UNIVERSITY OF OSLO
Department of Informatics

Generating Synthetic VoIP Traffic for Analyzing Redundant OpenBSD-Firewalls

Master Thesis

Maurice David Woernhard

May 23, 2006

Generating Synthetic VoIP Traffic for Analyzing Redundant OpenBSD-Firewalls

Maurice David Woernhard

May 23, 2006

Abstract

Voice over IP, short VoIP, is among the fastest growing broadband technologies in the private and commercial sector. Compared to the Plain Old Telephone System (POTS), Internet telephony has *reduced availability,* measured in uptime guarantees per a given time period. This thesis makes a contribution towards proper *quantitative statements* about *network availability* when using two redundant, state synchronized computers, acting as firewalls between the Internet (WAN) and the local area network (LAN).

First, methods for generating adequate VoIP traffic volumes for loading a Gigabit Ethernet link are examined, with the goal of using a minimal set of hardware, namely one regular desktop computer. *pktgen*, the Linux kernel UDP packet generator, was chosen for generating synthetic/artificial traffic, reflecting the common VoIP packet characteristics *packet size, changing sender* and *receiver address,* as well as typical *UDP-port* usage. pktgen's three main parameters influencing the *generation rate* are *fixed inter-packet delay, packet size* and *total packet count*. It was sought to relate these to more user-friendly values of *amount of simultaneous calls, voice codec employed* and *call duration*. The proposed method fails to model VoIP traffic accurately, mostly due to the currently unstable nature of pktgen. However, it is suited for generating enough packets for testing the firewalls.

Second, the traffic forwarding limit and failover behavior of the redundant, state-synchronized firewalls was examined. The firewalls were running OpenBSD 3.8 and used the Common Address Redundancy Protocol (CARP) and the packet filter state synchronization protocol (pfsync) for achieving redundancy, with one acting as *master*, and the other as *backup*. Empirical measurements show that the *upper limit for unidirectional traffic* is at about 125,000 packets per second, independent of packet sizes typical for VoIP media packets (less than 220 bytes). This is far below the traffic capacity of Gigabit Ethernet, and is caused by a "receive livelock": full system load due to non-optimized interrupt handling. The obtained measurements allow for questioning the suitability of a default OpenBSD installation for *firewalls in high packet rate networks.*. The network connectivity glitch in *failover situations* was measured at: when *turning CARP off administratively* while processing circa 80,000 packets per second, the maximum glitch was in the magnitude of 300 milliseconds.

When *power-cycling* the master firewall, maximum connectivity interruptions of circa 3,000 milliseconds occurred. In all cases, series with much lower values were measured, but may not be representative.

Keywords: Voice over IP, VoIP, pktgen, artificial, synthetic, traffic, OpenBSD, redundant, firewall, high-availability, CARP, pfsync.

Acknowledgments

> As we enjoy great advantages from inventions of others, we should be glad of an opportunity to serve others by any invention of ours; and this we should do freely and generously. – *Benjamin Franklin*

In the spirit of this dictum, I would like to express thanks to all who have joyfully contributed to this project with their own "inventions" – be it ideas, technical expertise or feedback, as well as by bearing over with me during this interesting albeit intense 17 weeks.

From academia, my supervisor Dr. Hårek Haugerud deserves thanks for both professional and practical guidance; Professor Mark Burgess for being the *spritus rector* of scientific system administration at Oslo University College and passing on a scholarly spirit to his students. The project idea originated with doctoral candidate Kyrre Begnum, leading to FreeCode providing the firewalls, a short technical introduction and office space during the project. Working in the FreeCode atmosphere was very pleasant, thanks to all the outstanding employees.

Last, I would like to mention Senior Engineer Tore Øfsdahl from Oslo University College, for always being ready to help when the need for replacement hardware arose.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Voice over IP, short VoIP, is among the fastest-growing broadband technologies. According one specialist, VoIP "has moved to a level of reliability and capability such that mainstream users are adopting it at a rapidly increasing pace."[1] This is also seen in the Norwegian Internet telephony marked, with an increasingly varied spectrum of VoIP providers. Forerunners like Telio that have been active for several years are facing more competition from the broadband providers that start offering this service themselves, among them Telenor, Nextgentel or Bluecom/Ventelo.

Motivation for VoIP adaptation or "the move from POTS [Plain Old Telephone System] to PANS [Promise of Internet-based pretty amazing new services]" [2], is multifarious[3, 4] and depends on the target segment, e.g. private or corporate. Some motivating key factors can be identified

- reduced operating cost
- more efficient use of resources
- possibility of value-added services
- progressive deployment (instead of a one-time technology migration)

There exist important inhibiting factors. Of special interest for this thesis are *availability* and *security*, two key elements for a professional deployment of VoIP. Traditional telecommunication companies have a long history and therefore much experience with managing downtime; the VoIP community is still working hard for achieving uptime guarantees that are somewhat comparable. The same can be said about security – since traditional phone networks were often owned by the state, very few people or companies had direct access to it, limiting the exposure. This is no longer true with the Internet's global infrastructure, so adequate means have to be used in order to achieve an acceptable level of protection. While it may be up to a private consumer's preference how

much attention these factors should receive, they are of pivotal importance for a company. A redundant architecture for securing VoIP traffic is needed.

This thesis focuses on two aspects: first, evaluating possibilities for generating VoIP traffic, and presenting a simple method for creating synthetic/artificial traffic with Linux' `pktgen` kernel module, with the packets reflecting typical VoIP characteristics; second, the use of two redundant state-synchronized firewalls running OpenBSD for internetwork-connectivity. Answers were sought for the following questions:

- How can VoIP-traffic be generated with a moderate set of commodity hardware? How much traffic can be generated? Which properties must synthetic/artificial VoIP packets have in common with real ones?
- What is the measurable traffic forwarding limit of one firewall? What are the limiting factors?
- What can be said about packet loss and it's influence on ongoing "calls" in failover situations, either administratively (soft-failover), or by power-cycling (hard-failover)?

This document is structured in the following way:

Chapter 2 enlightens the reader with historical and technical background information on Voice over IP, and mentions previous work done on the topic.

Chapter 3 explains the available hardware, the architecture of the testnetwork and issues related to traffic generation, capture and forwarding. Also the simple model for relating pktgen's three main parameters (fixed inter packet delay, packet size, total packet count) to more user-friendly values (amount of simultaneous calls, voice codec employed, call duration) is presented..

Chapter 4 contains the description as well as the obtained data of the experiments about generating artificial VoIP traffic with pktgen, and examining the forwarding capacity and failover behavior of the firewalls. The *black-box* and *inside-the-firewall* perspectives are presented and used for data interpretation.

Chapter 5 concludes with a discussion of results, and possible future research.

Chapter 2

Technical VoIP background

In order to give the reader a more complete view of the rather vast area of VoIP, a few pages of background information are provided. The material includes parts of a scientific literature survey,[5] written by the author himself in spring 2005. New sources have been added for reflecting recent research.

2.1 A paradigm change

The history of VoIP can roughly be divided into three stages: *technology discovery (1970s), pre-commercial (1980-1995), PC-centric (1995-1998) and carrier grade (1998 onwards).* The first and second stage are characterized by research activities, and the lack of standards. During the third, PC-centric period, VoIP underwent the change from an almost exclusively academic domain to end-user targeting technology, but with many deficiencies (half-duplex, no program compatibility etc.) and proprietary solutions; the question of of *call signaling* (providing a "virtual dial- and ringtone") was unsolved. A good end-user experience was still miles away.

Only after 1996 did inter-operability take shape, with the ITU's H.323 protocol suite, followed by Reston, VA's Internet Society publishing the "Session Initiation Protocol (SIP)" (approved proposed IETF standard in March 1999). Inter-connectivity between networks (both the PSTN[1] and other VoIP-networks) began to be heavily researched and tested. [3, 6]

Today, the VoIP telephony services fall into two basic categories: carrier and "free"; general differentiation is possible by a simple rule of thumb, namely: if a user gets special networking hardware, it is probably a carrier solution.

The approach of using packet-switched networks like intranets or the Internet differs greatly from the traditional circuit-switched telephone networks.

[1] Public Switched Telephone Network

[7, 4] With traditional telephony, "voice as an analog signal [is sent] through a system of wires and cables connected to incredibly smart central computers, called switches. At those switches, the voice signals are digitized and routed to other switches, which then ultimately route them to quite stupid devices, old-fashioned analog telephones." [7] VoIP does the exact opposite: voice is turned into data packets by smart devices and then sent through a relatively dumb network - the Internet. The receivers are also smart devices: computers, PDAs, IP phones. Therefore, the paradigms are diametrically opposed; an "qualitative comparison" of differences is given in ACM's VoIP-paper.[3, table 1, page 90]

Cable telephone companies take pride in being able to deliver a high availability, high quality service. Availability is often measured in "9's": "five 9's" means 99.999 availability, or "three 9's" stands for 99.9, signifying 5 minutes, or 8 hours, respectively, downtime per year - "six nines" (99.9999) correspond to 31 seconds a year! Serious conventional carriers strive for "six nines"[8]; most VoIP companies, big or small, do not dare to promise even three nines, since VoIP is at the mercy of the weakest link in a possibly very long chain of dependencies. Technically especially challenging is the conversation quality - a traditional phone call sounds so good because a devoted full-duplex 64 kilobit channel is allocated for each call.[2] With VoIP, the available bandwidth may be more than 64 kb/s (with roughly 14 Kbps needed), but congestion can arise anytime and bring the data rate temporarily down to almost zero. Such *hiccups* are unavoidable and exist even in conventional telephone networks.

As pointed out by several pivotal technology overviews [9, 3, 10], VoIP is technically complex and involves several difficult engineering decisions, namely the choice and deployment of speech codec, packetization strategy, efficient data transport and dealing with transport difficulties (delay/latency, jitter and packet loss), but also the choice of the call setup and signaling protocol.

2.2 Call signaling

Call signaling includes many functionalities essential to VoIP: establishing calls, providing call control (manage different types of media transmitted at the same time), call termination, user registration (authentication), locating users (directory services), feature invocation (transferring, conferencing, hold, message waiting) and interoperability between different architectures.[3, 4, 11]

The two most widespread signaling protocols are H.323 and SIP[12, 11],

[2]The expression "switched network" for the traditional telephone network may be misleading today, but historically speaking the telephone operators flipped physical switches to open dedicated electrical circuits between two phones.

with IAX2 (Asterisk's native protocol) gaining much popularity. A short overview follows.

ITU-T H.323 is the foundation protocol suite for audio/video over IP based networks, and moved the industry away from the initial proprietary solutions during the mid-1990's. A default H.323 network consists of 4 basic entities. The *terminal* is the end-user device, also called H.323 client. It provides real-time two-way media communication with another H.323 client. The *gatekeeper* is responsible for address translation, bandwidth management and call control services, while the *gateway* provides inter-network connectivity, both to the PSTN or to other networks (like ISDN, ATM). A *Multipoint Control Unit (MCU)* supports multi-conferencing between several terminals or gateways.

H.323 relies on many other protocols. "H.323 uses a number of protocols for call control and singnaling: H.225.0 Call Signaling Messages, which is based on Q.931, for call setup; H.245 for exchanging terminal capabilities and creation of media channels; RAS for registration and admission control; RTP/RTCP for sequencing audio and video packets; G.711/712 for codec specification. T.120 may also be used for data conferencing althought it is not an integral part of the protocol."[11]

The call signaling processes work like this: "First ... an H.323 terminal registers with a H.323 gatekeeper using an *registration request message*. After receiving a *registration confirm message* from the H.323 gatekeeper, the H.323 terminal queries a H.323 gatekeeper for the address of another terminal using an *admission request message*. The terminal then establishes a session with the other terminal using H.225.0 Call Signaling Messages setup message, possibly routed via the H.323 gatekeeper. The other terminal obtains admission from H.323 gatekeeper using an admission request message. Once the session is established, the two terminals will negotiate the available features of each terminal using .245 as specified in the H.323 document. Finally, the two terminals can exchange media data with the RTP/RTCP channels that were created during negotiation." In short and less technical language, the steps are *registration, confirmation, admission request, session establishing, feature negotiation, media exchange, teardown.*
For the taste of many [13, 14, 12, 6, 11], H.323 is too complex to encourage migration to VoIP. In 1999, a new proposed standard was approved:

Session Initiation Protocol (SIP), which was developed by the IETF, and has two components: User Agents and SIP servers (including SIP proxies, SIP registrars, and SIP redirect servers). A *user agent* is a logical entity that acts as both a client and a server. A user agent client *initiates* a SIP transaction with a request. A user agent server *responds* to a SIP request by accepting, rejecting or redirecting the request. A *SIP server* is a server that accepts requests and

sends responses back to those requests. [11] SIP is a request-response protocol that closely resembles HTTP. A SIP request and the appropriate response are grouped into a SIP transactions, as defined in RFC 3261 (INVITE, ACK, OPTIONS, BYE, CANCEL, REGISTER) and RFC 3311 (UPDATE). (This rather high-level description was inspired by Ahuja/Ensor [15, box 'What is SIP?', page 52].)

SIP is gaining momentum and is not only used in VoIP applications, but also in instant messaging programs and game consoles. Big efforts are made to ensure the interworking between these protocols[3]; since both H.323 and SIP use RTP for transferring the data, no media translation needs to be done (as long as the same codec is chosen).

IAX(2) is a rather new addition.[16] It is a binary protocol and uses therefore the bandwidth efficient for voice, yet may not be as efficient for other media stream types (like video). It uses a single well-known port (4569) and sends both the signaling and media packets in the same channel. This has the big advantage of being Network Address Translation (NAT) friendly, and causing only few firewall problems.

In IAX2 lingo, packets are called "frames". A *full frame* has a 12-byte-header and is mostly related to connection control (NEW – ACCEPT – RINGING – ANSWER – HANGUP packets). Full frames require a receiver confirmation; every frame contain a 15-bit "call number" that allows an end station to multiplex connections; it also contains a 32-bit timestamp that expresses how many milliseconds have elapsed since the conversation started.[4] *Mini-frames* have a shorter header (4 bytes), and contain only a the lower 16 bits of the conversation timestamp. When this counter wraps, a full frame is used to synchronize the short timestamp.

Connections can be *trunked*, meaning that packets belonging to multiple connections are sent in one *meta trunk frame* with 8 header bytes and 4 bytes header per call/packet. Again, this makes the protocol even more efficient, since the input-output overhead is very notable for small voice packets.

Many other protocols have been suggested, among those Megaco (ITU H.248), IETF's Media Gateway Control Protocol (MGCP), and Distributed Open Signaling Architecture[14, 9]. They are not discussed further in this thesis.

[3] see the IETF's draft-agrawal-sip-h323-interworking-reqs-07.txt

[4] `http://unleashnetworks.com/articles/asterisk-call-analyzer-for-iax2.html`

2.3 Data Transport

Packet transport behavior is highly complex and dynamic. [17, Understanding Internet Traffic Dynamics] explains how by generating tightly controlled test traffic streams, a detailed analysis of delay and loss patterns can be made. The method succeeds in providing high resolution packet departure timestamps and excludes timing errors attributed to complex variations in clock rates. In [18], the authors present the results of RTP/RTCP measurements, and come to the conclusion that the Internet is capable of carrying voice with acceptable delay and quality. Some concern remained about possible difficulties regarding asymmetric paths. Kampicher and Goeschka [19] introduce a performance measuring method (and tool) for assessing the "VoIP-readiness" of a LAN by generating and observing imitated VoIP-traffic. The proposed procedure consists in sending sequences of UDP-packets to uncommon high destination numbers (higher than 30,000), assuming an ICMP `port unreachable` answer will be generated; they further assume the ICMP handling runs at a high priority on the host. The port numbers are implicitly used as sequence number, enabling the detection of lost packages. The packet round-trip delay can be calculated, based on a pair of sent/received timestamps A sophisticated model aims at ensuring that errors and uncertainties are kept within specified boundaries. They recognize the further potential of this method and develop a client-server application, eliminating the dependency on ICMP.

For transporting voice data in IP networks, the User Datagram Protocol (UPD) is used. On top of UDP, the Real Time Protocol (RTP) provides packet sequence information so endpoints can determine arrival order. RTCP (Real Time Control Protocol), RTP's companion, sends feedback about the quality of the stream. RTSP (Real Time Streaming Protocol) is used for streaming prerecorded data, a possible application in VoIP is a user listening to his voice messages or sending prerecorded conference calls. RTSP is the only protocol that can use a large buffer since this is a one-way stream; bi-directional communications are not buffered, unless for balancing out jitter (using a small "dejitter buffer"[20]).

Time- and sequence-critical data like audio or video depend to a larger degree than other data streams on a minimal service quality for the end-user to be satisfied. *Quality of Service* stands for the effort to ensure defined quality levels; the term is the 'sum' of factors like availability of the network, throughput (effective data transfer) and packet loss (congestion) rate, latency (total time from source to destination) and jitter (variation of time between arriving packets). [7, 8, 21, 22] Some delays are hard to predict, like processing or re-ordering packets the packets at a router, since they depend on the vendor-specific implementation and traffic at a given moment. Below follows a tabular overview of the VoIP delay factors, taken from [3, table 2, page 91]:

Transport problems can be tackled differently, since possible sources vary.[7,

Cause of Delay	Length of Delay
processing at a switch/router	5-10msec per packet
time to put packets online	packet size divided by line speed
propagation delay	proportional to segment length
jitter (reordering, buffering)	variable
speech encoding	5-10msec

Table 2.1: Traffic Delay Factors Overview

19, 3] Points of application include making routing decisions based on overlay network info like in MPLS (Multi Protocol Layer Switching), where a "packet label" is the key to faster routing decisions. Other approaches are Integrated Service (IntServ), where the Resource Reservation Protocol (RSVP) tries to ensure before call setup that all devices along the network path have the needed resources; or DiffServ (Differentiated Service), where each packet gets tagged as belonging to a certain service class. In pure IPv6 networks, traffic prioritization is supported natively. A carrier may also use overprovisioning, a technique where the available bandwidth is bigger than the required one in order to have a safety margin for congestions. The concept of having VoIP systems test different route(r)s before sending the packets does not always yield good results, since the weak link in the chain can exists at a later stage.

An interesting question raised was wether if extensive QoS-provisions are needed at all, since end-to-end delays overseas where within the 150 ms range; [23] or for quoting Goode [9]: "Essentially, the debate is over when excess network capacity [...] is less expensive than QoS implementation."

Security Since *security* is one of the decisive factors for new technologies in enterprise environments (and also climbs the charts of private users), it has to measure up to at least the current PSTN standard.

