HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Degree Programme of Computer Science and Engineering

PERFORMANCE ANALYSIS OF A HIGH AVAILABILITY FIREWALL

Master's Thesis

Teemu Takanen

Laboratory of Information Processing Science
Espoo 2006

HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author: Teemu Takanen

Title of thesis:
Performance Analysis of a High Availability Firewall

Date: Jan 30 2006 **Pages:** 9 + 71
Professorship: Software Systems **Code:** T-106
Supervisor: Eljas Soisalon-Soininen, prof.
Instructor: Kaj Mustikkamäki, M.Sc. (tech)

High availability firewalls are used in mission critical roles to protect networks. Telecommunication applications are increasingly connected to both PSTN and Internet networks and we have a need to know the exact capacity of all involved systems. Reliable capacity information for firewalls is not generally available. In this work we analyze performance of a high availability firewall.

We form a theoretical model which converts the information on a firewall's configuration and an installation environment's network traffic profile, into expected load of the system. The model is based on the expected complexity of the internal operations of a firewall. We form the model for an OpenBSD Packet Filter and provide suggestions for changes for other firewalls.

The model contains multiple constant factors which have to be determined experimentally for the exact firewall software and hardware combination used. We describe a set of test cases and a method with which the constants can be determined.

We also run the tests on our reference system and form a complete usable load model for the system. The resulting model is adequate for a limited set of operating environments and can be used to estimate the number of firewalls required for a particular installation beforehand.

Keywords: Firewall, High Availability, Performance
Language: English

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ
Tietotekniikan osasto
Tietotekniikan koulutusohjelma

Tekijä:	Teemu Takanen	
Työn nimi:	Performance Analysis of a High Availability Firewall	
Päiväys:	30. Tammikuuta 2006	**Sivumäärä:** 9 + 71
Professuuri:	Ohjelmistojärjestelmät	**Koodi:** T-106
Työn valvoja:	prof. Eljas Soisalon-Soininen	
Työn ohjaaja:	DI Kaj Mustikkamäki	

Televiestintäjärjestelmiä liitetään nykyisin usein sekä julkiseen puhelinverkkoon että Internet-verkkoon ja niitä suojataan muun muassa palomuureilla. Tällaisilla järjestelmillä on korkeita käytettävyysvaatimuksia, minkä vuoksi myös niiden yhteydessä käytettävien palomuurien tulee olla korkeasti käytettävissä.

Järjestelmien mitoituksessa pyritään mallintamaan sen kapasiteetti mahdollisimman tarkasti. Palomuurien tarkkoja kapasiteettilukuja ei usein ole saatavilla. Lisäksi niiden kapasiteettia on vaikea ilmoittaa yhtenä selkeänä lukuna, mikä tekee mimimaalisen palomuurikapasiteetin myymisestä vaatimuksiltaan vaihteleviin järjestelmätoimituksiin vaikeaa.

Tässä diplomityössä muodostetaan korkean käytettävyyden palomuurijärjestelmän teoreettiseen kuormakäyttäytymiseen perustuva kuormamalli, sekä määritellään joukko testejä, joita voidaan käyttää mallin vakiokertoimien selvittämiseen.

Työn kokeellisessa osassa suoritetaan määritellyt testit referenssinä käytettyä OpenBSD-käyttöjärjestelmän "Packet Filter" palomuuriohjelmistoa vasten. Kokeiden tulokset sovitetaan teoreettiseen malliin käyttökelpoisen kuormamallin aikaansaamiseksi.

Saavutettu malli on käyttökelpoinen rajoitetulle joukolle palomuurien käyttötilanteita. Mallia voidaan soveltaa tavoitellussa määrin televiestintäjärjestelmien mukana toimitettavien palomuurien mitoitukseen.

Avainsanat:	Palomuuri, Korkea käytettävyys, Suorituskyky
Kieli:	Englanti

Acknowledgements

I would like to thank professor Eljas Soisalon-Soininen for supervising this work and providing valuable suggestions for improvements.

I thank Kaj Mustikkamäki for his guidance during the work.

Special thanks to Pauli Laukkanen, Matthew Wooller, Kari Haapala and Riku Saikkonen for their help and support during the work.

This work would have not been possible without the financial and material support of Tecnomen Oyj.

Espoo Jan 30th 2006

Teemu Takanen

Abbreviations and Acronyms

ACK	Acknowledgement (TCP flag)
CARP	Common Address Redundancy Protocol
CPU	Central Processing Unit
FIN	Finish (TCP flag)
HA	High Availability
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MTU	Maximum Transfer Unit
NAT	Network Address Translation
NIC	Network Interface Card
OS	Operating System
PF	Packet Filter
PSTN	Public Switched Telephony Network
RAM	Random Access Memory
RFC	Request For Comments
RTP	A Transport Protocol for Real-Time Applications
SIP	Session Initiation Protocol
TCP	Tranmission Control Protocol
UDP	User Datagram Protocol

Contents

List of Figures

Chapter 1

Introduction

There is an increasing trend to connect telecommunication systems not only to the public switched telecommunication networks, but also to the public Internet. In a telecommunications environment, the exact capacity requirements of the connected systems is usually known. In Internet servers, the exact capacity requirements are often not known or are not trivially deducible due to numerous factors which affect the performance of the system.

The Internet connectivity also means that there is a need for traffic filtering to protect the system. One way to offer protection is to use firewalls. As with all telco-equipment, there must be no single point of failure. This means that the firewalls have to support high availability functionality: state synchronization over the cluster and fail-over capabilities are a must.

From this background we can see the need for accurate capacity numbers for high-availability firewalls. A simple capacity statement in terms of bandwidth or frame passing rate is not practical for a firewall. The performance model must take in account both the configuration of the firewall and the profile of the passed traffic to accurately predict the capacity of the system.

The purpose of this work is to deduce a reasonably accurate capacity calculation method for a high available packet filtering firewall pair. The method should be usable for system sizing calculations during pre-sell discussions.

We have a specific need for this kind of a model. We have a case where one vendor ships a complete network of servers. Making capacity statements on firewalls would be easy if all or even part of the shipments were identical. However there have been no two identical shipments. The capacity for the firewalls just like for the other servers must be verified without a test system from the estimated configuration of the new system.

In chapter 2 we will first introduce the background material on firewalling. Firewall functionality and the conceptual structure of a firewall will be shown. We

will also discuss about OpenBSD Packet Filter, the software used as a test case in this work. We discuss performance metrics terminology in chapter 3. We selected OpenBSD/PF as a reference firewall for this work because it had all the needed high availability features, its source code was available for investigation and we can use its load model in practise.

Later in chapter 4 we will analyze the theoretical load generated by a packet in a firewall as a function of the complexity of the rule-set and other factors. This work will be based on the knowledge on the algorithms used in the firewalls. Since we are particularly interested in the capacity of the firewalls in extreme situations, the analysis will concentrate on the worst case behavior of the system. This analysis produces a theoretical load-formula for the firewalling system.

After the theoretical analysis, we will be perform practical tests to find out the co-efficients of the load-formula for a particular combination of firewalling software and underlying hardware. These tests are the topic of chapter 5. The chapter will describe a method which can be used to test a firewalling system and the test results on the test case platform.

We analyze test results and load model factor relevance in chapter 6. We also present a method for defining load model co-efficients from the tests and calculate the actual load model co-efficients.

The correlation between the theoretical model and the practical performance together with the overall model usability will be analyzed in chapter 7.

1.1 Previous Work

Firewall performance analysis works which could be used as a load model practically do not exist at the moment.

Vendor information on the exact firewall load figures are pretty much useless because of impractical metrics used in the materials. Many firewall producers with software only solutions do not provide any kinds of performance metrics because the performance depends on the hardware. Hardware firewall vendors often give bandwidth metrics (1Gbit/s) which can of course not to be sustained with all configurations and traffic patterns.

There are numerous papers comparing different firewalls. Most of the comparisons are made to show improvements in the latest releases of the firewall software. We found one good OpenBSD/PF centric performance paper[18]. Other similar papers for different firewall brands exist.

The second type of scientific paper often written on firewalls is a demonstration of some new attack and its effect on various firewalls. These kind of papers usually also show some solutions to the problem at hand. One example of such a work

is Gill's paper on firewall session table denial of service attacks. [15]

We can easily see why there are no papers available for exact load models. The resulting model would be rather specific for one version of firewalling software modelled. More importantly the numeric results would be usable only on the exact hardware configuration used.

Chapter 2

A Firewall

After the Internet gained popularity and moved from being just a research network, there became need to isolate private networks from the public Internet. Some early solutions were gateways which allowed specific services to be used [12].

The early gateways were custom ad hoc solutions based on the organizational needs. Later the idea was productized. A host in a network which allows network traffic only if specified by a policy is called a **firewall**. [31]

In this chapter we describe the functionality of a firewall. We assume familiarity with the internals of the Internet Protocol (IP). A summary of the protocol can be found in appendix A.

Most of the following discussion is based on [37].

A firewall is a network interconnect device which allows traffic pass through only if it is allowed by a configured policy. The configuration definitions for passed or blocked traffic are called **firewall rules**.

There are two basic ways to build a firewall. It can be either a **proxy** or a **packet filter**. Proxying firewall is a **multi-homed** host which has an application proxy for one or more protocols. All connections from one network with this protocol must be made into the proxy firewall. The firewall will then make the real connection into the target server and relay the application traffic. Packet filter acts on lower level. It will either pass or block individual network packets.

A proxy is somewhat more flexible way to implement a firewall. However, a separate stack is needed for all network protocols which need to be passed through. In some cases this is feasible, but more often it is not. A packet filter firewall does not have to understand the semantics of application protocols, so it can be used in more diverse cases. It will not be able to filter traffic based on content. Many firewalling solutions combine both approaches: packet filter is used as a base, but proxies are offered for some common protocols.

Predicate	Typical Values
Direction	in, out
Network Interface	interface name
Protocol	TCP, UDP, ICMP
Flags	Ack, Syn, others
Source IP address	address of a host or a network
Source Port	0 – 65535 or range
Target IP address	address of a host or a network
Target Port	0 – 65535 or range

Table 2.1: Most common firewall rule predicates.

Another fundamental differentiation of firewalls is whether they are **transparent** or **non-transparent**. The difference is whether the firewall can be introduced in a network without any configuration changes in other devices. A transparent packet filter is a network **bridge**: it will connect two segments of the same network together and filter packets in between. A non-transparent packet filter is a **router**: it will route traffic between networks and selectively allow connections.

2.1 Firewall Rules

A firewall uses patterns called **rules** to select whether some traffic should be passed or blocked. A single rule is the basic unit of firewall configuration. The exact syntax and semantics or firewall rules are heavily implementation dependent. However, the basic idea is the same.

2.1.1 Structure of a Rule

A rule contains two parts: **rule condition** and **action**. The condition part is a list of logical predicates combined with logical-and operations. Usually negation of individual predicates is also supported. The condition part can be evaluated in a context of a single packet, and if its value is *true*, the specified action is taken.

The most common predicates for firewall rules are listed in table 2.1. Usually every rule uses at least direction predicate (is the packet going in or out from the firewall) in addition to checks for source and target IP addresses. The network interface is usually also checked to prevent address **spoofing** from locally connected networks.

All firewalling systems support the direction predicate, but the semantics differ. Some implementations (like Linux IPTables) have three directions: in, out and

forward. Forward direction is used when the packet is forwarded through the system and the in and out directions are used for connections to and from the firewall host itself. On some other systems (like OpenBSD/PF) there are only in and out directions and two rules are needed to forward traffic: one to let the packet in and another one to pass it out again.

When the condition of the rule has been evaluated as true, the action specified in the rule is taken. Usually this action is either **pass**, **drop** or **reject**. Pass means that the packet is passed and processed normally by firewall's IP stack. The two other actions are used on traffic which is not allowed. Packets can be dropped: they are simply ignored. This is a usual default configuration on firewalls. Sometimes it is preferable to provide response for the traffic initiator by sending ICMP Unreachable (for UDP packets) or TCP Reset (for TCP packets) packet. Packet denying together with response sending is called rejecting. Even with rejecting, denied ICMP packets are not usually responded at all.

Besides conditions and actions firewall rules usually contain statistics information. Often at least packet counters for matching and not matching packets are maintained by the firewall on rule basis.

2.1.2 Example Rule

Below we have one example of a typical firewall rule. The purpose of this rule is to allow all hosts in 10.0.0.0/8 network to use a web server (TCP/80) in a host 192.168.10.5. Note how the source port is not fixed into any particular port number. The operating system of the host from which the web server is accessed allocates the source port as it wishes. It is typical for most firewall rules that the source port is not specified.

Direction	Source IP	Target IP	Proto	Src Port	Tgt Port	Action
in & out	10.0.0.0/8	192.168.10.5	TCP	any	80	pass

When looking at the rule above we notice that it does not allow the web server to send any return traffic back to the clients. The problem could be solved with two rules. We could add an additional rule which allows 192.168.10.5 send frames with source port TCP/80 to any host and port in the 10.0.0.0/8 network. We should of course add an additional predicate which does not allow the web server to send TCP packets with SYN-bit set if the ACK-bit is not also set to prevent it initiating arbitrary TCP connections into the 10.0.0.0/8 network.

This method for allowing two-way traffic is cumbersome and it still allows some unwanted extra traffic to flow through. We come back to this issue with a better solution in the stateful filtering section below.

2.1.3 Rule Processing

When a packet is received by the input interface of a firewall it will cause rule processing. The firewall evaluates rule conditions for the packet for each rule in the order in which the rules have been defined. The first matching rule defines the action taken.

In practise many firewalls implement the rule processing in a different way. The action taken might be the last matching rule, not the first one. There might be rules which simply define that some other rule-set is used for processing of a matching packet. In most cases these are just ways to make the configuration easier for a human: the rules could be presented also in a form where the first matching rule specifies the action, but they would be more complicated and error prone.

2.1.4 Rule Processing Optimizations

If implemented with lists which are scanned through for each and every frame the rule processing time would scale linearly on number of rules. This would be rather inefficient even with small amount of rules. Usually some optimization method is used. Key feature of any optimization is transparency: the optimization must not alter the semantics or the order of rules.

One way to optimize the rule processing is to chain the rule predicates with multiple linked lists. This method relies on two key facts. First, the rule condition clauses are composed of predicates connected with conjunctives, there are no disjunctions. Second key finding is the fact that the same predicates tend to be used on multiple rules.

The conjunctive normal form means in practice that when the rule condition is evaluated, the evaluation can be terminated when one predicate is false. The truth value of the whole condition is then also false and we can move on to process the next rule.

When used alone, this optimization still keeps the overall complexity in $O(R)$ where R is the number of rules. Now we can use the second key fact. There is no point to evaluate next rule condition at all if it contains the same predicate which failed the current rule in processing. Therefore every predicate has its own next-pointer which points to the next rule which does not have the same failed predicate. The next-pointers are set up in rule pre-processing which needs to be done only when the rules are loaded or re-loaded into the firewall. This optimization method is used at least in our reference firewalling system OpenBSD/PF. [18]

While the multiple linked list optimization can reduce the processing time of rules

significantly and will never increase it compared to singly linked list implementation it has unpleasant complexity characteristics. If the rule set happens to be in such an order that no skipping can be done, the complexity is exactly the same as with no optimizations at all. If skipping can be used optimally, the complexity can drop to $O(1)$. This depends on the structure and the order of the rules.

The list based firewall rule processing is conceptually simple and it is also easy to implement. The complexity of the rule processing can be reduced with more sophisticated methods like binary decision trees or tuple space hash-tables.

In a decision tree every rule predicate is checked at most once. The rules are organized as a binary decision tree where the truth value of the checked predicate selects the predicates which still need to be checked [19]. This is obviously an improvement on the list skipping optimization above, since the maximum complexity of the rule search now linearly depends on the number of different predicates, not on the number of rules.

In a tuple space search method the rules are organized as a hash-table. The relevant fields of the incoming frame are hashed and only the rules which have the same hash value have to be checked. This method is especially suitable for a huge number of firewall rules. The hash calculation and the extra memory usage of this method make it somewhat unsuitable for a small number of rules. [35]

2.1.5 Default Rules

There are two different approaches in configuring a firewall. The rules can be written either with **default allow** or **default deny** policy. Almost all firewalling solutions support both ways, so the choice depends on the environment in which the firewall is used.

With the default allow policy the firewall will pass all traffic which is not explicitly denied by the rules. This approach is good when we do not know the all protocols or even hosts involved. For example, the default allow policy is often used by Internet Service Providers (ISP). They will typically allow almost all traffic, but they still want to block a few protocols which will cause trouble if passed into machines of typical home users. In the default allow, the number of firewall rules is usually quite small.

