2005.eurobsdcon.org

EuroBSDCon 2005

Proceedings of the Fourth European BSD Conference

November 25 – 27, 2005
University of Basel

Conference Organization:
micro systems
Wiesendamm 2a · Postfach
CH-4019 Basel
Tel. +41 61 383 05 10
Fax +41 61 383 05 12
E-Mail info@eurobsdcon.org

Program Committee: Prof. Dr. Christian Tschudin, Computer Science Department, University of Basel · André von Raison, iX Magazin · Marc Balmer, The Organizing Committee · Wim Vandeputte, The OpenBSD Team · Max Laier, The FreeBSD Team · Emmanuel Dreyfus, The NetBSD Team · Felix Kronlage, bytemine

www.gkhcom.ch

EuroBSDCon 2005

Proceedings of the Fourth European BSD Conference

November 25 – 27, 2005
University of Basel, Switzerland

Event organization:

micro systems marc balmer
Wiesendamm 2a, Postfach
CH-4019 Basel, Switzerland

Copyright © 2005 by micro systems marc balmer. All rights reserved.
Printed in Switzerland

Published by micro systems marc balmer
Wiesendamm 2a, Postfach
CH-4019 Basel, Switzerland

Tel. +41 61 383 05 10, Fax +41 61 383 05 12
Email info@msys.ch

http://www.msys.ch/

Conference website: http://2005.eurobsdcon.org/

The copyright of the respective papers remains with the original authors.

While every precaution has been taken in the preparation of these proceedings, the publisher and the authors assume no responsability for errors, omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

The Call for Papers 1

Signal Handlers 3
Henning Brauer, with Wilhelm Bühler

Single User Secure Shell 11
Adrian Steinmann

Improving network security by adding randomness 15
Ryan McBride

Complete Hard Disk Encryption Using FreeBSD's GEOM Framework 21
Marc Schiesser

Improving TCP/IP security through randomization without sacrificing interoperability 49
Michael James Silbersack

A Machine-Independent Port of the MPD Language Run Time System to the NetBSD Operating System 67
Ignatios Souvatzis

New Evolutions in the X Window System 73
Matthieu Herrb and Matthias Hopf

Design and Implementation of OpenOSPFD 87
Claudio Jeker

Remote user access VPN with IPsec 113
Emmanuel Dreyfus

Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack 125
Robert N. M. Watson

Failover Mechanisms for Filtering Bridges on the BSD's 139
Massimiliano Stucchi

DVCS or a new way to use Version Control Systems for FreeBSD 145
Ollivier ROBERT

Porting NetBSD/evbarm to the Arcom Viper 161
Antti Kantee

New Networking Features in FreeBSD 6.0 171
André Oppermann

Embedded OpenBSD 179
Niall O'Higgins, Uwe Stühler

rthreads: A New Thread Implementation for OpenBSD 195
Ted Unangst

FreeBSD Jails in depth. An implementation walkthrough and usefulness example 199
Matteo Riondato

Conference Sponsors 209

EuroBSDCon 2005 - Call for Papers

4th European BSD Conference
November 25 - 27, 2005
University of Basel, Switzerland
http://www.eurobsdcon.org/

Introduction

The Berkeley Software Distribution (BSD) family of computer operating systems is derived from software developed at the University of California at Berkeley. The various family members (Free-, Net- and OpenBSD, among others) are extensively used both for embedded appliances and for large internet servers and have an excellent reputation for stability and state-of-the-art technology. BSD-derived software is a driving force for IT research and development and is well-received as a building block in commercial software due to its unique license scheme.

The fourth European BSD conference is a great opportunity to present new ideas to the community and to meet some of the developers behind the different BSDs.

The two day conference program (Nov 26 and 27) will be complemented by a tutorial day preceeding the conference (Nov 25).

Call for Papers

The program committee is inviting authors to submit innovative and original papers not submitted elsewhere on the applications, architecture, implementation, performance and security of BSD-derived operating systems. Investigations on economic aspects regarding the operation of BSD systems are also welcome. Topics of interest for the Euro BSD Conference 2005 include, but are not limited to:

- kernel hacking
- embedded application development and deployment
- device drivers
- security and safe coding practices
- system administration: techniques and tools of the trade
- operational and economic aspects

Prospective authors of contributions to the technical program are requested to submit an extended abstract through the web-interface on the conference website. All submissions will be reviewed by the program committee. The extended abstract should be at least two but no longer than four pages in either PostScript or PDF format. Submissions accompanied by a non-disclosure agreement are not acceptable and will be returned unread.

Authors of accepted submissions have to provide a full paper for publication in the conference proceedings and give permission to the organizers to publish the results in the printed proceedings and on the conference web site. Instructions to authors will be available on the conference web site.

Call for Tutorial Proposals

Selected tutorials on practical and problem-solving aspects of BSD-derived operating systems will be offered on the day before the Euro BSD Conference. The tutorials will be presented by speakers who have wide experience in developing and administering the different BSDs. Potential tutorial themes include, but are not limited to:

- Using FreeBSD in a datacenter environment
- Firewall configuration with OpenBSD
- Porting NetBSD to embedded devices
- Safe coding practices to provide secure solutions

If you are interested in presenting a tutorial, please contact the program committee at pc@eurobsdcon.org with details about the topic, intended audience, required room and facilities as well as a meaningful CV before August 1, 2005.

Important Dates

Extended abstracts due:	August 1
Tutorial proposals due:	August 1
Notification to speakers:	August 31
Final papers due:	October 20
Tutorial day:	November 25
Conference:	November 26 - 27

Conference Organizers

General Chairs <chair@eurobsdcon.org>

Marc Balmer, micro systems
Vera Hardmeier, micro systems

Program Chair <prog-chair@eurobsdcon.org>

Christian Tschudin, CS Department, University of Basel

Program Committee

Marc Balmer, micro systems
Emmanuel Dreyfus, the NetBSD project
Felix Kronlage, bytemine
Max Laier, the FreeBSD project
André von Raison, iX Magazin
Christian Tschudin, University of Basel
Wim Vandeputte, the OpenBSD project

Local Organizers

Marc Balmer, micro systems
Giacomo Cariello
Marcus Glocker, UBS AG
Vera Hardmeier, micro systems
Massimiliano Stucchi, WillyStudios.com
Marc Winiger, micro systems

Signal Handlers

Henning Brauer, with Wilhelm Bühler

October 29, 2005

1 Abstract

Signals are used to notify a program of events in Unix. Programs install signal handlers to catch them and react. The major problem with signal handlers is that non-atomic operations can get interrupted, and may end up in an unexpected state.

This paper will show these problems in detail. It talks about safe function in POSIX, ANSI C and the addional safe functions in the current OpenBSD version. It will show how to cope with possible solutions and finally shows a safe signal handler.

2 Introduction

2.1 Signal Handlers

Signals are used to notify a program of some events in Unix.

SIGTERM asks a program to shut down and exit, SIGCHLD tells a program that a child process changed its state. On most unix 31 signals are defined.

Programs can install signal handlers to catch them and react, with one exception: SIGKILL is not catchable. The program will just exit without any cleanup that might happen, e.g. if SIGTERM has been caught or the program exits by itself.

There is a default action for each signal defined.