Traditional security aspects are confidentiality, integrity, authentication, authorization and availability.[24] "VoIP is not easy to secure", state Sicker and Lookabough, referring to the combination of PSTN-interconnection and complex networking functions. "Privacy and confidentiality are aided by the difficulty in physically accessing wires in order to tap them"; a better solution would be encryption, which is feasible and deployed for the signaling channel. H.323 offers specific hooks for each of these security features, and SIP uses IPsec and SSL/TLS for (partially) securing the signaling channel (not the whole request can be encrypted, since some fields need to be visible to proxies).[9] It is to remember that using encryption enhances security, but creates additional traffic overhead, as well as requiring more computing power for en- and decrypting. Therefore, the voice data itself is not encrypted, unless

IETF's SRTP (secure RTP) is used.

Firewalls are an essential part of a network defense system, being often the primary traffic security access checkpoint. Firewall policies have the tendency to be rather strict (and as a consequence, static). It is inconceivable to jeopardize a current security status by deploying VoIP. Due the still prevalent lack of public IPv4 addresses, many companies and Internet providers use Network Address Translations (NAT) in different flavors. This challenges the interoperability, since call signaling requires the caller to be able to contact the desired recipient (callee).

The issues related to availability can be broken down into four areas: *dynamic port allocation, embedding port addresses in the packet payload, private IP end-user addresses and session initialization from public IP to private network behind NAT.* In [25], Stukas and Sicker provide an overview of existing solutions: MIDCOM, STUN, Sen, FANTOM, STEM, with the recommendation of MIDCOM. MIDCOM's solution moves the application intelligence off the firewall and into trusted external MIDCOM agents; these agents control the "middlebox" (the firewall or NAT), using a standard control protocol and thus allowing signaling and media streams to pass through the firewall according to strict, secure policies. This solution removes the burden of performing application specific processing by the firewall and NAT as well as removing the vendor-dependency for support of H.323 or SIP. The security of the link between the agent and the middlebox is critical. Bur Goode[9] mentions two other solution types: (1) a *proxy* placed at the border between two domains that handles the VoIP-related traffic, and (2) a *firewall* that understands the application logic. The MIDCOM solution type seems to gain momentum; the IETF has a Midcom Working Group.

In this thesis, the firewalls are looked at as independent from the remaining firewall architecture. More comments on this decision will be offered in the "Methodology" chapter.

2.4 Conversation Quality: Codecs and QoS

Voice can be transmitted uncompressed, but this tends to be rather ineffective since it contains much redundancy; transmission facilities are expensive in some parts of the world and merit therefore more efficiency. Most codecs perform voice activity detection, silence suppression and comfort noise creation during each silence period. Many good codecs exist, and have been compared.[9, 23] When it comes to the codec choice, a balance must be found between codec complexity, payload efficiency and packetization delay. Voice encoding/compression may shorten the transmission time ("putting the data on the wire"), but increase total end-to-end time due to the computing delay.

The process of *en- and decoding*, estimated with 5-10 ms per packet, was

empirically examined in [23]. μ-law compression (G.711) was compared to Adaptive Differential Pulse-Code Modulation (ADPCM, G.729), cutting the 128 kbit/s bandwidth needed for uncompressed voice (PCM) to 64 kbit/s and 32 kbit/s. The study concluded that uncompressed voice could be used in intranets, but recommended ADPCM for Internet usage. Impact of variation in packet size was linear, therefore negligible.

Seen on a high level, the goal of any implementation is to deliver good conversation quality. Some of the technical details mentioned before have a direct influence on the voice quality. *Delay* provokes two problems: echo (quote: "signal reflections of the speaker's voice from the far end telephone equipment back into the speaker's ear") and *talker overlap* ("one talker stepping on the other talker's speech"). Echo is very disturbing and must be addressed by some form of echo cancellation; overlap is problematic if the round trip delay (the time between emission and reception of the data) becomes greater than 250ms. The International Telecommunication Union (ITU) recommends a limit of the round-trip-delay for telephone traffic to 300ms[26] - yielding 150ms as maximum one-way delay. Opinions have changed little over time, a value of 150 - 200 ms is still a valid threshold.[4, 23, 27, 9, 1]. ACM's VoIP paper [3] mentions an interesting historical anecdote about the 1980s tests with voice over geosynchronous satellites where users deemed 270 ms latency as unacceptable, and the tolerable maximum was set to 200.

Jitter is the "inconsistent time spacing between each packet at the receiving host"; since normal voice sources generate a constant stream, jitter can make a conversation sound unnatural. A study performed at the Ghent University [20] states "delay jitter has a devastating influence on the perceived quality ... if the received signal is dejittered, the degradation due to jitter is similar to the one caused by packet loss". A jitter-buffer damps the variability of arrival rates, but adds to the total latency - a balancing act between performance and reliability is needed.

Packet loss effect depends on two parameters: frequency (how often) and contiguity (how many successive). How much packet loss a codec can handle depends on bitrate and codec design; the percentage lies between 1 and 10 percent [14, 20, 1]. It can be handled better if the lost packets are randomly distributed, and don't occur in bursts.

Bandwidth, the throughput of the network, needs to be large enough to accommodate the full data traffic.

The "subjective experienced quality" stands at the center of the end-user perception. For tests, the user pronounces his (subjective) judgment after *listening to* or *engaging in* a conversation. The ITU has given guidelines on how to perform listenings. [28, 29]

Several often combined methods have been used to investigate this, among

those Mean Opinion Score (MOS), Perceptual Speech Quality Measurements (PSQM, KPN Research, now ITU-T P.862[30]), Perceptual Analysis Measurement System (PAMS, British Telecom) and Perceptual Evaluation of Speech Quality (PESQ).

A short explanation: PSQM assesses the voice quality by comparing the original voice signal with the voice signal that is delivered to the end-user after transmission over the network. It scores the voice quality on a scale of 0 (excellent) to 6.5 (bad). PAMS conducts quality evaluation using an automated auditory model. It also assigns MOS scores based upon the quality detected. PESQ combines PAMS and PSQM techniques to generate voice quality scores on a scale of 0.5 (bad) to 4.5 (excellent).[23, 27] MOS testers judge the quality of voice on a scale of 1 (bad) to 5 (excellent); the scores are averaged to a mean value. MOS seems to be the most extensively used method. Since MOS scores are averaged and therefore somewhat test-dependent, MOS tests need to be long or abundant in number to give reliable and concrete measures. "It is particularly ill-suited for long-term measurement, such as making measurements every 5 minutes for an entire week."[27, page 63].

Empirical numeric measurements do not focus on subjective user options, but measures numerically precise data, like the delay in milliseconds, or the amount of lost packets. Such measurements haven been done in different granularities, with pure user-perspective ("microphone" in - "speakers" out) [23] or differentiating between underlying reasons [19, 31, 20, 18, 32]. When doing such measurements, the processing power of the device, memory availability at a given moment, efficiency of the protocol and driver and general OS design need to be considered when interpreting the data.

The conversation quality has reached acceptable levels, but the goal has not been fully reached yet. Especially *jitter* handling still seems to be an ongoing issue.

2.5 Other issues in VoIP

There exist some semi-technical issues that only relatively recently have reached an awareness-level that fostered an organized solution discussion. These topics are included for completeness, but will not be dealt with further in this thesis.

Handling of emergency calls According to [33, 7, 1], a big technological challenge lies in the the *911 problem* - routing calls to a "public safety answering point (PSAP)" in order to be able to provide local emergency assistance. With traditional PSTN, finding the nearest PSAP is very easy since the terminal de-

vices (telephones) are fixed and at known, hardwired locations. Even mobile phones offer a good tracking method, due to the registration mechanism of the active cell (area). With VoIP-terminals, it can be very difficult to determine where the call is coming from - if a "school district has five or six buildings, where are the paramedics going to show up?"

Several possible solutions are proposed: (a) the user defines his geographic location beforehand, (b) automatic routing of 911 calls to a regular PSTN-line, and (c) the PBX maintains and provides location information about the IP device.

All these solutions have (dis)advantages - (a) is very straightforward, but provides only a limited quality since the location information is invalid if the phone is used by somebody in a different location. (b) implies the subscriber still needs to maintain a PSTN connection, which is highly unlikely in the case of an individual, and an overhead in case of a company. (c) raises questions about privacy and possible abuse, but would be the most accurate solution.

Spit - spam over internet telephony The problem of email spam has been discussed widely. For a summary, see the 2004 "Spam" Research Survey [34] Spit, *spam over internet telephony*,[35] will get much attention soon since spammers and telemarketers are about to discover this new, promising market. No extensive analysis has been made on current status or countermeasures, but this is only a question of time since analogous developments as seen in email threaten to eliminate the technology's benefits.[24]

P2P (peer-to-peer) approach Skype announced in late 2003 a peer-to-peer (P2P) application for internet telephony; its success has given internet telephony in general much attention [36, 37]. Dan Sweeney's article in *America's Network* [38] targets clearly telecom executives, and asks some poignant questions. Is there a good business case behind this hype? Does it scale (due to the diluted P2P paradigm) and meet increasing demands? Will this technology strengthen the position of the big telephone companies while the small ones go bankrupt, leading to an oligopoly?

The economic impact of this new flavor of VoIP is also in the center of relatively few (semi)academic articles having examined it. [7, VoIP Myths] links the myth "VoIP is free" to the publicity of this software - this is a common misunderstanding, since calls are only free computer-to-computer. For calling regular phones ("SkypeOut"), the user is charged a fee. Additionally, there are implicit costs like the broadband internet connection itself.

In MIT's *Technology Review* [39, Skype beyond the hype], Khamsi points to technical specialties of VoIP with P2P: the search for a *unique* hit (read the *callee*), opposed to the multiple existence of shared files in a regular P2P-network. A "global index" - complete current directory of online users - is maintained

by "supernodes". Skype uses proprietary solutions (codec, signaling, routing) and claims their solutions obliviate the need for *quality of service*, a fact contested by some specialists like Mark Kaish of Bell South.

This precondition made Skype a technically more complex software than Kazaa, and re-introduced he master-list-concept (dubbed "Global Index"), thus moving away from "true completely-distributed P2P". This index is maintained by *supernodes*, randomly chosen powerful computers that are connected to the network; these special nodes exchange updates to the index and thereby have collectively seen a complete current directory of online users. The company does not need to provide any infrastructure itself (this statement is only valid for inter-skype calls, not connectivity with traditional PSTNs).

It is assumed that corporate users will hesitate to switch to this kind of implementation, since P2P networks lack *service-quality guarantees* and – due to the lack of control of the network – *efficient support possibilities*. Dennis Bergström [40] concludes that it cannot be recommended for corporate users to switch since (a) traffic cannot be confined by reason of supernodes being outside the company's influence, (b) controlling Skype traffic is extremely hard because of varying ports and protocols, (c) no content scanning is possible on account of encryption, (d) the "end user license agreement" (EULA) raises concerns, especially the section that specifies that no action may be taken to technically analyze Skype traffic, (e) known "bad" people – "famous" for embedding spyware in Kazaa - are behind Skype, and finally (f) there is complete lack of information about used encryption schemes, meaning the content *may or may not be readable by Skype Inc.*, representing a possible danger of information disclosure.

So far, no scientific study comparing classical VoIP with P2P VoIP has been undertaken, covering the broad range of topics related to VoIP.

Chapter 3

Architectural Issues and Implementation

One of the motivating factors for this thesis was the possibility to work on a "real-world" scenario. FreeCode[1] is a Norwegian company that creates, implements and supports Open Source (OSS) products. Since OpenBSD has an outstanding positive record for being *security-aware and -conscious,*[2] the idea was born to analyze redundant OpenBSD-firewalls for VoIP .

Combining the topic of redundant firewalls with Internet telephony yielded a possible research area. It turned out that only few academic articles have dealt with the issue of redundant firewalls, and none of them in the context of VoIP.[41, 42] Yet it was acknowledged in some strategic articles that single points of failure pose a security threat, and high-availability architectures were recommended.[43, 44].

Integrating a firewall solution into the already existing security landscape is not a trivial task if done in a responsible way. Most companies of a certain size abhor the idea of exchanging such a central element, yet they may consider enhancing the current architecture by splitting the traffic handling and thus delegating a specific task to dedicated hardware. Thus, it would be of interest to see if an OpenBSD-firewall is apt for handling exclusively VoIP traffic.

For live time-sensitive bidirectional VoIP network traffic, the presence of multiple, redundant firewalls is a must in order not to loose the communication channel to the outside world in case of a firewall failure.

As mentioned in the introduction, the focus was to find answers to the following questions:

- How can VoIP-traffic be generated with a moderate set of commodity hardware? How much traffic can be generated? Which properties must

[1] `http://www.freecode.no`

[2] Many contributions from the OpenBSD-community have made their way into other operating systems, like the *SSH* server and client implementation, or the *packet filter*.

synthetic/artificial VoIP packets have in common with real ones?

- What is the measurable traffic forwarding limit of one firewall? What are the limiting factors?
- What can be said about packet loss and it's influence on ongoing "calls" in failover situations, either administratively (soft-failover), or by power-cycling (hard-failover)?

Before being able to conduct meaningful experiment for finding answers, a *reasonable framework* had to be built. This chapter aims at making the architectural decisions *comprehensible*.

3.1 Hardware and Configuration

The following list gives an overview over the available hardware for this project. One of the issues was to find out how much hardware was needed for researching this, so the exact quantity of the hardware was not carved in stone beforehand, yet it was aimed at using the bare reasonable minimum, namely the firewalls plus two commodity desktop computers on the network edges.

SMC Network 8648 Tiger Gigabit Ethernet Switch (provided by FreeCode) was used to connect the firewalls and the computers together. The switch has 48 10/100/1000 capable ports, supports ISO-layer 2/3/4 switching and many more features. The full technical datasheet is available at the SMC homepage.[3]

The configuration follows Cisco-IOS-standards; the switch was configured as follows:

VLANs (a) One default administrative VLAN (ID 1) having three ports assigned to it - the uplink port with IP `10.0.0.30` and the two ports to the administrative interface of the *dells*. (b) The WAN VLAN (ID 31) with *dell1* and the WAN-ports of *firewall1* and *firewall2*. The network does not exchange any data with other networks. (c) The LAN VLAN (ID 32) with *dell2* and the LAN-ports of *firewall1* and *firewall2*. Also this network is closed.

Ports All ports are disabled explicitly, only the active ports are opened and configured for auto-negotiation. The negotiable operation mode set was reduced to `capabilities 1000full`, allowing only gigabit speed. Setting this manually did not work, even though the documentation contained such an example. A bug report was filed with SMC.[4]

[3] `http://www.smc.com/`

[4] Also another bug was reported, namely the crash of the HTTPS server when entering the IP-addresses of the NTP-servers through the web-interface.

MAC-address table The switch's static MAC-address-table was populated with the addresses for the *dells* and the *firewall*'s real hardware addresses.

Two Dell OptiPlex GX270 computers were provided by Høgskolen i Oslo; their task is to generate and capture traffic.[5] The machines contain an Intel Pentium 4 2.6 GHz processor (800 MHz system clock, hyper-threading disabled) on an Intel 865G chip set and are equipped with 512 MB DDR SD-RAM (dual 333 MHz).[6] The built-in Intel 82540EM Gigabit Ethernet interface was connected to the WAN/LAN, and an additional 3Com Fast EtherLink Xl 100 MBit PCI card (revision A and B) was used for admin network connectivity.

On the 40 GB Western Digital hard disk, Ubuntu Linux release "Breezy Badger" was installed with the "server" template. On *dell1*, 20 GB were set aside for an OpenBSD-installation in order to be able to create and maintain the images for the firewalls.[7]

The reader shall not be bored with a long elaboration of how these Linux machines were configured; however, a few points deserve attention:

kernel The latest Linux-kernel 2.6.16.14 was patched with PF_RING for allowing fast capturing rates.[8]

network configuration Upon taking the network interfaces up or down, the routing table was updated so it would reflect the status of network availability.[9]

serial console Since the author suspected to commit errors while writing the *packet filter* ruleset, precautions were taken so that `minicom` can be used to administer the firewalls. Unfortunately, the serial port on *dell2* did not work, and it was therefore only possible to access *firewall1* by this means from *dell1*.

[5] One computer was dead-on-arrival and had to be replaced with a new one immediately.

[6] Detailed specs see `http://support.euro.dell.com/support/edocs/systems/opgx270/en/ug/specs.htm`

[7] To the big surprise and disappointment of the author, no other operating system supports even reading the UFS2-filesystem used by OpenBSD which is a subtype of the *fast file system (FFS)*. The most current Linux kernel crashed upon mounting it, *ffsdrv* for Windows and BSD-based Mac OS X did not recognize the partition at all, and FreeBSD was also unable to mount it.

[8] In the context of traffic capturing, the decision to use Linux and not FreeBSD was a conscious one. Fabian Schneider empirically compared capturing in Gigabit environments and concluded in 2004[45, p. 35] that Linux with the PF_RING patch was best suited for a single capturing process. Several months later (in 2005) he conducted extended experiments[46, p.68] and found the combination FreeBSD with AMD Opteron processors was superior, yet this hardware was unavailable.

[9] Example in `/etc/network/interfaces`: `up /sbin/route add -net 10.2.0.0 netmask 255.255.0.0 gw 10.0.1.1 dev eth1` upon taking the WAN-interface of *dell1* online.

Hewlett-Packard Pro Liant DL140 (provided by FreeCode)[10] equipped with a Intel Xeon Pentium IV 3.6 GHz processor and 1 GB RAM. They are natively equipped with 2 Broadcom BCM5721 Gigabit Ethernet network cards, and a third one (Linksys EG1032) was added for being able to connect them to a third network.

Due to their planned function as firewalls, the harddisk and CD-ROM was removed and replaced with an PCI Reiser card that allows the system to use a Compact Flash (CF) memory card as hard disk. On the bootable compact flash card, a stripped-down *flash-boot* kernel image (with an UFS2-filesystem inside) [11] contains OpenBSD 3.8. During the boot process, the flash is mounted at `/flash`, the image is expanded in RAM, and the whole operating system is loaded into memory. No swap partition (on the flash card) is used since it would be the bottleneck of the system.

For allowing user-friendly configuration of the system, `/etc/rc` – the init-script – copies any files found in `/flash/conf` to `/` before any services are started. This mechanism was used to set the correct hostname, the network-related configuration (interfaces, hosts, resolver), the timezone, the packet-filter configuration, allowing login from the serial console, and having persistent ssh-server-keys. The file `rc.more` is run at the end of the init-script and contained a single command for re-mounting the flash card in read-write mode.[12]

Additionally, `/etc/rc` also extracts `/flash/*.tgz` files in `/`. This mechanism is used to make additional binaries available, for example the compiled C-tools `pf-query` for querying the *packet-filter* state information or `tod` for returning the *time of the day*.

3.2 Network Topology

The network topology has evolved during the project. The network was born at Høgskolen, with *dell1* running FreeBSD 6.0 and acting as gateway to the Internet. Unfortunately this network was stillborn and died after a short time – for troubleshooting the firewall hardware problems, it was necessary to move the equipment up to FreeCode at Forskningsparken. There, the network was built anew.