When we use default deny policy, everything which is not explicitly allowed is denied. This policy is good for well defined service networks. If we know that only our web-server and mail-server need to be accessed from the public Internet we can just allow traffic to them. In default deny, the number of firewall rules is a function of allowed services. This means also that it is roughly a function of the network complexity.

Often the default allow and deny are both used in the same firewall. Many

organizations use default deny for incoming traffic and default allow for outgoing connections.

2.2 Stateful Filtering

Firewall rules provide a way to configure a firewall so that only desired traffic can pass it. However, because IP traffic is packet switched it is sometimes hard to see which packets are actually wanted and which are not.

The problem can be easily seen in passing of TCP traffic. As we saw in the example rule above if we want to allow external hosts to access a HTTP server on TCP port 80 in a private network, we write a firewall rule which allows traffic into this port and another rule to let traffic from TCP port 80 on the HTTP server to pass into the external network.

This configuration, while it works, will however let additional packets pass. The HTTP server can make TCP connections into any external hosts by using port 80 as the source port. The external hosts can send packets into the web server without TCP connection by never sending TCP SYN packet to open the connection. The first problem can be remedied by adding a requirement in the outgoing rule that the TCP SYN bit can not be set if TCP ACK bit is not also set and thus deny the TCP connection initiation packets going out. The latter problem can not be solved without keeping track of ongoing connections.

Connection tracking is called **stateful filtering**. In stateful filtering we keep a separate **state table** in addition to firewall rules. The state table contains one entry for each active connection through the firewall. When a new connection is opened, an entry is added. When connections are closed, or state timeout timer expires, the state entry is removed from the table. [34]

Stateful filtering has couple of additional advantages. State entries can contain various counters which can be used to calculate statistics per connection basis. In addition, since incoming packet can be compared directly against existing states instead of the set of firewall rules, stateful filtering can improve performance firewall's performance.

Some firewall maintain more than one state entry for each passed connection. For example in OpenBSD/PF which does not support the forwarding direction predicate in the firewall rules has two states for each connection: one created by the incoming traffic rule and one by the outgoing traffic rule.

2.2.1 State Entries

A state entry is a n-tuple which contains enough information to determine whether an incoming packet is part of the connection tracked by the state and what is the current state of the connection.

A state entry contains at least the following fields for connection identification:

- Protocol version (IPv4, IPv6)
- Connection source IP address
- Connection target IP address
- Protocol (TCP, UDP, ICMP)
- Source port number (for TCP and UDP)
- Target port number (for TCP and UDP)

These fields can be considered constant during the lifetime of the state. They are set up once during the state creation and they will not change during the connection.

Some dynamic information is also stored in the state entries. All states contain at least a time-stamp of the latest packet seen. Some age limit is used by all firewalls to clean up states for broken or finished connections. In addition, states usually contain statistics information, like the number of packets matched in the state and number of bytes transfered.

In our reference firewall OpenBSD/PF one state entry needs approximately 1kB of memory. State entries contain information on the relevant firewall rule, various counters and protocol specific information fields.

2.2.2 States with TCP

While TCP is a complex protocol it is conceptually simple to model with firewall states. This is due the fact that it is inherently stateful. The following discussion assumes in-depth knowledge of TCP. A summary of TCP is available in appendix A.2

TCP uses a tree-way handshake in connection initiation. The termination of a TCP connection also takes several packets. During the the connection both ends of the stream also have separate sequence numbers.

We can think a TCP state in firewall's perspective as a combination of states and sequence numbers in the both ends of a TCP stream. A firewall updates state

information only for the side sending the frame currently in the firewall. The firewall can not know for certain that any single frame gets to its recipient and thus find out the new state of the receiver, but it can find out the state of the sender from the frames it sees.

Thus if we view the TCP states without the sequence numbers as a pair (SourceState, TargetState) a running TCP stream would then be in state (ESTABLISHED, ESTABLISHED).

A half closed TCP connection would be in state (CLOSE_WAIT, FIN_WAIT_2). A fully closed connection would be (CLOSED, TIME_WAIT). During the timeout period in the connection termination initiating side one end of the connection has already closed it and the other is waiting for re-send timeout period to end.

The firewall has to keep the connection in its state table for the full timeout period to be able to pass the possible FIN re-transmissions from the other side.

The state entry must of course also track the connection initiation in the same manner with relevant timeout values. The sequence numbers in all frames can be checked against the known previous state and out-of sequence transmissions can be rejected. The firewall can also trivially reject frames which appear to be sent by a host in a different TCP state than the host should be.

As we discussed above all state entries in the firewall have timeout counters which allow the state entry to be removed when the connection has been terminated without proper termination packet sequence. TCP state entries have different timeout values depending on the state of the connection tracked connection. Timeouts are large when the connection is in (ESTABLISHED, ESTABLISHED) state. During connection initiation and termination the timeouts are smaller.

We can also see that the final removal of the state entry also happens due to timeout expiration even when connection is properly terminated by both ends. Therefore the only needed state removal condition for TCP state entries is a timeout expiration. In the following discussion about UDP and ICMP states we find that the same is true for their state entries. In fact usually no state entries in a firewall are removed for any other reason than timeout expiration.

2.2.3 States with UDP and ICMP

Both UDP and ICMP are stateless protocols. A firewall with stateful filtering can still make state entries for these protocols.

States for UDP allow two way traffic with a one way rule. Most UDP based applications send responses to requests. With states the responses come through, without allowing random UDP packets to pass the firewall.

The firewall can create a UDP state entry when a passing packet matches a

UDP firewall rule. Since there is no way to see the end of a UDP transaction, the firewall must rely only on timeouts on UDP state clearing. Usually timeout values for single UDP frames are low (at most tens of seconds perhaps) and timeouts for "connections" where multiple packets have been sent to both directions are higher.

ICMP traffic can be divided into queries and error messages by the firewall. Querying ICMP traffic is quite rare and it is usually used only for diagnostics. The most typical ICMP query is the ICMP ECHO REQUEST (usually called ping) and its response message the ICMP ECHO REPLY. The querying ICMP messages can be handled in the firewall with separate firewall rules just like with TCP and UDP traffic. State entries can contain message types and response traffic can then be passed by the state entry.

ICMP error messages are sent by routers and hosts in response to some packet which is not itself usually a ICMP message. The error messages are usually signals for missing routes, unreachable hosts and networks and connection refusals.

There is no need to allow reactive messages with the firewall rules. The firewall can use the state information for the packet which triggered the ICMP response to allow it. We can think the reactive ICMP messages as part of the normal TCP, UDP or IP traffic.

2.2.4 A State Table

A **state table** is a data structure in which a firewall keeps the current state entries. Ideal structure allows fast inserts, deletions and searches. In addition the structure should be resistant to abuse by sending frames with carefully selected header values to exploit the worst case behavior of the structure. The resistance can in practise be achieved by selecting a structure with a fast worst case performance.

The firewall searches state entries every time it receives a frame from the network to see if the frame matches any existing state. The speed of this search is thus crucial for the performance of the firewall. The search uses the basic packet information as a search key. The searched entry must match source and target addresses, source and target ports, protocol versions and types of the currently processed frame. Since the search key uniquely identifies a connection, there should never be duplicate keys in the state table.

State tables are usually implemented with hash-tables or balanced binary search trees. Both methods have their strong and weak sides. [18]

A hash-table can in theory provide fast constant time searches. In addition, both inserts and deletions are usually fast operations. On the other hand computation of a good hash value from IP addresses and port numbers can be rather slow.

Hash value collisions must be handled with some fast fallback structure, perhaps a balanced binary search tree. In addition a full search through all available states can be a relative slow operation when the hash-table is rather empty.

Balanced binary search trees guarantee $O(log\ n)$ behavior in the worst case. There is no need for a secondary fallback structure because there can be no collisions. Trees can be slower than hash-tables with a large number of state entries. Walk-through of all entries in a tree is a cheaper operation than in a hash-table because there is no need to look into empty slots.

As we discussed above the state entries are inserted and searched from the state table based on the packet header information. However we also saw that deletion actually happens due to timeouts. If the connection identification information is used as a key in the state table the search for expired states must either go through the whole state table or use some secondary search structure for timeouts.

The OpenBSD/PF implements the state table with a balanced binary search tree. The exact implementation is a red-black tree (for PF in OpenBSD 3.7, older versions used AVL trees). A red-black tree has maximum depth of $2+2log_2(n) = 2log_2(2*n)$. The complexity of insertion, deletion and search operations depend on the depth of the tree. [13]

The PF performs periodic sweeps in its state table to purge expired states. By default this is done every 5 seconds.

2.3 Traffic Normalization

Rule based stateful filtering protects networks against access attempts on services which are not meant to be used outside the protected networks. It does not as such protect the host operating systems if the IP stack implementations are attacked with custom tailored packets. In addition to this, the stateful filtering engine itself can be fooled with ambiguous traffic.

To protect hosts against attacks which exploit bugs in the IP stack implementations, the firewall should reconstruct all passed packets. This process may change the IP flags and TCP window parameters, renumber TCP sequence numbers and recompute the packet checksum. While at the first glance this seems to be an expensive operation, we have to remember that the firewall would have to recompute the checksum anyway because it has to lower the Time To Live value in the packet. The recomputation goes through the whole header, so changing it causes no significant extra overhead.

The problem with stateful filtering is that there are cases when the firewall and the destination host might interpret the incoming packet differently. What happens if the TCP window is set incorrectly or some of the reserved fields of IP flags

are set? The firewall cannot be sure whether the destination host will accept or reject the packet. If it passes the packet and the destination host drops it, the state information is not consistent with the actual state of the connection as seen by the destination host.

To prevent state table inconsistency, the firewall can either block inconsistent packets or rewrite them with parameters which interpretation is not open to implementation dependencies.

One rather problematic source of inconsistency is fragmentation [21]. IP packets can be fragmented both at the source of the packet or at any intermediate router. The packet is considered to be received only when all fragments have been received and the packet has been reconstructed.

If the firewall passes fragmented packets without fragment bookkeeping, it cannot be certain when to update the state information. If it keeps track of the fragments, it must in practise imitate fragment reassembly. Even then it might do the reassembly somewhat differently than the destination host. Overlapping fragments are the most problematic cases where the firewall cannot know how the destination host will handle the packets.

Since the firewall is already imitating the fragment reassembly, it can actually do it completely and forward the packet only after it has been received completely and correctly. After reassembly, it will resend the packet and fragment it again if needed.

The process of reconstruction of inconsistent packets is called **traffic normalization** or **scrubbing**. [16] [23]

2.4 Network Address Translation

Since firewalls are connection points into networks they are often used also as network address translators.

In **network address translation** (NAT) the router (or firewall) changes either source or target address information in passing frames. In port-NAT multiple hosts use the same public address and the NAT router must also change UDP and TCP port numbers of passing traffic. NAT allows the actual hosts use private network IP-addresses and still communicate with the rest of the Internet. [14]

Network address translation is problematic for firewalls when protocols with embedded IP addresses are used. Many common protocols, like FTP, SIP and H323 contain embedded addresses. These addresses are not translated by the normal NAT procedure. There is no general solution to this problem. Sometimes it can be solved by properly configuring the public routable address into the application (typical solution with SIP), but often the only real solution is to use application

proxy to properly translate the protocol traffic.

NAT with changing port numbers causes an additional problem: it needs state information on allocated ports to support connections through the NAT gateway. The states can be stored and handled the same way as firewall rule states.

Actually in some firewalls, like OpenBSD/PF, there is no separate state table for NAT. Instead the normal connection tracking state entries have contain both the address IP packet has when it enters the firewall and the address it should have when it is sent forward. This way the actual NATting is a rather cheap, almost free, operation.

We do not study the effects of NAT to the firewall load in this work. However, since NAT is such a cheap operation the resulting load model should be rather accurate even if NAT is used in the firewall since most of the NAT load is included in stateful filtering and normalization load.

2.5 Logging

Most firewalls produce logs packets matching certain rules. The logging is triggered as an additional action on rules. Logs typically contain the matched packet, time-stamp and the rule which was matched.

Logs are extremely useful for problem solving and intrusion detection, but log generation can be both CPU and disk intensive. If all matched packets are logged on high-traffic firewall, disk I/O can easily become limiting factor on the overall throughput.

Logging is usually performed with lower priority in the firewall than the frame forwarding and filtering. In OpenBSD/PF the firewall itself runs in the kernel and the logging into a file is done with a userland process. [4]

While this arrangement is good in the sense that the disk I/O bottleneck does not stop the packet filtering it has couple of unpleasant side effects: Log events can be lost in situations where load is high and the load testing for logging load is harder because the logging load usually does not cause frame losses in the system. Instead, log events are lost.

In this work we use the typical real-world logging configuration, where we log all blocked packets and the first packet of all allowed connections.

2.6 On IP Network Traffic

The basic issue in load analysis of any software system is how the complexity of the operations performed respond to the input fed into the system. The input

of a firewall is a set of input network frames and the output of the system is a subset of these same frames.

The fundamental variable is the rate in which input frames are fed into the system. Framerate defines the overall event rate of the system. Together with frame sizes the framerate defines the input data rate.

Frame sizes in IP traffic are limited not only by maximum and minimum frame sizes allowed by the Internet Protocol itself, but also the physical network used to carry the IP packets. Most typical way to connect a firewall into the network is to use Ethernet [20]. The IP packets can by the design of IP travel on multiple networks in their path from the sending host to the receiving host. The firewall usually still sees them encapsulated into Ethernet frames.

Ethernet supports data payload sizes from 46 octets to 1500 octets. IP supports packet sizes up to 64kB. Fragmentation is used if the IP packet does not fit into its carrier frame. We do not study effects of fragmentation to the firewall load in this work (although the effect may be significant). Therefore we can restrict our IP packet sizes between 46 and 1500 octets.

While the framerate and the frame size are the most important factors in the firewall input, also the protocols used on top of IP also differ. Particularly the normalization code for different protocols varies, but as we discussed above, also state handling is somewhat different.

One important category of input traffic of a firewall is attack traffic designed to halt or disrupt the firewall. The traffic stream is usually designed specifically the disruptive effect in mind and it usually also exploits some particular weakness in the firewall in question. One example is to attempt to fill the firewall state table so that it can not accept new connections [15].

Analysis of load effects of attack traffic is hard in general, because usually the firewall manufacturers change their implementations after some particular weakness is found. We do not include attack traffic in our load model.

2.7 High Availability

High availability (HA) is a design feature of a computing system which allows continuous operation of the provided service of the system despite of a failure of an individual component of the system. The system includes both hardware and software components. [30]

To achieve high availability, the system must have redundant components. HA can be designed at hardware level or software level. Hardware solutions tend to be much more expensive than software solutions. Both approaches need redundant hardware components, but the management of the redundancy is cheaper on the

software level.

High available system must have no single points of failure. A **single point of failure** is a component of the system which failure cannot be tolerated by the system without loss of service.

2.7.1 Typical High Availability Solutions

Typically highly available solutions are based either on **hot-standby** or **cluster** approach.

In hot-standby there are two identical units of which one is providing the service and the other is ready to take over the service should the primary unit fail. Hot-standby is typical on state intensive services, where it is inconvenient to have more than one unit operating at a time. File servers are typical hosts which often use hot-standby.

In cluster approach there are two or more units which all provide the service. If a unit fails, the rest of units in the cluster handle the service. Clustered approach is often used in services where one unit cannot handle the whole service. Since the clustering is needed for performance reasons, it is also used to provide high-availability. Many internet services, like Domain Name System and Network Time Protocol are inherently clustered by providing clients with addresses of multiple server hosts.

When a unit fails in a high available system, the service will **fail-over** to some other unit. The fail-over should preferably be transparent for users of the service.

One important factor in the complexity of a high available system is whether the service provided is **stateful** or **stateless**. If the service is stateless, the responses to client requests depend only on the server configuration and the contents of the client's request, not on previous requests of the client itself or other clients. In stateful service there is state information which needs to be available before requests can be served.

Obviously it is easier to provide high availability when the provided service is stateless. With stateful services the server hosts need to synchronize state information between the hosts to be able to fail-over.

2.7.2 Firewall High Availability

The whole network connectivity of a network depends on the firewall in front of it. If the rest of the system is designed to be high available, the firewall should of course also be. [9]

The main difference in a firewall and any other type of a high availability server

is that a firewall is a multi-homed host. All firewalls in a cluster or a HA-pair share the same service IP addresses, at least one per connected network. The fail-over in functionality must thus happen approximately at the same time in multiple networks. In practise the problem is usually handled so that should one network interface of a firewall fail, the software forces the other interfaces to fail. This way the firewall is not left in some network as a black hole of traffic where it would receive frames on one side, but would be unable to forward them.

Fail-over functionality in a firewall is typically rather simple. It is enough that the machines in a firewall cluster or pair setup repeatedly send heartbeat information into each-other. When a heartbeat is no longer seen, the responsibility for the cluster shared IP addresses can be taken over. The exact mechanisms vary from one firewall to another, but the basic principle is the same. The monitoring features required for fail-over typically require neglible resources and their load impact in a firewall is not an interesting question.