2.2 Signal Handlers in the News

Signal handlers are very important. There have been numerous exploits in the past making use of incorrect signal handlers.

2.2.1 ftpd Signal Handling Vulnerability

Ftpd was open for intruders 1997:

This vulnerability is caused by a **signal handling routine** increasing process privileges to root, while still continuing to catch other signals. This introduces a race condition which may allow regular, as well as anonymous ftp, users to access files with root privileges. Depending on the configuration of the ftpd server, this may allow intruders to read or write to arbitrary files on the server.

This attack requires an intruder to be able to make a network connection to a vulnerable ftpd server.

For details see CERT Advisory CA-1997-16 http://www.cert.org/advisories/CA-1997-16.html

In this case a remote attacker could take over the entire machine - remote root. If anonymous ftp was enabled, he didn't even need a valid user account.

2.2.2 Sendmail Unsafe Signal Handling Race Condition Vulnerability

Sendmail before 8.11.4, and 8.12.0 before 8.12.0.Beta10 is vulnerable beause of unsafe signal handling.

Sendmail **signal handlers** used for dealing with specific signals (SIGINT, SIGTERM, etc) are vulnerable to numerous race conditions, including handler re-entry, interrupting non-reentrant libc functions and entering them again from the handler. This set of vulnerabilities exist because of unsafe library function calls from signal handlers (malloc, free, syslog, operations on global buffers,etc).

This vulnerability allows local users to cause a denial of service and possibly corrupt the heap and gain privileges.

For details see CVE-2001-1349 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-1349

2.2.3 Procmail Unsafe Signal Handling Race Condition Vulnerability

Procmail 3.20 and earlier is vulnerable beause of unsafe signal handling.

The problems lie in several **signal handlers** used by the program. By generating a signal while a signal handling operation is already in progress, an attacker could interrupt a non-reentrant libc function and enter it again from the handler. Precise timing in such an attack could possibly result in, for example, heap corruption or interruption during privilege lowering.

This vulnerability allows local users to cause a denial of service or gain root privileges.

For details see CVE-2001-0905 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0905

2.2.4 stunnel signal handler race DoS

The stunnel 4.0.3 and earlier allows attackers to cause a denial of service (crash) via SIGCHLD **signal handler** race conditions that cause an inconsistency in the child counter.

For details see CAN-2002-1563 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1563

3 The Problem

Signal handlers run upon receival of the associated signal. They can run basically at any time and they can **interrupt** basically **anything**, even syscalls.

3.1 Simple example

```
void
sighdlr(int sig)
{
        printf("signal %d received", sig);
}

int
main(int argc, char *argv[])
{
        signal(SIGHUP, sighdlr);
        signal(SIGUSR1, sighdlr);
...
```

The printf. being part of the stdio functions. uses buffers internally.

Imagine your program receives a SIGHUP while it does a printf("%d", 1); . The printf is halfway through changing some internal data structures. Now your signal handler calls printf, uses the very same data structures, and returns to the main program.

When we have a look at lib/libc/stdio/fvwrite.c we see it fiddles with all sorts of pointers. Upon return to your main program the actual pointers and your assumptions are out of sync.

Trying to write behind your allocated buffers will crash most time, but it might even end up in an **exploitable buffer overrun**.

3.2 malloc

The function malloc uses incredibly complicated data structures internally to keep track of the allocations, but without locking.

Sending a signal while main program is deep in malloc() will result in a half-recorded allocation, signal handler malloc()s too, and then return to main program.

The malloc internally now has a half wrong recording of the allocation it is about to give out.

The pointer it returns might be completely wrong. It is unpredictable what happens when you write to it or free() it.

3.3 exit()

```
void
sigterm(int sig)
{
        exit(1);
}
```

The function exit flushes stdio. You must not asume that all the internal stdio structs are in a consistent state when your signal handler runs.

The function exit runs atexit handlers. Again, you must not assume that all the atexit structures are in a consistent state when your signal handlers runs.

The function _exit() is safe, but then atexit handlers don't run and stdio is not flushed. You must understand the consequences before using _exit() in a signal handler.

Additionally, the registered atexit handlers are likely not signal handler safe.

The function _exit() is safe, but then atexit handlers don't run and stdio is not flushed. You should get clear about the consequences.

3.4 first summary

All the previous examples show basically the same issue: Non-atomic operations can get interrupted and you end up in an inconsistent or otherwise unexpected state.

Due to that, most functions are not signal handler safe.

4 safe functions

4.1 POSIX: safe functions

POSIX demands that these are safe:

_exit(), access(), alarm(), cfgetispeed(), cfgetospeed(), cfsetispeed(), cfsetospeed(), chdir(), chmod(), chown(), close(), creat(), dup(), dup2(), execle(), execve(), fcntl(), fork(), fpathconf(), fstat(), fsync(), getegid(), geteuid(), getgid(), getgroups(), getpgrp(), getpid(), getppid(), getuid(), kill(), link(), lseek(), mkdir(), mkfifo(), open(), pathconf(), pause(), pipe(), raise(), read(), rename(), rmdir(), setgid(), setpgid(), setsid(), setuid(), sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), signal(), sigpending(), sigprocmask(), sigsuspend(), sleep(), stat(), sysconf(), tcdrain(), tcflow(), tcflush(), tcgetattr(), tcgetpgrp(), tcsendbreak(), tcsetattr(), tcsetpgrp(), time(), times(), umask(), uname(), unlink(), utime(), wait(), waitpid(), write().

4.2 POSIX realtime extensions: safe functions

POSIX realtime extensions demand these too to be safe:

aio_error(), clock_gettime(), sigpause(), timer_getoverrun(), aio_return(), fdatasync(), sigqueue(), timer_gettime(), aio_suspend(), sem_post(), sigset(), timer_settime().

4.3 ANSI C: safe functions

ANSI C Interfaces, OpenBSD determined to be safe:

strcpy(), strcat(), strncpy(), strncat(), and perhaps some others.

This is documented in OpenBSD in the signal(3) manpage.

4.4 OpenBSD: safe functions

Additionally, we made some more functions safe in OpenBSD:

- strlcpy()
- strlcat()

- syslog_r()
- snprintf()
- vsnprintf()

It was a two years effort to make (v)snprintf() and syslog_r() safe.

5 more issues

5.1 errno...

```
do {
        pid = waitpid(-1, &stat, WNOHANG);
} while (pid == -1 && errno == EINTR);

if (pid == -1)
        err(1, NULL);
```

Imagine your signal handler runs after waitpid returned -1 and set errno to EINTR, yet before the errno check.

Your signal handler causes errno to be set to something else, and upon return you exit from the loop and quit your program. This is not exactly intended and can lead to Denial of Service attacks.

5.2 Another example

Another example for the same problem:

```
do {
        n = write(fd, buf, len);
} while (n == -1 && errno == EINTR);
```

In this case, you'd try to cope with a write error, but there was only an interruption by a signal.

The solution is to save and restore errno in signal handlers.

5.3 the _r functions

It is often heard that the *_r functions are reentrant and thus safe to use, even in signal handlers.

Many are made reentrant in a threads environment by acquiring pthread locks. Obviously this does not help at all in signal handlers.

6 Possible solutions

Now we saw the problem, but how to cope? There is so little allowed in signal handlers.