[10] During the first 8 weeks of the project, two no-name firewalls with a VIA motherboard were used; they had to be exchanged since both OpenBSD 3.6 and 3.8 kernel-dumped when initializing the PCI Gigabit Ethernet cards. The problem persisted with different network cards, leading to the hypothesis that the hardware was somewhat incompatible or dying. Many weeks have been spent on trying to find the exact error, but in vain.

[11] `http://www.mindrot.org/flashboot.html`

[12] In other words, the following files were in `/flash/conf`: `hostname.{bge0 | bge1 | sk0 | carp0 | carp1 | pfsync0}`, `hosts`, `localtime`, `mygate`, `myname`, `pf.conf`, `rc.conf`, `rc.more`, `resolf.conf`, `ssh/`, `sysctl.conf`, `ttys`.

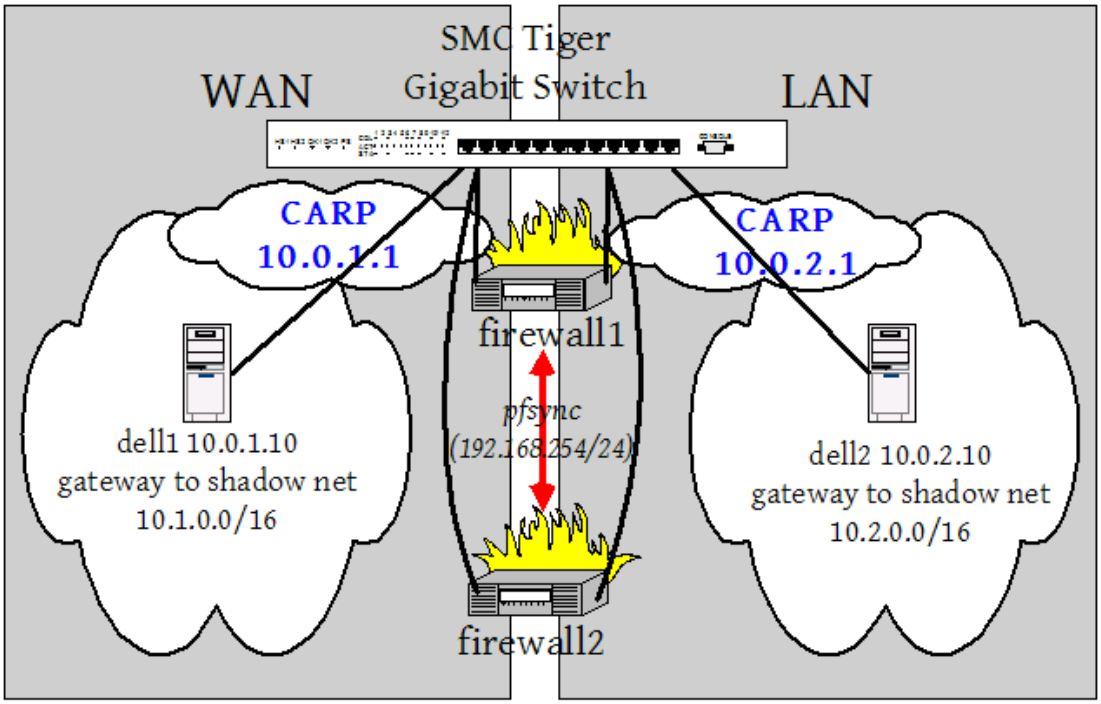


Figure 3.1: Testnetwork Topology

Diagram 3.1 shows the network topology. A short explanation of the selected address-spaces:

WAN `10.0.1.0/24`, populated with *dell1 - 10.0.1.10*, *firewall1/WAN - 10.0.1.2* and *firewall2/WAN - 10.0.1.3*. The routing tables of the firewalls contain *dell1* as the gateway to a *shadow* WAN-network `10.1.0.0/16`.

LAN `10.0.2.0/24`, populated with *dell2 - 10.0.2.10*, *firewall1/LAN - 10.0.2.2* and *firewall2/LAN - 10.0.2.3*. The firewalls point to *dell2* as gateway for the *shadow* LAN-network `10.2.0.0/16`.

pfsync Since only two firewalls were in use, they were connected together with a crossover cable. They were assigned the addresses `192.168.254.{2 | 3}`.

The CARP-cloud designates the two network segments where the redundancy element of the firewall activity lives. CARP is explained in the next section about *the concept of redundant firewalls.* On the WAN side, the master firewall responded at IP `10.0.1.1`, and at `10.0.2.1` on the LAN side.

Admin Network (not drawn) The two *dell* machines were accessible from the FreeCode network at `10.0.0.33` and `.34`, respectively. The ports on the switch were put into the administrative VLAN so the switch could be administered and queried by SNMP.

The *shadow* WAN- and LAN-networks were necessary for a two-folded reason. First, the class C-networks (`10.0.1.0/24` and `10.0.2.0/24`) can only provide addresses for 254 hosts, but traffic from more sources was to be simulated. Second, if the sending/receiving hosts lived on the same network as the respective firewall, a proxy ARP would have to be run on the *dells*, with the overhead of maintaining a mini-ARP-database and flooding the network with many ARP queries and replies. Defining the *dells* as gateways was the easier solution.

3.3 The Concept of Redundant Firewalls

Firewalls sitting on the edge of the network are often given much attention, since they are a key element in most security architectures. As elaborated in Ryan McBride's introduction to CARP and pfsync[47], there is often a strong pressure to keep the network up at all times. Several factors may contribute to such a demand; from a human standpoint, the person (e.g. manager) responsible for network issues needs to find the balance between defending legitimate downtime and measuring up to the upper management's expectations. It can also be a challenge from a technical standpoint since big organizations may not be able to map out consequences for all connected system if the network is detached for a time period.

Logically, keeping the firewalls up no matter what inhibits proper firewall maintenance, especially patching or upgrading that requires single-user access. Such an attitude is of course counter-productive in the long-run, since known security issues may not be addressed duly. The problem is strongly mitigated by using multiple firewalls. Many firewall manufacturers have understood this issue and offer solutions; the terminology used is "firewall clustering", "hot-standby firewall", "firewall redundancy" or "firewall failover". Two RFCs comment on protocols used to exchange state information: RFC 3768 on *Virtual Router Redundancy Protocol* (VRRP, Nokia/IETF authorship and the de-facto standard), and the older RFC 2281 *Cisco Hot Standby Router Protocol* (HSRP, Cisco/Juniper authorship). There's more than a side note[13] about patent problems with these, so both the OpenBSD developers and the Linux community decided to write their own protocols.[14]

[13] `http://www.openbsd.org/lyrics.html#35`

[14] The uCarp project at `http://www.ucarp.org/project/ucarp` and the ct_sync netfilter module `http://people.netfilter.org/hidden/ct_sync/` provide this functional-

OpenBSD supports redundant firewalls since version 3.5. This is achieved by combing technologies from OSI-network layers 2 and 3 (link and network layer). The Common Address Redundancy Protocol (CARP) allows several computers to share an IP address, and *pfsync* takes care of replicating the firewall states and was explicitly designed for dealing with known security problems of VRRP and HSRP.[15]

CARP is IP protocol number 112. Said with few words, CARP allows one virtual[16] IP address to be shared by several computers, either based on availability of a master node, or round-robin. This is accomplished by periodically sending CARP-advertisement messages to the network, saying "The IP x has MAC-address y, valid for virtual host n." The recipients – other network components like switches or routers – take note and update their routing tables, consisting of the *(switchport, IP, MAC)* tuple. In such a default configuration, there is only one virtual MAC address per virtual host, but the "location" of the host can change (moving to another port on the switch).

Several important configuration options can or must be set on any participating firewall; this is done by appending the configuration string to the `ifconfig` command, or writing it in the correct `hostname.carp`*N* file. The following parameters can be tuned:

advbase The frequency of the ARP updates is defined on a participating firewall with the *advbase* ("advertisement base") parameter; possible values are `1..255` seconds, default is 1. There is only one host advertising. According to McBride[47], it takes about 3 seconds for backup firewalls to realize the master has vanished, electing a new master and finally forward traffic.

advskew In order to allow the participating firewalls to elect the master, the metric *advskew* – default value of 0 – can be set to a value in the range `1..255`. The sender with the lowest *advskew* wins. The process of electing a master is only started if and only if the current master vanishes or sends a message with an "infinite" `adskew`. This means that the previous master, once online again, simply participates as a backup firewall.

state The *state* reflects the current role the participating client has at this instant: it can either be in `init` (finding current role or administratively

ity for Linux; this information is only included for completeness.

[15]Chris Russel of Infosecalliance has written *Understanding Dynamic Route Protocol Vulnerabilities* in late 2001 where he explains weaknesses of routing-related protocols, among them VRRP and HSRP (section 4, pages 7–8). The main problem is related to authentication of the packets. The document is indexed by `scholar.google.com`.

[16]"Virtual" is used in the sense of not being (a) statically assigned to a host (for IP) or (b) the real hardware-address of a NIC.

down), `backup` or `master`. If this parameter is set manually, it overrides the automatic election.

VHID and group password Since a physical device can participate in multiple CARP groups, the CARP packets contain the *virtual host ID* (VHID); this parameter is numeric. In order to ensure integrity of the packets they are signed cryptographically with the SHA-1 HMAC and a pre-shared group *password*.

arpbalance CARP supports the *arpbalance* feature; this feature allows multiple hosts to share a single IP address simultaneously. When *arpbalance* is used, there are multiple virtual MAC address (one per host), in contrast to normal CARP-configuration with one "moving" virtual MAC.

The kernel needs to be configured correctly so that CARP works as expected. This is either done on the command line by calling `sysctl`, or making a permanent entry in `sysctl.conf`.

carp.allow `net.inet.carp.allow=1` has to be set for accepting CARP-updates. It is important that the packets have a *pass*-rule in the packet filter ruleset, e.g. `pass quick on $phys_if proto carp keep state`

preemption If `net.inet.carp.preempt` is set to 1, then the firewall that was master before failing will take back his role once online again. For this to work, the process of electing a master is *continuous*: the hosts compare their own *advskew* value with the one in the packets they receive. If their own value is lower, they start advertising themselves, and the other host bows out after having sent bulk *pfsync*-updates to the new master. In addition, this option also enables failing over all interfaces in the event that one interface goes down. If one physical CARP-enabled interface goes down, CARP will change advskew to 240 on all other CARP-enabled interfaces, in essence, failing itself over.

arpbalance `net.inet.carp.arpbalance=1` must be set.

CARP can be turned off manually with `ifconfig carpN down`, forcing sending a last advertisement with `advskew 255` (infinity). Thus, any present peer takes over immediately as soon as the packet is received. This manual failover is handy for maintenance, and it will be examined if traffic passes through the routers more smoothly than when one firewall is power-cycled.

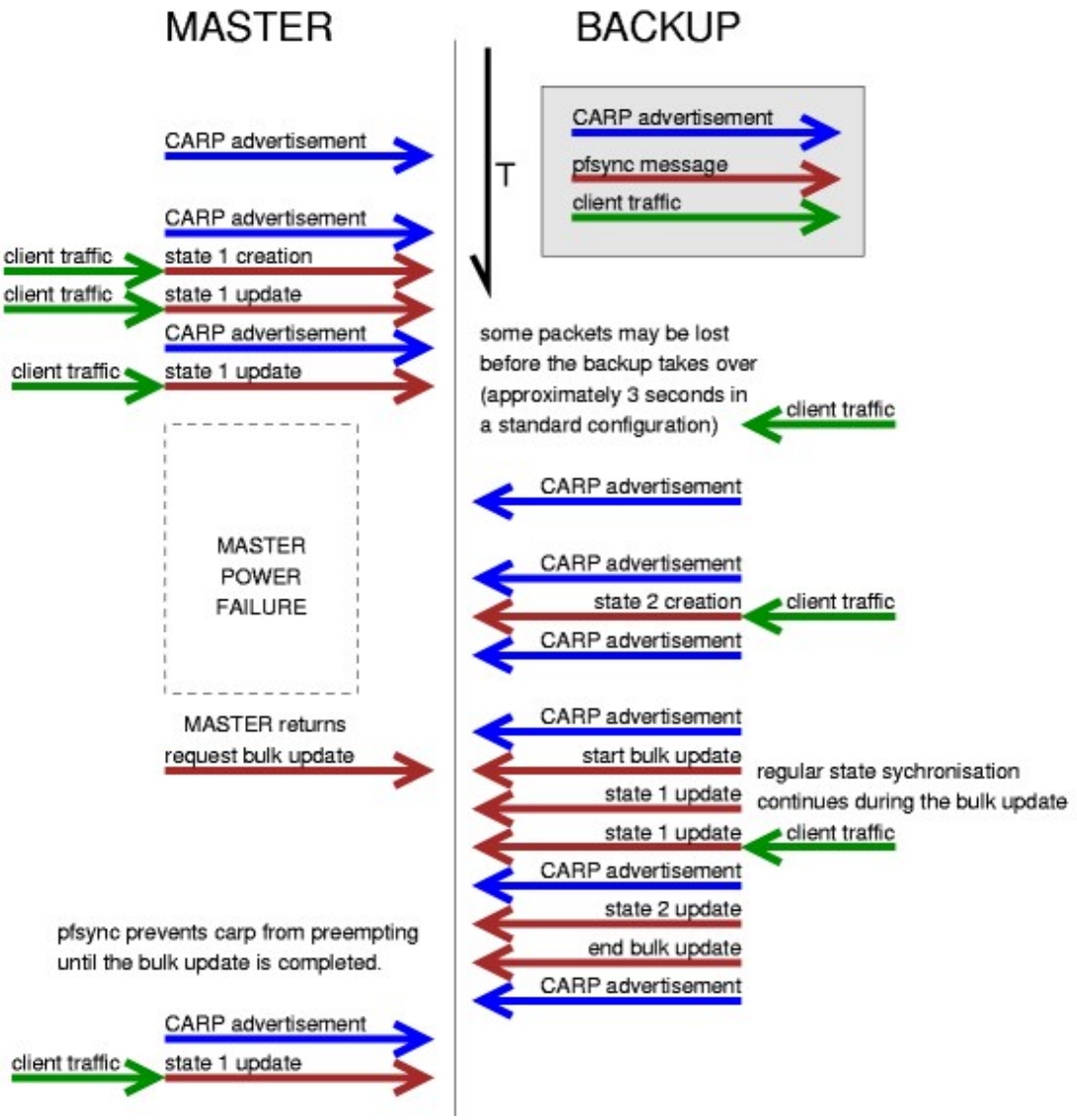


Figure 3.2: The CARP/pfsync Failover Sequence

pfsync is IP protocol 240. This protocol takes care of communicating the current *firewall state information* to the others.

By default, multicast updates are sent to the local network (224.0.0.0/4), but this can be overridden by the *syncpeer* parameter, forcing unicasts to the specified peer. Since pfsync is not cryptographically secured due to speed advantages, the traffic has to use a secure network link. In its simplest form, this can be a crossover-cable; if more than two firewalls participate, they could be part of a closed VLAN or an otherwise dedicated, secure network segment. The *pfsync*-traffic amount scales linearly with the number of participating hosts.

Figure 3.2 (diagram taken from [47]) gives a visual representation of the failover sequence, with *preemption* enabled; with `preemption` disabled, the lower part – below "master power failure" does not apply but the the master/backup roles are simply switched.

The packet filter is responsible for making the pass or block decision for traffic. OpenBSD has a very elegant configuration language: the Berkeley Packet Filter (BPF).[17] Only a limited subset of *pf*'s capabilities were used for this setup.

In this experiment setup, CARP was configured on both firewalls with `advbase 1`. Even though this is the default, explicitly stating it rules out misunderstandings. *firewall2* was configured with an `advskew 128`, so it would automatically assume the role as a backup as long as the master is present. Preemption is turned off.

The `hostname.carpN` configuration file therefore looks like this:

carp0 attached to the WAN interface

```
inet 10.0.1.1 255.255.255.0 10.0.1.255 vhid 1 pass v0ip
carpdev bge0 advbase 1
```

(*firewall2* contains additionally `advskew 128`)

carp1 on fw2 attached to the LAN interface

```
inet 10.0.2.1 255.255.255.0 10.0.2.255 vhid 2 pass v0ip
carpdev bge1 advbase 1
```

(*firewall2* contains additionally `advskew 128`)

The `hostname.pfsync0` contains the line `up syncdev sk0 syncpeer 192.168.254.{2 | 3 }`.

The complete *pf* ruleset can be found in the appendix; with normal language, the ruleset can be described as follows:

[17] For an introduction and advanced topics, see `http://www.openbsd.org/faq/pf/`.

- allow all traffic on the loopback-interface
- *general options:* set a limit of 50,000 states, block-policy return
- allow pfsync on `sync-if`, and carp on WAN and LAN, keeping state information
- allow new SSH-connections from/to WAN/LAN, keeping state
- allow incoming WAN UDP traffic for ports 20,000 to 30,000 going to LAN with a first-packet-timeout of 5, a single-direction-timeout of 15 and a connection timeout of 15, keeping state
- allow incoming LAN UDP traffic for ports 20,000 to 30,000 with the same timeouts and also keeping state
- allow incoming WAN UDP traffic from and to IAX2 port (4569) with a first-packet timeout of 45, a single-direction-timeout of 15 and a connection timeout of 15, keeping state
- allow incoming LAN UDP traffic from and to IAX2 port with the same timeouts and also keeping state
- block other traffic coming from WAN
- allow from LAN to WAN, keeping state

3.4 Traffic generation

Traffic used to load the firewalls can either be *replayed* or *generated*. The pros and cons of these alternatives had to be pondered so that a reasonable choice could be made.

Traffic replay implies that traffic was captured and stored previously, and several tools could be used for this, for example tcpreplay or tcpivo. Harpoon[18] or "Monkey See, Monkey Do"[19] only rely incoming traffic and do not generate any traffic themselves.

There are inherent difficulties with replayed traffic so that packet sending times correspond to the original ones. Feng et al.[48] (authors of TCPivo, a "High Performance Packet Replay Engine") identify the following areas:

[18] `http://www.spirentcom.com/documents/atp/University_of_Wisconsin-Whitepaper-978.pdf`

[19] `http://www.usenix.org/publications/library/proceedings/usenix04/tech/general/cheng.html`

- preloading the trace file(s) for quick availability
- sending the packets with correct inter-packet-gaps, which requires an in-kernel sending procedure
- process scheduling

Several elements important for the planned experiments were already known. Since VoIP traffic is time-sensitive, the inter-packet-gap of "outgoing" packets needs to be correct. This does not exclude bursts. Feng was able to send correctly packets with an inter-gap of 20 milliseconds (ms), coming from 64 MB 1 million packet traces. Two of the voice codec candidates use packet rates of 50 packets per seconds, yielding 20 ms inter-packet-gap for one voice call/connection. This poses a serious limitation for generating simultaneous calls.