A firewall with only routing and basic firewall rule features is a stateless device in terms of high-availability. We are however interested in load behavior of firewalls with stateful filtering and perhaps also NAT. Both of these features need state information which must therefore be shared with the active and standby firewalls. As we discussed above NAT state information is practically identical to connection state information at least in state synchronization point of view.

State information sharing details again depend heavily on the firewall in question, perhaps even more than the fail-over mechanisms. Most firewalls which support state sharing are commercial products and we do not have access to their exact implementations of state sharing.

In principle the state sharing operates simply with sending of state creation, deletion and update events from each firewall in a cluster or a HA-pair to others through a (usually dedicated) network. The receiving machines can then perform the same operations in their own state tables and maintain a shared state.

This kind of implementation is of course not as reliable as mechanisms used for example in databases. There is usually no guarantee that the state information is identical in both machines in a HA-pair or a cluster. However, the lost states in case of sudden break of one machine are usually those which had been created about the same time that the break happened. The re-try mechanisms of most practical protocols will re-initiate the connection when they get no response. The re-try will also re-create the state in the new active firewall.

In this work we handle only hot-standby high availability firewall pairs, not clusters. We go through the fail-over and state sharing mechanisms of our reference firewalling system in the next section.

2.8 OpenBSD and PF

OpenBSD[8] is a free operating system derived from 4.3BSD Lite[1]. The OS includes wide set of system utilities and a complete development environment. OpenBSD has two main focus areas: software freedom and security. The slogan of the system is "Secure by Default" which means that most relevant security features are included in the default system and enabled by default [6]. This includes the firewalling code and networking high-availability features.

The operating system has a built in firewalling system called Packet Filter[18] (PF). As the name implies, PF is a stateful IP packet filter. Both bridging mode and routing mode are supported, but in this work we concentrate on the routing mode. [7]

We selected OpenBSD and PF as an example case in this study because the default system had all the required features and the full source code was available for reference.

2.8.1 High Availability Features

To be usable in an environment with high-availability requirements, a firewall has to support redundant operating mode. In PF the redundancy can be achieved by using both Common Address Redundancy Protocol[3] (CARP) and pfsync[5].

CARP is an Ethernet level protocol which allows multiple hosts in the same physical network segment to share common Ethernet- and IP-addresses. It can be used both in clustered and hot-standby configurations, but we use only hot-standby setup. CARP is conceptually somewhat similar to Virtual Router Redundancy Protocol[25] (VRRP). Since VRRP is patent encumbered in the USA, it is not used in OpenBSD [1].

Hosts which have CARP shared addresses send CARP announcements approximately every second into the network. Announcements are deliberately delayed by a time called advertisement skew. The skew is configured to be low on the master host and high on the backup host. The machine which can announce its CARP address most often is seen as the master and the rest are backup hosts.

State tables on clustered PF hosts are synchronized with pfsync protocol. The protocol uses multicast to announce new state entries to other cluster members. Several new state entries may be combined into a single announcement packet for efficiency. When a PF host boots up, it will request mass update of state entries. Each other host will then send all known states to the newly started host.

[1]Actually OpenBSD is an offshoot of NetBSD, which is derived from 4.3BSD Lite and 386BSD. 386BSD is the x86 port of 4.3BSD Lite. [22]

Chapter 3

Performance Metrics

To evaluate performance of a firewall, we must first define what we mean by performance. In this work we are interested in the amount of network traffic the firewall can handle without overloading itself.

3.1 Terminology

We will first define what we mean by load in this work. After load, the rest of the definitions are from RFC1242[10], which defines vendor independent terminology for network interconnection device benchmarking.

3.1.1 Load

The load of a system is L, $L \geq 0 \in \mathbf{R}$. When $L = 0$ we say that the system is **idle**, it is not doing any processing and thus has no load. When $L = 1$ the system is **fully loaded**, it is using some limiting resource to the maximum capacity and it cannot process data any faster. If $L > 1$ the system is **overloaded**. Input is fed into overloaded system faster than it can process it.

3.1.2 Throughput

Throughput is "the maximum rate at which none of the offered frames are dropped by the device." Throughput is measured in number of N-octet input frames per second. In terms of load this means that we measure the number of frames per second when $L = 1$.

We can measure throughput by sending N-octet frames at constant rate through the measured device. If all frames pass, the throughput of the device when using

N-octet frames is at least the send rate. We can then rise the send rate and re-run the test until we find the maximum rate.

Of course in practise, without specialized testing hardware, it is impossible to distinguish frame loss caused by the measured device from frame loss caused by testing equipment or environment. Some frame lossage occurs in all heavily loaded Ethernet networks because of frame collisions and other issues. Because of this, in this work we use somewhat arbitrary limit of 0.005 as the loss rate where we say that the system is running with its maximum throughput.

3.1.3 Overloaded Behavior

A system is said to be overloaded ($L > 1$) "When demand exceeds available system resources." An ideal network interconnect device would forward the same number of frames in all cases when $L \geq 1$. In practice, the the processing delays in most systems increase when they are overloaded. This causes increased **latency** and **data frame lossage**.

3.1.4 Latency

Latency is defined to be "the time interval starting when the last bit of the input frame reaches the input port and ending when the first bit of the output frame is seen on the output port." In practice, to measure the latency we will actually send a packet through the tested device and use a time interval starting when a test frame has been sent and ending when is has been received again.

3.1.5 Frame Loss Rate

Frame loss rate is defined to be "percentage of frames that should have been forwarded by a network device under steady state (constant) load that were not forwarded due to lack of resources."

3.2 Defining Firewall Performance

It would be nice is we could state the firewall performance as a single number like 1Gbit/s or 150000 frames/s. As we discussed in the previous chapter the load of the firewall is dependant on many configuration and traffic aspects. Due to this a simple statement is not really practical.

What we can get is a model with which we can calculate the load of a firewall with particular settings and traffic patterns. We can form this model based on

computational theory and knowledge of the internal workings of a firewall.

We can measure the sustained framerate with acceptable frame loss ratio with different parameter settings. From the measurements we can get the co-efficients for the model terms.

Creation of this model is the topic of the next chapter.

Chapter 4

Theoretical Load Model

In this work we will discuss the static load behavior of high availability routing packet filters with stateful filtering which are configured with default deny policy. We assume hot stand-by redundancy based high availability. We use the OpenBSD PF implementation as a reference case.

The purpose of this chapter is to provide a hardware independent mathematical model for a firewall load calculation. In the next chapter we will formulate tests which can be used to determine constant factors in the load model for a particular software/hardware combination. After this the model should be able to tell us the load based on information on the handled IP traffic.

We will consider only a case of stable load where all factors are kept constant for a period of time. We are looking for a formula which can be used to find out the maximum performance of the system. If the model is used for non-stable traffic the worst case traffic scenario of the traffic can still be used with the model.

The model will be specific to the examined firewall software. The method is general and the built model can be used for other firewalling solutions with small modifications.

4.1 Factors of Firewall Load

The most important factor in firewall load is the amount of IP packets it needs to process. We will express this as frames per second, or p. The frames can be divided into four different groups: TCP frames (p_{tcp}), UDP frames (p_{udp}), ICMP frames (p_{icmp}) and into frames which are not allowed by the firewall rules, regardless of protocol ($p_{blocked}$). Therefore $p = p_{tcp} + p_{udp} + p_{icmp} + p_{blocked}$.

The amount of data flowing through the system is also an important factor. Due to this we include the average frame size (s) into the model.

As we are considering a stateful packet filter, we are also interested in the rate of new connections initiated through the system. This will be expressed as number of new connections per second, or c.

The firewall needs to process the firewall rules for those packets which are not yet part of any firewall state entry. Due to this the number of configured firewall rules (r) is needed in the model.

The last pair of factors are slightly more complicated. To estimate the complexity of firewall state processing, we need to know the approximate state tree depth, which we will call z. To calculate the depth, we need to know the number of simultaneously open connections through the firewall (which we will call q). For this, we will need the average connection age (l), expressed in seconds. We also need the configured timeout period during which the firewall does not remove a state entry even if it has detected the end of a connection. The timeout (t) is expressed in seconds.

If the firewall would erase the state entries as soon as the connection has been closed, the number of simultaneous connections would be $q = c * l$.

However, as explained in section 2.2, the state entries can be erased only after a timeout period has passed. The timeout is configurable separately for all protocols, but we will here assume that it has been set to t seconds for all traffic types, because that is a realistic setting at least in our tests later. This setting effectively adds t seconds to the length of every connection. Therefore the actual number of simultaneous connections is:

$$q = c * (l + t) \tag{4.1}$$

The firewall keeps track of these connections with state entries. The firewall under investigation keeps separate state entries for incoming and outgoing connections and thus needs two entries for one forwarded connection. Therefore the number of state entries in OpenBSD PF is $2 * q$.

Now we can calculate the state tree depth. Since the tree in the modelled firewall is a red-black tree, the maximum depth of the tree is two times a logarithm of two times the number of nodes in the tree (See 2.2.4 for details). We leave the multiplication by two out because it will be included later in the constant factor. Tree depth parameter thus becomes:

$$z = log_2(2 * 2 * q) = log_2(4 * c * (l + t)) \tag{4.2}$$

In the following discussion we will use factors p (and its components p_{tcp}, p_{udp}, p_{icmp} and $p_{blocked}$), s, c, r, q and z. Factors l and t are needed only to figure out q and z.

4.2 Routing Load

First we will consider the load behavior of a router. When firewalling is disabled on a routing firewall it becomes a normal router. Therefore the load behavior of a router can be used as base for load modelling. We will assume that the number of routes is small and constant. This reflects to the typical firewall use scenarios.

IP routers are stateless frame forwarders. A router does not need to understand the protocols used on top of the IP-protocol. Therefore it does not need to have a concept of connection, it simply forwards incoming frames into correct networks. This means that c (connections per second), q (number of connections), z (state tree depth) and r (number of firewall rules) are not factors in router load.

A router performs a same set of operations for each received frame. The frame header is examined to find out the target IP address. The frame is then sent forward to the target network. The routing load itself is thus dependent mainly on the rate of frames processed.

On the other hand, when frame forwarding is done in software, frames need to copied to and from the network interface cards. Frame checksums need to be computed either by the operating system or by the networking hardware. These operations have complexity directly proportional to the amount of data. Therefore the traffic data rate is a factor in router load. The data rate equals to number of frames per second (p) times the average frame size (s).

The basic router load thus comes to:

$$L_r = C_1 * p + C_2 * p * s \tag{4.3}$$

4.3 Normalization

The next step in the packet's flow through the firewall is the normalization code. Normalization checks the integrity of the frame and sometimes reconstructs the entire frame. IP fragments are the most common cause for frame reconstruction. We will (somewhat unrealistically) assume below that frame reconstruction occurs either with all frames or with no frames.

Normalization is problematic for load analysis because the normalization load depends heavily on frame contents. Fragmentation could be modelled with a separate term which would have parameters for fragmented and not fragmented frames. Similarly we could have different terms for frames which need tuning of TCP or IP flags and other parts of frames which might need normalization checks.

Since we do not have enough time to test the combinations of these parameters

later, we will ignore the effect in this work. This would be rather unacceptable in a general case, but since the networks where we usually use the measured firewalls are already behind another firewall we can get quite usable model without including different normalization parameters in the model.

If we assume that the normalization time is independent on the frame contents we get two terms for the normalization load. First of all normalization checks and sets fixed fields in frame headers. This is an about constant time operation so the load is directly proportional to framerate (p). In addition, the checksum recalculations and refragmentation operations need to go though the whole contents of the frame. Therefore we expect this part to scale linearly proportional to the amount of data processed or $p * s$.

Looking at the equation 4.3 above we can see that it contains exactly these two terms. Since it would be hard in practise to differentiate the effects of the basic firewall load and the normalization load we assume that the normalization is included in the basic firewall load.

Why do we assume that the effect differentiation would be hard? In theory we can test basic firewalling both with and without normalization. However since the normalization code used the exactly same bits of data in memory as the basic firewalling code and the amount of data is quite small the cache effects would probably make it practically impossible to differentiate the effects. We make the assumption on caches based on some very preliminary load tests we made with and without normalization.

4.4 State Table

The state table is perhaps the most implementation specific part of a firewall. Due to this we have already encapsulated the complexity of state table operations into the factor z which represents the complexity of a state table search, add or removal. Of course searches usually are cheaper operations than additions or removals, but we will address this in respective constant terms.

In state table processing the firewall compares the incoming frame against the table existing connections. If it finds a match, it passes the frame and updates the state table information. If it doesn't find a match, it proceeds to firewall rule comparison.

State table search complexity is thus proportional to z. Therefore the search causes load proportional to both framerate and tree complexity, or $p * z$.

Frames for which the firewall does not find a match for are either part of $p_{blocked}$ or c. The firewall checks these frames against the firewall rules (see next section). Frames which are allowed by firewall rules cause new state creation.

New states are therefore created at rate c. Since we are forming a model only for stable constant load, we consider c and z to be constant during the time frame of our load calculation. This means that the rate of connection termination is also c. Thus the firewall has to both create new state entries and remove old entries at rate c.

The creation or deletion of a state entry is nearly a constant time operation if we do not think the insertion and deletion in the state table structure. Therefore this part has complexity directly proportional to the connection rate (c).

The order of complexity for a single state addition and removal in the state table is z. Therefore the load of state table manipulation is proportional to $c * z$.

State removals in our reference firewall are actually done only based on the timeout. Timeout is not a key in the state table search tree. Therefore the firewall periodically checks all state entries for timeout expiration and removes expired states. This search through the whole structure has complexity directly proportional to the number of state entries (q).

The state table load thus comes to:

$$L_t = C_3 * p * z + C_4 * c + C_5 * c * z + C_6 * q \tag{4.4}$$

OpenBSD/PF creates actually two state entries for each connection. We do not have to figure this into the formula above since this fact is already part of the state tree depth (z). Factors $C_4 * c$ and $C_6 * q$ do not have z component, but multiplier two can be thought to be part of the constant term.

4.5 Firewall Rules

If an incoming packet didn't match any existing entry in the state table, the firewall must check it against the firewall rules.

As we discussed in section 2.1.4, there are multiple ways in which the firewall rule processing can be arranged for optimal efficiency. Since we are looking for an equation which can be used to give safe estimates on firewall capacity, we will assume that all optimization have failed and degraded into a simple linked list. We will later manually ensure that this degrade happens with all test cases. This assumption is not far away from the reality in our test-bed firewall since its rule optimizations are quite primitive.

The firewall checks the rule list for each frame which doesn't match any existing state table entry. Rules can then either pass the frame or block it. Both actions usually cause logging (see next section). Passes cause state generation, but this is already included in L_t. The actual frame passing is part of the basic routing

load L_r.

For each blocked frame the firewall needs to scan through the entire rule list. The complexity of this operation is directly proportional to the length of the rule list. OpenBSD/PF matches firewall rules with last matching rule policy. Therefore it needs to scan the entire rule list even for passed frames.

Firewall rule predicates are based in fixed position fields in frame and protocol headers. For this reason, the length of the frame does not affect the complexity of rule checking. Therefore the rule checking load scales linearly on the number of rules.

$$L_f = C_7 * (c + p_{blocked}) * r \tag{4.5}$$

4.6 Logging

Next we have to include the load which is caused by generation of log entries. We assume that one log entry is generated for each new connection and for every blocked frame by relevant rule actions.

Log entries are approximately equivalent in length. The logging itself is done one entry at a time as new rule actions with logging match in firewall rule processing. Therefore the load caused by logging is directly dependent on the rate of log entry generation, or:

$$L_l = C_8 * (c + p_{blocked}) \tag{4.6}$$

4.7 Synchronization

The components of firewall load we introduced above cover the load for a single firewall. To provide a load model for high availability hot-standby firewall, we have to include the state synchronization load.

In hot-standby model only the currently active firewall processes traffic. It sends state information updates to the standby firewall as the state information changes to keep the standby firewall ready to take over should the active firewall fail. We will first consider the load effects of state synchronization on the active firewall.

When the firewall creates a new state for a newly passed connection it must send information on this creation to the standby firewall. In addition, to track the status and timeout values of the connection, it needs to send updates whenever a frame is passed because it matches the state (to update counters and timeouts).

Thus there will be one synchronization event for each passed frame. The complexity of this event may depend on whether it is a state creation event or an update event. Therefore the load comes to:

$$L_s = C_9 * (p - p_{blocked} - c) + C_{10} * c \tag{4.7}$$

Or in terms of passed frames:

$$L_s = C_9 * (p_{tcp} + p_{udp} + p_{icmp} - c) + C_{10} * c$$

In practise the active firewall will group synchronization messages. In a busy firewall, this divides the complexity of the synchronization events by some implementation specific constant (part of C_9 and C_{10}). A firewall which receives input frames so infrequently that grouping cannot be waited for is not an interesting case for load analysis anyway.