6.1 the flag solution

A first possible solution could be to set a flag in the signal handler and react upon later:

```
int quit = 0;

void
sigterm(int signal)
{
        quit = 1;
}

int
main(int argc, char *argv[])
{
        signal(SIGTERM, sigterm);

        while (!quit) {
                /* do something */
                poll()
                /* do something */
        }
}
```

The flag solution needs a main loop or a similar central place to check the flag.

Many, but not all programs have an appropriate place for the check.

When a program spends extended amounts of time outside said main loop (or, more generally, without checking for the signal flag), it will react late on signals. You must get clear about the consequences of this late reaction, maybe you need to cope.

But this is still not safe, because not all data types can be accessed atomically – same old locking issue again.

int should be safe everywhere, but there is no guarantee.

Fortunately, there is sig_atomic_t that is guaranteed to be atomically accessible.

6.2 compiler

And we still have an issue. The compiler can reorder reads and writes, even to your sig_atomic_t flag, and we're back in locking hell.

For once, there is an easy solution, it's the volatile keyword.

6.3 safe signal handle

Finally, we have a safe signal handler here:

```
volatile sig_atomic_t quit = 0;

void
```

```
        sigterm(int sig)
        {
                quit = 1;
        }
```

6.4 Signal Races

While the signal handler is now safe, there are still signal races. Let's have a look at the simple case.

```
        volatile sig_atomic_t gotsigchld = 0;

        void
        sigchld(int sig)
        {
                gotsigchld = 1;
        }

int
main(int argc, char *argv[])
{
while (!quit) {
poll()
if (gotsigchld)
childhandler();
gotsigchld = 0;
}
}
```

There is several problems in here. First, we do not keep track of the number of SIGCHLDs received. If childhandler() only cleans up after one child while actually two went away there is an obvious problem. Thus, childhandler() needs to loop until it handled all dieing child processes.

Second, if we receive a second sigchild **after** childhandler() was run, but before gotsigchld gets reset to 0, we will not react on that signal at all. The usual solution is to reset the flag first.

```
if (gotsigchld) {
gotsigchld = 0;
childhandler();
}
```

7 next paper

The signal handler is safe, now we have still signal races, this could be another paper.

8 Acknowledgment

We would like to thank Theo de Raadt for his continued help and support. as well as Wim Vandeputte for moral, hard(ware) and liquid support.

9 Authors

9.1 Henning Brauer

Henning Brauer. 27. lives in Hamburg/Germany and runs bsws.de. an ISP focussing on online solutions for corporations and running most of their network on OpenBSD. He has been an OpenBSD developer for some years now, focused on network stuff. In the past he spent a lot of work on the packet filter. pf. including the altq merge. thus providing stateful bandwidth management. He also wrote the chroot and privilege revocation extensions for apache. and did the privilege revocation/seperation for the dhcp related programs. as well as several other smaller network daemons. He started and still works a lot on OpenBGPD and OpenNTPD. shipping with OpenBSD since 3.5 and 3.6. respectively. When you meet him without a notebook he's likely mountainbiking. hiking in canada's fantastic landscape. or hanging out with friends. likely paired with enjoying brewer's art.

9.2 Wilhelm Bühler

Wilhelm Bühler, 39, lives in Karlsruhe/Germany and works at a computing center. He likes stuffed mascots. The 85-centimeter BSD-daemon was initiated by him. Ever saw a stuffed blowfish? He supports the OpenBSD-project with donations on hardware and time. but not as a developer.

Single User Secure Shell

Adrian Steinmann
ast@webgroup.ch

Abstract—**Unix systems traditionally do integrity checks and other initialization before bringing up network services. System administration tasks, for example operating system upgrades, system disk reformatting, or system disk partitioning, very often need to be done in single user mode locally, not via the network. We describe how a 'Secure Shell Maintenance RAMdisk Environment' can be built and launched very early in the boot process. This environment can be used to remotely fix a problem when the machine is stuck in single user mode. Our method has already been in use for a number of years to upgrade remote managed firewall systems [3] from one release to the next.**

I. Introduction

ANYONE who has needed to unexpectedly commute to a production machine wedged in single user mode has wished that it would possibly still allow a remote SSH login. In most cases, the problem in question does not require network access to be blocked at all. It is much more a policy decision that the system first checks the root filesystem and runs other early startup scripts which may stall before launching network services.

In this paper we describe a way to build a 'Secure Shell Maintenance RAMdisk Environment' which can be started even before the root filesystem is checked. This environment can be useful in many different situations:

- *Root filesystem fails to check:* When a system crashes, it sometimes damages the root filesystem so that it cannot be automatically fixed. Traditionally, the system then stays without networking enabled awaiting input on the console. Missing or corrupt files in the `/etc` hierarchy could also cause the system to never reach network initialization.
- *System partitions need to be resized:* As more software is installed, the operating system partitions occasionally need to be resized. This usually calls for a `dump`, `bsdlabel`, `newfs`, `restore` cycle, which may not be possible in multi user mode.
- *Pristine operating system upgrade:* As is the case for the FreeBSD 4.x to 5.x migration, it may be desirable to `newfs` all the system partitions to take full advantage of new features or simply to do a 'clean' install.
- *Changing root filesystem to RAID:* It is difficult to transform the system partitions of an already installed operating system onto a GEOM-based RAID because the system is using the non-RAID devices. Similarly, `atacontrol create` will fail on disks with system partitions because they are busy.
- *Minimal installations on small systems:* Full installations on single board computers (SBCs) with only compact flash (PC-Engines [1], Soekris [2]) from a standard distribution or from CD may not be practicable.

II. Building the Secure Shell RAMdisk Image

SIZE of the Secure Shell Maintenance RAMdisk filesystem is a prime concern: it should include all the important tools needed for remote system administration, in particular a SSHv2 daemon, yet it should not fill out too much system memory because it is retained there during the lifetime of the system.

Earlier releases managed to fit a gzipped kernel and a gzipped RAMdisk image onto a single 1.44 MB floppy by using 'small' versions of utilities – see, for example, ports `shells/sash` and `security/ssh` (SSHv1), as well as `release/picobsd/tinyware`.

As of FreeBSD release 5.x, fitting everything on one floppy became impossible, and in fact most systems deployed nowadays do not even carry a floppy drive. Nonetheless, the Secure Shell Maintenance RAMdisk Environment is still viable because even systems with only 64MB of compact flash have ample space to store such a RAMdisk image alongside a whittled-down FreeBSD distribution. Without such a strict size limitation, we now have the additional advantage that we always have the native implementation of the commands instead of their often deficient 'small' substitutes.

