resp_time = 0;
21 my $par_var = 0;
22
23 my %responsetimes = (
24     "E" => {},
25     "2.01" => {},
26     "2.2" => {},
27     "2.4" => {},
28     "2.6" => {});
29 my @alphas = ( "E",
30   "2.01",
31   "2.2",
32   "2.4",
33   "2.6");
34
35 # run the simulation
36 foreach my $key(keys %responsetimes ) {
37   # Is it exponential?
38   if($key eq "E") {
39     $VEA_EXP= $EA**2;
40   } else {
41     $VEA_EXP = find_variance_for_alpha_and_es ($EA, $key);
42   }
43   foreach my $alpha(@alphas) {
44     # Is it exponential?
45     if($alpha eq "E") {
46       $VES_EXP= $ES**2;
47     } else {
48       $VES_EXP = find_variance_for_alpha_and_es ($ES, $alpha);
```

```
49      }
50      # calculate the response time
51      $resp_time = calc_GG1_queue($VEA_EXP,$VES_EXP, $EA,$ES);
52      $responsetimes{$key}{$alpha} = $resp_time;
53    }
54  }
55
56  # Print the results
57  my $string = "";
58  for my $k(sort keys %responsetimes) {
59    if($string eq "") {
60      $string = "A\\S\t";
61      my @header = sort keys %{$responsetimes{$k}};
62      foreach(@header) {
63        $string .= $_."\t";
64      }
65      $string .= "\n";
66    }
67
68    $string .= $k."\t";
69    for my $cle(sort keys %{$responsetimes{$k}}){
70      $string.= "\$".$responsetimes{$k}{$cle}."\$\t";
71    }
72    $string .= "\n";
73  }
74  $string .= "-------------------------------------------------------\n\n";
75
76  print $string;
77
78  # Sub routine for finding variance based on
79  # expected value and alpha parameter
80  sub find_variance_for_alpha_and_es {
81    my ($ES, $ALPHA) = @_;
82    my $B = $ES * ($ALPHA - 1) / $ALPHA;
83    my $VAR = ($ALPHA * ($B**2))/
84        ((($ALPHA - 1)**2)*($ALPHA - 2));
85    return $VAR;
86  }
87
88  # Subroutine for calculating average response time
89  sub calc_GG1_queue
90  {
91    my ($var_arr, $var_serv, $ETA, $ETS) = @_;
92    my $RHO = $ETS/$ETA;
93    my $C_ARR = $var_arr/($ETA**2);
```

```
94    my $C_SERV = $var_serv/($ETS**2);
95    my $first = $RHO*$ETS*($C_ARR + $C_SERV);
96    my $second = 2*(1 - $RHO);
97    my $answer = ($first/$second) + $ETS;
98    my $rounded = sprintf("%.5f", $answer);
99    return $rounded;
100 }
```

A.1.5 Pareto and exponential distribution generator

```
1  : # *-*-perl-*-*
2  eval 'exec perl -w -S  $0 ${1+"$@"}'
3      if 0;
4  =pod
5  exponentialParetoMaker.pl: Script for generating distributions
6  with desired expectation values based on Pareto and
7  exponential parameters.
8
9  Usage: ./exponentialParetoMaker.pl
10 =cut
11
12 use strict;
13 use POSIX;
14 # Alpha values for pareto distribution
15 my @alpha = (2.01,2.2,2.4,2.6);
16 # Hash containing filenames for later
17 # evaluation of mean values
18 my %distros = ("ARRIVALS" => [], "SERVICE" => []);
19 # beta pareto parameter
20 my $beta = 0;
21 # number of entries in each file
22 my $number = 100000;
23
24 # Experimentally decided quantities by linear
25 # regression.
26 my $time_iteration = 0.0000018588;
27 my $overhead_time = 0.00099379;
28
29 # Set test parameters
30 my $rho = 0.95;
31 my $service_time_exp = 0.09;
32 # RHO = lambda/mu = E[S]/E[A] => E[A] = E[S]/RHO
33 my $arrival_time_exp = $service_time_exp/$rho;
34
```

```
35 # Seed for randon number generation
36 my $seed;
37
38 #service expectation value
39 my $iter_expectation_value =
40   ($service_time_exp - $overhead_time)/$time_iteration;
41
42 #Put all filenames in hash for comparison of expected values
43 my %expect_values =
44             (
45                "ARRIVALS" => $arrival_time_exp,
46                "SERVICE" => $service_time_exp);
47
48 foreach my $keys(keys %expect_values) {
49   # Set seed for Perl random variable generator
50   my $rand_seed =
51   `head -2 /proc/interrupts | grep -v CPU | awk '{print \$2}'`;
52   chomp($rand_seed);
53   srand $rand_seed;
54   # Create seed for distribution generators
55   $seed = floor(rand()*1000);
56   foreach (@alpha){
57     $beta = get_beta($expect_values{$keys}, $_);
58     my $paretofn = $keys."_paretodist_alpha_".