Additionally, if the captured traffic is 'real', for example from a VoIP provider or a company, many issues need to be addressed concerning storage and maintenance, both on a technical level (network traffic traces with full payload grow to huge sizes over short time), as well as on a legal level (privacy law). Due to well-known time limitations for this project, this would have been a hindering factor. Last, replayed traffic does not adapt to new situations, e.g. new protocols, or increased network capacity. This is a significant drawback for future research because new traffic traces would have be organized.

The biggest advantage in replayed traffic is the *representative* property: the packet pattern (packet size, size distribution, packet amount, etc.) do reflect a real-world situation.

Traffic generation The alternative is to generate the traffic. Taking this road would make it much easier to repeat the experiments later, yet it is not without pitfalls. The following demands must be addressed:

time needed for packet generation Generating network traffic needs processing resources. Traditional userland programs need to traverse a long generation path from user- through kernelspace until the packet is put on the wire. The overhead of handling the packet down from user- to kernelspace is so big that a rather impressive machine park is needed to load a Gigabit Ethernet link. Alternatively, a much faster *kernel packet generator* can be used.

packet characteristics The generated traffic - both from a single-packet perspective and seen as a set - must reflect the "true" characteristics. For Internet telephony traffic, this includes varying packet sizes for different voice codecs, changing IP and UDP information, as well as the packet intensity.

3.4.1 Real or Synthetic?

A natural choice would have been to use programs like Asterisk or Yate to generate calls, since they can generate real VoIP traffic, including eventual media flow control traffic like SIP/IAX and RTCP. The voice payload is normally created by streaming a recorded sound file; bidirectionality can be achieved by automatically answering incoming calls and sending back the received packets. A quick test showed that the upper call generation limit for this approach was about 400 calls per computer, when generating the media itself (encoding the voice payload) taking most of the resources.[20] The load two computers could generate would therefore not be enough to load a Gigabit link. Since the machine park for this thesis *was* to be moderate, this possibility was excluded. Not to forget that the software may or may not support new developments (meaning new control protocols), making it harder to have an extensible framework.

The focus was turned toward generating "dummy" VoIP traffic that has enough "true" characteristics. Generating such synthetic traffic could be done in kernel level, so generation would be much faster. This is only possible since the "call voice" is almost exclusively wrapped in UDP or RTP traffic; the more complex TCP protocol has a much larger overhead and is harder to handle in-kernel.

Since most firewalls (and specifically the OpenBSD-firewalls in question) look at the IP/UDP protocol information and do not examine the payload itself, having non-voice payload does not pose a problem - the idea was born to use "pktgen the linux packet generator" (capitalizing by Robert Olsson, author of pktgen)[49]:

interface `/proc/net/pktgen/` is the interface to the kernel module.

parameters A full overview of all parameters is available online.[21] Of pivotal interest were the parameters *count, delay, pkt_size, udp_src_min/max, udp_dst_min/max, dst_min/max (IP) and src_min/max (IP).*

packet size distribution In order to model traffic realistically, Schneider has extended pktgen for his engineering thesis in 2005.[46]. His version allows packet size to be selected according to a statistical distribution, instead of either having a fixed size, increasing monotonically or just have a random size in the range a..b. Since Olsson did a major rework of pktgen

[20] Programs exist for only generating VoIP control protocol traffic, both Open-Source and Commercial: SIPptester, HCL SIP Conformance Tester, sipsak any many others. Other software can be used for generating payload, like rtptools, or both control and payload traffic (Asterisk, Yate, Candelatech's LANforge FIRE).

[21] `ftp://robur.slu.se/pub/Linux/net-development/pktgen-testing/pktgen-HOWTO.txt`

for kernel 2.6.11, Schneider's distribution enhancement is not available yet.

Schneider, again, [45, p. 30] explains that he was able to fully load Gigabit Ethernet with a single computer running pktgen, sending 1,500 byte packets. With 180,000 packets per second (pps), rates of approximately 915 Mb/s were reached, corresponding to an almost fully utilized Gigabit link. pktgen reached its upper sending limit at an inter-packet-pause of 3,000 nanoseconds.

At a later stage of the project, it was discovered that there exists an alternative to pktgen: KUTE (*K*ernel-based *U*DP *T*raffic *Engine*).[50] . KUTE has the following features pktgen is missing or has implemented in a different way:

link layer independence KUTE can run on any link layer and not just Ethernet; this is possible since the level-2 header is set by the kernel's `output` function and not by KUTE itself.

additional adjustable parameters KUTE's configuration options include the packet payload (cannot be controlled in pktgen), time-to-live (TTL) and type-of-service (TOS) as well as flags for turning on/off UDP checksum and IP identification fields.

receiver module KUTE includes a receiver kernel module; it creates a inter-arrival histogram that can be accessed through the `/proc` file system (the number of histogram bins and their size in microseconds is configurable). Upon unloading the module, the mean and and standard deviation are calculated and written to `/var/log/messages`. The receiver can filter the traffic based on source IP and UDP port number, or it can simply measure all incoming UDP traffic. For best performance, the Linux kernel must be patched. For inter-arrival time measurement, KUTE uses the timestamps present in the socket kernel buffer (skb).

The biggest incentive for using KUTE would have been the availability of the receiver module. Since KUTE in its current version 1.3 only runs on old kernels (either 2.6.4 or 2.6.11.10) which are problematic with Deri's PF_RING patch, KUTE was not used.[22]

3.5 Voice codec selection

As mentioned in the VoIP background chapter, many different voice codecs are deployed, but only a few have gained widespread acceptance among non-telecommunication-companies and private parties. With the advent and spread of

[22]Sebastian Zander as one of the authors of KUTE was so kind to send me the KUTE 1.4 prerelease, but kernel support had only arrived at 2.6.11.12.

codec	payload (bytes)	pps
G.711	160	50
G.726 (ADPCM32)	80	50
GSM (slow mode)	66	25
GSM (fast mode)	33	50

Table 3.1: Voice codec candidate overview

open-source telephony software like Asterisk[23] or Yate[24], two main issues for selecting the speech codec became apparent:

Processing intensity Since the vast majority of these "VoIP switchboards" run on commodity hardware, the en- and decoding intensity should be moderate. Eventually - if interconnectivity with another VoIP switchboard is desired - transcoding (converting from one codec to another) has to be done; example transcoding delays when using Asterisk on a Pentium III 300 MHz can be found in [51, table p. 194]. The delays vary greatly. G.711 with its two dialects a-law and μ-law turned out to be the codec of choice for many since it has a low processing intensity, with Speex (an variable bitrate codec licensed inder the Xiph.org variant of the BSD license[52, p. 147]) gaining popularity.

Licensing issues Releasing software open-source touches on licensing issues. Many possible codecs are patent-encumbered in one way or the other (G.723, G.279, iLBC). Therefore G.711, G.726 (ADPCM) and GSM (fast and slow mode) are unproblematic and therefore most used.

Out of the plethora of possibilities, the three community favorites *G.711, G.726 and GSM (slow)* were selected as candidates.[52, p. 144] Table 3.1 lists their *data payload size* and the *default*[25] *pps (packet-per-second) rate*.

3.6 Modeling VoIP traffic

Since pktgen as a "dumb" UDP-packet-generator only allows few parameters to be tuned in order to mimic more sophisticated traffic patterns, a simple model for imitating VoIP traffic had to be devised. The following enumeration

[23] `http://www.asterisk.org`

[24] `http://yate.null.ro`

[25] Some codecs allow for choosing the amount of milliseconds that is encoded and put in one packet. Increasing this amount leads to bigger payload/packets while decreasing the pps, and vice versa. For keeping the possible combinations at a concise level, only the default was chosen.

lists typical properties of VoIP traffic, and how these parameters were managed with pktgen.

IP header A regular IP-header with no special parameters set consists of 20 bytes. In this header, the source and destination address is contained; these can be set at runtime in pktgen. Since type-of-service (TOS) field is not used consistently as criteria for routing or firewalling on the Internet, and pktgen does not expose an interface for modifying this parameter, it was not taken into consideration.

UDP header The normal UDP-header is of length 8 bytes, and contains the source and destination port. Also these parameters can be set in pktgen. It is important to note that IAX2 only uses one well-known 4569 port in both directions, whereas RTP uses one port combination per stream and direction.

RTP (real-time protocol) and IAX2-headers The headers for the specific media stream can be already considered "payload" and are not taken into consideration when passing through a firewall. Therefore, only the size of the headers are important: 12 bytes for the RTP-header and 4 bytes for the header of a IAX2-mini-frame. The length of the header was used for calculating the total packet size.

Voice payload Different codecs have varying payload sizes; again, this belongs to the packet payload and influences only the total packet size. The payload sizes listed in table 3.1 were used for calculating the total packet size.

packets per second With the exception of GSM in slow mode, all codec candidates have a packets-per-second rate of 50. The packet rate can be modeled by setting the appropriate inter-packet-gap (or delay) in pktgen.

Table 3.2 gives an overview of the packet sizes used for modeling the possible *codec* and *media stream type* combinations. pktgen's documentation states that the network card will add 4 bytes for the Ethernet checksum (CRC) to the configured packet size.

While IAX2 supports "trunking", this could not be included in this model since it would involve a distribution of different packet-size, not all meta-frames being of the same size. Using Schneider's distribution enhancement for pktgen, the model could be enhanced in the future.

Of course, VoIP traffic consist of more than voice packets, yet it was decided not to take into account the signaling and control traffic. *For IAX2*, once the connection is established, control traffic consists of "full frames" having a bigger header (12 bytes), and occurs according to [16] only when 16 bit

codec	mediatype	pkt size (bytes)	max packets	calls
G.711	RTP	214	615,677	12,313
G.711	IAX2	206	639,132	12,782
G.726	RTP	134	972,592	19,451
G.726	IAX2	126	1,032,444	20,648
GSM slow	RTP	120	1,082,401	43,296
GSM slow	IAX2	112	1,157,049	46,281
GSM fast	RTP	87	1,474,920	29,498
GSM fast	IAX2	79	1,617,081	32,341

Table 3.2: Maximum packet count for one Gigabit Ethernet second per voice packet

`timestamp` in the "mini-frame" wraps. This happens every 65,535th millisecond, or roughly once a minute. Compared to the much higher media packet rates, it was felt that this control traffic could be neglected. The IAX2-documention does not explain how feedback about the media stream is exchanged between the peers. The frequency of signaling traffic, using full frames for call setup (NEW - ACCEPT - ACK / RINGING - ACK / ANSWER - ACK) and teardown (HANGUP - ACK), is also much lower than the media packet rates, and was not taken into account either. For modeling a real-world scenario with many simultaneous VoIP users initiating or ending conversations, this traffic would have to be incorporated.

The same thought about the huge frequency difference between signaling and media traffic was applied to SIP, and therefore no SIP packets were modeled. When it comes to the real-time control protocol (RTCP), RFC 3550 says: "It is RECOMMENDED that the fraction of the session bandwidth added for RTCP be fixed at 5%." With default settings, a RTCP packet is 128 bytes long, and therefore in a similar range as a VoIP packet. For simplicity reasons, no special accounting for RTCP was implemented; for calculating the relation generated packets per second -> calls, these 5% RTCP-overhead could be subtracted.

The *CARP-advertisement* packet is sent once per second, and since only the master of one network segment advertises, this single packet was assumed to be neglectable as well.

The exchange of *packet filter states* was done through a separate network and did not affect the VoIP-network.

In order to make pktgen more user-friendly for simulating traffic, a correlation was sought between the parameters *delay* and *count (total packets to send)* and more intuitive parameters for VoIP, namely *(simultaneous) calls* and *call duration*. The *delay* parameter of pktgen is nanoseconds, which in reality is much

too fine-grained, since most kernels do not have such a precise timer.

There are 10^9 nanoseconds per second, so for finding the correct packet frequency the following formula was used:

$$packet\ intergap = \frac{1,000,000,000}{call\ count \times packets\ per\ second_{codec}} \tag{3.1}$$

For example, simulating 1 call for 1 second with a codec using 50 pps yields a packet intergap of 20,000,000 (1,000,000,000 / 1 x 50) and a total packet count of 50. Since [45] managed to almost load a Gigabit Ethernet link with a single computer, it was assumed that pktgen was able to generate packets at Gigabit "wire speed"[26]; the following calculation was used to find the correct *packet size* value (all sizes in bytes):

$$packet\ size_{codec} = (Ethernet + IP + UDP + RTP/IAX2)\ header + payload_{codec}$$

$$max\ packet\ count_{codec} = floor\left(\frac{134,217,728\ [GB\ Ethernet\ capacity]}{packet\ size_{codec}}\right)$$

For finding the approximation on how many calls this means:

$$max\ calls_{codec} = \frac{sent\ packets\ per\ second}{packets\ per\ second_{codec}} \tag{3.2}$$

Table 3.2 contains the "max packet count" and "calls" values for the chosen VoIP codecs and media types. For getting the the correct *duration*, the *count* was set to

$$calls \times packets\ per\ second_{codec} \times duration\ in\ seconds$$

The accuracy of this approach is commented in the "Experiments" section.

In oder to test the firewall's packet filter, the sender and destination information – IP addresses and UDP ports – was dynamic. pktgen offers two possibilities for this: either to monotonically increasing the addresses or ports in range, or to randomize the fields. For every fields, a randomization flag can be turned on or off. Since randomizing *any* of the fields would have invalidated the allowed IP/UDP permutations – once the "conversation" is started, per-direction-socket will not change (same sender-IP/UDP receiver-IP/UDP combination for the whole call) – this option was not used.

Based on this information, the `pktg-config-voip.sh` Bash-script was written that configures all the necessary pktgen parameters. The idea is to shield the user from having to know many internals of pktgen. The script's usage:

[26]Gigabit capacity = 1 GBit/s = 1,024 MBit/s = 1,048,576 KBit/s = 1,073,741,824 Bit/s = 134,217,728 Byte/s

```
USAGE: pktg-config-voip.sh
 -mediatype (rtp | iax2)
 -codec (g711 | g726 | gsms | gsmf)
 -calls n (integer)
 [-singlehost]
 [-nocarp -fw(1 | 2)]
 -direction ( W2L | L2W)
 -duration s (seconds)
 [-clone_skb n (packets)]
 [-softirq n (packets)]
 [-debug]
```

Most options and their effect are self-explanatory; nonetheless some comments may be useful:

-mediatype Mediatype *rtp* uses different UDP ports for simulating the media streams; *iax2* employs only the well-known port 4569.

-calls The amount of calls determines directly how big the inter-packet-gap is, as well as how many different IP-addresses and UDP-ports are used. IP-addresses start at `10.{1|2}.0.1`, UDP-ports (for mediatype RTP) start at 20,000 and end at 30,000.

-singlehost If only one IP address should be used, this option can be given. Combining this with `-mediatype iax2` allows to send all traffic with one IP address and one UDP port.

-nocarp with -fwN (must be given before -*direction!*) pktgen needs to know the Layer-2-address of the recipient; this is normally set to the virtual CARP MAC based on the `-direction` option. If this is not desired, it can manually be set with this option.

-direction can either be WAN-to-LAN (W2L) or LAN-to-WAN (L2W).

-clone_skb and -softirq These two settings have only effect when initializing pktgen. `-clone_skb` sets how many identical *copies with packet-payload* should be sent; when not specified, it equals the calculated total packet count. `-softirq` sets the interval on after how many packets pktgen should "simulate" an irq.

3.7 Traffic Forwarding, Capture and Analysis

As discussed in several scientific papers,[53, 54, 55, 56], software-based traffic handling often suffers from the general-purpose architecture of many drivers;

this is especially valid for *NIX-platforms. The problem is called "receive livelock" and denotes system overload due to lots of interrupt request. Techniques like memory-mapping[27] or using a ring-buffer[28] lessen input-output overhead. *Interrupt mitigation* allows combining multiple interrupts into one single one, preferably with the operating system polling the device and not vice versa. Interrupt handling is a major issue for high-speed network cards and the inability to deal with it can lead to process starvation since all CPU resources go into handling interrupts. Newer PCI and PCI-X specifications also reflect ongoing efforts to address these issues.

A few academic papers gave insight on other researcher's activities in this area. A Korean Gigabit-packet-header capture framework [57] can collect 100% of 385-byte-packets while "existing software-based systems can collect less than 50% in spite of using more than twice of CPU resources". This is achieved by modifying the firmware in the network interface card and let it transfer the packet header data into RAM using Direct Memory Access (DMA). Additionally, the data can be encapsulated in a UDP packet and sent to other computers, opening the possibility for load-balancing. *Fairly Fast Packet Filters* [58] presents the approach to "minimize both packet copying and context switching, pushing processing to the lowest levels, and executing computationally expensive functions as native code", with the focus being more on the *filtering* part of traffic capture. Finally, [59] mentions that it's hard to monitor and analyze traffic with a single general purpose system, and propose a *distributed capturing architecture*.

Armed with this information, the decision was made to patch both Linux machines with Deri's PF_RING and setting the `bucket_len` (snaplen) to 68 bytes, and the `num_slots` (ring size) set to 16,384 bytes, as well as increasing Linux' receive buffer default and maximum values[29] to 128 MB (134,217,728 bytes), as recommended in [45]. This ensured having reasonable traffic capturing rate. In order to keep the capturing processing overhead small, no pcap-filter was passed to tcpdump; if only the number of received packets was of interest, tcpdump was called with the command-line parameters `tcpdump -w /dev/null -v`. The causes tcpdump not do do any lookups, no information is written to the harddisk, and no output is shown on screen but the total count of received packets so far, updated every 10 seconds.

One important side-effect of interrupt mitigation(device polling) must be mentioned: if it is the operating system to timestamp the incoming packets

[27] When using memory-mapping, the kernel does not copy the received network data to a new location but allows the userspace program to access the same memory, thereby saving one copy operation.

[28] A ring-buffer does not allocate constantly new memory but overwrites old entries after some time.

[29] The entries in the `/proc/sys/net/core/` are `rmem_max` and `rmem_default`.

(and not the NIC), then these timestamps will not be accurate since the packets are processed in a batch.

"Receive livelock" is also a problem when forwarding traffic, and may affect the OpenBSD-firewalls. In contrast to FreeBSD which has excellent device polling (and is therefore often considered the best platform for capturing traffic), no such kernel enhancements seem to exists for OpenBSD. The problem is not unknown, of course – OpenBSD 3.8 uses the *bge* ethernet driver for the Broadcom gigabit Ethernet cards and according to a message on kerneltrap[30], the "idle loop fix" should have made OpenBSD more resilient. Yet discussions in the OpenBSD-community show two distinct different opinions:

- *either* the network card driver is assumed to handle the problem, and therefore all network cards from the same manufacturer should be on the same irq
- *or* irq-sharing is bad and creates additional load; a kernel compiled for multi-processor-support be used instead, for taking advantage of the I/O Advanced Programmable Interrupt Controller (IOAPIC) on the chipset. No solid reasons were given for this rather unusual recommendation.