A. *Use* `crunchgen` *to minimize RAM utilization*

In our implementation, the following programs are available in the RAMdisk environment:

```
RAMdisk# ls /bin
-sh              ex           kill              reboot
[                expr         kldconfig         red
atacontrol       fastboot     kldload           restore
badsect          fasthalt     kldstat           rm
boot0cfg         fdisk        kldunload         rmdir
bsdlabel         fsck         ldconfig          route
bunzip2          fsck_4.2bsd  link              rrestore
bzcat            fsck_ffs     ln                scp
bzip2            fsck_ufs     ls                sed
camcontrol       gbde         mdconfig          sh
cat              gconcat      mdmfs             sleep
chflags          geli         mini_crunch       slogin
chgrp            geom         mkdir             ssh
chmod            ggatec       mknod             sshd
chown            ggated       mount             stty
chroot           ggatel       mount_cd9660      swapctl
clri             glabel       mount_devfs       swapoff
cp               gmirror      mount_fdescfs     swapon
date             gnop         mount_linprocfs   sync
dd               graid3       mount_nfs         sysctl
df               gshsec       mount_procfs      tar
dhclient         gstripe      mount_std         test
dhclient-script  gunzip       mv                touch
diskinfo         gzcat        newfs             tset
disklabel        gzip         pax               tunefs
dmesg            halt         ping              umount
du               hostname     ps                unlink
dump             ifconfig     pwd               vi
dumpfs           init         rdump             zcat
ed               kenv         realpath
```

```
RAMdisk# du -k /bin/*
2256    /bin/-sh
8       /bin/dhclient-script
```

Note that SSHv2 as well as the network filesystem utilities `mount_nfs`, `ggated`, and `ggatec` are present with the requisite network configuration utilities `ifconfig`, `route`, and `dhclient`. The standard archiving tools `dump`, `restore`, `tar`, and `pax` with `gzip` and `bzip2` are also available.

Since this is primarily an environment for system administration, the low-level `fdisk`, `bsdlabel`, `newfs`, and `tunefs` utilities are included, with the added luxury of the `vi` editor (albeit with a small `termcap` file supporting only `xterm`, `screen`, `vt220`, `at386`, and `cons25` terminals). Finally, `atacontrol`, `camcontrol`, DHCP, GEOM-based RAID, GBDE, and GELI are available when the underlying kernel is adequately configured.

B. *Use* `mdconfig` *to create a RAMdisk image*

The script `release/scripts/doFS.sh` creates a file containing a RAMdisk image of a given filesystem hierarchy. For example, `release/Makefile` uses this script to create the 'Install' and 'Fixit' environments for the standard FreeBSD distribution CD.

Since the loader supports uncompressing gzip files on-the-fly, this RAMdisk image can be gzipped and placed into, say, a `/boot/maint/` subdirectory, where it can be accessed at boot image load time.

C. *Use the loader to boot into RAMdisk*

While FreeBSD starts up, one can enter the boot loader environment by choosing the menu item *6. Escape to loader prompt* on the console, whereupon one is presented with an `OK` prompt as shown in figure 1.

```
OK ls /boot/maint
/boot/maint
    k.CUSTOM.gz
    fs_img.gz
    params
    loader.rc
OK unload
OK load /boot/maint/k.CUSTOM
/boot/maint/k.CUSTOM text=0x22ed1b data=0x28ba4+0x12748
OK load -t md_image /boot/maint/fs_img
OK include /boot/maint/params
OK set vfs.root.mountfrom=ufs:/dev/md0
OK autoboot
Hit [Enter] to boot immediately, or any other ...
Booting [/boot/maint/k.CUSTOM] in 9 seconds...
```

Fig. 1. Booting a RAMdisk maintenance environment from the loader.

By pre-loading the RAMdisk maintenance image and setting the `vfs.root.mountfrom` variable, the kernel mounts it as the root filesystem instead of the one specified in the `/etc/fstab` file on disk. Note that if the already loaded kernel `/boot/kernel/kernel` supports the 'md' device, there is no need to unload and load the custom kernel.

III. Some Minor Hurdles

The implementation of the described system is straightforward, except for the following minor difficulties:

Crunching SSHv2: The standard build of FreeBSD `sshd` requires many libraries, yet most are unnecessary in the RAMdisk environment. We will show how we can get by with only linking a fraction of those libraries.

Supporting runtime loader in a crunched binary: Some programs require runtime loading; this means we must link some libraries statically and some dynamically – although we are using `crunchgen`, which by default links everything statically.

Parameterizing a generic RAMdisk image flexibly: It is better to keep the machine parameterization separate from the binary RAMdisk image, so that deployment over many similar machines consists of identical binary files and one machine-specific text file.

A. *Crunching SSHv2 without too many libraries*

Even if we specify lots of `NO_*` options in the `crunchgen` configuration file (figure 2), the link phase fails because `libpam.a` – among others – remains referenced.

```
# LIBS_AS_SHARED_OBJECTS = -lmd -lcrypto
buildopts -DNO_CRYPT
buildopts -DNO_INET6
buildopts -DNO_KERBEROS
buildopts -DNO_PAM
buildopts -DNO_X
srcdirs /usr/src/secure/usr.bin
srcdirs /usr/src/secure/usr.sbin
progs scp ssh sshd
libs -lssh -lutil -lz -lcrypt
ln ssh slogin
```

Fig. 2. A `crunchgen` configuration file fragment for a 'mostly statically' linked SSHv2.

By setting the correct compile time flags for `PAM`, `LIBWRAP`, and `XAUTH_PATH` in the `crypto/openssh/config.h` file *directly*, the `crunchgen` fragment in figure 2 then links successfully. The other additionally required libraries not mentioned on the `libs` line are made available as dynamically loadable shared objects in the RAMdisk environment.

B. *Building 'mostly statically' linked crunched binaries*

Although the rest of the `crunchgen` configuration file is straightforward, the geom programs require the runtime loader for their `dlopen(3)` calls. As an added complication, some `/lib/geom/geom_*.so` libraries also expect `libmd.so` and `libcrypto.so` to be dynamically loaded. For this reason we need to include them together with `rtld(1)` on the RAMdisk:

```
RAMdisk# ls -sFR /lib*
/lib:
total 1928
   2 geom/                  992 libcrypto.so.4
 880 libc.so.6               54 libmd.so.3

/lib/geom:
total 156
14 geom_concat.so          10 geom_nop.so
42 geom_eli.so             22 geom_raid3.so
12 geom_label.so           14 geom_shsec.so
26 geom_mirror.so          16 geom_stripe.so

/libexec:
total 134
134 ld-elf.so.1*

RAMdisk# du -k /lib*
158     /lib/geom
2086    /lib
136     /libexec
```

Mostly statically linked binaries can be built simply by replacing "`$(CC) -static ...`" in the makefile created by `crunchgen` with "`$(CC) -Xlinker -Bstatic`

`... -Xlinker -Bdynamic ...`", where the second set of "`...`" mention the libraries that are left to be linked dynamically at runtime. Between the `crunchgen` invocation and the subsequent `make -f mini_crunch.mk` command we substitute completely static linking with 'mostly static' via a simple `sed(1)` command on the makefile.

We have chosen this particular set of shared objects because `geom/geom_eli.so` requires `libcrypto.so` in addition to `libmd.so`. Special attention must be given to dynamic objects requiring others to make sure that the RAMdisk image remains self-sufficient. For our purposes the goal is to save as much space as possible by maximizing the number of libraries which are statically linked.