Now we have to take a look into the standby firewall. Since we are interested only in the overall performance of the HA pair, we will only consider whether the active or the standby firewall is the limiting resource for performance.

The active firewall is loaded by all the factors described above. The standby firewall does not process frames, so it is not loaded by routing, normalization, firewall rules or logging. The load therefore comes from state table operations and synchronization traffic.

The standby firewall receives the synchronization events sent by the active firewall. We expect this to be approximately as complex operation as sending the events. The standby firewall also mimics the state table operations performed by the active firewall based on the synchronization traffic. Therefore we can safely assume that the standby firewall is never a limiting resource in HA firewalling system and it can be ignored in the load analysis which search the performance limits of the overall system.

The high availability heartbeat traffic is sent only infrequently (about once per second) and does not depend on the input frames to the system. We assume that it causes no measurable load on the system.

4.8 Summary of Load Components

By adding together all components of load in equations 4.3-4.7 above, we get:

$$L = L_r + L_t + L_f + L_l + L_s \tag{4.8}$$

Which expands to:

$$\begin{aligned} L &= C_1 * p + C_2 * p * s + C_3 * p * z + C_4 * c + C_5 * c * z + C_6 * q + \\ & C_7 * (c + p_{blocked}) * r + C_8 * (c + p_{blocked}) + \\ & C_9 * (p - p_{blocked} - c) + C_{10} * c \end{aligned} \quad (4.9)$$

The equations above do not have a constant term. The term is missing because by the definition of load (in section 3.1.1) $L = 0$ when the system is idle. When the system is idle, there are no incoming frames, which makes $q = c = p = 0$. As we can see, this makes $L = 0$.

4.9 Performance Constraints

Simple load formulas presented above can not possibly contain enough information to model the firewall in all possible usage scenarios. There are many fixed limits which cause rapid performance drops. These drops can not be modelled with a simple linear load model.

We need to specify the constraints for the load formula applicability domain. At least the following issues must be considered.

All firewalls have some absolute maximum numbers of firewall rules and simultaneously tracked connections. These limits usually, but not always, depend on the available memory.

In practise the firewall is probably not useful when it is operating near the absolute rule and connection limits. Caches are the main reason for this. Only some number of rules and state entries can be kept in caches and the performance drop might be huge especially when some of the rules are not in cache since rules are usually processed linearly as a list. With state entries the situation is somewhat better at least if some states are used much more frequently than others. The most critical state entries can then be kept in caches more frequently.

The number of rules and states where performance of the system drops dramatically can not be accurately pre-determined. We can find these limits while we perform tests to find out the fixed co-efficients in the load formula. We can then limit the applicability domain of the model in the area where performance can be modelled with our linear model.

The physical environment also sets some hard limits. The most significant limit is the maximum capacity of the networks into which the firewall is connected to. The networks set hard limits on the traffic bandwidth and frame rates.

4.10 Other Issues

The model presented above assumes that the performance of the firewall is identical with different types of traffic. The model takes into account the frame size, but does not address protocol issues.

With unfragmented UDP frames the model is probably quite near to reality since it is constructed with UDP traffic in mind. TCP is more complex to track and normalize. This can affect performance and separate fixed co-efficients for state operations and basic firewalling might be needed. ICMP traffic is quite similar to UDP in tracking complexity so we assume they cause nearly the same load on the firewall.

Perhaps the biggest issue is fragmentation. The normalization load can vary significantly when fragmentation reassembly is done.

We will not model these issues in this work, but doing so might be an interesting topic for further study.

4.11 Model Modifications for Other Firewalls

Now we have formed a load model for OpenBSD/PF firewall. Is this model usable for other HA firewalling systems? For most parts, we expect it to be.

The routing load is accumulated from the same factors on all routers, so for L_r the model should be directly usable with all firewalls.

Traffic normalization operations performed by different firewalls vary significantly, but the complexity should still be dependent on frame rate and size of frames. If we configure a firewall with the same logging policy as we used for PF the logging complexity should also be similar. There is also no way to get around the fact that the state synchronization must be done for passed frames. Therefore the synchronization load should also be of the same complexity class as in our reference firewall.

Differences arise in firewall rules and state table operations. Many firewalls have better rule optimizations than PF and the r component of the load model needs to be changed to match the particular firewall being modelled. For example in case of a decision tree based firewall, this can be done simply by defining r to be log_2 of the number of firewall rules.

State table implementations tend also to differ. Many firewalls use hash tables instead of trees. We have defined z to be the maximum depth of the search tree used to store the state entries. If we change the definition to complexity of an individual state operation, z stays the same for red-black tree implementations. For other implementations we can define z again using its subcomponents c, l

and t and use the rest of the model unmodified.

We use also the number of simultaneous connections (q) in the model. This parameter is used only in the term which describes the load effects of expired state cleaning. If we define q to be the complexity of this operation on average instead of the number of connections we can use the same term for other firewalls.

As we can see, the model should be usable on most firewalls with a simple redefinition of r, q and z. The load formula itself can usually be used unmodified.

Chapter 5

Tests

In the previous chapter we formed a model for firewall load. The model, although specific to our test-bed firewall, still has ten undetermined constant factors. In this chapter we will present a method for performing tests which can be used to determine these factors.

In addition, the tests should verify that the complexity of individual firewall load components in reality matches adequately to the complexity assumed in the load model.

5.1 Measurement Techniques

The basic problem in load tests like this is that we cannot reliably measure the exact load of the tested system (L). There are two exceptions. The trivial case of no load ($L = 0$) is easy, because it needs no measurements.

The other reliable measurement point is the maximum load ($L = 1$). This is the point where the system can barely manage the input flow, but cannot take any more. We can not measure this point directly. However, when the firewall is overloaded ($L > 1$) we can see that frame lossage occur. As we discussed in section 3.1.3, frame lossage in a frame forwarding system indicates overload condition.

Thus to find out the combination of parameters which cause the system to be fully loaded we will select one of the parameters as a free variable. We will then feed input to the system, varying the free variable. If the system does not lose frames with the current value of the free variable, we know that the searched value is higher. If the system loses frames, we know that the value should be lower. We will use binary search to find out the value of the free variable when $L \approx 1$.

We will use p (frames/s) as the free variable in all test cases. We use $p = 0$ as the low bound for binary search. We will perform preliminary tests to find a reasonable upper bound for p. Since we will handle p as an integer, the search will stop when the upper and lower limit in the binary search are the same. We have no way to check for an exact match.

In theory we could use the definition above: any frame loss indicates overload and the search will continue with lower upper bound for p. However, in practice some frame lossage can occur regardless of the load of the system because of effects in switches and load test machines. Therefore we will consider that frame lossage occurs only if we lose more than five frames in a thousand.

For tests including logging, we need also consider the rate of lost log entries. Since logging is done with lower priority than the firewalling itself, overloaded system might forward frames perfectly and lose logs. We will use the same limit as for lost frames to see whether log entries are lost. Since lost frames are usually not logged, the loss rates should correlate heavily anyway.

5.2 Test Environment

Test environment is of course limited by economical constraints. Therefore we can only use 4 machines for the tests. Two of these are configured as firewalls, one as a load generator and one as frame receiving host. More hardware would make it possible to perform tests with wider scale of parameters. This set should however be adequate to simulate most realistic installation environments.

The two firewall hosts have 3GHz Intel Xeon CPUs, 1GB RAM and Intel gigabit Ethernet NICs. This CPU has 8kB L1 data cache, 12kB L1 code cache, 512kB L2 combined cache and 2 MB L3 cache. The firewalls run OpenBSD 3.7 with the default installation of PF.

Similar hardware is used for the load generator and frame receiving hosts. We use FreeBSD 5.4 operating system in these machines, mainly because of hardware compatibility issues which require changing of parts before installation of OpenBSD.

Two 3Com 3824 (3C17400) model switches are used to interconnect the hosts.

The networking topology is illustrated in figure 5.1. Both tester hosts have their own networks and they are connected to the firewalls through the switches. In addition, the firewalls are connected to directly to each-other with cross-over cables for state synchronization traffic. The dashed lines in the figure show the test software control connections (see below for details) and the thick lines show the test frame flow direction.

This setup is not equivalent to the tests described in RFC2544[11] which defines

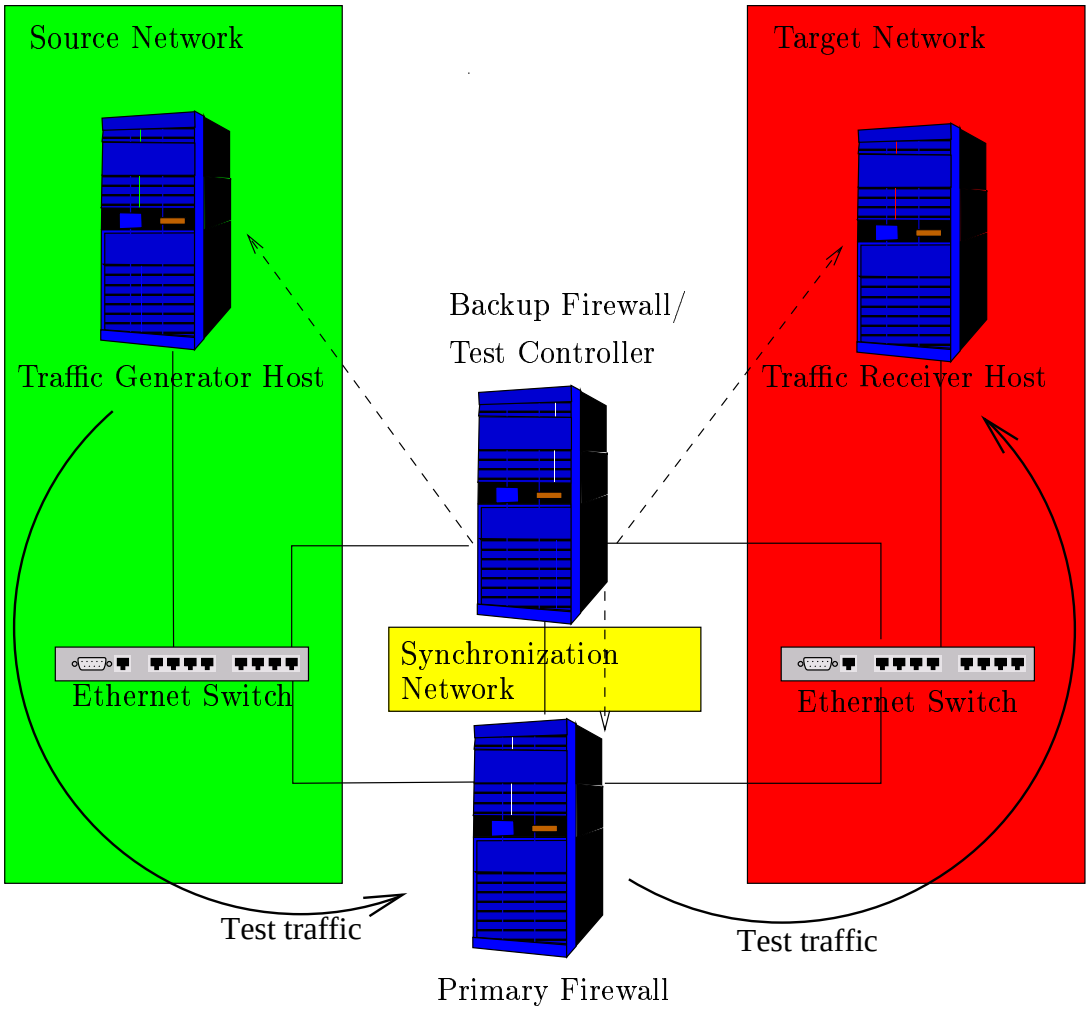

Figure 5.1: Test Setup

the reference tests for network interconnect device performance measurements (This reference test set is a companion to the above-mentioned performance terminology paper RFC1242[10]). The RFC requires that the traffic is looped back into the sending hosts so that the same software both sends and receives the traffic. This setup is practically only needed to reliably measure latency. Since we are not measuring latency, but only frame loss rate, we decided to use simpler two machine setup.

5.3 Test Software

Tests are controlled by a master test script which runs on the backup firewall. This is the only machine in the test configuration which is in all cases in relatively small load. All other hosts run a test server application which listens on a TCP port for commands from the master test script.

On the primary firewall the test server application waits for reconfiguration commands. With these commands the master test script can install any available PF configuration file into the firewall. Installation of a new configuration file also flushes the state table.

On the test traffic generation and receiving hosts the test server executes traffic sending and receiving applications. Test parameters are part of the command from the master test script. After the specified time has expired, the traffic generator and receiving applications will return with traffic statistics. The test server then returns the results to the master test script for aggregation.

The traffic sending side will return the number of frames sent. The receiving end reports the number of received frames. The master test script can calculate the number of frames which should have been sent from the test parameters.

The master test script runs the above described test runs repeatedly to do binary search on the frame loss rate. It varies the frame sending rate and stops the search as it has found the integer framerate which is closest to the point where frame loss rate is 0.005.

Originally we intended to use this test script set to run two tests simultaneously so that both test hosts would send and receive traffic. The master test script would then aggregate results from both simultaneous runs. However, the combined framerate with two separate test flow was considerably lower than with a singe flow even without the firewall in the middle. Because of this somewhat unexpected phenomena we decided to use only one test traffic flow in the tests.

We are not certain of the cause of the performance degrade of the network with multiple flows. The switch ports were in full-duplex mode and the load reported by the OS in the test hosts was rather low. The switch backplane should be capable handling the traffic easily. The reason for degrade was probably combination of traffic collisions in the switch and effects of load in the test host kernels.

We originally intended to use a traffic testing tool Iperf[2] as the back-end to the test scripts. It turned out not be a suitable tool for the job for two reasons. First of all, it lost considerable amount of frames in the sending side. In addition, it could not be configured to use multiple sending and receiving ports which was needed to simulate hundreds of thousands of connections.

Because of these problems we wrote our own tool for traffic generation and re-

ceiving. The simple tool allocates a configurable number of sending and receiving UDP ports and sends traffic with specified framerate and frame size. The tool reports both the actual number of frames sent during the test (on the sending side) and the number of frames actually received (on the receiving side).

5.4 Preliminary Tests

We ran a few preliminary tests to set up rest of the tests properly.

First we checked what is the maximum framerate which the test hosts can sustain without the firewall in the middle. This provides an upper bound on the rest of the results. The maximum rate was about 300000 frames/s. We call this test "the base test case" because it sets the limit for rest of the tests.

Next we checked whether we can overload the firewall with our test setup with minimal set of firewall rules (one rule which allows all traffic). It turned out that this is not possible because routing and normalization are together cheaper operations than frame sending from a userland program.

By trial and error we found that with at least 50 firewall rules we can reliably overload the firewall with any frame size. Therefore we use at least 50 rules in all actual tests. This is unfortunate since in many practical scenarios we do not need 50 rules.

We also verified that the selected frame loss limit (0.005) for fully loaded system was reasonable. When we used lower value, the measured frame rate dropped significantly (in some cases more than 90%) as some frame lossage seems to happen with even small framerates. With higher values the results were approximately the same as with 0.005. We decided to use 0.005 as the limit in all test cases.

Preliminary tests also revealed the problems with IPerf and two way testing detailed in the section above.

5.5 Test Cases

The basic goal of test planning is to isolate different phenomena in different tests. We plan to test one part of the firewall in one set of tests whenever possible. However, the preliminary tests showed that we can not measure the routing and normalization loads alone with the available hardware. These parts of the firewall do not together cause load to raise to 1 or higher with 1Gbit/s frame flow.

The second constraint in test planning is that we must have enough tests to make regression analysis on the constant possible.

All test cases below are run with the test environment and software detailed

above. In all cases we run the tests repeatedly with different value of framerate (p) to find the maximum sustained framerate.

5.5.1 Basic Firewalling Tests

In the first set of tests we run the firewall in the most minimal configuration possible. The goal is to have enough data to find out values of constants C_1, C_2 and C_7. Therefore the relevant part of the load formula 4.9 is:

$$L = C_1 * p + C_2 * p * s + C_7 * c * r$$

For these components we need to have basic routing and normalization. In fact, these two components can not be avoided in any test case. In addition to these required minimum features we set up variable number of firewall rules. All rules allow traffic to pass.

We want to use realistic values for frame size (s) and number of firewall rules (r). For frame size we limit the test in sizes which can be fit within one Ethernet frame. Thus the value range for s is theoretically from 58 (sum of header lengths) to 1500 (maximum Ethernet frame size). In practise we decided to use 50, 100, 300, 500, 700, 900, 1100 and 1400 octet UDP payloads. The actual frame size is 58 octets more because of UDP, IP and Ethernet frame headers.