`dmesg` on reveals that all Broadcom NIC's *and* the Linksys Gigabit card share irq 5, on both firewalls.

```
bge0 at pci2 dev 0 function 0 "Broadcom BCM5721"
irq 5 address 00:15:60:ed:da:06
brgphy0 at bge0 phy 1: BCM5750 10/100/1000baseT PHY
bge1 at pci3 dev 0 function 0 "Broadcom BCM5721"
irq 5 address 00:15:60:ed:da:07
brgphy1 at bge1 phy 1: BCM5750 10/100/1000baseT PHY
skc0 at pci5 dev 1 function 0 "Linksys EG1032": irq 5
skc0: Marvell Yukon (0x1)
sk0 at skc0 port A: address 00:12:17:54:f2:96
eephy0 at sk0 phy 0: Marvell 88E1011 Gigabit PHY, rev. 3
```

Available information was insufficient for deciding that this standard configuration should be changed, and was left untouched.

Analyzing the live or captured traffic is a further step; "analysis" can stand for anything from a simple bandwidth check to sophisticated in-depth breakdown. On a high-level, VoIP makes network traffic increase considerably; if not enough resources are available, this is very problematic because "voice is

[30] `http://kerneltrap.org/node/5169`

more sensitive to network slowdowns and glitches than data (...) Utilization should be monitored over a period of time", as Emily Hollis[60] states in an *Certification Magazine* article from 2005. She gives an overview of companies and their products that can be used for analyzing VoIP traffic.[31]

Luca Deri [61] has extended *ntop* in order to be able to identify and analyze VoIP traffic. ntop allows extracting predefined metrics, covering both the SIP controlling protocol and the RTP media stream; for media streams, the metrics are available per direction. A quick glance shows that RTP_IN_JITTER and RTP_IN_PKT_LOST are interesting candidates. [62] mentions the same metrics for VoIP traffic, yet on a protocol-independent level: packet loss, delay, and delay variation (jitter).

Since using ntop would have increased the need for more computers (running contrary to the project goal of minimizing hardware) and introduced an external dependency for the analysis process, it was decided to look for other possibilities.

KUTE 1.3 has a receiver module which would have facilitated creating histograms of inter-arrival times; yet as mentioned previously, it is not working with a recent kernel and could therefore not be used.

One viable alternative when only counting received packets was to use Deri's example PF_RING program `pcount`. It was felt the handling was somewhat clumsy, tcpdump remained the tool of choice.

Plab[63], a "Traffic capture and analysis architecture" from the University of of Naples, Italy, was evaluated since it seemed to have interesting properties for analyzing flows of traffic, e.g. groups of packets with common criteria. The publicly available version (2.2) was only able to identify TCP steams on port 80; Alberto Dainotti sent me a more recent development version, which sadly did not work either. Quoting his own words: "we are currently working on an improved version of Plab with many new features to make it more flexible. It will take a while though" – unfortunately, time constraints prevented more testing. Plab could become an interesting candidate in the future for extended analysis.

After some more research focusing on how to be able to measure packet loss with the eventual possibility of correlating them to an ongoing call, it was decided to use the information provided in the pktgen header itself, as defined in `pktgen.c`:

[31] *Acterna* PVA-1000 VoIP Network Analysis Suite with a field tester unit (HST-3000) supporting call storage and later analysis in PVA-1000); *Agilent*'s Telephony Network Analyzer; *Brix Network* offers a free online service at `http://testyourvoip.com`, but also advanced tools; *Finisair*'s Surveyor Network Monitoring and Analyisis Console; *NetIQ*'s VoIP Management Solution; *Analyser Sales* Sniffer Voice (add-on for Sniffer Portable, Distributed, and Netasyst); *Viola Networks*' NetAlley VoIP; *WildPackets* EtherPeek VX; *AppareNet*; *Avaya*; *Empirix*; *Iaxia* and *Unleash Networks*' Ruby-based Unsniff Network Analzyer (having even a new IAX2-addon).

```
322 struct pktgen_hdr {
323   __u32 pgh_magic;
324   __u32 seq_num;
325   __u32 tv_sec;
326   __u32 tv_usec;
327 };
```

The first 16 bytes of every pktgen packet consist therefore four 4-byte-fields: 1. a magic number (0xbe9be955), identifying the packet as coming from pktgen, 2. a sequence number, 3. the "second"-part of the the `gettimeofday` system call, and 4. the "microsecond"-part of `gettimeofday`. If *all* traffic can be captured at the receiver end, these fields together with the packet-capture timestamps (from pcap) would provide enough material to look for packet loss, delay and jitter.

A Bash-shell script was written (`d2t.sh`[32]) that processes the very verbose text output of `tcpdump -ttNnXr file.pcap`. The `-X` switch produces all of the captured packet payload to be printed. The script reads the input line per line, checks the magic number, and then compacts the information into one single line with the following fields:

- pcapsec - the "second" part of the pcap timestamp
- pcapusec - the "microsecond" part of the pcap timestamp
- fromsocket - the IP/UDP information of the sender
- tosocke - the IP/UDP information of the receiver
- seq - the pktgen sequence number of the packet
- sec - the "second" part of pktgen's timestamp
- usec - the "microsecond" part of pktgen's timestamp

While designing and testing the script, two important observations were made. The first concerns *network byte order*. Attention needs to be given if the bytes are in big- or small-endian. A program reading the packets directly from the network needs to convert to the byte order of the local architecture. When using libpcap, the problem is delegated to that library and tcpdump passed the string in correct byteorder. The second is about *performance:* some important operations, especially `cut`ting the fields and converting the pktgen payload strings from hex into decimal, are done by proxy in a subshell, the speed is *very* slow. It took about 1 hour to process a dump file containing 280,000 packets - which does not correspond to a very long experiment run under high load. For future work, this analysis-script has to written in C, preferably also using libpcap.

[32] see listing in the appendix

Chapter 4

Experiments

After the rather long and tedious process of *getting an overview* of involved technical issues, *selecting options*, organizing *working* hardware and writing the script framework for being able to run tests, the focus was *finally* turned to the experiments themselves. Three areas were targeted: the *suitableness* of pktgen for generating synthetic VoIP traffic, the *traffic forwarding limits* of the firewalls, and *observations when firewall failover occurs.*

4.1 Traffic generation with `pktgen`

This section describes the experiments that were aimed at getting an empiric understanding of pktgen's traffic generation properties and capabilities.

Ruling out clone_skb and randomization overhead

A base value had to be established in order to understand how many packets could be generated. At the same time, it was necessary to find out if the following factors influence the speed:

clone_skb pktgen allows for creating a new payload every *n* packets. This is necessary for simulating Denial-of-Service (DOS) attacks[49, p. 5].

IP- and UDP-randomization Both the source- and destination-addresses can be randomized, as well as the UDP source- and destination ports. If this randomization is not turned on, the addresses/ports are increased monotonically.

`pktgen` exposes the packet-per-second (pps) rate after the generation process is done; a series of 14 2-million-packet sets with an IP/UDP range of

10,000 each were generated and the pps rates were recorded, with all possible variations of *dell*s and *firewalls*. The test was done with the above mentioned parameters turned off (no new payload, no address/port randomization) and turned on (new payload per packet, IP and UDP randomization), for the biggest and smallest packet size (214 and 79 bytes, see table 4.1). It was expected that the effect would be more notable the smaller the packets were, more packets need to be generated per second.

The results allow for the conclusion that there is no notable effect. The standard deviation of single measurements per batch is well below half a percent (one exception with 1.5% exists), and the difference of the averaged values is between 0.58 and 0.76 percent for the bigger packet and between 3 and 6 percent for the smaller one. *Translated into "calls":* a reduction of less than 0.01 percent, for both packet sizes. As assumed, the measurable change was bigger for the smaller packet.

Randomization will not be used in the firewall-tests since this would make the state tables of the firewalls explode (due the the immense large possible amount of host/port combinations), thereby partially invalidating the tests.

4.1.1 Maximum packet generation

As elaborated on in chapter 3, it was assumed that pktgen would be able to saturate a Gigabit Ethernet link since it was understood that [46] succeeded in doing that. This assumption proved quickly to to be wrong. Measuring the packets per second rate as reported by pktgen with `delay 0` yielded the results recorded in table 4.1. The last column, "percent", expresses the effective rate in percent of the theoretical Gigabit Ethernet packet limit for the given packet size.

The maximum packet generation average for the biggest packet (G.711/RPT packet of 214 Bytes) turned out to be around 277,000 (corresponding to 5,500 calls), and for the smallest (GSM fast/IAX2 packet of 79 Bytes) around 400,000 (8,260 calls). These values are assumed to be the "upper bound" of pktgen for this type of traffic.

Rereading closely Schneider's report,[45, 46] it was discovered that he had a *dual-processor* and used much larger *packet-sizes, up to 1500 bytes.* Since the overhead of constructing the header and payload, and putting the packet on the wire occurs for every packet, the I/O overhead in this experiments here is bigger by magnitudes. This becomes more visible the smaller the generated packets are, with the load percent decreasing.

As a practical consequence, the *packet intergap calculation* (equation 3.1, page 42) is wrong, and off by 50 to 75 percent. A new approach was needed.

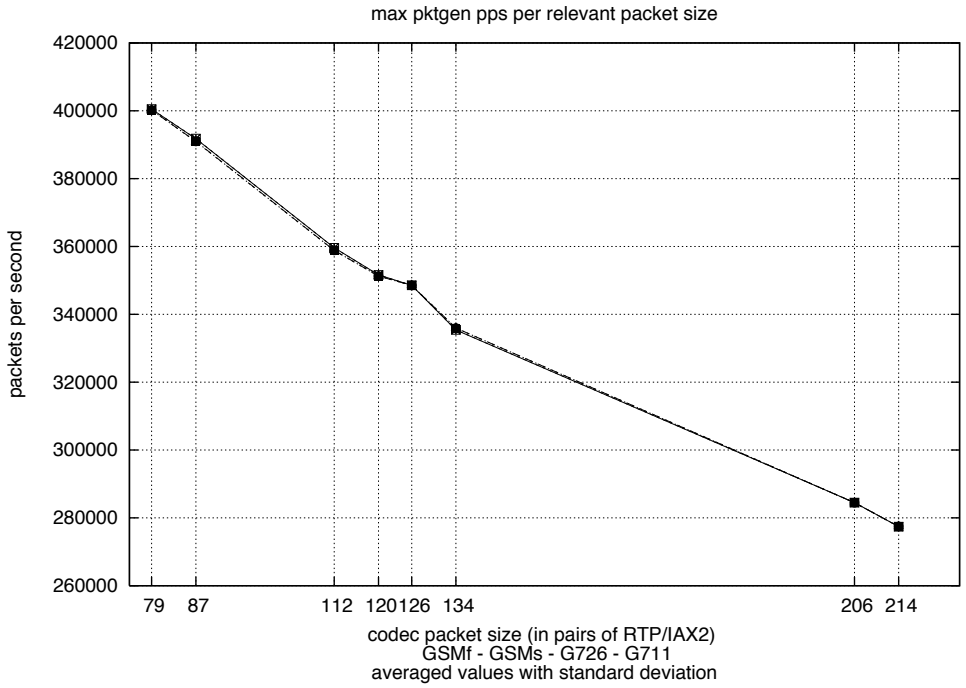


Figure 4.1: Max pps per codec

codec	type	pkt size	theoretical pps	effective pps	percent
G.711	RTP	214	615,677	277,448	45.1
G.711	IAX2	206	639,132	284,485	44.5
G.726	RTP	134	972,592	335,690	34.5
G.726	IAX2	126	1,032,444	348,528	33.8
GSM slow	RTP	120	1,082,401	351,440	32.5
GSM slow	IAX2	112	1,157,049	359,219	31.0
GSM fast	RTP	87	1,474,920	391,365	26.5
GSM fast	IAX2	79	1,617,081	400,360	24.8

Table 4.1: pktgen packet sizes and effective pps for voice packets

4.1.2 Generation time

The next idea for making pktgen more user-friendly by approximating some $function f(calls, duration) = (delay, totalpacketcount)$ was to measure the time needed for generating a packet of a given size with *delay 0* and use this value as some kind of "time0" or generation time.

$$generation\,time_{codec} = \frac{1,000,000,000\,[ns]}{max\,packet\,count_{codec}} \tag{4.1}$$

Once a stable "time0" value was obtained, the `delay` can be calculated as follows:

$$calculated\,packet\,count = calls \times packets\,per\,second_{codec} \tag{4.2}$$

and

$$delay_{packet} = \frac{1,000,000,000}{calculated\,packet\,count_{codec}} - generation\,time_{codec} \tag{4.3}$$

.

While experimenting with generation time, a problem was discovered: the packet generation rate fell dramatically by 90% when the delay value passed from 900 to 1,000 nanoseconds. The very low packet generation rate stayed constant until about 40,000 nanoseconds. This made it impossible to increase the network load *smoothly.* (Figure A.1 in the appendix visualizes this.)

Since most tests so far were conducted with either delay 0 or delays $>$ 25,000, the problem had not become visible yet. A counter-check was done with the old version of pktgen (1.6, pre-2.6.11-kernel)[1]; *NSPT* (no such phenomenon there).

After investigating with the author, Robert Olsson,[2] he suggested commenting out "debug line" 1,660 in `pktgen.c` and setting the *timer frequency* to 1,000 Hertz.[3] New tests showed that pktgen was now generating traffic smoothly, depicted in figure 4.3.[4]

With a working pktgen, the "generation time" was researched in the area from 0 to 250 bytes, results shown in diagram 4.2. Noteworthy is the fact that packets up to size 60 bytes have a constant generation time and accordingly

[1] All scripts interacting with `/proc/net/pktgen/` had to be extended since the generation process/thread management had been reworked substantially; also some variables were renamed (`ipg` became `delay`, `dstmac` became `dst_mac`).

[2] See appendix for details.

[3] Kernel-parameters `CONFIG_HZ=1000` and `CONFIG_HZ_1000=y`, found in section *Processor type and features*.

[4] The numbers reported as maximum packets per seconds are not identical with the ones indicated in table 4.1, probably due to the changed timer frequency.

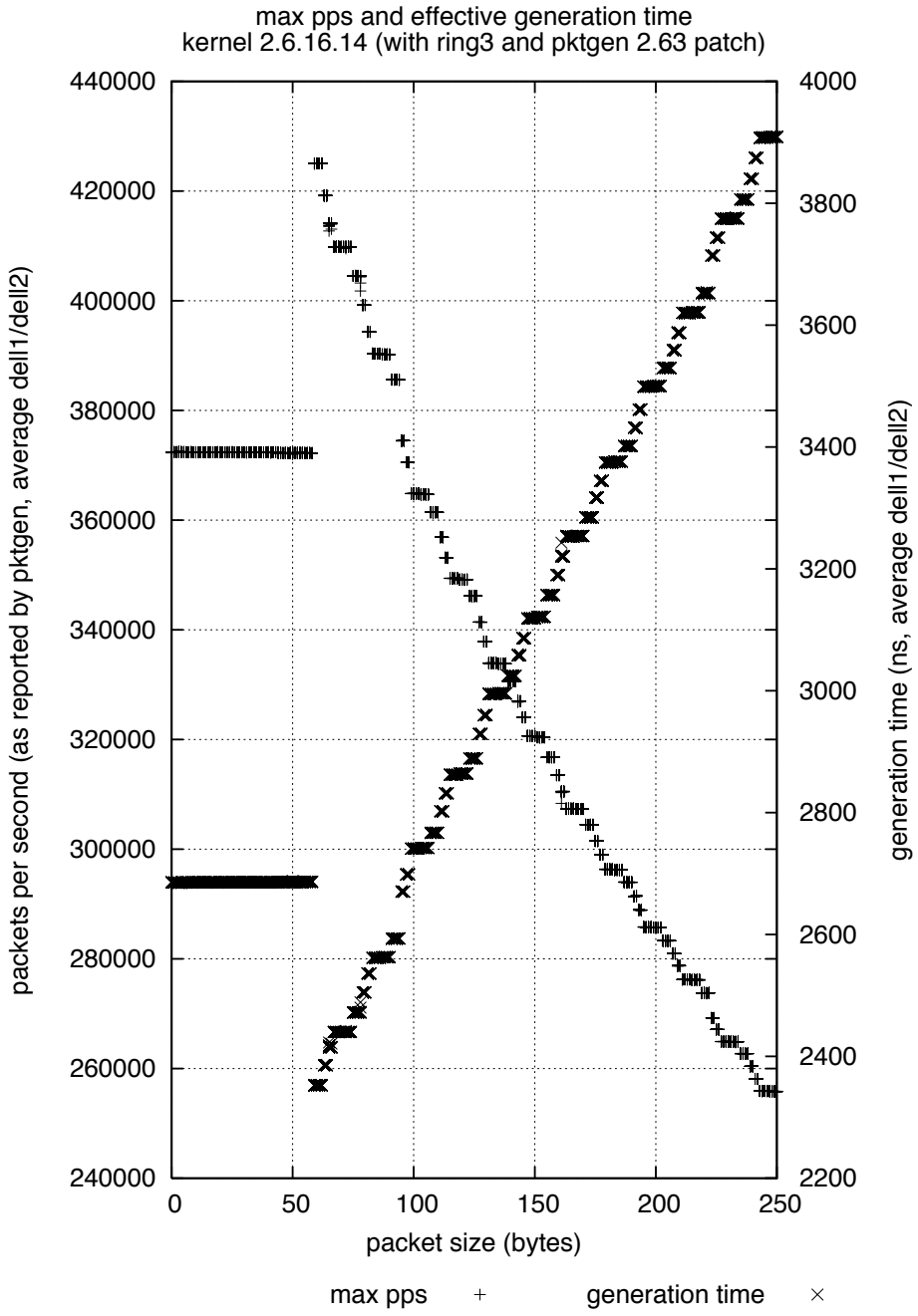


Figure 4.2: Max packets per second and generation time

.