C. *One RAMdisk image for many systems*

Another important design goal is to have one RAMdisk image for all machines and yet configure the network and other machine specific parameters via a separate text-based configuration file. This is resolved by passing information via the kernel environment. The RAMdisk environment then uses `kenv` calls to configure the network and, in particular, to create the `/root/.ssh/authorized_keys` file there.

```
set maint.ifconfig_XX0="192.168.0.254/24"
set maint.ifconfig_XX1="192.168.1.254/24"
set maint.ifconfig_YY0="dhcp"
set maint.defaultrouter="192.168.0.1"
set maint.host="GENERIC"
set maint.domain="SETME.com"
set maint.sshkey_01a="ssh-d.. (120 chars) ..qP"
set maint.sshkey_01b="leQXQ.. (120 chars) ..9d"
set maint.sshkey_01c="b7Zd+.. (120 chars) ..zu"
set maint.sshkey_01d="KrdBn.. (120 chars) ..tw"
set maint.sshkey_01e="7eMec.. (120 chars) ..4G"
set maint.sshkey_01j="hdTLKVUokhU41Q== 200507"
```

Fig. 3. A `/boot/maint/params` file describing a specific machine parameterization.

In our setup, the `/boot/maint/params` file describes the machine parameterization (see figure 3) and is included by the loader when booting into the RAMdisk environment (figure 1).

A limitation of `kenv` is that the key and value lengths may not exceed 128 characters. Since we need to craft a `/root/.ssh/authorized_keys` file in the RAMdisk environment, we split its contents into smaller pieces and then paste them back together before launching the SSH daemon.

IV. Putting it all together

LAUNCHING the 'Maintenance Single User Secure Shell' as early as possible during the boot sequence is achieved by placing a startup script in the `/etc/rc.d/` directory with the correct `REQUIRE` and `BEFORE` keywords:

```
#!/bin/sh
PATH=/rescue:/usr/bin:/bin:/usr/sbin:/sbin
export PATH

# REQUIRE: initrandom
# PROVIDE: maint_sshd
# KEYWORD: nojail
# BEFORE:  disks
...
```

On a standard FreeBSD 6.0 system, this means it will be invoked in fourth place, immediately after `initrandom`:

```
6.0-current$ rcorder /etc/rc.d/* | head -4
/etc/rc.d/rcconf.sh
/etc/rc.d/dumpon
/etc/rc.d/initrandom
/etc/rc.d/maint_sshd
```

To control launching of the Single User Secure Shell, `/etc/rc.conf` is equipped with these knobs and tunables:

```
## maint_sshd_enable="NO"
## maint_sshd_mntdir="/boot/maint"
## maint_sshd_fs_img="/boot/maint/fs_img"
## maint_sshd_port="22222"
```

When its enable switch is set to `"YES"`, the startup script `/etc/rc.d/maint_sshd` mounts the given RAMdisk image onto `$maint_sshd_mntdir` with executables from `/rescue`. Then, under invocation of the `chroot(8)` command from RAMdisk, the `/dev` mount and network initialization is started inside the RAMdisk, which also starts a SSHv2 daemon on port `$maint_sshd_port`. Provided the network configuration in the `/boot/maint/params` file is correct and the private SSH keys are known for the public keys therein, a root shell can be opened remotely – even when the machine is stuck in single user mode.

```
6.0-current$ /bin/ls -lsFR /boot
   4 beastie.4th.gz
   2 defaults/
   2 device.hints
   2 frames.4th.gz
   2 kernel/
 110 loader*
   4 loader.4th.gz
   2 loader.conf
   6 loader.help.gz
   2 loader.rc
   2 maint/
   2 screen.4th.gz
  10 support.4th.gz

/boot/defaults:
 6 loader.conf.gz

/boot/kernel:
3216 kernel

/boot/maint:
1840 fs_img.gz
1056 k.CUSTOM.gz
   2 loader.rc
   2 params
```

Fig. 4. A minimal `/boot/` directory hierarchy with gzipped loader files, RAMdisk image, custom RAMdisk kernel, and RAMdisk configuration files.

The additional disk space requirements are modest and hence this method is also very well applicable to SBCs. A gzipped kernel and a gzipped Secure Shell Maintenance RAMdisk image will be at most 4 MB and are integrated into the `/boot` hierarchy (figure 4). Note that the loader itself can also be gzipped, roughly halving its space requirements. All files which the loader may load can also be gzipped because it searches for `*.gz` files and uncompresses them on-the-fly.

Alternatively, instead of launching the `maint_sshd` startup script at boot time, the loader can be instructed to boot directly into the RAMdisk by including the lines in figure 1 into `/boot/loader.rc`. For this case we generally prefer to sup-

ply a custom kernel /boot/maint/k.CUSTOM which has the device md compiled in and is otherwise stripped of extraneous options not needed in the RAMdisk. The machine can then be rebooted into the Single User Secure Shell for a console-free system upgrade.

V. Experience with Single User Secure Shell

As our own practice shows, the Single User Secure Shell has twofold usage:

Staging Single Board Computers: While deploying SBCs by Advantech, PC-Engines [1], and Soekris [2], we found it inconvenient or impossible to utilize a Preboot Execution Environment (pxeboot(8)) for the initial install. These systems are generally left in the field without keyboard nor console and only with compact flash (CF) for disk space. Before assemblage, the CF is populated with the maintenance RAMdisk, from which the final installations are done as if they were an upgrade. The CF can be copied as a 'disk image' (see also the NanoBSD tool in the FreeBSD tree) or, more portably, it can be initialized in just a few steps (figure 5) on a host system with PCcard-CF adapter.

Upgrading remote systems: Once the systems are in the field, the CF certainly cannot be replaced easily yet may need to be fully initialized and repopulated with new software. The systems are rebooted into a current RAMdisk, from which all operations are possible.

```
# double check that these are correct!
disk=ad4; slice=s1

# initialize the MBR
dd if=/dev/zero of=/dev/${disk} bs=512 count=64
fdisk -I ${disk} auto

# initialize the FreeBSD bootblocks and label
bsdlabel -w -B ${disk}${slice} auto
bsdlabel ${disk}${slice} | \
  sed -e '/a:/ s/unused.*/ 4.2BSD * * */' | \
    bsdlabel -R -B ${disk}${slice} /dev/stdin

# create root fs: no snapshots, space optimize
newfs -n -o space /dev/${disk}${slice}a
mount -o noatime,sync /dev/${disk}${slice}a /mnt

# force serial console; populate /boot hierarchy
echo " -h" > /mnt/boot.config
(cd /mnt && tar vjxpUBf -) < boot+maint.tbz
```

Fig. 5. Initializing a CF from scratch before deployment in SBC.

As always, there are some caveats and pitfalls which should be considered:

4MB or more of RAM remain occupied: System memory for the RAMdisk is allocated forever when maint_sshd is enabled. On small platforms it may be unacceptable to sacrifice so much space continually. In this case, the RAMdisk would be loaded only for upgrades by rebooting after /boot/loader.rc is replaced by /boot/maint/loader.rc.

Network reconfiguration may interfere: Setups with complicated network configuration could become error-prone in combination with the RAMdisk network configuration which takes place earlier. Moreover, when experimenting with IP filter rules remotely, it is always wise to define a special early rule to assure access in the failure case. On our systems, such interference occurs seldom because the network configuration defined in the RAMdisk environment is identical to the multi user configuration.

Booted kernel may not support 'md' devices: When the running kernel does not support memory disks because device md is not configured, then maint_sshd will fail to launch. For this reason a custom kernel is kept in the /boot/maint directory.