59             $_."_expect_".$expect_values{$keys}.".dat";
60     $distros{$keys}[$#{$distros{$keys}} + 1] = $paretofn;
61     open(DIST, "gsl-randist $seed $number pareto $_ $beta|");
62     open(PARETO, ">$paretofn");
63     while (my $line = <DIST>) {
64       if($keys eq "SERVICE") {
65       my $value = ( ($line - $overhead_time) < 0 )
66                     ? 0 : ($line - $overhead_time);
67         $line = int(($value/$time_iteration));
68                                 $line .= "\n";
69       }
70       print PARETO $line;
71     }
72     close(PARETO);
73     close(DIST);
74   }
75   my $exp_expo = $expect_values{$keys};
76   my $exponentialfn=$keys."_exponentialdist_es_".
77             $exp_expo."_count_".$number.".dat";
78   $distros{$keys}[$#{$distros{$keys}} + 1] = $exponentialfn;
79   open(EXPONENTIAL,">$exponentialfn");
```

```
80
81    #We're interested in interarrival time, not
82    # rate.... need therefore use exponential
83    open(DIST2,
84        "gsl-randist $seed $number exponential $exp_expo|");
85    while (my $line2 = <DIST2>) {
86      if($keys eq "SERVICE"){
87        my $value = ( ($line2 - $overhead_time) < 0 )
88              ? 0 : ($line2 - $overhead_time);
89        $line2 = int(($value/$time_iteration));
90                          $line2 .= "\n";
91      }
92      print EXPONENTIAL $line2;
93    }
94    close(DIST2);
95    close(EXPONENTIAL);
96  }
97
98  my @service_check;
99  my @arrivals_check;
100
101 foreach my $key(keys %distros) {
102   foreach(@{$distros{$key}}) {
103     open(FILE,$_);
104     my $sum = 0;
105     while(my $line = <FILE>) {
106       $sum += $line;
107     }
108     print "File: $_\tMean: ".($sum/$number)."\n";
109     if($_=~/SERVICE/) {
110       $service_check[$#service_check + 1] = ($sum/$number);
111     } else {
112       $arrivals_check[$#arrivals_check + 1] = ($sum/$number);
113                 }
114     close(FILE);
115   }
116 }
117 # Print originally desired values
118 print "Original values: Service iterations: "
119       .$iter_expectation_value.
120       "\t Arrival time: $arrival_time_exp\n";
121 print "Rho: ".
122   ((($iter_expectation_value*$time_iteration)+$overhead_time)
123    /$arrival_time_exp)."\n";
124
```

```
125 # Calculate traffic intensity and mean value for each
126 # constellation of service and inter-arrival distribution
127 for my $arr (@arrivals_check) {
128   for my $serv (@service_check) {
129     my $rho = (($serv*$time_iteration)+$overhead_time)/$arr);
130     print "Arr: $arr\tServ: $serv\tRho: $rho\n";
131     }
132 }
133
134 # Subroutine to get Pareto beta value
135 sub get_beta
136 {
137   my ($es, $alpha) = @_;
138   my $beta = $es * ($alpha - 1) / $alpha;
139   return $beta;
140 }
```

A.1.6 Self-similar distribution generator

```
1  : # *-*-perl-*-*
2  eval 'exec perl -w -S  $0 ${1+"$@"}'
3      if 0;
4  =pod
5  selfSimDistroMaker.pl: A script for generating self-similar
6  distributions with by employing functionality in
7  Math::Random::Brownian
8
9  Usage: ./selfSimDistroMaker.pl
10 =cut
11
12 use strict;
13 use Data::Dumper;
14 use Math::Random::Brownian;
15
16 # Values for the Hurst exponent
17 my @HURST_VALUES = (0.8,0.7,0.6,0.5);
18 my %DISTRIBUTIONS;
19 my $NUM_ARRIVALS = 100000;
20 my $RHO = 0.95;
21 my $ETS = 0.09;
22 my $ETA = $ETS/$RHO;
23 my $VARIANCE = $ETA**2;
24 my $IDEAL = $ETA*$NUM_ARRIVALS;
25
```

```
26 print "Starting generation...\n";
27
28 foreach(@HURST_VALUES) {
29   print ".";
30   my $noise = Math::Random::Brownian->new();
31   my @ans = $noise->Hosking(
32             LENGTH                    => $NUM_ARRIVALS,
33             HURST                     => $_,
34             VARIANCE        => $VARIANCE,
35             NOISE                     => 'Gaussian');
36   my $min = 0;
37   foreach (@ans){
38     if($_ < $min){
39       $min = $_;
40     }
41   }
42
43   # Turn minimum value to positive value
44   $min = ($min < 0) ? $min * -1 : $min;
45   print ($#ans + 1);
46   print " queries....\n";
47
48   my @dist=();
49   for(my $i=0; $i <= $#ans; $i++) {
50     $dist[$#dist + 1]= ($ans[$i] + $min);
51   }
52   my $sum_now = sumDistro(\@dist);
53
54   # Adjust parameter for obtaining correct
55   # expectation value.