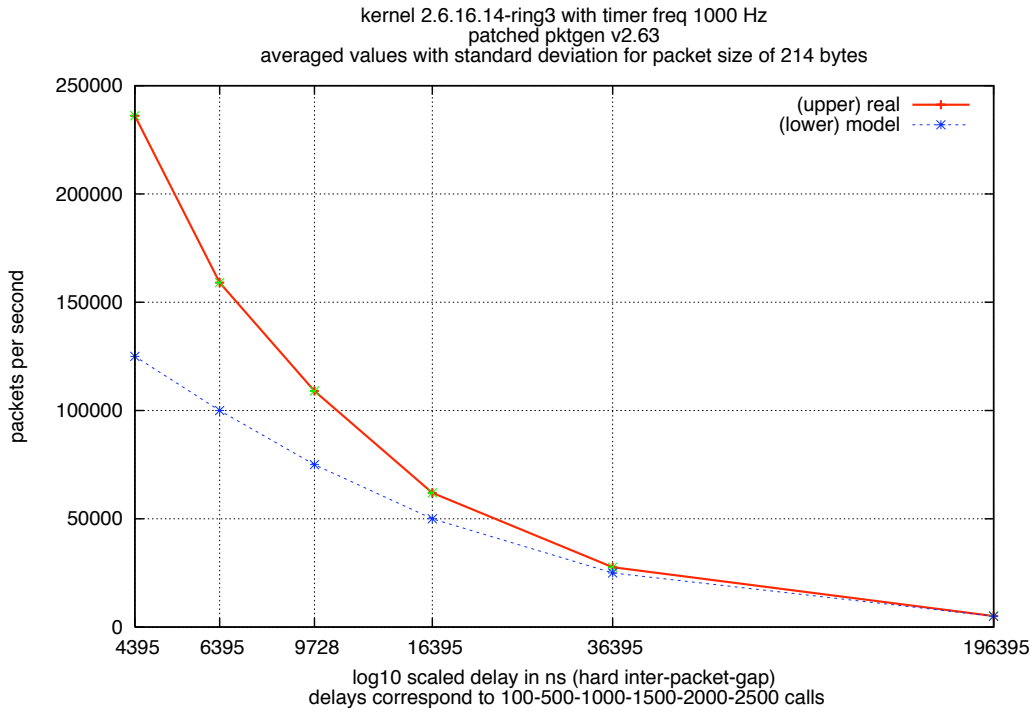


Figure 4.3: Smooth generation rates after removing `printk` and setting timer frequency to 1,000 Hertz

also a stable max packet per second rate. Once over this threshold, the generation time drops to circa 2,350 ns and increases linearly.

Unfortunately, new measurements with the fixed pktgen showed that also the second model using "generation time" for correlating pktgen's *delay* and *count (total packets to send)* with the more user-friendly *calls* and *duration* was far from accurate. As it can be seen in figure 4.3., the real packet per second rate is almost the double of the calculated one.

Since the antecedent mentioned hardware problems had delayed the project considerably, no more resources could be used to improve the model further.[5]. Yet, the inaccuracy of the model had to be addressed: since `pktg-conf-voip.sh`

[5] A probably more precise approach would be to analyze pktgen's "active" and "idle" time counters, since the problem seems to be the sleep-behavior. According to Olsson, optimizing sleeping behavior is work-in-progress.

underestimates the real packet per second rate by almost 50% when using short delays, the value of the configuration shell script's *duration* parameter was increased from 7 to 14 seconds. This should ensure a sane close-to-expected packet rate with the desired expected minimal delay.

4.2 Forwarding capacity of the OpenBSD-firewalls

After having looked at pktgen's properties, the attention was directed toward the actual object of interest for FreeCode: the capabilities and behavior of the OpenBSD-firewalls, starting with unidirectional traffic. The *packet filter* was enabled and configured with the previously mentioned simple ruleset. If the bottleneck of the system turned out to be the packet filtering, then it was considered to optimize the ruleset.

There are several different perspectives one can assume for analyzing the behavior. One perspective is the "black box", where the the internals of the firewalls are not looked at, but instead the amount of traffic sent/received is compared. In our case, the switch was queried for seen IPv4 unicast packets on the interfaces in question.

This perspective is adopted for some experiments in order to be able to make a statement about the probable "end-user" experience. Yet, this perspective falls short of giving a complete understanding: there may be multiple factors that can only be discerned if the firewalls themselves are monitored from the inside. As usual it's important to keep in mind: measuring on live hosts uses resources, so this uncertainty needs to be addressed.

4.2.1 Blackbox

The steps for conducting this experiment are as follows:

1. *Run the configuration script:* this step erases old configuration values and sets the new ones.

2. *Run the experiment:* every experiment was run 7 times in order to make a statistically sane assertion about the outcome. The experiment-independent variables were source (*dell1* or *dell2*) and gateway (*firewall1* or *firewall2*). This yields 4 possible combinations per other variable that were supposed to be conducted.[6]

3. *Verify:* verify that the output generated corresponds roughly to the expectation (e.g. file size, number magnitude).

[6]Unfortunately, hardware problems forcing a repetition of all previous experiments did not allow for testing all possible voice codec / media type permutations; instead, the smallest and biggest voice codec (GSM-fast and G.711) were tested.

In order to measure the forwarding capacity, the switch was queried by SNMP for the number of seen IPv4-packets on a given interface for a given direction. The number was queried using a script:

```
./qif-smc.sh ( -fw1 | -fw2 ) -direction ( W2L | L2W )
```

The numbers had to be read twice, once for getting the starting value and at the end for getting the end value per interface. Internally, the switch uses 32-bit counters which wrap at 4,294,967,295; this was taken into account when calculating the difference.

Since the interaction with pktgen is done easily through the shell, the whole command sequence was written as one long concatenation of shell-commands. They are listed in the appendix, section C.3.1; for better understanding, they are reproduced here in pseudo-code:

```
set variables: source, gateway, direction,
 duration, codec, mediatype
loop through call-values from 500 to 3000
 configure pktgen, based on the variables
 loop seven times
  query switch for start values
  start packet-generator
  sleep 5 seconds
  query switch for end values
 repeat
repeat
```

The program returned one line per run, with the following fields:

pktsize The packet size (minus 4 bytes for the Ethernet-CRC).

host_count Count of different source- and destination IP-addresses.

port_count Count of different source-and destination UDP-addresses.

pps Packets-per-second, as reported by pktgen.

mpps Packets-per-second (calculated by the configuration script)

counters_start The start count at the switch.

counters_end The end count at the switch.

dur_us Duration in microseconds, as reported by pktgen.

mdur_us Duration in microseconds (calculated).

calls Amount of calls (parameter).

inter-packet-delay The inter-packet-delay (calculated).

source The sending computer.

gateway The forwarding firewall.

codec The VoIP codec chosen.

mediatype The media type chosen.

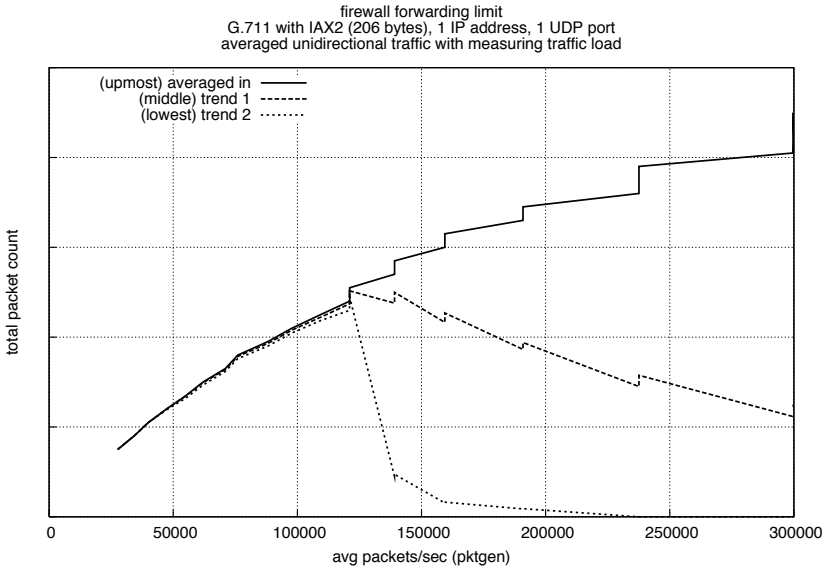
The information was recorded in a text file with a unique name and then fed into a database where calculating the average and standard deviation as well as grouping could be performed easily. First, the numbers were grouped by *source* and *gateway*; if not abnormal numbers were discovered,[7] the the whole data set was averaged. Then, the numbers were visualized with gnuplot for involving the left brain hemisphere even more in the analysis process.

In the already well-known series of unfortunate events, *dell2* became very unstable during the two last weeks of the master project and had to be completely exchanged on May 15th. This is problematic, since it cast doubts on the reliability of the data produced by *dell2* previously. Time only allowed to repeat the most crucial "packet forwarding" experiments (biggest/smallest voice codec) that had been conducted earlier more exhaustively, e.g. with the full voice codec collection.

The firewall capacity was tested – again – with the biggest and smallest codec and with single port and single host traffic in order to minimize the effect on the packet filter. The standard deviation was moderate in all cases ($<$ 2%); figure 4.4 shows that the firewalls start dropping traffic at approximately 125,000 packets per second, corresponding to an inter-packet-delay of approximately 12,500 nanoseconds. Two trends became visible: *trend 1* with the packet forwarding rate dropping more evenly over time, and *trend 2*, where the rate drops quickly to a small fraction of received packets. The reasons for these distinct trends are unknown for the time being, but should be examined more in detail.

Nonetheless, the measurements show that all firewalls independent of traffic source and packet size start dropping packets at around 125,000 packets per second. These results are *surprising* since the forwarding rate is far below the possible traffic generation speed of pktgen, and also much lower than Gigabit Ethernet capacity. With the biggest packet (G.711 with RTP media steam), the forwarded traffic corresponds to 218 MBit/second.

[7] For example a wrapped packet counter on the switch, as mentioned previously.



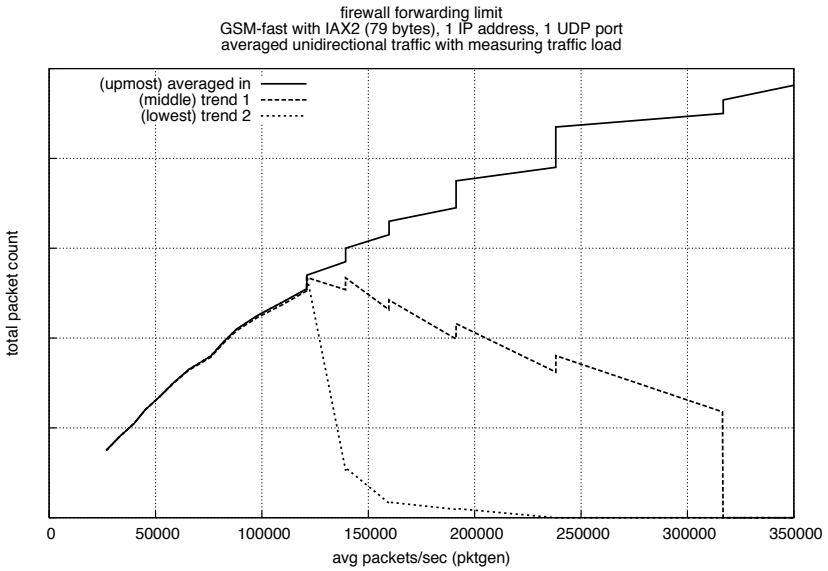


Figure 4.4: Firewall packet forwarding drop around 125,000 packets per second with biggest and smallest codec.

For gaining a better understanding on possible reasons, a look at the inside happenings of the firewalls is needed.

4.2.2 Inside the Firewalls

For the firewalls, the questions of main interest was to find out what the dominant CPU-state of the firewall was under a given load. For monitoring the performance of the packet filter, statistics about state table entries and count of forwarded or blocked packets can be analyzed.

kern.cp_time The `kern.cp_time` parameter was read through sysctl, getting hold of the amount of tics spent in the states *user, nice, system, interrupt, idle*. Since – in analogy to the counter numbers of the switch – also the tics are simply monotonically increasing, they need to be compared to the previous numbers for understanding the change in the system.

pf-statistics The packet filter statistics of the WAN and LAN interfaces were queried by calling the C-program `pfquery-if` (a rip-off of *pfstats*'[8]). This information is obtained by directly reading the packet-filter structures; this would not have been possible with a shell script, and grepping/cutting the output of `pfctl -s info` would most probably have required much more computing.

For collecting this statistics, a shell script was run in an endless loop; after every reading, the script slept for 1 second. The output of these three calls (sysctl, pfquery-if LAN and pfquery-if WAN) was written to a file on the flash card.

In order to have a reasonable amount of precision when measuring, the scripts were started remotely with ssh. Since having an open SSH-connection would generate additional traffic, it was desirable to close the ssh-session to the server while the measurement script was running and before starting the experiment. After some trial and error, it was found that closing the standard output and standard error makes an immediate disconnect possible. For managing this, the load-measurement script had to be wrapped in a "kicker" script that started the load-script with these two file descriptors closed. It was not possible to do that directly in one ssh-call on the command-line.

As soon as pktgen had terminated, the load-script was killed by a calling a *kill.sh* wrapper script via SSH that terminated the running measuring process.

[8] `http://www.benzedrine.cx/pfstat.html`

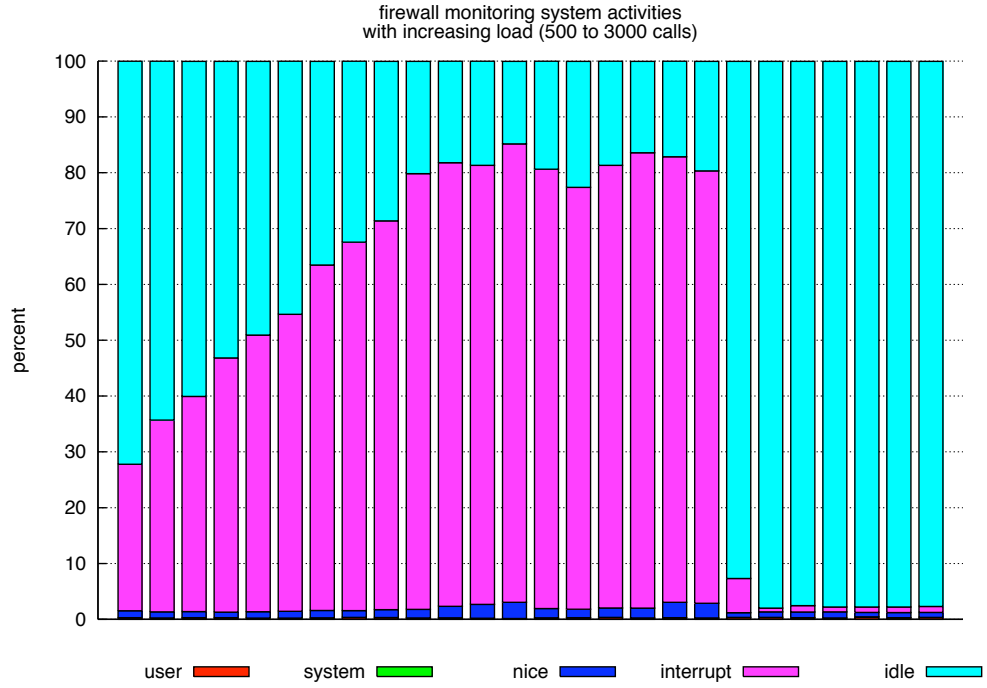


Figure 4.5: OpenBSD CPU-states under increasing load

Due to the aforementioned *dell2* hardware – and ensuing time – problems, the load behavior was only analyzed based on the batch of tests from *dell1*, but again with the biggest and smallest codec. The drop rate was the same as without running the measuring scripts, which allows for the conclusion that the resource measuring does not have a significant impact on the firewall' s overall performance.

In contrast, figure 4.5 shows the result graph of CPU-states and gives a plausible explanation of the unstable behavior when increasing the load: the system spends more and more time in *interrupt*-mode and all resources are bound, independently of the packet size of received traffic.

The reader will notice the significant increase of *idle* state towards the end of the test, where the load is heaviest. This is very unexpected. In order to get an explanation for this phenomenon, the load statistic files were analyzed with focus on *how many measurements were taken during one run, and in what intervals*. Since the "kill"-script was not started until 5 seconds had elapsed after pktgen finished, the last 5 measurement lines were disregarded (not so for the CPU-states!). This calculation yielded the data visualized in figure 4.6 and was the key to understanding why the firewall seems to be idle, yet drops packets: *the more the firewall is loaded, the less measurements are taken.* The measuring process is simply *starved* – the system is just busy handling the interrupts. So when the the load is highest, no information about the CPU states is recorded, but as soon as the 5-second-pause is in effect, the measuring process gets CPU time again. The approximately 2,400 "calls" – calculated with equation (3.2)– correspond to the 120,000 packets per second as reported by pktgen. The measuring process is starved, but still a large amount of packets is forwarded until we reach 125,000 pps (2,500 calls).

Here, the circle to the open question of OpenBSD mechanisms for avoiding "receive livelock" situations closes. The performance is disappointing since no special handling seems to be implemented at present.

A batch of bi-directional traffic tests involving all 32 combinations of *firewall, source computer, codec and media stream type* variables had been performed before the aforementioned *dell2* problems surfaced. The testing procedure, described on page 56, was extended by starting the tests on both machines simultaneously. Frustratingly, a several-hour analysis of the data identified the numbers as unusable: there was no recognizable pattern at all, and the numbers did not make any sense. Impossible packet forwarding rates were sighted, bigger then Gigabit Ethernet allows, and other weird phenomena. As a consequence, a closer look at bi-directional traffic is left for the future.

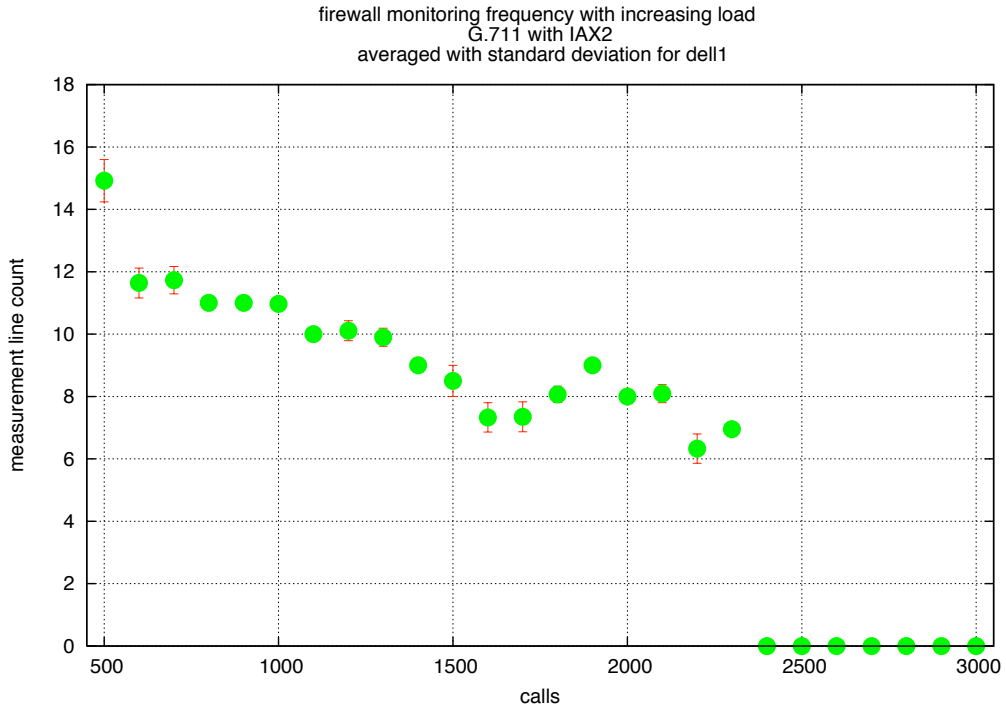


Figure 4.6: Measurement process starvation with increasing load

4.2.3 Behavior when failing over

Taking down a firewall can have two reasons: either administrative for maintenance, or unexpected due to a failure. OpenBSD's FAQ-document on CARP and pfsync[9] explains that turning off CARP manually with `ifconfig carpN down` "will cause the master to advertise itself with an 'infinite' `advbase` and `advskew`. The backup host(s) will see this and immediately take over the role of master." When the firewall simply "vanishes", it takes about three seconds before the new master starts forwarding traffic again.