SSH daemon in RAMdisk may be a security risk: Access to the system's root account is effectively controlled by the /boot/maint/params file because the SSH deamon running in RAMdisk must allow root logins.

VI. Future Directions

None of the methods we have employed are unique to FreeBSD. In fact, over the lifetime of this development, we were able to simplify and go 'back to the basics' as FreeBSD evolved. For this reason, we believe implementing an analogous RAMdisk environment for the other *BSD systems would be very straightforward.

The mostly statically linking capability should be integrated into crunchgen via a new keyword. The chain of library dependencies could be checked automatically to ensure that none of the required shared objects are linked statically.

Looking ahead, we plan to investigate if a Secure Shell Maintenance RAMdisk could be loaded via a kernel module so that the dependence on the /etc and /rescue directories can be completely removed. In fact, the RAMdisk environment could fully replace the present rescue utilities. Last but not least, it might be practical to present the choice *"Enter Maintenance RAMdisk"* directly from loader menu.

Adrian Steinmann earned a Ph.D. in Mathematical Physics from Swiss Federal Institute of Technology in Zürich and has over 15 years experience as a technical consultant and software developer. He is founder of Webgroup Consulting AG, CH-8032 Zürich, Switzerland.

He has been working on FreeBSD since 1993 (version 1.0) and since 1997 he maintains and develops the base system for a remote managed firewall called 'STYX' [3]. He is fluent in Perl, C, English, German, Italian, and has passion and flair for finding simple solutions to intricate problems.

During his free time, he likes to play Go, to hike, and to sculpt soapstone. Some sculptures are on display at www.steinmann.com/sculptures.

References

[1] PC Engine WRAP: *Wireless Router Application Platform; 2002–2005*; 266 MHz AMD Geode SC1100 CPU, 128MB SDRAM, CF, 2-3 LAN, 1-2 Mini-PCI; www.pcengines.ch

[2] Soekris net4501: *Compact, low power, low-cost, communication computer; 2001–2005*; 133 MHz, 64MB SDRAM, CF, 3 LAN, 1 Mini-PCI; www.soekris.com

[3] STYX Firewall: *FreeBSD-based Remote Managed Firewall; 1997–2005*; www.styx.ch

Improving network security by adding randomness

Ryan McBride
mcbride@openbsd.org

Abstract

Poorly specified or poorly implemented protocols often contain fields for which the value is essentially arbitrary, but can be guessed by an attacker in order to perform a spoofing attack, or leak information about the system which provided the data. By using random or strong pseudo-random data for these fields, many protocol attacks can be prevented or made impractical, and information leakage can be minimised.

The OpenBSD project has been very aggressive in its use of pseudo-random data in its network code; as a policy pseudo-random data is used in protocol fields wherever possible, in many cases in a way not envisioned by the protocol designers.

This paper outlines the reasons for this approach, discusses how and where it is implemented in OpenBSD, and provides examples of attacks which this approach has mitigated. Randomness used within protocols explicitly for security purposes (such as randomness in IPSec, SSH, CARP, etc) is not discussed - the interest is in randomness which is not intended by the protocol designers.

Introduction

Network protocols often contain fields which require unique data per packet or per session, or counters initialised and then incremented. Often the protocol specification allows arbitrary values to be chosen, but does not insist on randomness; even when randomness is called for, a poor implementation can still employ weak values. The ability to guess these weak or predictable values offers an attacker the opportunity to inject malicious packets blindly, modifying the data or killing connections.

A secondary concern is that of information leakage: poorly selected values can disclose internal system state (such as system time), aiding in general information gathering or even providing data which can be used to facilitate other attacks. It should be noted however that generally such attacks indicate a weakness in the protocol being attacked. The ability of an attacker to determine the system time is the most commonly discussed form of information leakage - but the solution is not to avoid leaking the system time. The solution is to stop using system time as a "secret" in protocols.

The OpenBSD project has adopted a policy of aggressive use of random and good pseudo-random data in its networking code. Essentially: if the value of a protocol field can be set to an arbitrary value, use data that an attacker cannot guess, even if no current attack is known. This pro-active approach has repeatedly resulted in OpenBSD adding protection against protocol attacks before they were discovered.

For example, problems with predictable TCP sequence numbers have been known for some time, and most modern TCP implementations try to make the initial sequence number difficult to predict. Paul Watson's paper "Slipping in the Window"[1] expanded on threat of blind attacks. He pointing out that some attacks could be conducted without predicting the exact next sequence number: sequence numbers in attack packets only need to fall within the TCP window. An attacker who knows the source and destination addresses and ports can expect to reset a connection with a known source port in 13.6 seconds at T1 speed. By default, OpenBSD selects the source port pseudo-randomly from the range 1024 to 49151. If the source port of the connection has been selected in this way, the attack will take 7.5 days, making a trivial attack impractical[1].

OpenBSD's Implementation

Pseudo-random number generation

Most network "randomness" comes from arc4random(), a fast 32-bit pseudo-random number generator based on the alleged RC4 stream cipher. This cipher is not extremely strong cryptographically: there are significant weaknesses in the key scheduling algorithm[2], and the algorithm leaks information about its internal state[3]. OpenBSD attempts to work around these issues by reseeding arc4random() on a regular basis from OpenBSD

[1] In fact OpenBSD requires the sequence number for a TCP reset to be directly on the edge of the window; the attack would take several years at T1 speeds.

strong random number subsystem and throwing out the initial 256 words of output.

Despite these weaknesses, arc4random() is a reasonable compromise between security and performance; the goal being to keep the source of random data cheap enough that it can be used wherever necessary, without measurably reducing the performance of the system. Cryptographically strong random data is relatively expensive, and most systems cannot generate strong random data in sufficient quantities to use aggressively on the network.

OpenBSD's strong random number subsystem constantly gathers randomness from a number of sources and folds it into a randomness pool. A broad range of cryptographic random number generating hardware is supported, but IO data and timing measurements are also stirred into the randomness pool: all mouse movements, keystrokes, disk activity, network activity, audio playback and recording, and even the timing of the random number subsystem itself are used to feed the randomness pool.

While the strong random number subsystem gathers randomness from many sources, it is also very conservative in estimating the randomness provided by these sources, and a running estimate of the amount of entropy in the randomness pool is kept.

It is worth noting that because network packet timings feed into the randomness pool, while a busier system is drawing more heavily on the random number generator, it is also feeding more entropy into the randomness subsystem.

Having multiple consumers of the pseudo-random number generator also increases the quality of the numbers for any one consumer, as requests may be interleaved and each consumer no longer receives an uninterrupted sequence of bits from the function. An attacker attempting to determine the internal state of the arc4random() subsystem has no idea whether two random values are sequential or 5000 bytes apart in the stream. They may not even be based on the same key. Even full knowledge of the network traffic will not provide a clear picture to an attacker, as arc4random() is used heavily by other non-network subsystems.