56   my $adj = $IDEAL/$sum_now;
57
58   for (my $y = 0; $y <= $#dist; $y++) {
59     $dist[$y] *= 1;#$adj;
60   }
61   $DISTRIBUTIONS{$_} = \@dist;
62 }
63
64 print "\n";
65 # Write distributions to files
66 for my $key (sort keys %DISTRIBUTIONS)
67 {
68   my $filename = "HURST_".$key."TIME_".
69        sumDistro($DISTRIBUTIONS{$key}).
70        "LENGTH_".$NUM_ARRIVALS."_VARIANCE".$VARIANCE.".dist";
```

```
71    print "Writing distribution to ".$filename."\n";
72    open(FILE, ">$filename");
73    foreach(@{$DISTRIBUTIONS{$key}}) {
74                    print FILE $_."\n";
75    }
76    close(FILE);
77  }
78
79  print "Done....\n";
80
81  #Subroutine for finding sum of distribution
82  sub sumDistro
83  {
84    my $distro = shift;
85    my $sum = 0;
86    foreach(@$distro) {
87      $sum += $_;
88    }
89    return $sum;
90  }
```

A.1.7 Server side PHP script

```
1  <?php
2    /* index.php: The server side script
3    generating workload. */
4    $start = microtime(TRUE);
5    for($i = 0; $i < $_GET['iter'];$i++)
6    {/* Do nothing but iterate */}
7    $stop = microtime(TRUE);
8    echo "Proc time: ".($stop-$start);
9  ?>
```

A.1.8 Experiment automation script

```
1  #!/bin/sh
2
3  # runDistroBatch.sh: Bash script for automating
4  # experiment execution.
5  #
6  # Usage: ./runDistroBatch <arrival-dir> <service-dir>
7
8  CLIENT=/home/jonhenrik/thesis_code/client/client
```

```
9  HOST=10.0.0.5
10
11 if [ -z $1 -o -z $2 ]
12 then
13   echo "Usage: $0 <arrivals-directory> <service-directory>"
14   exit 1
15 fi
16
17 for i in `ls $1`
18 do
19   DIR=pcaps/$i