Quick tests done in the beginning of the project confirmed this. While sending 10 packets per second, the master firewall was taken down administratively. A count of received packets showed that all packets arrived consistently. When taking the firewall down by power-cycling it, about 30 packets were lost with a standard deviation of 1 packet, corresponding to the aforementioned 3 second pause.

For testing failover effects with higher volume, a *delay of 12,500 nanoseconds*was chosen, yielding a packet rate of 81,600 packets per second. This is well under the maximum forwarding capacity so that the firewall does not suffer resource starvation. Sending 1,000,000 packets takes 12.25 seconds. pkgten was configured with *mediatype rtp* with 3,000 hosts in order to have unique sockets. It was verified before running the tests that the firewall actually forwards all 1,000,000 packets and the receiver is able to receive and write them to the dump-file.

The test were conducted seven times, and the steps are as follows:

- turn off CARP on both firewall, then on at the master firewall, then on at the backup
- start tcpdump on the receiver with no filter
- start packet generator
- after 6 seconds, either (a) turn off CARP administratively or (b) power-cycle the master firewall
- when pktgen returns, kill tcpdump

As a first analyzing step, the UDP-packets in the dump file were counted and values averaged. The resulting interruption time ("glitch") of network connectivity was calculated as follows:

$$glitch = \frac{1,000,000\,[total\ packets] - received\ packets}{sent\ packets\ per\ second_{average} \times 1,000,000\,[\mu sec]} \tag{4.4}$$

[9]http://www.openbsd.org/faq/pf/carp.html

source	master	received	glitch
dell1	firewall1	998,786	14.87
dell1	firewall2	980,279	241.48
dell2	firewall1	999,948	0.636
dell2	firewall2	973,199	328.17

Table 4.2: Received packet count for soft-failover with 81,667 ($\pm$ 0.04%) pps and 214B packets, with network glitch in milliseconds.

source	master	received	glitch
dell1	firewall1	761,164	2,924
dell1	firewall2	778,084	2,717
dell2	firewall1	983,734	199
dell2	firewall2	829,688	2,085

Table 4.3: Packet receive rate for hard-failover (power-cycling) with 81,661 ($\pm$ 0.04%) pps and 214B packets, with network glitch in milliseconds.

Table 4.2 summarizes the numbers when turning off CARP administratively. The standard deviation for received packets was between 2 and 8%. Interestingly, when *firewall1* was master (and *firewall2* backup), the network glitch was much shorter than when the roles were reversed. Especially spectacular is the extremely low packet loss value for *dell2* sending packets and *firewall2* being backup! Repeating the experiments confirmed that this is a trend and not just a one-time phenomenon. The reason for this is unknown at the time, but does not seem to be dependent on the computers generating traffic since both *dells* were sending packets at basically the same rate. The interruption of network connectivity is indeed minimal when doing a "soft-failover" - less than one millisecond in best-case and 300 milliseconds in worst-case.

As mentioned earlier, VoIP traffic pauses bigger than 150ms can be perceived by the call participants.[4, 23, 27, 9, 1] On average, taking administratively down the current master firewall will cause a short audible hiccup, but since the traffic flow continues normally afterwards, is seems to be tolerable from an end-user perspective.

Also the results of hard-failovers (power-cycling the current master) are of interest. Table 4.3 contains the exact numbers; also here, the standard deviation was between 2 and 8%, but there is no obvious interrelation to the standard deviation for soft-failovers.

Since [47] states that a network interruption of about 3 seconds is to be expected when the master firewall "vanishes", the obtained numbers displayed a better behavior. The traffic was interrupted by about 2 to 3 seconds with the

exception of *dell2* (traffic source) and *firewall2* as backup (table-row 2), outperforming the others combinations by far. Also in this case, more repetitions confirmed the trend, with not clear cause. Since the other series yielded a much higher average, it is probably safer to assume that these reflect a real-world scenario more realistically. Such a long break is irritating for VoIP participants, but since hard failovers should be rare, it does not disrupt the service beyond reason.

One uncertainty factor is the non-bursty nature of the traffic generated by pktgen running at this "low rate". More bursty traffic could reveal a more stable failover-behavior, but with higher packet loss numbers.

For further analysis, the tcpdump-file was fed to the `d2t.sh` script. As mentioned before, the speed is abysmally slow; yet this was not the biggest problem. A quick glance at the output of `d2h` revealed that pktgen neither did increase the sequence number nor the IP-address or UDP-port. A countercheck with pktgen versions 1.3 and 2.58 on two kernels (2.6.16.14 with PF_RING patch and 2.6.11.12 with KUTE patch), as well as version 1.4 with kernel 2.6.10 (with Ubuntu-patches) displayed the same behavior. Consequently, further analysis would have to concentrate on time-stamp analysis – with *device polling*-biased pcap-stamps, and possibly also irregular pktgen-timestamps[10] – but in view of the limited information value it was decided not to process the data any further.

At present, it is unclear what the reason for this additional weird behavior of the packet generator might be. This discovery would have significantly relativized and state-statistics and performance information for the *packet-filter*. Since all traffic – probably – was sent with only one source- and destination-IP-address, as well as single UDP-port, most of the experiments would have to be repeated to get correct values.

Due to "receive livelock" being the main performance stoppage, analysis of the *packet-filter* numbers and thereby trying to classify the influence of the filtering process was considered superfluous at the moment. If the interrupt-handling problem can be mitigated, then this pktgen-bug needs to be addressed *before* any experiments are designed or repeated.

[10] By now, the trust in pktgen and the hardware had sunken to a *very* low level!

Chapter 5

Discussion and Conclusions

Most of the results of the experiments have been commented already in chapter 4. During the 17 weeks available for this thesis, the questions asked have changed several times, and have a big span width. So, what conclusions can be drawn? For answering this, the questions from the introduction that have shown themselves to be relevant are taken up again:

Can VoIP traffic be generated with a moderate set of commodity hardware? Yes, this is possible with certain restrictions. Pktgen, the Linux kernel UDP-packet generator, was used for generating artificial traffic. The "recently" reworked version of pktgen (2.63, included in the Linux kernel 2.6.16.14) seems to be "work-in-progress", and cannot be considered stable. Many problems were encountered, and both the traffic approximation model and the experiments themselves need to be revisited.

How much traffic can be generated? This depends on the packet size. Since all VoIP packets have a very small packet size, the producible loads are fractions of the Gigabit Ethernet capacity. Values range from 45% (G.711 codec with RTP media type, packet size of 218 bytes) to 24.8% (GSM fast with IAX2 media type).

Which properties must synthetic/artificial VoIP packets have in common with real ones? A relatively simple method has been presented for generation artificial, synthetic traffic that has common properties with true VoIP traffic. Variations in *packet size* mimic different voice codecs (G.711/G.726/GSM-fast/GSM-slow) and media stream types (RTP/IAX2), as well as *variation in IP-addresses and UDP-ports* for simulating different numbers of hosts.

What is the upper forwarding limit for the OpenBSD-firewalls? What are the limiting factors? The upper limit for one firewall for unidirectional traf-

fic was found to be about 125,000 packets per second, independent of packet size in the range from 83 to 218 bytes. This corresponds to about 2,500 VoIP calls with voice codecs sending 50 packets per second, no control traffic (SIP, IAX2, RTCP) included in the calculation.

The firewalls suffer from "receive livelock", system overload due to lots of interrupt request caused by incoming packets. The problem is aggravated by the typical small payload packets size (<= 160 bytes); the interrupt handling consumes all system resources and leaves the firewall in an unstable forwarding state.

What can be said about network interruption in failover situations, either administratively (soft-failover), or by power-cycling (hard-failover)? While sending 81,500 pps, turning the current CARP-master administratively off lead to a network glitch between one millisecond in best-case and 300 milliseconds in worst-case. This causes a perceivable hiccup in a conversation, but since the traffic flow continues normally afterwards, is seems to be tolerable from an end-user perspective.

When power-cycling the active master, the traffic was interrupted for about 2 to 3 seconds. Such a long break is irritating for VoIP participants, and may lead to isolated individuals ringing off, but since hard failovers should be rare, it does not disrupt the service beyond reason.

During the project, new questions surfaced:

How can the pktgen-interface be made more user-friendly for simulating VoIP traffic? An approximative approach for correlating user-friendly *call count and duration simulation parameters* with pktgen's *total packet count and inter-packet-gap* was proposed. A first and a second model were implemented and tested; the latter is still off by 50% when it comes to real vs. expected packet rates. A more precise model is needed, but depends on a more stable "sleep" behavior of pktgen.

How can pkgten's payload yield useful information for stream analysis? The *sequence-number* in the payload of every pktgen-packet together with the IP-addresses and UDP-ports could be used to correlate packets to "connections" and thereby enabling a statement about the probable end-user experience. When combined with the analysis of both the pcap-timestamp and the timestamp in pktgen's packet, an solid analysis of packet loss and inter-arrival-times (jitter) is possible.

The unstable nature of ptkgen inhibited the successful analysis. The sequence number was not increased, adding one major "bug" to pktgen. It cannot be excluded that the problem is rooted in other code than pktgen itself.

On a more general level, the accomplished measurements allow for questioning the suitability of the OpenBSD operating system for *firewalls in high packet rate networks.* Using specialized, expensive server hardware (running OpenBSD) as "dedicated VoIP firewall" seems to be a waste of resources; such a hardware/operating system combination could better be used in networks with more heterogeneous packet sizes, since the bigger the packet size, the fewer packets per second there are. Having fewer but bigger packets may weaken the "receive livelock" problem.

5.1 Future Work

Many open questions and possible research areas were encountered; in the author's opinion, the most interesting ones for future research are these (with decreasing importance):

- Improving the user-friendliness of pktgen's interface for simulating VoIP traffic, especially making the

  $$function f(calls, duration) = (delay, total\ packet\ count)$$

  more accurate.

- Repeating the experiments with the firewalls running an operating system that has a proven solution to the "receive livelock" problem. FreeBSD (with CARP and pfsync) and Linux (with uCarp and ct_sync) can be patched with device polling enhancements.

- Retesting the forwarding capacity if the OpenBSD-community comes up with solutions for "receive livelock" for Gigabit Ethernet cards.

- Enhancing the model of artificial/synthetic VoIP traffic by using Fabian Schneider's enhanced `pktgen`, supporting a statistical packet size distribution. This allows for including signaling and control traffic, different codecs simultaneously, and even IAX2-trunking: since one Asterisk "meta-frame" contains payloads from multiple connections, the packet sizes could reach a size where "receive livelock" is not the main inhibitor any longer.

- Analyzing jitter behavior when a failover occurs; once running on a recent kernel, the KUTE-receiver seems to be an interesting candidate for classifying the inter-arrival times, as well as PLAB in its next version.

- If the "receive livelock" problem is solved, does the firewall have spare resources to do NATting, or acting as a secure gateway for IPsec or SRTP?

Bibliography

[1] Phil Sherburne and Cary Fitzgerald. You Don't know Jack about VoIP. *ACM Queue*, 2(6):30–38, September 2004.

[2] C.A. Polyzois, H.K. Purdy, Yang Ping-Fai, D. Shrader, H. Sinnreich, F. Menard, and H. Schuzerinne. From POTS to PANS: a commentary on the evolution to Internet telephony. *IEEE Network*, 13(3):58–64, May/June 1999.

[3] Upkar Varshney, Andy Snow, Matt McGivern, and Christi Howard. Voice over IP. *Communications of the ACM*, 45(1):89–96, January 2002.

[4] Aleš Vigrinec and Sašo Tomažič. IP telephony from a user perspective. *Proceedings of the 10th Mediterranean Electrotechnical Conference (Melecon)*, 2:344–7, 2000.

[5] Maurice David Woernhard. VoIP – Next Generation Telephony. Literature Survey, Høgkolen i Oslo, Oslo, Norway, May 2005. `http://cube.iu.hio.no/~s117181/voip-survey.pdf`.

[6] Alan E. Frey and Guy J. Zenner. The Role of SIP in the Migration of Service Provider Networks to VoIP. *Bell Labs Technical Journal*, 9(3):199–216, 2004.

[7] Steven Cherry. Seven Myths About Voice over IP. *IEEE Spectrum*, pages 53–7, March 2005.

[8] A. Dutta-Roy. The cost of quality in Internet-style networks. *IEEE Spectrum*, 37(9):57–62, September 2000.

[9] Bur Goode. Voice over Internet Protocol (VoIP). In *Proceedings of the IEEE*, volume 90, pages 1494–1517. IEEE, Sep 2002.

[10] Ulysses Black. *Voice over IP*. Prentice Hall, 2002.

[11] Ligang Wang, Anjali Agarwal, and J. William Atwood. Modelling and verification of interworking between SIP and H.323. *Computer Networks*, 45:77–98, 2004.

[12] Hong Liu and Petro Mouchtaris. Voice over IP Signaling: H.323 and Beyond. *IEEE Communications Magazine*, 38(10):142–148, 2000.

[13] Ted T. Kwon, Mario Gerla, Sajal Das, and Subir Das. Mobility Management for VoIP Service: Mobile IP vs. SIP. *IEEE Wireless Communications*, 9(5):66–75, October 2002.

[14] Pawan Goyal, Albert Greenberg, Charles R. Kalmarek, William T. Marshall, Partho Mishra, Doug Nortz, and K.K. Ramakrishnan. Integration of Call Signalling and Resource Management for IP Telephony. *IEEE Internet Computing*, 3(3):44–52, May 1999.

[15] Sudir R. Ahuja and Robert Ensor. VoIP: What is it good for? *ACM Queue*, 2(6):48–55, September 2004.

[16] Mark Spencer and Frank W. Miller. IAX Protocol Description. *http://www.cornfed.com/iax.pdf*, March 2004.

[17] J. Andren, M. Hilding, and D. Veitch. Understanding end-to-end Internet traffic dynamics. In *IEEE Global Telecommunications Conference 1998 (GLOBECOM 98)*, volume 2, pages 1118–22, 1998.

[18] O. Hagsand, K. Hanson, and I. Marsh. Measuring Internet telephony quality: where are we today? In *Global Telecommunications Conference*, volume 3, pages 1838–42, 1999.

[19] W. Kampichler and K.M. Goeschka. Measuring voice readiness of local area networks. In *IEEE Global Telecommunications Conference 2001 (GLOBECOM '01)*, volume 4, pages 2501–5. IEEE, 2001.

[20] B. Duysburgh, S. Vanhastel, B. De Vreese, C. Petrisor, and P. Demeester. On the influence of best-effort network conditions on the perceived speech quality of VoIP connections. In *Proceedings of the 10th International Conference on Computer Communications and Networks 2001*, pages 334–9, 2001.

[21] Paul Ferguson and Geoff Huston. *Quality of Service: Delivering QoS on the Internet and in Corporate Networks*. Wiley, New York, 1998.

[22] W. Kampichler and K.M. Goetschka. A light-weight measuring method for QoS performance of IP networks. In *Proceedings of the PDCS 2000*, pages 297–301, 2000.

[23] S. Zeadally, F. Siddiqui, and P. Kubher. Voice over IP in intranet and Internet environments. In *Communications of the IEEE Proceeings*, volume 151, pages 263–269, June 2004.

[24] Douglas C. Sicker and Tom Lookabough. VoIP Security: Not an Afterthought! *Queue*, 2(6):56–64, September 2004.

[25] Michael Stukas and Douglas C. Sicker. An Evaluation of VoIP Traversal of Firewalls and NATs within an Enterprise Environment. *Information Systems Frontiers*, 6(3):219–228, 2004.

[26] International Telecommunication Union. ITU Recommendation G.114: One-way transmission time. Technical report, ITU, 1996.

[27] P. Denisowski. How does it sound? [voice clarity analysis]. *IEEE Spectrum*, 38(2):60–4, February 2001.

[28] International Telecommunication Union. ITU Recommendation P.800 (08/98): Methods for subjective determination of transmission quality. Technical report, ITU, 1998.

[29] International Telecommunication Union. ITU Recommendation P.830: Subjective performance assessment of telephone-band and wideband digital codecs. Technical report, ITU, 1996.

[30] International Telecommunication Union. ITU Recommendation P.862 (02/2001, superseeds P.861): Perceptual evaluation of speech quality (PESQ). Technical report, ITU, 2001.

[31] T. Kushida. The traffic measurement and the empirical studies for the Internet. In *IEEE Global Telecommunication Conference (GLOBECOM 98)*, volume 2, pages 1142–7, 1998.

[32] A. Lakaniemi, J. Rosti, and V.I. Räisänen. Subjective VoIP speech quality evaluation based on network measurements. In *IEEE International Conference on Communications ICC 2001*, volume 3, pages 748–52, 2001.

[33] Ray Peckham. 911 over VoIP: whose responsibility? *Communication News*, 41(7):6–+, 2004.

[34] Bård M. Bergersen. Spam. *Network and System Administration: Researy Surveys*, 1(24), 2004.

[35] K. Anderberg. SPIT on VoIP. *Communications News*, 42(1):4, 2005.

[36] D. Roth. Catch us if you can. *Fortune*, 149(3):64+, 2004.

[37] J. Karlin. Skype hunt. *Fortune*, 149(5):34, March 2004.

[38] D Sweeney. Sum Of All Fears [peer-to-peer technology]. *America's Network*, 108(17):16–20, November 2004.

[39] Roxanne Khamsi. Skype Beyond The Hype. *Technology Review*, 107(5):44–7, June 2004.

[40] Dennis Bergström. An Analysis of Skype VoIP application for use in a Corporate Environment. based on Skype 1.0.0.29; availabe online at http://www.geocities.com/bergstromdennis/, October2004.

[41] M. Popovic and V. Kovacevic. An Approach to Internet-Based Virtual Call Center Implementation. In *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2001. University of Novi Sad.

[42] Masayuki Kumazawaa, Taisuke Matsumoto, Shinkiechi Ikeda, Makoto Funabiki, Hirokazu Kobayasi, and Toyoki Kawahara. Router Selection for Moving Networks. In *First IEEE Consumer Communications and Networking Conference (CCNC)*, 2004.

[43] Anton Batchvarov. Security Issues and Solutions for Voice over IP compared to Circuit Switched Networks. INFOTECH Seminar Advanced Communication Services, University of Stuttgart, Germany.