Randomness Constraints

Good targets to search for in protocol standards documents include counters, time stamps, and packet, session, or host identifiers. However, randomness needs to be added carefully - constraints can include a minimum or maximum gap between sequential numbers, avoiding "magic" values, or ensuring that values are not repeated within a minimum time interval.OpenBSD employs a number of different techniques to provide values which maximise entropy while meeting these constraints:

Some protocols require a *locally non-repeating* field - that the number used does not repeat within a certain number of uses or within a certain time. This property is often required of fields which are used to disambiguate packets, such as the DNS id, IP id and IPv6 fragment ID. Values for these fields can be effectively generated by using a Linear Congruential Generator (LCG), a pseudo-random number generator where the next number is generated from the current one by $r_{n+1} \equiv ar_n + b(\mod m)$, where a and m are relatively prime numbers. In order to foil attempts to guess the internal state of the LCG, a number of values are discarded for each one generated. Pseudo-random data from arc4random() is used to make the selection. The LCG is re-keyed every N values, or every M seconds, where N is shorter than the full cycle length of the LCG to avoid blackjack prediction of the next values. Because re-keying could lead to repetition, one bit of the final output is set to a fixed value which is toggled when re-keying occurs.

Less-common requirements are *minimum gap* or *maximum gap* between sequential numbers; A minimum gap can be enforced by simply forcing the appropriately valued bit to a fixed value, while a maximum gap can be enforced by remembering the previous number generated and adding or subtracting a pseudo-random value generated by taking arc4random() modulo the maxiumum gap. A field which requires *increasing* values can be handled similarly, by only adding the pseduo-random value. Where protocols require a *monotonically increasing* counter a random value can often be used to initialise the counter rather than starting from zero; care must be taken in cases where the protocol makes no provisions for wrapping of the number, as is the case with the IPSec replay counter.

Protocol implementations which ask for the time generally fall into two categories: those requiring a locally non-repeating number, and those which require a *timer* - not the real time but only require a number that increases at a regular rate. The former can be dealt with by using an LCG as discussed above, while the latter can be handled by keeping a normal timer with some random modulation: If the timer must be uniform across connections, it can be initialised to a pseduo-random value. Otherwise, the timer state can be kept independent by offsetting the timer by a pseudo-random modulator value generated on a per-connection basis.

End-point randomness

The following are the network fields initialised with pseudo-random data in OpenBSD's native TCP/IP stack:

TCP initial sequence number The Transmission Control Protocol (TCP), requires a sequence number in

order to provide ordering of packets[4]; much of TCP's resistance to blind spoofing attacks is derived from the difficulty of guessing what a valid sequence number might be. The ISN must be locally non-repeating and requires a minimum gap of 32768 bytes between numbers. Because of these constraints, OpenBSD does not use the output of arc4random() directly here but rather a combination of 15 bytes from a pseudo-random generator seeded from arc4random(), and 15 bytes of pure arc4random(). Byte 15 is always set to 0 in order to ensure the minimum gap between sequence numbers, and the most significant byte is toggled each time the generator is re-seeded, to ensure that the same number is not re-used within a short period.

TCP Timestamp Modern TCP stacks set a Timestamp option on TCP packets which contains the local system time and an echo of the timestamp from the other end[5]. This timestamp is used to calculate round trip times, and as a mechanism for protecting against wrapped sequence numbers. In OpenBSD, each TCP connection uses a different initial TCP timer value initialised with arc4random(). This makes spoofing valid TCP timestamps in a connection more difficult as an attacker can no longer poll for the current time to use in attacking an unknown connection.

TCP/UDP ephemeral source port Randomisation of the ephemeral source port makes all blind attacks against TCP connections more difficult as an attacker must correctly guess this portion of the (source address, source port, destination address, destination port) tuple. Ephemeral source ports are selected using arc4random() from a range of available ports; if the selected port is not available a linear search of the space up or down from that port is used to find the next available port.

DNS query id Thes 16 bit value is used to disambiguate DNS requests. An attacker who can guess this value and the 16 bit source port can blindly spoof DNS replies and mount DNS cache poisoning attacks. In order to ensure that sequential valid DNS responses are not confused, the DNS query id must be locally non-repeating, and OpenBSD uses an LCG to generate the bottom 15 bits of this value, toggling the most significant bit on re-keying.

IP id This 16 bit value is used as an identifier for assembling fragmented IP packets. The ability to predict this value could permit an attacker to perform a denial of service attack against applications which make heavy use of fragmentation, by spoofing fragments which would be correctly reassembled but result in the entire packet being discarded because of a failed checksum. The IP id has the same constraints as the IP id, and uses the same algorithm to generate it's value.

IPv6 fragment id Similar in purpose to the IPv4 ip_id, this 32 bit value is used to identify multiple IPv6 fragments as a single packet. An LCG is used to generate the bottom 31 bits of this field, toggling the most significant bit on re-keying.

tun virtual mac address 3 bytes of the virtual MAC address are generated directly by arc4random().

ping / ping6 icmp id These programs use their pseudo-random process ID as the 16 bit icmp id.

ping6 icmp6 node information nonce, used in an ICMPv6 Node Information Node Addresses query to match a query to the response; this 64 bit value is generated directly with arc4random(). While technically this value must be locally non-repeating, the likelyhood of repetition is not sufficiently high to warrant the use of more complex code to guarantee this property.

rpc message transaction identifier Each RPC transaction requires a unique 32-bit ID, which is directly generated with arc4random()

rdate and ntpd The time client from both of these utilities use a 64-bit number generated with arc4random() as the transmit time rather than the actual system time[6, 7]. The responding server copies this field into the originating time field on the response that it sends back. This makes it much more difficult for a blind attacker to spoof responses from the NTP server, and incidentally prevents the leakage of the real system time.

timed initial sequence number This value is used as a sequence number to identify time packets send out by timed.

All the above uses of pseudo-random are enabled by default in OpenBSD's network stack, and they cannot be disabled by the user.

Randomness in PF

In situations where a system acts as an intermediary between other systems rather than an endpoint, such as when it is operating as a firewall, fields can have their potentially insecure values replaced by pseudo-random data. This can protect hosts with weaker network stacks from some types of attacks.

In OpenBSD, this functionality has been implemented in pf(4), the packet filter. This works well technically

as the packet filter already requires code to track packet flow on a per-connection basis, and maintains a state table with relevant connection information - a logical place to store the mappings between the old and new values of fields which have randomness injected. Moreover, The firewall already does the packet manipulations, including IP packet reassembly and moving packet data into contiguous memory required to look into the IP packet payloads; making changes to the packets at this point is relatively simple. And finally, some of the randomness is also injected in places where packet changes are required for other functional reasons in the firewall code, for example the port and address changes which allow for address sharing or load balancing.

There is also an issue of expectations management: conceptually a firewall is already expected to make security-related changes to packet flow, as opposed to a simple router which is expected to get packets from A to B with the minimal set of changes.

Finally, the fine-grained packet matching based on a number of packet criteria allows the features to be applied only as needed, which is particularly important for dealing with situations where modifying the packets in this way has negative side-effects.

PF injects randomness in the following places:

TCP initial sequence number With the 'modulate state' keyword, PF adds a pseudo-random value to the sequence numbers in each direction of a connection. In versions of OpenBSD up to 3.8, this was simply 32 bits of data from arc4random(); a newer algorithm generates a new initial sequence number with the same algorithm as OpenBSD's tcp stack (described above), and stores the difference between this and the original timestamp. This ensures that PF generated timestamps have the same properties regarding minimum difference between timestamps and minimum repetition times.

nat source port By default the source port of all connections which match a 'nat' rule in PF are randomised. This helps to protects hosts with predictable source ports from blind spoofing attacks. It is possible to use the 'nat' keyword to protect hosts behind a PF firewall by translating the port while not changing the source address.