20   mkdir $DIR
21   for y in `ls $2`
22   do
23     # Create directory for the experiment
24     mkdir $DIR"/"$y
25     # Initiate tcpdump
26     tcpdump -w $DIR"/"$y"/dump.pcap" "port 80 and host $HOST" &
27     sleep 20
28     # Start the client
29     $CLIENT -h $HOST -a $1"/"$i -s $2"/"$y \
30     -p >> $DIR"/"$y"/php_output.txt"
31     sleep 20
32     # Kill tcpdump
33     killall tcpdump
34     sleep 30
35   done
36 done
```

A.2 Figures

A.2.1 Probability distributions, Exponential - Pareto

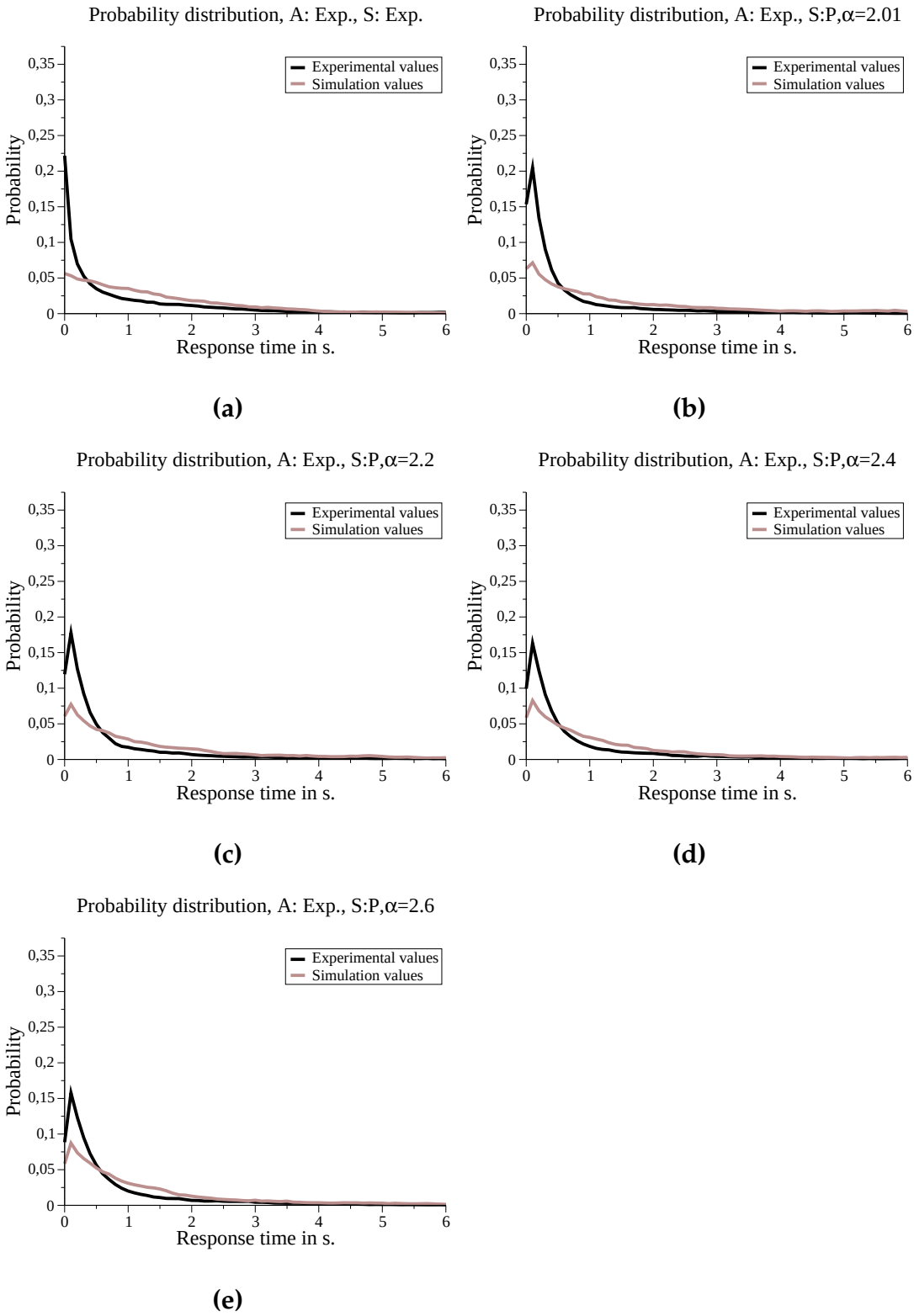


Figure A.1: *Probability distributions of response times for simulation and experimental values. The axises have been truncated for formatting purposes.*

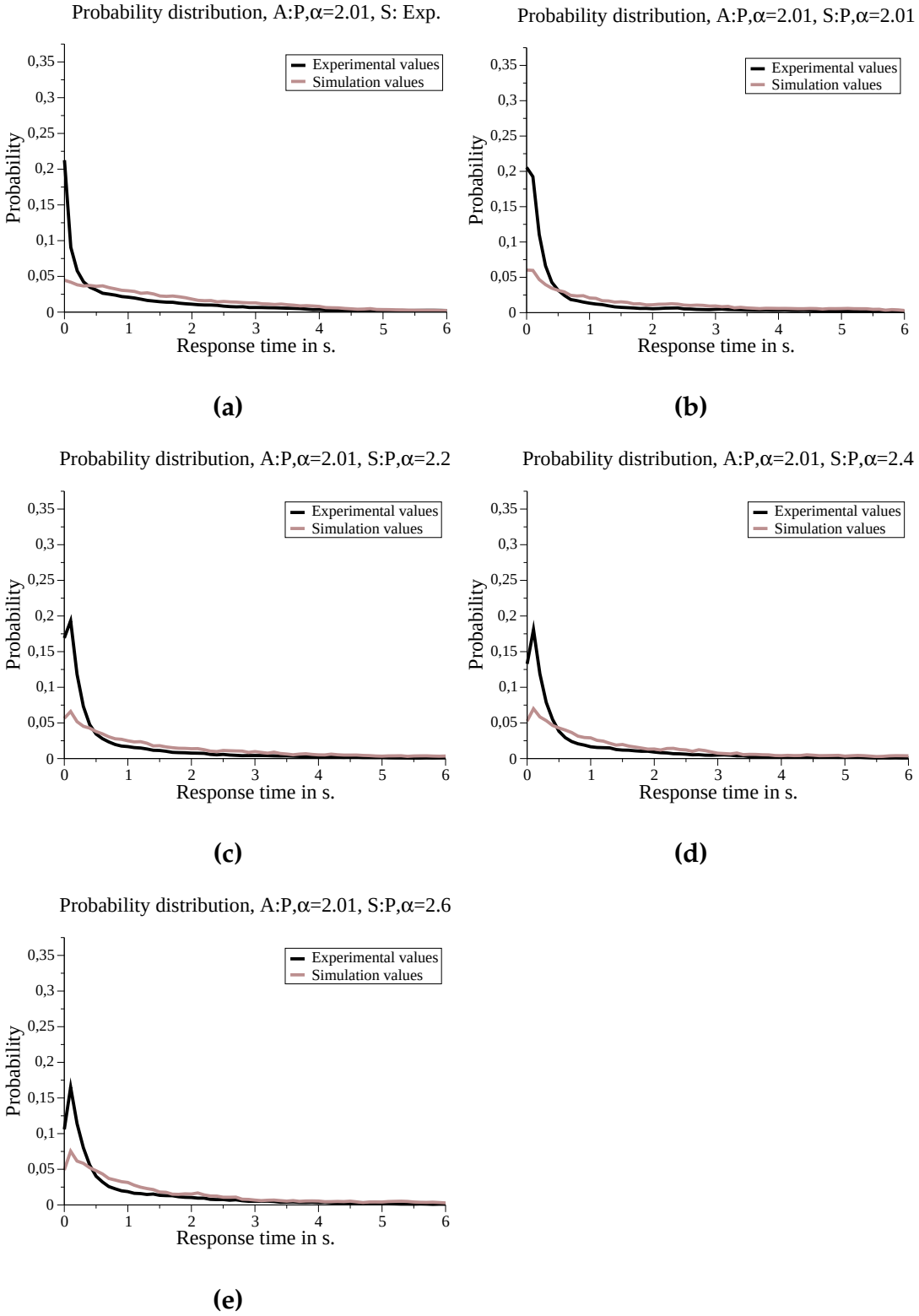