Since we do not use firewall's state engine all frames are new connections and cause rule checkup. Due to this the connection rate (c) and frame rate (p) are identical.

For firewall rules we would like to cover a realistic range of values. The optimal set would be from 1 to 1000. Unfortunately preliminary tests showed that we cannot get reliable results with less than 50 rules. Therefore we use $50 - 1000$ rules as the test range.

We configure the firewall not to form state entries for connections during these tests.

5.5.2 State Tests

We divide state handling tests into two cases. One set of tests covers the creation and deletion of states. The second set measures effects of searches from existing states. The state creation and deletion are both covered by term state creation below since in stable situation the rate of deletions is equal to the rate of creations. In this section we assume that the state synchronization mechanism of the firewall is disabled.

The relevant part of the load formula for state creation tests is:

$$L = C_1 * p + C_2 * p * s + C_3 * p * z + C_4 * c + C_5 * c * z + C_6 * q + C_7 * c * r$$

In theory we could use only one firewall rule in the state tests. However, since we can not get reliable load results from other tests with such a low number of rules we use 50 rules also for this test set.

We run the state creation tests by using 10000 sending UDP ports on the traffic generator and 1000 receiving ports on the receiving host. This way we can get 10000000 different connections in the perspective of the firewall state engine. By only sending one frame with each port combination until all combinations have been used we can thus simulate a huge number of connections.

We vary the state expiration timeout (t) between 1-30s in these tests. Therefore as long as we keep the framerate below $\frac{10^7}{timeout}$ the old state for the port combination has always expired before the port pair is used again. With 30s timeouts the maximum usable framerate in these tests is well above the maximum framerate we measured in the preliminary zero test case.

The port pair rotation causes the connection rate (c) to be equal to frame rate (p) in these tests. Because the frame size is not a factor in state load, we use only the small 158 octet frame size in these tests.

As always, we find out the maximum p. We vary the number of connections (q) and therefore also the state tree depth (z) by altering the state timeout settings. By setting timeout check interval to 1s and state timeouts to 1-30s we can get 12 different values for z.

We can find out the value of q by equation 4.1: $q = c * (l + t)$. Since in this test set $c = p$ and the length of a connection (l) is zero we get: $q = p * t$. Thus we know the exact value of our primary test parameter only after the test has been done.

To find out z for a test case we can use equation 4.2: $z = log_2(4 * q)$ or $z = log_2(4 * p * t)$. After these calculations we can present the test results in a form where we can see the maximum framerate for a value of q or z.

In the second set of state tests we will perform state searching tests. In state searching we have a constant number of connections during the whole test set. Every connection has the same framerate.

The relevant terms in the load formula are:

$$L = C_1 * p + C_2 * p * s + C_3 * p * z + C_6 * q$$

We will perform these tests in two phases. In the first phase one frame for each connection is sent trough the firewall with a low framerate (500/s). These frames

force the firewall to form the state entries for connections. After that the actual test begins and it is done just like the state creation tests. We will flush the state table of the firewall between every test run.

We configure the test traffic generator and receiving application both to have number of UDP ports equal to the square root of the number of connections. Due to this we select all our connection counts to be squares of some integer.

Because we form all connections before the actual test the connection rate (c) is 0 in these tests. This is why we have no terms with a c component in the formula above.

The frame rate (p) is again the measured variable. We define frame size (s) to be a constant 158 like we did in state creation tests.

Unlike in state creation tests we now know values of q and z in advance. Connection count (q) is our primary test variable. State tree depth can be calculated with equation 4.2: $z = log_2(4 * q)$.

In these tests we configure the firewall to use stateful filtering. In addition to prevent accidental state expiration during tests with high number of connections and low framerates we set the state expiration time to 120s.

5.5.3 Synchronization Tests

We keep the state synchronization feature of the firewall disabled in the state tests described above. To measure the performance impact on state synchronization we will run these same tests again with synchronization enabled.

In this set of state creation tests we add the term for state creation and deletion synchronization to the load formula. It thus becomes:

$$L = C_1 * p + C_2 * p * s + C_3 * p * z + C_4 * c + C_5 * c * z + C_6 * q + C_7 * c * r + C_{10} * c$$

Similarly we add the state update term to the formula for state searching tests. Since there are no blocked frames or new connections in these tests c and $p_{blocked}$ are zero. Thus the load formula for these tests is:

$$L = C_1 * p + C_2 * p * s + C_3 * p * z + C_6 * q + C_9 * p$$

5.5.4 Logging Tests

To measure the effects of logging we will again use the state creation tests. This time we use the synchronized version of the test cases as a base and enable the logging in the firewall.

We configure the firewall to log first frames of each connection (and to log all rejected frames but there are non in the tests). Since we use state creation tests where $c = p$ all frames get logged.

The load thus comes to:

$$L = C_1 * p + C_2 * p * s + C_3 * p * z + C_4 * c + C_5 * c * z + C_6 * q + C_7 * c * r + C_8 * c + C_{10} * c$$

To save time we use only a few of the 13 state creation tests in the logging test set. We select timeout values 1s, 5s, 10s and 20s for tests cases in this set.

As explained in the test tools section above we will use both frame loss rate and log loss rate in these tests. The test script will find out both the actual number of passed frames and the number of log entries the firewall generated. The lower number of these two is used to calculate the frame loss ratio.

5.5.5 Summary of Test Cases

We have five separate test case sets. The tests, parameter settings and test objectives are summarized in table 5.1.

Total number of test cases in basic firewalling tests is 88. We have $13 * 2$ test cases for state creation, $14 * 2$ cases for state searching, and 4 cases for logging. The total number of test cases is 146.

The run time for one test measurement is 5 minutes and we need to binary search the limit value for framerate with $log_2(p_{max}) \approx 20$ test measurement runs. This means that the total runtime for all tests is about one 10 days.

We had to run majority of the tests multiple times because of errors in the test scripts and test setups. We had no time for more tests which was one reason for dropping the fragmentation and TCP issues out of the scope of this work.

Due to long run time of the test sets we did not have time to run them repeatedly with identical parameters to get results with less measurement errors.

5.6 Results

In this section we present the results of the tests. Analysis of results is available in the next chapter. Results are mostly presented with graphs, full numerical results are available in appendix B.

Name	**Target**	p	s	c	r	q (z)	L
Preliminary Tests	None	Variable	108, 158, 258, 358, ..., 1458	equals p	1, 5, 10, 20, 50, 100, 200, ...	0	
Basic Firewalling	C_1, C_2 & C_7	Variable	108, 158, 258, 358, ..., 1458	equals p	50, 100, 200, 300, ..., 1000	0	$C_1*p+C_2*p*s+C_7*c*r$
State Creation (no synchronization)	C_4, C_5 & C_6	Variable	158	equals p	50	depends on c	$C_1*p+C_2*p*s+C_4*c+C_5*c*z+C_6*q+C_7*c*r$
State Creation (with synchronization)	C_{10}	Variable	158	equals p	50	depends on c	$C_1*p+C_2*p*s+C_4*c+C_5*c*z+C_6*q+C_7*c*r+C_{10}*c$
State Search (no synchronization)	C_3	Variable	158	0	50 (irrelevant)	1-255530 (2-20)	$C_1*p+C_2*p*s+C_3*p*z$
State Search (with synchronization)	C_9	Variable	158	0	50 (irrelevant)	1-255530 (2-20)	$C_1*p+C_2*p*s+C_3*p*z+C_9*p$
Logging (with synchronization)	C_8	Variable	158	equals p	50	depends on c	$C_1*p+C_2*p*s+C_4*c+C_5*c*z+C_6*q+C_7*c*r+C_8*c+C_{10}*c$

Table 5.1: Summary of test cases

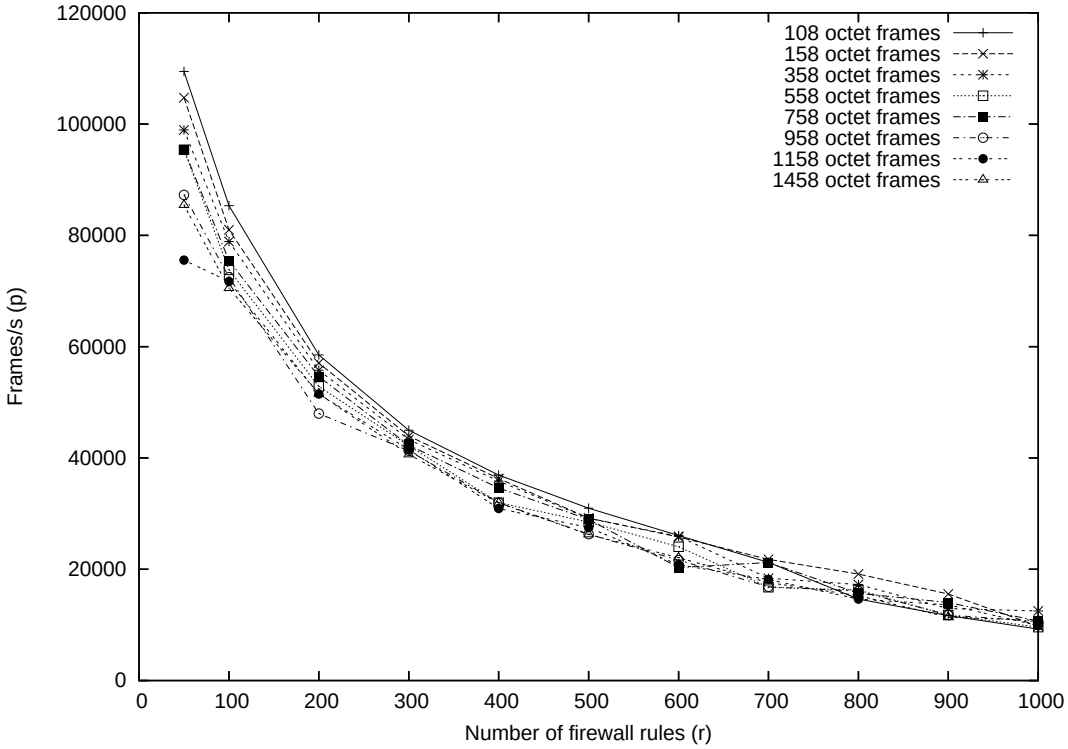

Figure 5.2: Basic firewalling capacity as a function of firewall rules and frame size

5.6.1 Basic Firewalling

The measured maximum framerate (p) with basic firewalling can be seen in figure 5.2. The number of firewall rules used in the test ($50 - 1000$) is shown on the x-axis. Separate plots are shown for each frame size ($108 - 1458$ octets).

The maximum measured framerate was 109460 frames/s. This rate was achieved with 50 firewall rules and 108 octet frames. The smallest measured rate was 9233 frames/s with 1000 firewall rules and 108 octet frames.

As can be seen in the figure, the frame rate decreases as the number of firewall rules and the frame size grow. However, after the number of rules is above 500 the plot lines start to cross each-other frequently.

5.6.2 State Creation and Searching

State creation test results can be seen in figure 5.3. The vertical axis shows the obtained framerate. The configured connection timeout value can be seen on the horizontal axis. Results with and without synchronization are shown in the same

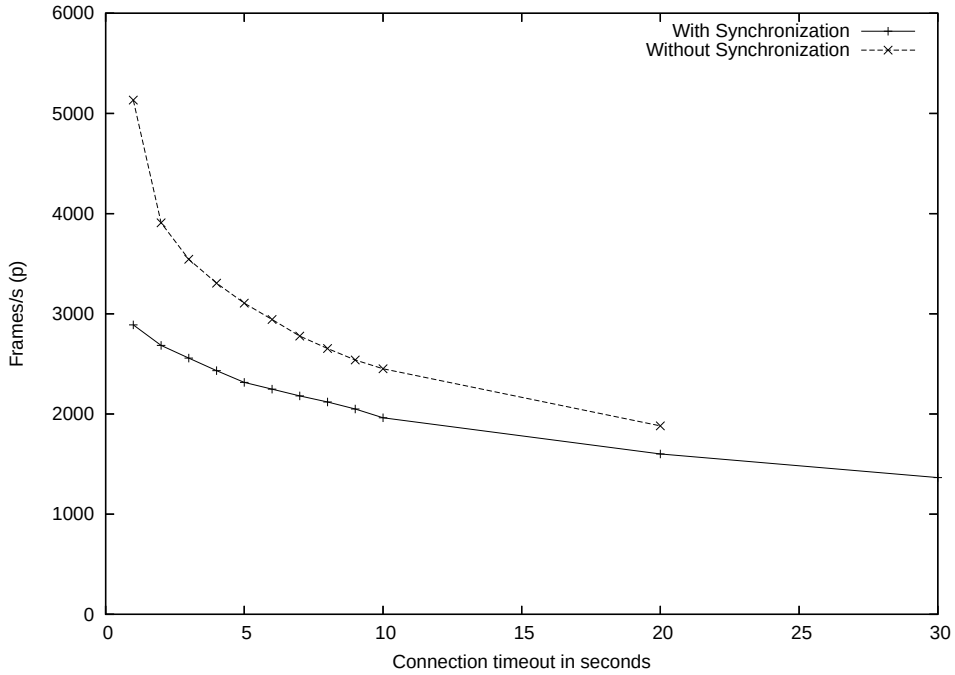

Figure 5.3: Connection creation rate with varying connection timeout values

figure.

As discussed above, the connection timeout value defines the number of connections kept in the state table. The connection count under stable load with only one frame for each connection is timeout multiplied with connection initiation rate (framerate in this case). The same results as above are illustrated in figure 5.4 with the number of simultaneous connections (q) in the horizontal axis. Since the x-axis is a derived value the data points are not in the same x-coordinate for both synchronized and unsynchronized tests in the figure.

With synchronization the peak framerate, and thus the peak connection creation rate, was 2889 frames/s with the 1s timeout (about 2889 simultaneous connections). The lowest rate was 1365 frames/s with the 30s timeout (about 40950 simultaneous connections).

Without synchronization the framerates are higher. The peak connection creation rate was 5134 frames/s with the 1s timeout (about 5134 simultaneous connections). The lowest measured rate was 1881 frames/s with the 20s timeout (about 37620 simultaneous connections).

The result for 30s timeout without synchronization was unfortunately lost.

Results from the state searching tests are illustrated in figure 5.5 (Note the logarithmic scale on the horizontal axis!). The vertical axis shows the framerate

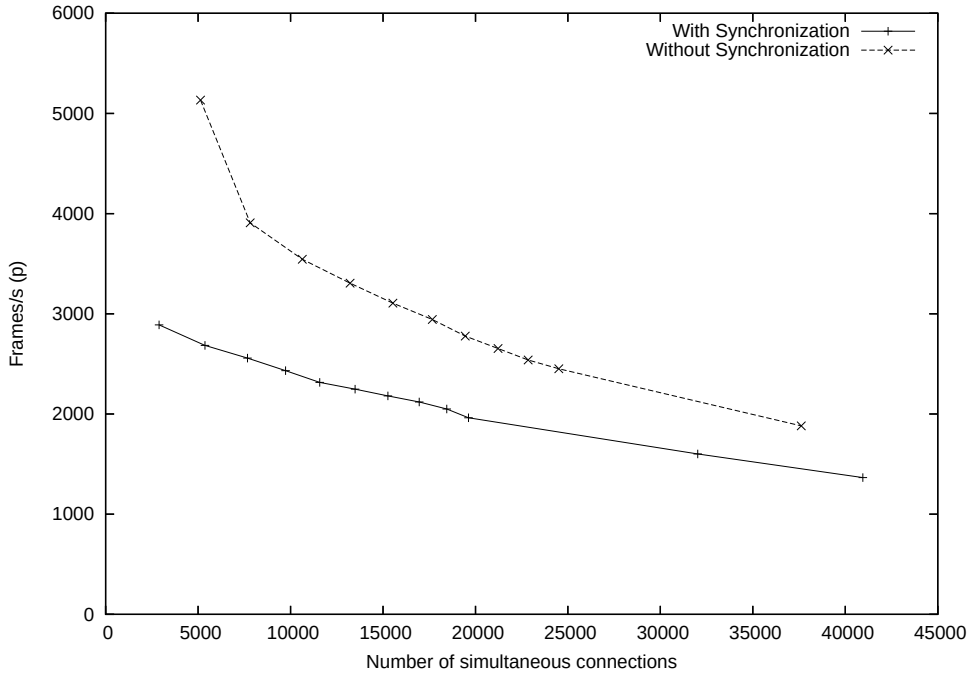

Figure 5.4: Connection creation rate with varying number of simultaneous connections

obtained and the horizontal axis shows the number of simultaneous UDP "connections" used while measuring. The figure shows results both with and without synchronization.