[44] Nils Ohlmeier. Design and Implementation of a High Availability SIP Server Architecture. Technical University Berlin, July 2003.

[45] Fabian Schneider. Analyse der Leistung von BPF und libpcap in Gigabit-Ethernet Umgebungen. Ausarbeitung zum Systementwicklungsprojekt, Technische Universität München, Munich, Germany, Oktober 2004. `http://www.net.in.tum.de/~schneifa/papers/sep_ausarbeitung_sep.ps`.

[46] Fabian Schneider. Performance evaluation of packet capturing systems for high-speed networks. Diplomarbeit, Technische Universität München, Munich, Germany, November 2005. `http://www.net.in.tum.de/~schneifa/papers/da.ps`.

[47] Ryan McBride. Firewall Failover with pfsync and CARP. http://www.countersiege.com/doc/pfsync-carp/.

[48] W. Feng et al. TCPivo: A High-Performance Packet Replay Engine. In *ACM Sigcomm Workshop*, 2003.

[49] Robert Olsson (Uppsala University). pktgen the linux packet generator. In *Proceedings of the 2005 Linux Symposium, Ottawa, Canada*, July 2005.

[50] Sebastian Zander, David Kennedy, and Greenville Armitage. KUTE – A High Performance Kernel-based UDP Traffic Engine. *CAIA Technical Report*, January 2005.

[51] Ted Wallingford. *Switching to VoIP*. O'Reilly, 2005.

[52] Jim Van Meggelen, Jared SMith, and Leif Madsen. *Asterisk - The Future of Telephony*. O'Reilly, 2005.

[53] Luca Deri. Improving Passive Packet Capturing: Beyond Device Polling. In 4^{th} *International System Administration and Network Engineering Conference*, 2004.

[54] Loris Degioanni, Mario Baldi, Fulvio Risso, and G. Varenni. Profiling and Optimization of Software-Based Network-Analysis Applications. In *Proceedings of the* 15^{th} *IEEE SBAC-PAD 2003 Symposium*, 2003.

[55] I. Kim, J. Moon, and H. Y. Yeom. Timer-based Interrupt Mitigation for High Performance Packet Processing. In *Proceedings of the* 5^{th} *International Conference on High-Performance Computing in the Asia-Pacific Region*, 2001.

[56] Luca Deri. nCap: Wire-speed Packet Capture and Transmission. In *Proceedings of E2EMON 2005*, May 2005.

[57] Teruyuki Hasegawa, Tomohiko Ogishi, and Toru Hasegawa. A Framework on Gigabit Rate Packet Header Collection for Low-cost Internet Monitoring Systems. In *IEEE International Conference on Communications ICC 2002*, volume 4, pages 2206–2211, 2002.

[58] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: Faily Fast Packet Filters. In *6th Symposium on Operating Systems Design and Implementation OSDI 2004*, pages 347–362, 2004.

[59] Se-Hee Han, Myung-Sup Kim, Hong-Taek Ju, and James Won-Ki Hong. The Architecture of NG-MON: A Passive Network Monitoring Systems for High-Speed IP Networks. In *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2002.

[60] Emily Hollis. Monitoring and Analyzing VoIP Traffic. *Certification Magazine*, February 2005.

[61] Luca Deri. Open Source VoIP Traffic Monitoring. In *Proceedings of SANE 2006 (to be published)*, 2006.

[62] S. Chatterjee, B. Tulu, T. Abhichandani, and H. Q. Li. SIP-based enterprise converged networks for voice/video-over-IP: Implementation and evaluation of components. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 23(10):1921–1933, Oct 2005.

[63] Alberto Dainotti and Antonio Pescapé. Plab: a packet capture and analysis architecture. http://www.grid.unina.it/Traffic/pub/TR-DIS-122004.pdf, December 2004.

Appendix A

pktgen

While experimenting with pktgen, it was discovered that the packet rate generation drops significantly when arriving at at delay value of 1,000 nanoseconds (figure A.1). In response to an SOS mail to Olsson, he answered on May 8th:

> OK Något är skumt.... Lennert Bytenheack hackade för att inte spinna bort CPU medans vi väntade.
> Kolla i spin() pktgen.c
> Ta bort prink-raden.
> Testa.
> Vad har du HZ till? Sätt HZ=1000.

So a close look at the code in question (starting with line 1645 in pktgen.c, kernel 2.6.16.14) revealed:

```
1645 static void spin(struct pktgen_dev *pkt_dev, __u64 spin_until_us)
1646 {
1647   __u64 start;
1648   __u64 now;
1649
1650   start = now = getCurUs();
1651   // printk(KERN_INFO "sleeping for %d\n", (int)(spin_until_us - now));
1652   while (now < spin_until_us) {
1653   /* TODO: optimize sleeping behavior */
1654     if (spin_until_us - now > jiffies_to_usecs(1)+1)
1655       schedule_timeout_interruptible(1);
1656     else if (spin_until_us - now > 100) {
1657       do_softirq();
1658       if (!pkt_dev->running)
1659         return;
1660       if (need_resched())
```

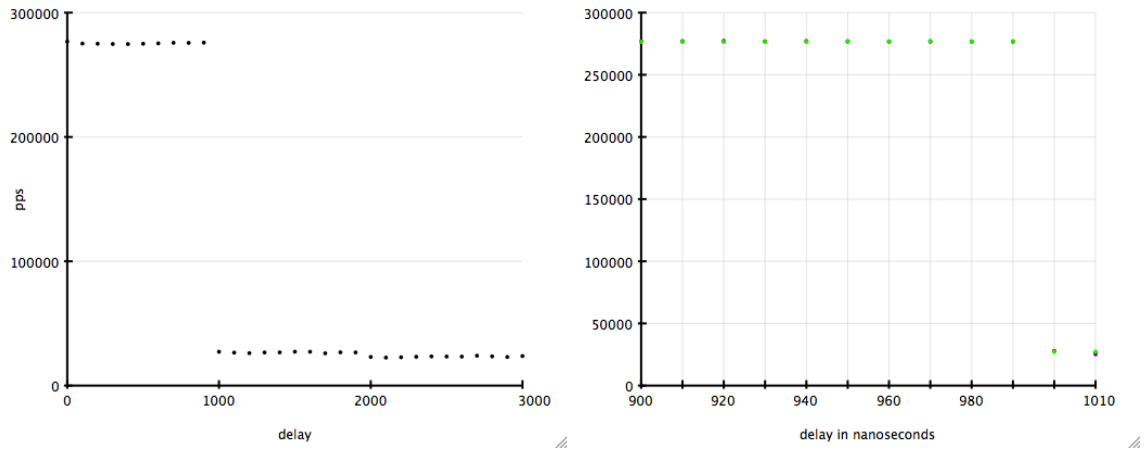


Figure A.1: Dramatic drop in packet generation rate per second at 1,000 ns

```
1661            schedule();
1662        }
1663
1664        now = getCurUs();
1665    }
1666
1667    pkt_dev->idle_acc += now - start;
1668 }
```

After commenting out the `pkrintk` line and recompiling the kernel with a timer frequency of 1,000 Hertz, pktgen's packet generation rate became stable.

However, doubts on the *real* stability and maturity of pktgen in it current version arose. Both the the *TODO: optimize sleeping behavior* comment in the pktgen-source and Olsson's mail suggest that this is *work in progress* and might be subject to change at any time. This situation of uncertainty poses a rather serious challenge to a simple yet accurate simulation of VoIP traffic.

Appendix B

Configuration Files

B.1 *pf* configuration file (firewall1)

```
# PF-conf for FW1
# Versions:
# 2006-03-20  0.01  initial draft, rip-off delivered version
# 2006-03-21  0.02  cont'd
# 2006-04-16  0.03  working on it
# 2006-04-25  0.4   wrong logical operator for udp range
# 2006-05-03  0.41  added udp timeouts

# interfaces
wan_if="bge0"
lan_if="bge1"
sync_if="sk0"
# addresses
voip_lan="10.2.0.0/16"
wan_ip="10.0.1.0/24"
lan_ip="10.0.2.0/24"

# voip protocols
sip="5060"
iax2="4569"
# voip payload
rtp_from="19999"
rtp_to="30001"

set skip on { lo }
set limit { states 50000, frags 20000 }
set block-policy return

#scrub incoming from WAN - not sure if I wanne use that
#scrub in all on $wan_if

# natting - not for now
#scrub in all fragment reassemble no-df
#nat on $ext_if from !($ext_if) -> ($ext_if:0)

# default policy
block in on $wan_if
pass in on $lan_if
pass out on $wan_if keep state

# pfsync and carp - filter on the physical interfaces!
pass quick on { $sync_if } proto pfsync
pass on { $wan_if $lan_if } proto carp keep state

# enable ssh
pass in on { $wan_if $lan_if } proto tcp to any port ssh flags S/SA keep state

# enable UDP/RTP between 20,000 and 30,000 for bi-directional traffic
# set aggressive udp timeouts
pass in on $wan_if proto udp from any to $voip_lan port $rtp_from >< $rtp_to keep state (udp.first 5 udp.single 15 udp.multiple 15)
pass in on $lan_if proto udp from $voip_lan to any port $rtp_from >< $rtp_to keep state (udp.first 5 udp.single 15 udp.multiple 15)

# enable IAX2 - since IAX uses same port for controlling, use longer initial timeouts
pass in on $wan_if proto udp from any port $iax2 to $voip_lan port $iax2 keep state (udp.first 45 udp.single 15 udp.multiple 15)
pass in on $lan_if proto udp from $voip_lan port $iax2 to any port $iax2 keep state (udp.first 45 udp.single 15 udp.multiple 15)
```

Appendix C

Scripts

All scripts can be found at `http://student.iu.hio.no/~s117181/thesis.tgz`

C.1 `pktg-conf-voip.sh` – modeling VoIP characteristics

The script is several hundred lines long and is included in the above mentioned tgz-file.

C.2 `d2h.sh` – consolidate `tcpdump` text output

```
1  #!/bin/bash
2
3  # extracts output fed from tcpdump to one-text liners
4  # 2006-04-28  0.1   initial version
5  # 2006-05-01  0.15  more stuff (first working version)
6
7  # returns
8  # pcap-sec pcap-usec src-ip.udp dst-ip.udp seq pktg-sec pktg-usec
9
10 # expects input from tcpdump -nNXttr dumpfile
11
12 myself=`basename $0`
13 prefix=""; postfix=""
14 if [ "x$1" != "x" ]
15 then
16  postfix=" $1"
17 fi
18 if [ "x$2" != "x" ]
19 then
20  prefix="$2 "
21 fi
22
23 gotsocket="no"
24 ispktgen="no"
25 skiprest="no"
26
```

```
# dehex - concatenate args and convert to decimal
dehex () {
 local hex=""
 for arg in $*
 do
  hex="${hex}${arg}"
 done
 printf '%d\n' "0x${hex}"
 return 0
}

while read line
do
 linea=( $(cut -d " " -f 1- <<< $line) )
 linetype=${linea[0]}
 if [[ "${linetype:0:2}" = "0x" && "$skiprest" = "no" ]] ; then
  # payload - does it belong to a socket?
  if [ "$gotsocket" = "no" ] ; then
   # abort, skip packet since payload
   skiprest="yes"
  else
   case $linetype in
    0x0000: )
     ;;
    0x0010: )
     # check for pktgen-magic
     if [[ "${linea[7]}" = "be9b" && "${linea[8]}" = "e955" ]] ; then
      ispktgen="yes"
     else
      # not pktgen
      ispktgen="no"
      packet=""
      gotsocket="no"
     fi
     ;;
    0x0020: )
     # get seq, sec, usec
     if [ "$ispktgen" = "yes" ] ; then
      seq=`dehex ${linea[1]} ${linea[2]}`
      sec=`dehex ${linea[3]} ${linea[3]}`
      usec=`dehex ${linea[5]} ${linea[6]}`
      packet="$packet $seq $sec $usec"
      echo "${prefix}${packet}${postfix}"
     fi
     packet=""
     gotsocket="no"
     ispktgen="no"
     ;;
    * )
     # set ispktgen to false since fields must come after ispkgen = true
     skiprest="yes"
     ;;
   esac
  fi
 elif [ "${linetype:0:2}" != "0x" ] ; then
  # first line of packet - get timestamp and socket
  gotsocket="yes"
  skiprest="no"
  # fields: time from-socket > to-socket: ...
  pcapsec=$(cut -d . -f 1 <<< ${linea[0]})
  pcapusec=$(cut -d . -f 2 <<< ${linea[0]})
  socket2=$(cut -d : -f 1 <<< ${linea[4]})
  packet="$pcapsec $pcapusec ${linea[2]} $socket2"
  #echo "socket ($packet)"
 fi
```

```
92 done
93
94 exit 0
```

C.3 Huge shell-commands for experiment control and logfile analysis

The beauty of script-coding becomes really visible in these examples. Either, the variables at the beginning of the statements can be changed, or they can be replaced by a `for ...in val1 val2 ; do ...; done` construct. This is useful for looping through multiple computers, firewalls, codecs and mediatypes.

C.3.1 Starting traffic generation

This command has to be started independently on every machine.

```
1  pc="dell1" ; fw="fw1" ; dir="W2L" ; duration="10" ; codec="g711";
2  mediatype="rtp" ;
3  for calls in `seq 500 100 3000` ; do
4    read delay mpps <<< `./pktg-config-voip.sh -codec
5      $codec -mediatype $mediatype -nocarp -${fw} -direction
6      $dir -calls $calls -duration $duration | grep delay |
7      cut -d " " -f 8,10` ;
8    let mdur_us=$duration*1000*1000 ;
9    pktsize=`./xpktsize.sh` ;
10   for try in `seq 1 7` ; do
11     counters_start=`./qif-smc.sh -${fw} -direction $dir` ;
12     pgstart ;
13     sleep 5 ;
14     counters_end=`./qif-smc.sh -${fw} -direction $dir` ;
15     pps=`./xpps.sh` ;
16     count=`./xcount.sh` ;
17     dur_us=`./xdur.sh` ;
18     echo "$pktsize $calls $calls $pps $mpps
19       $counters_start $counters_end $dur_us $mdur_us $calls
20       $delay $pc $fw $codec $mediatype" ;
21   done ;
22 done
23 > blackbox-${fw}-${pc}-${codec}-${mediatype}-callloop.data
```

C.3.2 Starting traffic generation with load analysis on the firewalls

In addition to starting the traffic generation, this command starts and kills the load measuring processes on the given firewall.

```
sleep="5" ; pc="dell1" ; fw="fw1" ; dir="W2L" ; duration="15" ;
codec="g711"; mediatype="iax2" ;
for calls in `seq 500 100 3000` ; do
  read delay mpps <<< `./pktg-config-voip.sh -codec $codec
      -mediatype $mediatype -nocarp -${fw} -direction $dir
      -calls $calls -duration $duration | grep delay
      | cut -d " " -f 8,10` ;
    let mdur_us=$duration*1000*1000 ;
    pktsize=`./xpktsize.sh` ;
    for try in `seq 1 7` ; do
    counters_start=`./qif-smc.sh -${fw} -direction $dir` ;
    ssh root@${fw} /flash/stats/kicker.sh
      ${pc}-${codec}-${mediatype}-nocarp-${calls}calls-try${try}
      $calls &> /dev/null ;
    pgstart ;
    sleep $sleep ;
    ssh root@${fw} /flash/stats/kill.sh &> /dev/null ;
    counters_end=`./qif-smc.sh -${fw} -direction $dir` ;
    pps=`./xpps.sh` ;
    count=`./xcount.sh` ;
    dur_us=`./xdur.sh` ;
    echo "$pktsize $calls $calls $pps $mpps $counters_start
      $counters_end $dur_us $mdur_us $calls $delay
      $pc $fw $codec $mediatype" ;
  done ;
done |
tee -a blackbox-${fw}-${pc}-${codec}-${mediatype}-callloop-load.data
```

C.3.3 Analyzing measurement frequency in the firewall logfiles

This command counts how many lines every measurement logfile contains, "minus 7" because the first 2 lines are comments, and the last 5 lines are measurements taken during the "sleep" period. *seq* was replaces with *jot* since the former was not installed on Mac OS X by default.

```
pc="dell2" ; fw="fw1" ; codec="g711" ; mediatype="iax2" ;
hosts="1" ; ports="1" ; minus="7" ;
for type in cp_time pfstat-lan pfstat-wan ; do
  for calls in `jot - 500 3000 100` ; do
    hosts=$calls ; ports=$calls ;
    for try in `jot - 1 7 1` ; do
      lc=`wc -l < ${fw}-logs/${pc}-${codec}-${mediatype}-nocarp-${calls}calls-try${try}-${type}` ;
      let "lc -= $minus " ;
      echo "${pc} ${fw} ${codec} ${mediatype} $hosts $ports ${calls} ${type} ${try} ${lc} $minus" ;
    done ;
  done ;
done
```

C.3.4 Combining the firewall's *cp-time* logfiles

This command filters out lines starting with #, hops over the last 5 lines (again due to the "sleep" period) and does not output the first line since the $n_{i-1^{th}}$ line is the base value for line n_i for calculating the CPU-state percent values.

```
minus="5" ; pc="dell1" ; fw="fw1" ; type="cp_time" ; codec="g711" ; mediatype="iax2";
for calls in `jot - 500 3000 100` ; do
  hosts=$calls ; ports=$calls ;
  for try in `jot - 1 7 1` ; do
    i=0 ;
    lc=`wc -l < ${fw}-logs/${pc}-${codec}-${mediatype}-nocarp-${calls}calls-try${try}-${type}` ;
    let "limit = lc - minus" ;
    while read counter call user nice system interrupt idle ; do
      if [[ "${counter:0:1}" != "#" && $i -lt $limit ]] ; then
        if [ $i -eq 0 ] ; then
          user0=$user ;
          nice0=$nice ;
          system0=$system ;
          interrupt0=$interrupt ;
          idle0=$idle ;
        else
          let "userN = $user - $user0" ;
          let "niceN = $nice - $nice0" ;
          let "systemN = $system - $system0" ;
          let "interruptN = $interrupt - $interrupt0" ;
          let "idleN = $idle - $idle0" ;
          let "sum = $userN + $niceN + $systemN + $interruptN + $idleN" ;
          echo "$i $call `bc <<< "scale=2; $userN*100 / $sum"`
            `bc <<< "scale=2; $niceN*100 / $sum"` `bc <<< "scale=2; $systemN*100 / $sum"`
            `bc <<< "scale=2; $interruptN*100 / $sum"` `bc <<< "scale=2; $idleN*100 / $sum"`
            $pc $fw $codec $mediatype $hosts $ports $try" ;
          user0=$user ;
          nice0=$nice ;
          system0=$system ;
          interrupt0=$interrupt ;
          idle0=$idle ;
        fi ;
        let "i += 1" ;
      fi ;
    done < ${fw}-logs/${pc}-${codec}-${mediatype}-nocarp-${calls}calls-try${try}-${type} ;
  done ;
done
```