Figure A.2: *Probability distributions of response times for simulation and experimental values. The axises have been truncated for formatting purposes.*

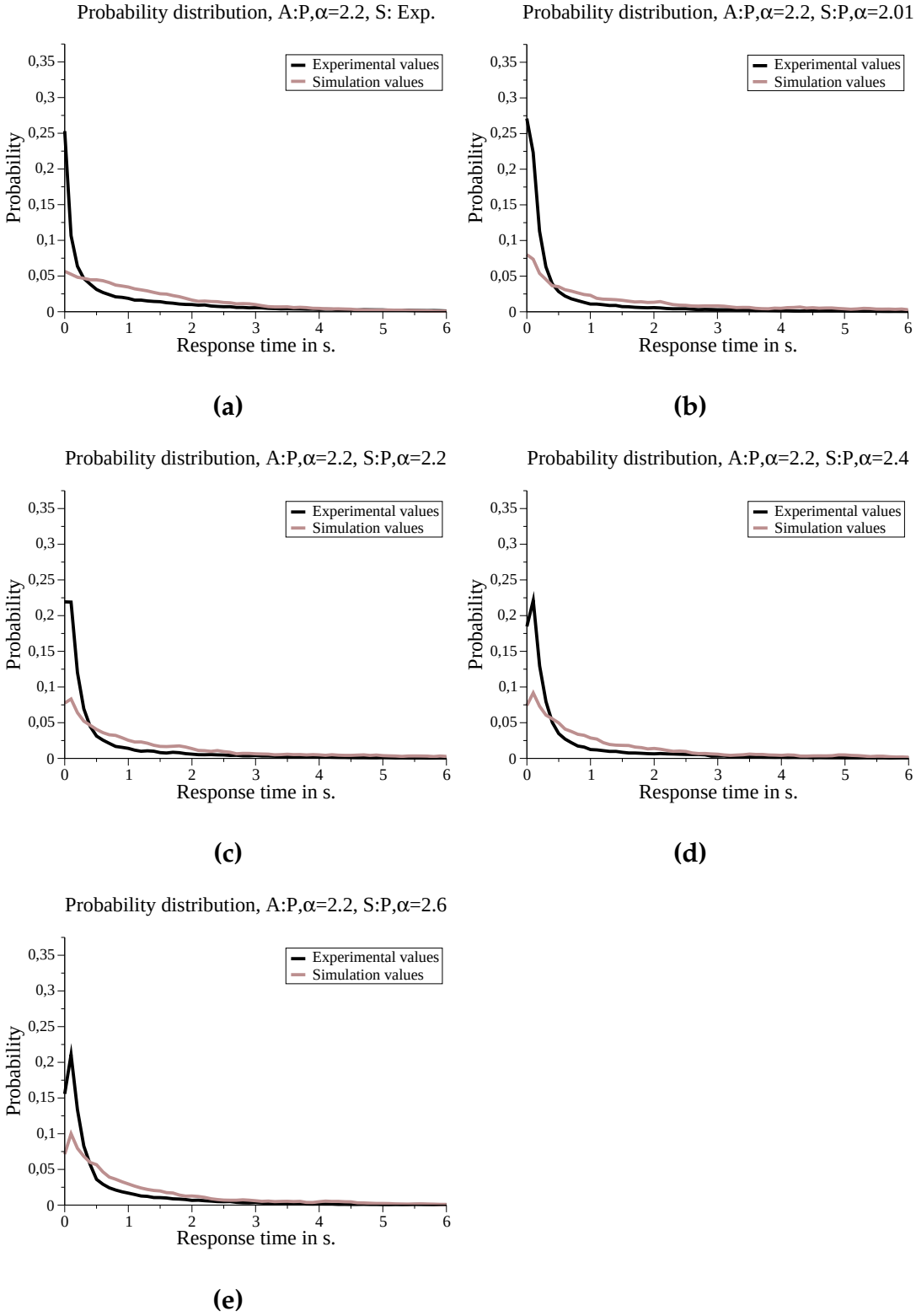


Figure A.3: *Probability distributions of response times for simulation and experimental values. The axises have been truncated for formatting purposes.*

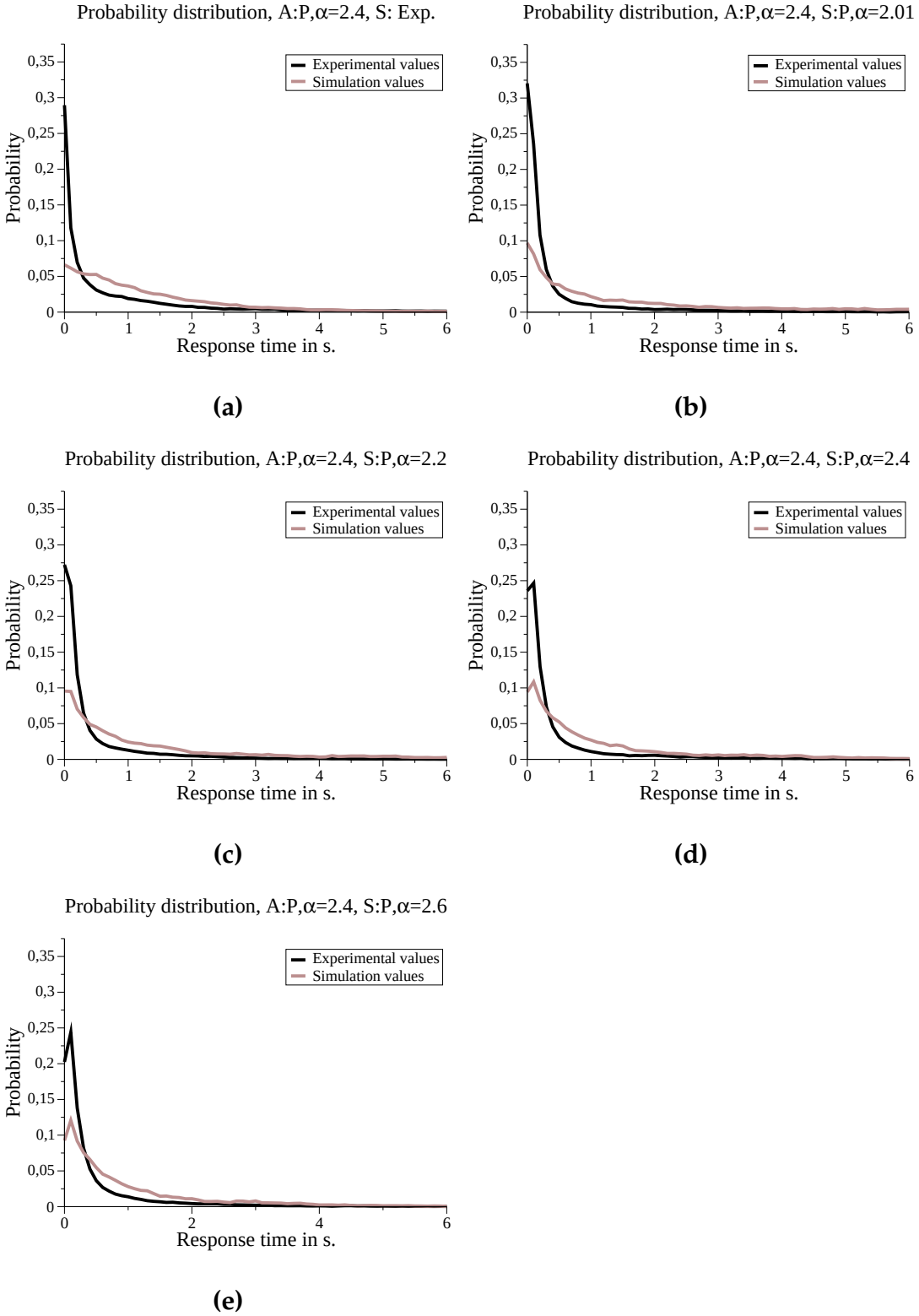


Figure A.4: *Probability distributions of response times for simulation and experimental values. The axises have been truncated for formatting purposes.*

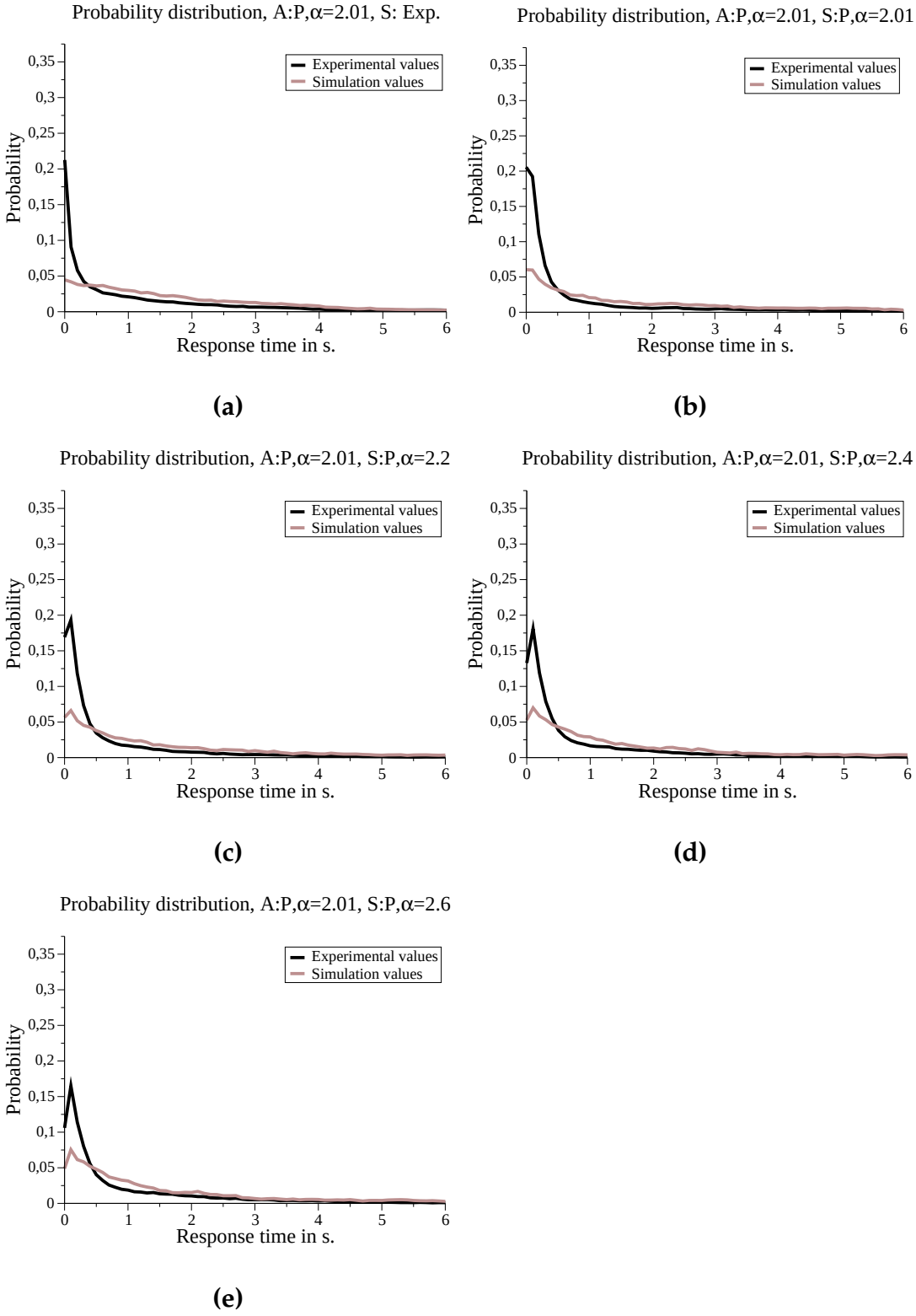


Figure A.5: *Probability distributions of response times for simulation and experimental values. The axises have been truncated for formatting purposes.*

A.3 Tables

A.3.1 Hurst values

$H = 0.5$	**Aggr. var.**	**R/S**	**Abs. mom.**	**Per.**	**Var. res.**	**Abry-Veitch**	**Whittle**