With synchronization the peak performance was 155748 frames/s, obtained with only one connection. The lowest measured framerate was 457 frames/s with 255530 connections.

The sustained framerate drops to about half from the rate with one connection when 100 simultaneous connections were used. Performance degrades steadily to about 30000 connections after which the framerate drops radically.

Without synchronization the peak framerate was 159436 which was measured with 484 connections. The result with one connection was slightly smaller, 155105 frames/s. The lowest result was 198 frames/s with 255530 connections. Framerates with $1 - 992$ connections were within 6.2% the peak value.

The sustained framerate drops less dramatically without synchronization than with it. Rather steady decline can be seen from the graph (remember the logarithmic x-axis!).

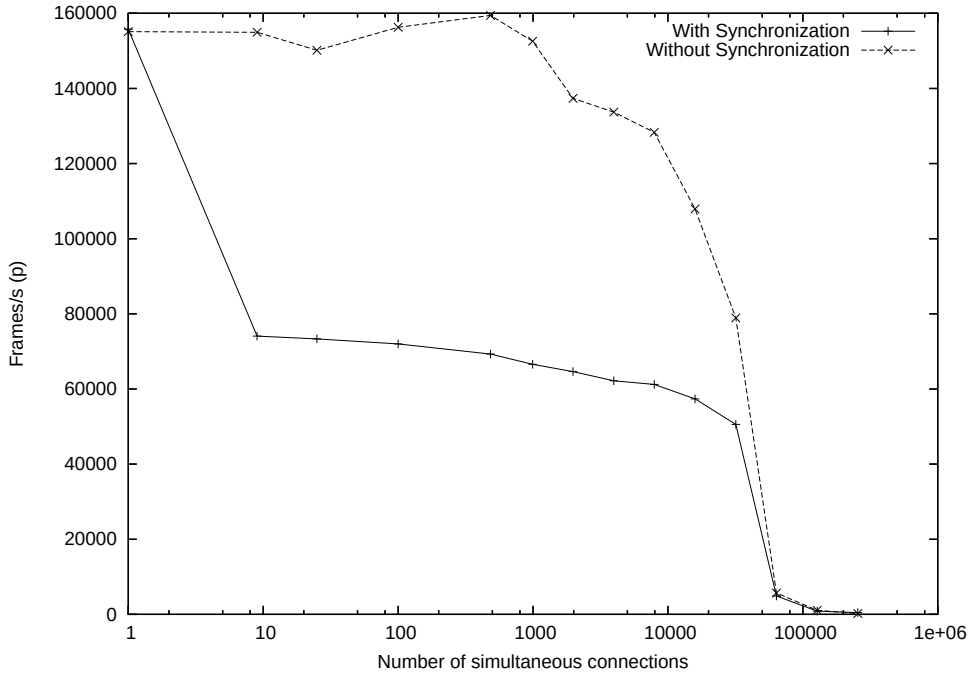

Figure 5.5: Framerate with existing connections with varying number of simultaneous connections

5.6.3 Logging

Timeout	With Logging	Without logging
1	2917	2889
5	2339	2316
10	1989	1962
20	1602	1601

Table 5.2: State creation results with and without logging

The selected data points where we performed state creation tests with synchronization both with and without logging are shown in table 5.2. The maximum framerate with at most 0.005 log and frame loss rates is shown for both types of tests.

Logging results are nearly identical to the tests without logging. The difference is about 1%. The framerate was higher with logging enabled than without logging.

Chapter 6

Analysis

In this chapter we will analyze the test results and form the actual load model for our test firewall. We will first eliminate clearly insignificant terms from the model. Next we will analyze test anomalies. After that we will define the applicability domain for the model. Then we will perform regression analysis to find out the constant terms of the formula. Finally we will analyze confidence intervals and term significance in the resulting formula.

6.1 Term Elimination

When looking at the results shown in the previous chapter we can clearly see that most tested parameters have measurable effect on the firewall performance. There is one exception. The logging tests show results which are within about 1% the results of the same tests without logging.

Since logging doesn't seem to have much effect on the performance we will leave the logging tests and the logging term ($C_8 * (c + p_{blocked})$ of the load formula out of the following analysis.

It should be noted that we didn't perform tests on the firewall blocking capacity. Therefore it might be that logging causes significant load in cases where the framerate is much higher than in our logging tests and most of the traffic is blocked.

6.2 Test Anomalies

The results of the logging tests are not only nearly identical to state creation tests without logging. They are actually consistently higher. At first this seems

counterintuitive. We would expect the performance of the firewall to be lower when it does more work.

The answer probably is in the delicate timing issues in the operating system kernel. If we would measure the absolute maximum throughput of the system we would most probably get lower results with logging enabled. But we actually measure the maximum throughput where we will not lose more than 0.5% of the frames. Since the frame lossage occurs in the firewall when it is too busy to receive or process the frame, timing changes can have surprising results.

6.3 Applicability Domain

As we discussed in section 4.9 we must define the applicability range for all our parameters. We can easily state that the maximum throughput bandwidth in a single interface of the firewall is 1Gbit/s since the host has only 1Gbit network interfaces. From the memory usage of the state table structure we can see that the system can not sustain much more that 900000 state entries, or 450000 simultaneous connections. For more limits we will take a critical look at the test results.

From the figure 5.2 in previous chapter we can see that the predictability of the framerate a function of both firewall rules and frame size is quite low when we have more than 500 rules. The framerate declines steadily as a function of rules, but the lines presenting the different frame sizes cross each-other multiple times. Due to this we will limit our later analysis and the applicability of our resulting model into cases with at most 500 firewall rules. Since we do not have results for any test with less than 50 rules the range of modelled firewall rules is $50 - 500$. This range is sufficient for most practical purposes.

The bandwidth used by majority of the test cases is significantly lower than the maximum physical capacity of the networks. Only couple of basic firewalling tests used the network to the limit. If we would have significant number of cases where this happens we would have to limit these tests out of the analysis because they wouldn't tell much about the actual performance of the firewall. Since the number of cases is so small, we leave them in the result pool.

State entries use about 1kB of memory for each state. Due to this we expected caches to cause rapid performance drop at some number of states. From the figure 5.5 we can see that this decline occurs somewhere between 30000 and 60000 connections. Since we have a measurement points at 31862 and 63756 we will set the applicability limit to about 32000 connections. Since we have measurement data from both 0 and 1 state cases we can use the range $0 - 32000$ as the applicability range of simultaneous connections.

Parameter	Low Limit	High Limit
Framerate (p)	0	159436
Frame size (s)	108	1458
Connection rate (c)	0	5000
Simultaneous connections (q)	0	32000
Firewall rules (r)	50	500
Blocked Framerate ($p_{blocked}$)	0	$\ll p$

Table 6.1: Applicability domain limits

The upper range for connections thus much lower than the memory based limit of 450000. On the other hand, with OpenBSD/PF default settings the configured maximum number of state entries is 10000 which allows 5000 connections.

We do not specify limits for state tree depth because the value is derived from the number of simultaneous connections for which we have limits.

The new connection creation tests illustrated in figure 5.4 do not show similar applicability limits. However, the maximum new connection rate obtained in any test case was 5134. With practical cases we would have synchronization enabled. With synchronization the maximum rate was 2889. Due to this we will limit the model to maximum of 5000 new connections/s.

The maximum measured framerate in any test was 159436 frames/s. This is our upper limit for the framerate parameter in the model. We made measurement only with frame sizes between 108 and 1458 octets (including all headers) so we restrict the frame size in the model between these limits.

Since we have no test cases on firewall blocking capacity (with our measurement model we couldn't have separated them from logging tests) we can not say much about the effect of $p_{blocked}$ component of the load formula. We will thus limit the model to cases where the number of blocked frames is much smaller than the total number of frames ($p_{blocked} \ll p$).

For a summary of these limits, see table 6.1.

6.4 Regression Analysis

Next we will find out the values for constant terms C_1 – C_{10} in the load formula. We will use standard multi-variable regression analysis. All mathematics below is based on [24].

We have results for 146 test cases. After filtering out all logging tests and cases which we dropped out in the application domain analysis above we have 90 usable

results.

When we form a linear equation for each test case based on the expected load model for each case we get the following linear equation group (the full matrix with numbers, 90 rows and 9 columns can be found in table C.1 in appendix C):

$$\begin{pmatrix} p_1 & p_1 * s_1 & \dots & p_1 - c_1 & c_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{90} & p_{90} * s_{90} & \dots & p_{90} - c_{90} & c_{90} \end{pmatrix} \begin{pmatrix} C'_1 \\ C'_2 \\ C'_3 \\ C'_4 \\ C'_5 \\ C'_6 \\ C'_7 \\ C'_9 \\ C'_{10} \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Note that we have left out term $C_8(c + p_{blocked})$ because we dropped it in the term elimination above. Since all tests didn't use all terms we have replaced the missing terms with zeros. The load L is 1 in every equation because $L = 1$ was our measurement point in all tests.

In a perfect world we could just solve the equation group. Of course due to errors in measurements and the model itself we have to use statistical methods.

We start the regression analysis by taking out the left hand side of the equation above to get a matrix of test results. We call this matrix R:

$$\mathbf{R} = \begin{pmatrix} 109460 & 109460 * 108 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 66545 & 66545 * 158 & \dots & 66545 & 0 \end{pmatrix}$$

We can now use the matrix based least square estimation to find out estimates for the constants:

$$\hat{b} = \left(\mathbf{R}^T * \mathbf{R}\right)^{-1} * \mathbf{R}^T * \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} C'_1 \\ C'_2 \\ C'_3 \\ C'_4 \\ C'_5 \\ C'_6 \\ C'_7 \\ C'_9 \\ C'_{10} \end{pmatrix} = \begin{pmatrix} 5.1811 * 10^{-6} \\ 2.9194 * 10^{-9} \\ 1.7793 * 10^{-7} \\ -3.4496 * 10^{-4} \\ 3.7485 * 10^{-5} \\ 1.1107 * 10^{-5} \\ 5.6783 * 10^{-5} \\ 4.4093 * 10^{-6} \\ 1.3248 * 10^{-4} \end{pmatrix}$$

The negative estimator C'_4 looks alarming. None of the constants should be negative because they all model increases in the firewall load. To find out if the

value could be 0 we find out 95% confidence intervals for the constant estimators with T-distribution.

First we have to find vector of errors in the estimated values:

$$\hat{e} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} - \mathbf{R} * \hat{b}$$

Next we need squares of the errors:

$$SSE = \hat{e}^T * \hat{e}$$

Now we can find out the estimated mean square of errors (σ^2). Since we have 90 results and 8 degrees of freedom in the variables:

$$\sigma^2 = SSE/(n-k-1) = SSE/(90-8-1) = 0.011187$$

From which we get $\sigma = 0.10577$. Now we can find out the variance-covariance matrix:

$$\mathbf{V} = \mathbf{R}^T * \mathbf{R}$$

We can now apply these to get the confidence bounds:

$$\hat{\beta}_i \pm t_{\alpha/2} S \sqrt{v_{ii}}$$

Since we are looking for 95% confidence, $\alpha/2 = 0.025$. We expect the errors to be distributed normally. We have $90 - 8 - 1$ degrees of freedom and the table value for T-distibution with these parameters is 1.990.

Thus we can get the confidence interval for C_1':

$$C_1 = C_1' \pm 1.990 * 0.10577 * \sqrt{v_{11}} = 5.2023 * 10^{-6} \pm 7.8202 * 10^{-7}$$

By performing the same operation for all constant estimators we get:

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \\ C_9 \\ C_{10} \end{pmatrix} = \begin{pmatrix} 5.2023 * 10^{-6} \pm 7.8202 * 10^{-7} \\ 2.9065 * 10^{-9} \pm 1.0576 * 10^{-9} \\ 1.2973 * 10^{-7} \pm 7.6275 * 10^{-8} \\ -2.6661 * 10^{-4} \pm 3.7728 * 10^{-4} \\ 3.2001 * 10^{-5} \pm 2.5159 * 10^{-5} \\ 1.2798 * 10^{-5} \pm 4.8088 * 10^{-6} \\ 5.6730 * 10^{-5} \pm 3.8016 * 10^{-9} \\ 4.6773 * 10^{-6} \pm 9.2012 * 10^{-7} \\ 1.2984 * 10^{-4} \pm 3.5848 * 10^{-5} \end{pmatrix}$$

Now we can get back to the problem of negative value of C_4'. As we can see from the confidence intervals above, the value of C_4' can be zero. Looking at the data and load equations carefully we can also find a reason for this.

The terms $C_4 * c$ and $C_6 * q$ are actually heavily linearly dependent. The reason lies in the definition of q. Since in the tests q often is $c * z$ and the value of z changes only between 2 and 17 the values are indeed quite dependent on each-other. In addition the terms with C_7 and C_5 are highly dependent on the value of c.

We solve this issue by removing term $C_4 * c$ from the equations. When we perform the exactly same mathematics as above with the remaining 8 terms in the equation (of course the degrees of freedom for T-distribution are now 92 with table value 1.989) we get the following estimators and confidence intervals:

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_5 \\ C_6 \\ C_7 \\ C_9 \\ C_{10} \end{pmatrix} = \begin{pmatrix} 5.2854 * 10^{-6} \pm 7.7735 * 10^{-7} \\ 2.8559 * 10^{-9} \pm 1.0610 * 10^{-9} \\ 1.1430 * 10^{-7} \pm 7.3478 * 10^{-8} \\ 1.4262 * 10^{-5} \pm 1.6991 * 10^{-6} \\ 1.4804 * 10^{-5} \pm 3.9028 * 10^{-6} \\ 5.6524 * 10^{-8} \pm 3.8110 * 10^{-9} \\ 4.6179 * 10^{-6} \pm 9.2126 * 10^{-7} \\ 1.2267 * 10^{-4} \pm 3.4569 * 10^{-5} \end{pmatrix}$$

The full **R**-matrix for this calculation is available in table C.2 in appendix C.

For most of the values the confidence range is within $\pm 20\%$. The estimation for the value of the C_2 constant is less accurate, the range is $\pm 37\%$.

6.5 Term Significance Analysis

By now we have eliminated two terms from the original load formula. Next we will analyze whether we can eliminate some additional insignificant terms.

We calculated the minimum and maximum values for each remaining term in the load formula based on the applicability domain limits shown in table 6.1. We made the same calculations with the actual minimum and maximum parameters used in the tests. Only those cases which were not dropped in the applicability domain analysis are used. The results are available in table 6.2. The calculations use the mean estimator for each constant, variance is not taken into account.

The maximum value for C_7 term in the tests is based on the state creation tests. The value which we could calculate from basic firewalling tests is not realistic since it assumes no synchronization and state operations are made by the firewall.

Term	Dom. Min	Test Min	Dom. Max	Test Max
$C_1 * p$	0	0.00105	0.84269	0.84269
$C_2 * p * s$	0	0.00062	0.66388	0.35575
$C_3 * p * z$	0	0	0.30919	0.21927
$C_5 * c * z$	0	0	1.20990	1.04900
$C_6 * q$	0	0	0.47373	0.47169
$C_7 * (c + p_{blocked}) * r$	0	0	0.15544	0.01451
$C_9 * (p - p_{blocked} - c)$	0	0	0.73626	0.71923
$C_{10} * c$	0	0	0.61334	0.35439

Table 6.2: Minimum and maximum term values in the final load formula

When we remember that the terms are totalled and total value of 1 means the firewall is fully loaded we can see that most terms are significant. The only questionable case is the firewall rule term with constant C_7. Even when including the highest bound from the confidence interval analysis the maximum value using the test set parameters is less than 2%.

In most practical uses of the model with small number of firewall rules (like less than 100) we can thus leave the term with C_7 out of the formula and use some safe constant value (e.g. 0.05) in its place. We keep the term in the model because it is needed for special cases with high number of rules without stateful filtering.

6.6 The Final Model

We are now ready to present the final load formula and its mean constants.