Exp.	0.516	0.433	0.629	0.511	0.507	0.547	0.503
P.,$\alpha = 2.01$	0.516	0.433	0.629	0.511	0.507	0.547	0.503
P.,$\alpha = 2.2$	0.516	0.433	0.629	0.511	0.507	0.547	0.503
P.,$\alpha = 2.4$	0.516	0.433	0.629	0.511	0.507	0.547	0.503
P.,$\alpha = 2.6$	0.516	0.433	0.629	0.511	0.507	0.547	0.503
$H = 0.6$	**Aggr. var.**	**R/S**	**Abs. mom.**	**Per.**	**Var. res.**	**Abry-Veitch**	**Whittle**
Exp.	0.604	0.535	0.708	0.592	0.632	0.654	0.604
P.,$\alpha = 2.01$	0.604	0.535	0.708	0.592	0.632	0.654	0.604
P., $\alpha = 2.2$	0.604	0.535	0.708	0.592	0.632	0.654	0.604
P.,$\alpha = 2.4$	0.604	0.535	0.708	0.592	0.632	0.654	0.604
P.,$\alpha = 2.6$	0.604	0.535	0.708	0.592	0.632	0.654	0.604
$H = 0.7$	**Aggr. var.**	**R/S**	**Abs. mom.**	**Per.**	**Var. res.**	**Abry-Veitch**	**Whittle**