When we take the equation 4.9 and remove the terms for logging and the term for state operations we get:

$$\begin{aligned} L &= C_1 * p + C_2 * p * s + C_3 * p * z + C_5 * c * z + C_6 * q + \\ &\quad C_7 * (c + p_{blocked}) * r + C_9 * (p - p_{blocked} - c) + C_{10} * c \end{aligned} \tag{6.1}$$

We relabel the constants to get rid of the gaps in the numbering. The values of the constants with confidence intervals is thus:

$$\begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_5 \\ C_6 \\ C_7 \\ C_9 \\ C_{10} \end{pmatrix} = \begin{pmatrix} 5.2854 * 10^{-6} \pm 7.7735 * 10^{-7} \\ 2.8559 * 10^{-9} \pm 1.0610 * 10^{-9} \\ 1.1430 * 10^{-7} \pm 7.3478 * 10^{-8} \\ 1.4262 * 10^{-5} \pm 1.6991 * 10^{-6} \\ 1.4804 * 10^{-5} \pm 3.9028 * 10^{-6} \\ 5.6524 * 10^{-8} \pm 3.8110 * 10^{-9} \\ 4.6179 * 10^{-6} \pm 9.2126 * 10^{-7} \\ 1.2267 * 10^{-4} \pm 3.4569 * 10^{-5} \end{pmatrix}$$

The formula thus becomes:

$$\begin{aligned} L &= A * p + B * p * s + C * p * z + D * c * z + E * q + \\ &\quad F * (c + p_{blocked}) * r + G * (p - p_{blocked} - c) + H * c \end{aligned} \tag{6.2}$$

We can simplify formula a little bit more for uses where we can be certain to need all terms. We regroup the multiplications and get:

$$L = p*(A+B*s+C*z+G)+c*(D*z+F*r+H-G)+q*E+p_{blocked}*(r*F-G) \tag{6.3}$$

6.7 Model Reliability

We now have a model for firewall load behavior. But does it actually represent reality?

To perform a reality check we will feed back values from the test sets trough the model. In theory, all formulas should return exactly $L = 1$. Of course this does not happen in practise. To find out a vector of load values by the model from the test sets we do:

$$L' = \mathbf{R} * \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{pmatrix}$$

Vector L' now contains 90 values which all should be quite close to 1. The minimum value in the vector is 0.79406 and the maximum is 1.64833. The model seems to overestimate the load quite much.

However when we take out the highest result which is for a state search test for only one connection (quite unrealistic case anyway) the next highest value is 1.1735. With 89 cases in 90 the model estimates the load from tests within about $\pm 20\%$.

Since the load formula is linear and most of the constants have 95% confidence range of $\pm 20\%$ this is of course to be expected.

6.8 Example of Use

To see how the model could be used in practise we will use it to estimate the following example case.

Before we begin, we simplify model usage a bit. In theory all mathematics done with the model should include the confidence intervals for the constant factors. In practise we do not want to use them in everyday work. On the other hand we want to be sure that the model will not underestimate the load allowing an overload condition. Since none of the test cases caused lower than approximately 0.80 result for L in the model, we make a brute force assumption that 0.80 can be used as the full load limit instead of 1.

Now we analyze the following case:

We have a system with servers providing RTP[33] protocol based voice streams to another network. RTP is run on top of UDP. Each call will last approximately 15s and there will be 10 new calls per second. The calls are initiated with SIP[32] protocol on top of UDP. The voice streams are two way streams with 64kbit/s bandwidth and they are split into packets containing 20ms of voice data. Can one firewall pair handle the load?

The number of firewall rules is minimal, we use the domain low limit $r = 50$.

One RTP frame contains 20ms of speech which at 64kbit/s makes 160 octets per frame plus headers. RTP header size is usually 12 octets. The UDP, IP, and Ethernet headers have total size of 58 octets. One RTP frame is thus about 230 octets. There will be 50 RTP frames for each call in each direction making RTP framerate for one call 100 frames/s.

We assume that there will be about 10 SIP signaling message frames for each call, each taking 1000 octets with headers.

The connection rate is one connection for SIP messages and two for RTP streams for each call per second. RTP streams for both directions are independent and do

not use same ports in both direction so they are essentially different connections. The total connection rate is thus $c = 10 * 3 = 30$ connections/s.

With 15s average call length (l) and 30s UDP state expiration timeout (t) in the firewall we get the number of simultaneous connections from the firewall's point of view by applying equation 4.1:

$$q = c * (l + t) = 30 * (15 + 30) = 1350$$

We find out z with equation 4.2:

$$z = log_2(2 * 2 * q) = log_2(4 * 1350) = 12.399$$

The RTP connection count in servers' perspective is $l * 10 * 2 = 15 * 20 = 300$. Therefore the framerate will be $300 * 50 = 15000$ frames/s. We can ignore the framerate caused by SIP because it is insignificant compared to RTP framerate. For the same reason the average frame size is about 230 octets.

We can now apply the formula 6.2:

$$L = (p \quad p*s \quad p*z \quad c*z \quad q \quad c*r \quad p-c \quad c) \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{pmatrix} = 0.20858$$

The result is below the brute force full load limit of the model (0.80). The system can handle the traffic.

Chapter 7

Conclusions

The main objective of this work was to create a usable load model for a highly available firewall pair.

The model formulation succeeded and we managed to get a sensible and working set of constant parameters for it with the tests. The resulting model is usable for most purposes we need it for.

The applicability domain of the model in its current form is somewhat limited. We model only cases with $50 - 500$ firewall rules, at most 32000 simultaneous connections and no fragmented frames. The logging part of the model is practically missing and we do not know how the system handles high amount of blocked frames.

These limits are problematic for general use of the model and results. On the other hand, the model and especially its constants are valid only for the specific hardware and software used in our tests. The general principle should though be useful for others. For our own use the model covers enough interesting use cases.

We would have liked to run the test sets repeatedly for a number of times to both verify the results and reduce errors in them. Unfortunately we didn't have enough time for this because a full test round takes about two weeks of calendar time.

Since we have used rather conservative settings in the tests the resulting model is somewhat pessimistic. In addition the round-robin circulation of connections in frame sending and the total lack of firewall rule optimizing in the tests yield near worst case results. The actual performance of the system should be higher. On the other hand we were looking after a reliable performance limits for the system. A pessimistic model can be used to get low bounds for performance and make performance guarantees.

7.1 Future Work

Time and other resource constraints forced us to leave many issues out of the original scope of the work.

For a generally useful model the behavior of ICMP and TCP traffic in the firewall should be verified. The load might be quite identical to UDP load, but we can not be sure without further tests.

The normalization parameter behavior should be verified and added into the model. Especially fragmentation can cause quite much extra load which is not modelled or tested in this work.

The logging behavior with a high number of blocked frames should be verified. The logging term could be added back to the model.

Firewalls often do network address translation (NAT) which we also left out of the model. Because of the internal operation of NAT is mostly integrated in state operations in OpenBSD/PF, we assume that NAT load is already included in the model. This assumption should be verified with further testing.

It would be useful if a complete test set with our tests and the missing features would be available for modelling firewall load behavior. This kind of test tool might be either free or a commercial product.

During the work we also found that the firewall rule optimization in OpenBSD/PF is quite primitive. It might be possible to get more performance out of the system by adding a tree or tuple based rule handling into the system. On the other hand we saw that with a small number of firewall rules this is rather insignificant.

We also found that the cleaning of state entries in PF is done periodically with a operation which scales linearly on the number of states. This fact causes unpleasant performance jitters in the system with a high number of states.

The expiration cleaning might be improved with a expiration queue based system. It might be possible to add a separate queue for each expiration timeout value in the configuration. Each state entry would always be in one queue representing its current active timeout value and moved to back of the queue in every timeout counter refresh. Timeout checks could then simply go through every timeout queue and remove states until a state which has not yet been expired is found.

The state removal might also be improved with mass removal tree operations [17]. The effects of these changes should be studied.

Bibliography

[1] IETF: About Cisco's VRRP Patent. http://www.ietf.org/ietf/IPR/VRRP-CISCO, Referenced 22.3.2005.

[2] NLANR/DAST: IPerf - The TCP/UDP Bandwidth Measurement Tool. http://dast.nlanr.net/Projects/Iperf/, Referenced 11.11.2005.

[3] OpenBSD Programmer's Manual: carp - Common Address Redundancy Protocol. http://www.openbsd.org/cgi-bin/man.cgi?query=carp.

[4] OpenBSD Programmer's Manual: pflogd - packet filter logging daemon. http://www.openbsd.org/cgi-bin/man.cgi?query=pflogd.

[5] OpenBSD Programmer's Manual: pfsync - packet filter state table logging interface. http://www.openbsd.org/cgi-bin/man.cgi?query=pfsync.

[6] OpenBSD Security. http://www.openbsd.org/security, Referenced 22.3.2005.

[7] PF: The OpenBSD Packet Filter. http://www.openbsd.org/faq/pf/index.html, Refenced 15.3.2005.

[8] The OpenBSD Project. http://www.openbsd.org/, Referenced 15.2.2005.

[9] Bjerke, J. The New Firewall Design Question. Private Distribution, http://cjlab.altervista.org/security/Firewalls/new_firewall_design.pdf, 2001.

[10] Bradner, S. Benchmarking Terminology for Network Interconnection Devices. Reguest For Comments 1242, IETF, July 1991.

[11] Bradner, S., and McQuaid, J. Benchmarking Methodology for Network Interconnected Devices. Reguest For Comments 2544, IETF, March 1999.

[12] Cheswick, B. The Design of a Secure Internet Gateway. In *Proceedings of the USENIX Annual Technical Conference* (1990), USENIX.

[13] Cormen, T., Leiserson, C., and Rivest, R. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1999, pp. 263–280. ISBN 0-262-53091-0.

[14] Egevang, K., and Francis, P. The IP Network Address Translator (NAT). Reguest For Comments 1631, IETF, May 1994.

[15] Gill, S. Maximizing Firewall Availability. Private Distribution, http://www.qorbit.net/documents/maximizing-firewall-availability.pdf, 2002.

[16] Handley, M., Paxson, V., and Kreibich, C. Network Intrusion Detection: Evasion, Traffic Normalization and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Symposium* (2001), USENIX.

[17] Hanke, S., and Soisalon-Soininen, E. Group updates for red-black trees. In *4th Italian Conference on Algorithms and Complexity (CIAC 2000)* (2000), vol. 1767 of *Lecture Notes in Computer Science*, Springer-Verlag GmbH, pp. 253–262.

[18] Hartmeier, D. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *Proceedings of the USENIX Annual Technical Conference* (2002), USENIX.

[19] Hazelhurst, S., Attar, A., and Sinnappan, R. *Algorithms for Improving the Dependability of Firewall and Filter Rule Lists. In proceedings of the International Conference on Dependable Systems and Networks (DSN 2000).* 2000, p. 576.

[20] Hornig, C. A Standard for the Transmission of IP Datagrams over Ethernet Networks. Reguest For Comments 894, IETF, April 1984.

[21] Kent, C., and Mogul, J. Fragmentation Considered Harmful. In *Proceedings of the SIGCOMM* (1987), ACM.

[22] Lévénez, E. UNIX History. http://www.levenez.com/unix/.

[23] Malan, G., Watson, D., Jahanian, F., and Howell, P. Transport and Application Protocol Scrubbing. In *Proceedings of the INFOCOM* (2000), IEEE.

[24] Milton, J., and Arnold, J. C. *Introduction to Probability and Statistics.* Probability and Statistics Series. McGraw-Hill, 1995, pp. 470–489. ISBN 0-07-113535-9.

[25] Minden, R. Virtual Router Redundancy Protocol (VRRP). Reguest For Comments 3768, IETF, April 2004.

[26] POSTEL, J. User Datagram Protocol. Reguest For Comments 768, IETF, August 1980.

[27] POSTEL, J. Internet Control Message Protocol. Reguest For Comments 792, IETF, September 1981.

[28] POSTEL, J. Internet Protocol. Reguest For Comments 791, IETF, September 1981.

[29] POSTEL, J. Transmission Control Protocol. Reguest For Comments 793, IETF, September 1981.

[30] PRAMANICK, I. IEEE Task Force on Cluster Computing - High Availability, 2005. http://www.ieeetfcc.org/high-availability.html, Referenced 11.10.2005.

[31] RANUM, M. A Network Firewall. In *Proceedings of the World Conference on System Administration and Security* (1992).

[32] ROSENBERG, J. SIP: Session Initiation Protocol. Reguest For Comments 3261, IETF, June 2002.

[33] SCHULZRINNE, H. RTP: A Transport Protocol for Real-Time Applications. Reguest For Comments 3550, IETF, July 2003.

[34] SENNER, L. Anatomy of a Stateful Firewall. SANS Institute Information Security Reading Room, 2001. http://cjlab.altervista.org/security/Firewalls/stateful_firewall.pdf.

[35] SRINIVASAN, V., SURI, S., AND VARGHESE, G. Packet Classification using Tuple Space Search. In *Proceedings of the SIGCOMM 99* (1999), ACM.

[36] STEVENS, R. *TCP/IP Illustrated*, vol. 1: The Protocols. Addison Wesley Longman, 1994. ISBN 0-201-63346-9.

[37] ZWICKY, E., COOPER, S., AND CHAPMAN, D. *Building Internet Firewalls*, second ed. O'Reilly & Associates, 2000. ISBN 1-56592-871-7.

Appendix A

Internet Protocol

Internet Protocol (IP) is a layered communication protocol for unreliable **packet switched** networks. As the name implies, the protocol is designed to be used for inter-network communications, regardless whether the networking technologies are similar. IP is the base protocol, subprotocols are **encapsulated** within IP packets. The main subprotocols are User Datagram Protocol (UDP), Tranmission Control Protocol (TCP) and Internet Control Message Protocol (ICMP). [36]

The currently widely used version of Internet Protocol is 4. This work concentrates on IPv4, but the new version (IPv6) is mostly identical for the parts which are relevant for firewalling load.

IP messages are transported in packets. The packet contains IP header and data. The IP header is illustrated in figure A.1. The most relevant field is the destination address. The IP packets are **routed** based on the destination address. [28]

The IP protocol as such enables networked hosts to send packets to each-other. Application software running on a host is supposed to use some of the subprotocols. Both TCP and UDP have a concept of a **port**. Port is a 16-bit unsigned number. Port number identifies the application software while the IP address identifies the host.

A.1 User Datagram Protocol

UDP is a simple datagram protocol for applications. It provides the user level interface to the packet oriented Internet Protocol. It adds the information on source and destination ports to the messages. In addition, data checksum is contained in the UDP header. The header is shown in figure A.2. [26]

4-bit version	4-bit hdr len	8-bit type of service	16-bit packet total length in bytes	
16-bit identification			3-bit flags	13-bit fragment offset
8-bit time to live		8-bit protocol	16-bit header checksum	
32-bit source IP address				
32-bit destination IP address				
Options, if any				

Figure A.1: IPv4 Header

16-bit source port number	16-bit destination port number
16-bit UDP length	16-bit UDP checksum

Figure A.2: UDP Header

A.2 Transmission Control Protocol

TCP[29] is a reliable connection oriented stream protocol. The protocol is far too complex to present here, but two reference figures are available.

The TCP header is shown in figure A.3. TCP headers are encapsulated within IP packets.

The state transion diagram (partial figure modelled after a figure in [36]) is shown in figure A.4. The thick solid lines represent state transitions for the client and the thick dashed lines transtions by the server. Thin solid lines show transitions which can be made by either end.

The figure does not show frames sent while transfering data. Data can be included

<table>
<tr><td colspan="3">16-bit
source port number</td><td>16-bit
destination port number</td></tr>
<tr><td colspan="4">32-bit
sequence number</td></tr>
<tr><td colspan="4">32-bit
acknowledgement number</td></tr>
<tr><td>4-bit
hrd len</td><td>6-bit
reserved</td><td>6-bit
flags</td><td>16-bit
window size</td></tr>
<tr><td colspan="3">16-bit
TCP checksum</td><td>16-bit
urgent pointer</td></tr>
<tr><td colspan="4">Options, if any</td></tr>
</table>

Figure A.3: TCP Header

into most of the frames indicated to be sent or received. The flags set in the TCP header to identify state transitions to the other end of the connection are shown as Sent: *FLAGNAME*. The flags which trigger transition are shown as Recv: *FLAGNAME*.

TCP flag bits are URG (Urgent), ACK (Acknowledge) , PSH (Push), RST (Reset), SYN (Synchronize sequence numbers), FIN (Finish). The SYN flag is used to initiate a new connection. ACK is used to acknowledge received frames. FIN is used to signal that the data stream from the host has ended. RST immediately closes the connection. It is primarily used to reject connections, normal closing is done with mutual FINs.

The URG and PSH flags are used to categorize the carried data. For more information on TCP, please see for example [36].

A.3 Internet Control Message Protocol

Internet Control Message Protocol[27] (ICMP) is a IP subprotocol used for error signaling and diagnostics. Most ICMP messages are reactive. They are sent for example when a router can not find a route to IP packet destination or when a frame has been rejected by a host. ICMP is not directly usable by applications.