Exp.	0.608	0.582	0.713	0.705	0.679	0.756	0.700
P.,$\alpha = 2.01$	0.617	0.582	0.723	0.711	0.671	0.756	0.700
P.,$\alpha = 2.2$	0.617	0.582	0.723	0.711	0.671	0.756	0.700
P.,$\alpha = 2.4$	0.617	0.582	0.723	0.711	0.671	0.756	0.700
P.,$\alpha = 2.6$	0.617	0.582	0.723	0.711	0.671	0.756	0.700
$H = 0.8$	**Aggr. var.**	**R/S**	**Abs. mom.**	**Per.**	**Var. res.**	**Abry-Veitch**	**Whittle**
Exp.	0.843	0.673	0.944	0.808	0.859	0.845	0.788
P.,$\alpha = 2.01$	0.818	0.646	0.928	0.803	0.837	0.858	0.801
P.,$\alpha = 2.2$	0.818	0.646	0.928	0.803	0.837	0.858	0.801
P.,$\alpha = 2.4$	0.818	0.646	0.928	0.803	0.837	0.858	0.801
P.,$\alpha = 2.6$	0.818	0.646	0.928	0.803	0.837	0.858	0.801

Table A.1: *Table listing the Hurst values of the measured interarrival time of experiments in 7.4.*