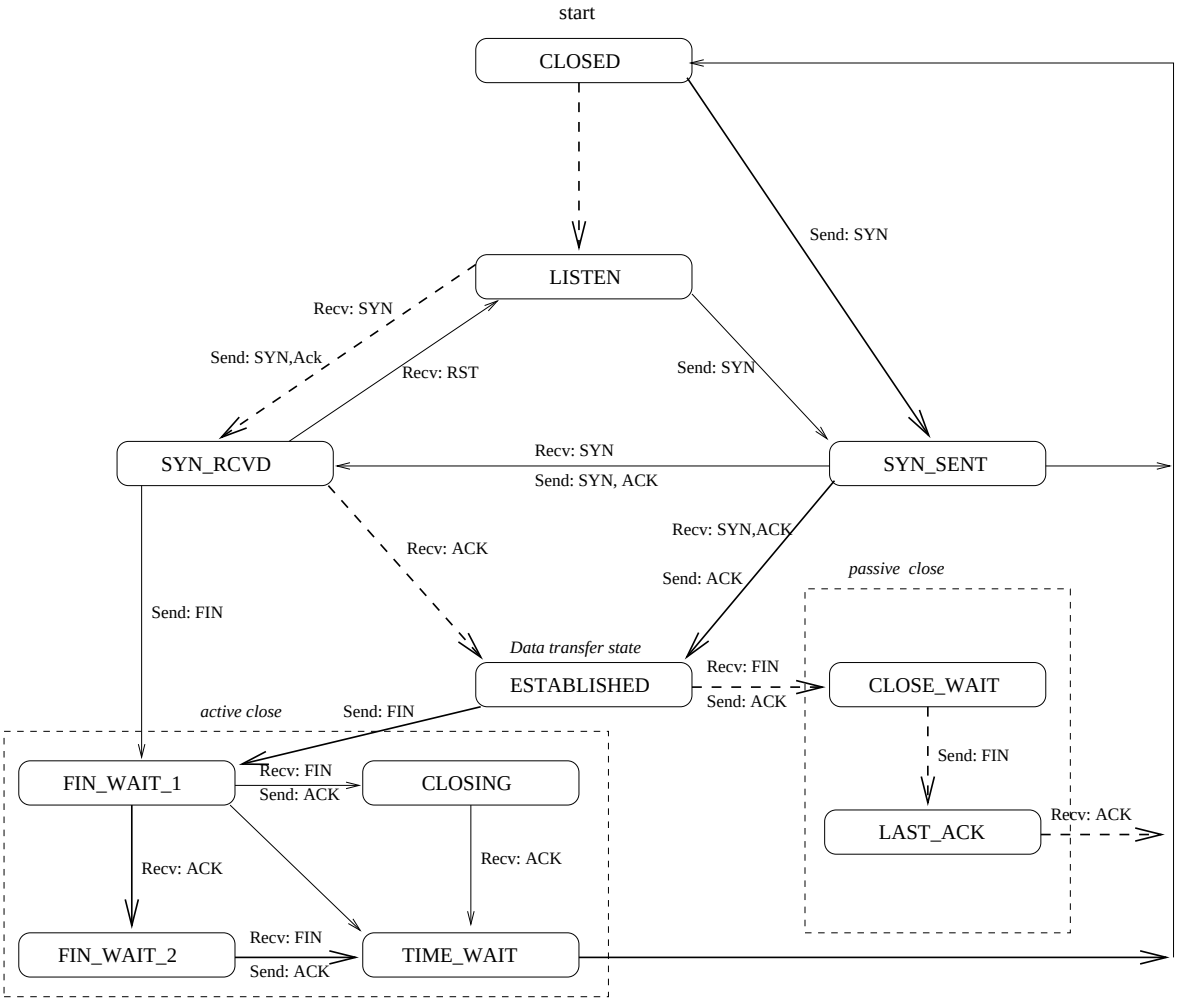

Figure A.4: TCP State Diagram

Appendix B

Test Results

B.1 Routing, Normalization and Rules Tests

Results of the basic firewalling tests are available in table B.1. The result number is the maximum framerate which could be used to feed frames through the system and getting at most 0.005 frame loss ratio. Variables r (number of firewall rules) and s (frame size) represent test settings.

B.2 State Table Tests

State creation and deletion test results are shown in table B.2. The result value is the maximum connection rate (c) with at most 0.005 frame lost ratio. The timeout (t) shows the configured timeout value in the firewall for each test.

State table search test results are available in table B.3. The connection count column shows the configured number of simultaneous connections (q) during the test. The state tree depth column shows corresponding state tree depth (z) based on equation 4.2 (this is a derived value, not actual measurement or setting). The framerate column shows the maximum framerate (p) with at most 0.005 frame loss ratio.

B.3 Logging Tests

Logging test results are shown in table B.4. The result value is the maximum connection rate (c) with at most 0.005 frame lost ratio. The timeout (t) shows the configured timeout value in the firewall for each test.

Variable	$s = 108$	$s = 158$	$s = 358$	$s = 558$	$s = 758$	$s = 958$	$s = 1158$	$s = 1458$
$r = 50$	109460	104748	98942	95396	95422	87279	75557	85436
$r = 100$	85336	80989	78898	73813	75472	72475	71737	70504
$r = 200$	58520	57060	55769	52847	54639	47983	51490	51527
$r = 300$	44981	43941	43307	42219	42292	41303	41506	40627
$r = 400$	36934	36313	35850	31966	34614	31957	30906	31910
$r = 500$	30977	29182	29129	28484	28947	26297	27494	26262
$r = 600$	26052	25786	25919	24039	20292	21633	20811	22077
$r = 700$	21232	21786	18377	16753	21206	16848	18173	17598
$r = 800$	14647	19129	17240	16254	15698	16073	14636	15257
$r = 900$	11683	15553	12929	11790	14027	11769	13608	11365
$r = 1000$	9233	9764	12517	9599	10714	10599	9857	10819

Table B.1: Throughput with varying frame size (s) and number of firewall rules (r)

Timeout (t)	**Rate with sync (c)**	**Rate without sync (c)**
1	2889	5134
2	2685	3908
3	2558	3545
4	2432	3306
5	2316	3107
6	2248	2944
7	2181	2778
8	2120	2653
9	2050	2539
10	1962	2451
20	1601	1881
30	1365	N/A

Table B.2: New connection creation rate

Connections (q)	**Tree Depth (z)**	**Rate with sync**	**Rate without sync**
1	2.000	155748	155105
9	5.170	74080	154930
25	6.644	73305	150143
100	8.644	71968	156275
484	10.919	69292	159436
992	11.954	66545	152548
1980	12.951	64618	137350
3969	13.955	62186	133687
7921	14.951	61216	128304
15876	15.955	57372	107930
31862	16.960	50554	78936
63756	17.960	4881	5693
127806	18.964	880	1076
255530	19.963	457	198

Table B.3: Framerate with varying number of simultaneous connections

Variable	$t = 1$	$t = 5$	$t = 10$	$t = 20$
c	2917	2339	1989	1602

Table B.4: New connection creation rate with logging

Appendix C

Parameter Matrices

Table C.1 contains the **R** parameter matrix for the first phase of regression analysis (before dropping out $C_4 * c$) in chapter 6.

Table C.2 contains the **R** parameter matrix for the second phase of regression analysis (after dropping out $C_4 * c$) in chapter 6.

p	ps	pz	c	cz	q	cr	$p-c$	c
109460	11821680	0	0	0	0	5473000	0	0
75557	87495006	0	0	0	0	3777850	0	0
85436	124565688	0	0	0	0	4271800	0	0
104748	16550184	0	0	0	0	5237400	0	0
98942	35421236	0	0	0	0	4947100	0	0
95396	53230968	0	0	0	0	4769800	0	0
95422	72329876	0	0	0	0	4771100	0	0
87279	83613282	0	0	0	0	4363950	0	0
85336	9216288	0	0	0	0	8533600	0	0
71737	83071446	0	0	0	0	7173700	0	0
70504	102794832	0	0	0	0	7050400	0	0
80989	12796262	0	0	0	0	8098900	0	0
78898	28245484	0	0	0	0	7889800	0	0
73813	41187654	0	0	0	0	7381300	0	0
75472	57207776	0	0	0	0	7547200	0	0
72475	69431050	0	0	0	0	7247500	0	0
58520	6320160	0	0	0	0	11704000	0	0
51490	59625420	0	0	0	0	10298000	0	0
51527	75126366	0	0	0	0	10305400	0	0
57060	9015480	0	0	0	0	11412000	0	0
55769	19965302	0	0	0	0	11153800	0	0
52847	29488626	0	0	0	0	10569400	0	0
54639	41416362	0	0	0	0	10927800	0	0
47983	45967714	0	0	0	0	9596600	0	0
44981	4857948	0	0	0	0	13494300	0	0
41506	48063948	0	0	0	0	12451800	0	0
40627	59234166	0	0	0	0	12188100	0	0
43941	6942678	0	0	0	0	13182300	0	0
43307	15503906	0	0	0	0	12992100	0	0
42219	23558202	0	0	0	0	12665700	0	0
42292	32057336	0	0	0	0	12687600	0	0
41303	39568274	0	0	0	0	12390900	0	0
36934	3988872	0	0	0	0	14773600	0	0
30906	35789148	0	0	0	0	12362400	0	0
31910	46524780	0	0	0	0	12764000	0	0
36313	5737454	0	0	0	0	14525200	0	0
35850	12834300	0	0	0	0	14340000	0	0
31966	17837028	0	0	0	0	12786400	0	0
34614	26237412	0	0	0	0	13845600	0	0
31957	30614806	0	0	0	0	12782800	0	0
30977	3345516	0	0	0	0	15488500	0	0
27494	31838052	0	0	0	0	13747000	0	0
26262	38289996	0	0	0	0	13131000	0	0
29182	4610756	0	0	0	0	14591000	0	0
29129	10428182	0	0	0	0	14564500	0	0
28484	15894072	0	0	0	0	14242000	0	0
28947	21941826	0	0	0	0	14473500	0	0
26297	25192526	0	0	0	0	13148500	0	0
5134	811172	73549	5134	73549	5134	256700	0	0
3908	617464	58355	3908	58355	7816	195400	0	0
3545	560110	54509	3545	54509	10635	177250	0	0
3306	522348	51874	3306	51874	13224	165300	0	0
3107	490906	49473	3107	49473	15535	155350	0	0
2944	465152	47423	2944	47423	17664	147200	0	0
2778	438924	45134	2778	45134	19446	138900	0	0
2653	419174	43438	2653	43438	21224	132650	0	0
2539	401162	41842	2539	41842	22851	126950	0	0
2451	387258	40640	2451	40640	24510	122550	0	0
2889	456462	38990	2889	38990	2889	144450	0	2889
2685	424230	38639	2685	38639	5370	134250	0	2685
2558	404164	38128	2558	38128	7674	127900	0	2558
2432	384256	37082	2432	37082	9728	121600	0	2432
2316	365928	35896	2316	35896	11580	115800	0	2316
2248	355184	35337	2248	35337	13488	112400	0	2248
2181	344598	34673	2181	34673	15267	109050	0	2181
2120	334960	34025	2120	34025	16960	106000	0	2120
2050	323900	33151	2050	33151	18450	102500	0	2050
1962	309996	31902	1962	31902	19620	98100	0	1962
155105	24506590	310210	0	0	1	0	0	0
156275	24691450	1350818	0	0	100	0	0	0
107930	17052940	1721975	0	0	15876	0	0	0
137350	21701300	1778858	0	0	1980	0	0	0
150143	23722594	997528	0	0	25	0	0	0
78936	12471888	1338718	0	0	31862	0	0	0
133687	21122546	1865543	0	0	3969	0	0	0
159436	25190888	1740859	0	0	484	0	0	0
128304	20272032	1918333	0	0	7921	0	0	0
154930	24478940	800976	0	0	9	0	0	0
152548	24102584	1823588	0	0	992	0	0	0
155748	24608184	311496	0	0	1	0	155748	0
71968	11370944	622081	0	0	100	0	71968	0
57372	9064776	915345	0	0	15876	0	57372	0
64618	10209644	836886	0	0	1980	0	64618	0
73305	11582190	487027	0	0	25	0	73305	0
50554	7987532	857373	0	0	31862	0	50554	0
62186	9825388	867778	0	0	3969	0	62186	0
69292	10948136	756589	0	0	484	0	69292	0
61216	9672128	915268	0	0	7921	0	61216	0
74080	11704640	382988	0	0	9	0	74080	0
66545	10514110	795491	0	0	992	0	66545	0

Table C.1: Parameter Matrix for 90 tests and 9 variables

p	ps	pz	cz	q	cr	$p-c$	c
109460	11821680	0	0	0	5473000	0	0
75557	87495006	0	0	0	3777850	0	0
85436	124565688	0	0	0	4271800	0	0
104748	16550184	0	0	0	5237400	0	0
98942	35421236	0	0	0	4947100	0	0
95396	53230968	0	0	0	4769800	0	0
95422	72329876	0	0	0	4771100	0	0
87279	83613282	0	0	0	4363950	0	0
85336	9216288	0	0	0	8533600	0	0
71737	83071446	0	0	0	7173700	0	0
70504	102794832	0	0	0	7050400	0	0
80989	12796262	0	0	0	8098900	0	0
78898	28245484	0	0	0	7889800	0	0
73813	41187654	0	0	0	7381300	0	0
75472	57207776	0	0	0	7547200	0	0
72475	69431050	0	0	0	7247500	0	0
58520	6320160	0	0	0	11704000	0	0
51490	59625420	0	0	0	10298000	0	0
51527	75126366	0	0	0	10305400	0	0
57060	9015480	0	0	0	11412000	0	0
55769	19965302	0	0	0	11153800	0	0
52847	29488626	0	0	0	10569400	0	0
54639	41416362	0	0	0	10927800	0	0
47983	45967714	0	0	0	9596600	0	0
44981	4857948	0	0	0	13494300	0	0
41506	48063948	0	0	0	12451800	0	0
40627	59234166	0	0	0	12188100	0	0
43941	6942678	0	0	0	13182300	0	0
43307	15503906	0	0	0	12992100	0	0
42219	23558202	0	0	0	12665700	0	0
42292	32057336	0	0	0	12687600	0	0
41303	39568274	0	0	0	12390900	0	0
36934	3988872	0	0	0	14773600	0	0
30906	35789148	0	0	0	12362400	0	0
31910	46524780	0	0	0	12764000	0	0
36313	5737454	0	0	0	14525200	0	0
35850	12834300	0	0	0	14340000	0	0
31966	17837028	0	0	0	12786400	0	0
34614	26237412	0	0	0	13845600	0	0
31957	30614806	0	0	0	12782800	0	0
30977	3345516	0	0	0	15488500	0	0
27494	31838052	0	0	0	13747000	0	0
26262	38289996	0	0	0	13131000	0	0
29182	4610756	0	0	0	14591000	0	0
29129	10428182	0	0	0	14564500	0	0
28484	15894072	0	0	0	14242000	0	0
28947	21941826	0	0	0	14473500	0	0
26297	25192526	0	0	0	13148500	0	0
5134	811172	73549	73549	5134	256700	0	0
3908	617464	58355	58355	7816	195400	0	0
3545	560110	54509	54509	10635	177250	0	0
3306	522348	51874	51874	13224	165300	0	0
3107	490906	49473	49473	15535	155350	0	0
2944	465152	47423	47423	17664	147200	0	0
2778	438924	45134	45134	19446	138900	0	0
2653	419174	43438	43438	21224	132650	0	0
2539	401162	41842	41842	22851	126950	0	0
2451	387258	40640	40640	24510	122550	0	0
2889	456462	38990	38990	2889	144450	0	2889
2685	424230	38639	38639	5370	134250	0	2685
2558	404164	38128	38128	7674	127900	0	2558
2432	384256	37082	37082	9728	121600	0	2432
2316	365928	35896	35896	11580	115800	0	2316
2248	355184	35337	35337	13488	112400	0	2248
2181	344598	34673	34673	15267	109050	0	2181
2120	334960	34025	34025	16960	106000	0	2120
2050	323900	33151	33151	18450	102500	0	2050
1962	309996	31902	31902	19620	98100	0	1962
155105	24506590	310210	0	1	0	0	0
156275	24691450	1350818	0	100	0	0	0
107930	17052940	1721975	0	15876	0	0	0
137350	21701300	1778858	0	1980	0	0	0
150143	23722594	997528	0	25	0	0	0
78936	12471888	1338718	0	31862	0	0	0
133687	21122546	1865543	0	3969	0	0	0
159436	25190888	1740859	0	484	0	0	0
128304	20272032	1918333	0	7921	0	0	0
154930	24478940	800976	0	9	0	0	0
152548	24102584	1823588	0	992	0	0	0
155748	24608184	311496	0	1	0	155748	0
71968	11370944	622081	0	100	0	71968	0
57372	9064776	915345	0	15876	0	57372	0
64618	10209644	836886	0	1980	0	64618	0
73305	11582190	487027	0	25	0	73305	0
50554	7987532	857373	0	31862	0	50554	0
62186	9825388	867778	0	3969	0	62186	0
69292	10948136	756589	0	484	0	69292	0
61216	9672128	915268	0	7921	0	61216	0
74080	11704640	382988	0	9	0	74080	0
66545	10514110	795491	0	992	0	66545	0

Table C.2: Parameter Matrix for 90 tests and 8 variables