The port selection mechanism in PF is less constrained than the standard BSD mechanism for selecting ephemeral source ports in that a port can be used simultaneously for multiple connections. A source port for the connection is selected using arc4random, and then the full tuple (address family, protocol, source address, source port, destination address, destination port) is checked against the state table. If there is no *exact* match, the port can be used. This tends to makes it more robust when a large number of connections are in progress, as the initially selected source port is much less likely to be in use; linear searches of the port space are thus less likely, and there is consequently less clustering of active ports.

nat source address By using a pool of multiple translation addresses with NAT and the 'random' address selection algorithm, it's possible to mask the source IP address of a connection. The network section of the address stays the same, while the host portion of the address is generated by arc4random();

One can also nat addresses from a net-block to the same net-block. This can protect from disclosure of IPv6 source addresses and thus leakage of host information, as automatically assigned v6 addresses are based on the MAC address of the host.

rdr destination address This randomises incoming connections being translated to a pool of servers, preventing an attacker from being able to predict which server in the pool a specific connection will be directed to.

ip id Many hosts generate predictable values for the ip_id; this defeats ip_id based NAT detection, and protects hosts behind the firewall from attacks on their poor id selection. The identifier field in the IPv6 fragment extension header cannot currently be modified by PF as PF does no reassembly of IPv6 fragments.

TCP timestamp modulation Modulation of TCP timestamp values can make blind spoofing attacks more difficult, makes it impossible to analyse TCP timestamp values to count the number of hosts behind a NAT device, and prevents the leakage of the real system time of machines behind the firewall.

Much of the above PF randomness is redundant for connection originating on an OpenBSD, however it *is* useful for protecting non-OpenBSD systems.

Like the end-point randomisations discussed above, the performance cost of these techniques is negligible. However, while these randomisation techniques are transparent to the majority of protocols and applications, a number of problems exist:

The first two problems are related in that they are the result of what would now be considered poor protocol design or implementation: given the prevalence of firewalls and network translation on the Internet, new protocols should not depend on untranslated end-to-end connectivity, or the ability to connect or arbitrary ports.

Firstly, some applications require the source port of a connection to be a specific value; examples include protocols which use the fact that the source port is lower than 1024 to infer that the originating user has superuser privilege, or certain IKE/ISAKMP implementations which require that the source port be 500. Other applications require the source address to remain constant across connections which take place within a single session. Address and port translation interfere with these types of applications.

Secondly, some legacy protocols make reverse connections to ports identified within the protocol. Because the host behind the firewall is unaware of the translation, it sends its actual IP and port, which is not valid from the perspective of the other endpoint. Examples of such protocols are FTP and H.323.

The third issue relates to integrity protection mechanisms. Network protocols as IPSec AH or TCP MD5 attempt to authenticate parts of the packet header. When authenticated values are changed by the firewall in an attempt to add randomness, the authentication fails and the recipient of the packet rejects the packet as invalid. Therefore these techniques cannot be used unless the firewall also knows the secret authentication key and can recalculate the authenticating hash when changes are made. Such a feature has not yet been implemented in PF, and thus there must be a mechanism to be selective about what traffic the firewall will attempt to modify.

The final issue concerns the robustness of the firewall platform. Because these randomness mechanisms are active and depend on the translation data mapping the original data to the random data inserted by the packet filter, connections using these techniques are brittle in the face of firewall reboots (due to failure or maintenance) or flushes of the state table. While the firewall rules may still allow packets from the session to pass, the necessary translations will not occur or a different mapping will be established by the firewall, and the packets will be rejected as invalid by the recipient.

Where possible, OpenBSD provides mechanisms to deal with these issues without requiring that all the functionality is disabled. Because this the randomness is being added by the firewall, rulesets can be configured to avoid the use of randomess only for protocols which break it; Additionally, for randomisation of source IP address, the 'sticky-address' keyword allows the same randomised mapping to be used for multiple distinct connections, allowing applications which require a constant source address to work correctly.

Application-level proxies can also be used to deal with protocols such as FTP to ensure that appropriate translations are made in the application data to match the randomisation of packet header fields. Many firewalls attempt with varying success to do this in the packet filter code

Finally, the pfsync and carp protocols can be used to provide firewall redundancy, ensuring that pseudo-random translations are not lost in the event of a firewall reboot or a hardware failure. Explicit flushing of the PF state table still results in the loss of these mappings.

Results and Conclusions

OpenBSD's policy regarding network stack randomness has proven itself by providing protection against vulnerabilities unknown at the time that the feature was implemented.

From a performance standpoint, the cost of these techniques is minimal. Furthermore, most of these techniques have no impact on the protocols. Where such impacts do occur, OpenBSD is careful to provide mechanisms to deal with this, for example by restricting the introduction of randomness. While some of the applications of randomisation in PF can cause application breakage, many of these problems are largely a fundamental problem with the hack of network address translation, and not the randomness being applied.

The OpenBSD project encourages other operating systems to implement these measures; as these techniques become ubiquitous broken broken protocols that depend on non-random behaviour are less likely to be designed or deployed.

Future Work

Over time OpenBSD has seen a convergence between the protections offered by the TCP/IP stack and the packet filter, most recently with TCP timestamp generation in the TCP stack being made to match that in PF, and the generation of initial sequence numbers in PF being made to match that in the TCP stack. This convergence will likely continue over time; because PF provides the ability to control where randomness is applied, it is an ideal testing ground for new techniques.

RFC 1323 [5] describes the use of TCP timestamps for round trip time measurement (RTTM) and protection against wrapped sequence numbers (PAWS). The randomisation of TCP timestamps lays the foundation for the PAWS code to provide additional checks to ensure that TCP timestamps are within an appropriate window. Based on the rules laid out in this RFC, we can track the timestamp sent by the other host as well as the echoed timestamp that we sent, and use them as additional sequence numbers to prevent blind insertion attacks. This functionality has already been implemented in PF by Mike Frantzen, as part of the TCP normalisation code, and

an implementation in OpenBSD's stack would likely be based on this.

As the connection rate that can be handled by a single system increases, we begin to push the limits of the techniques which have worked sucessfully in the past: the current algorithm for ephemeral port selection degrades as the number of ports in use increases, taking longer to find unused ports and clustering the values of new ports selected, making them easier to predict. One available approach is to track the least-recently used ports and select new ports from those; alternatively some preliminary design discussions have taken place regarding exposing PF's state table to other areas of the kernel. Aside from improving the performance of route and port lookups for connections being tracked by PF, use of the state table could allow more aggressive re-use of ephemeral source ports, as is done by PF's translation code; it would also allow both the stack and pf to select ephemeral ports from a larger space rather than having their own slice of the available ports. This technique must be implemented carefully as the TCP stack should work correctly without PF enabled.

References

[1] P. Watson, "Slipping in the window," 2004.

[2] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4," *Lecture Notes in Computer Science*, vol. 2259, pp. 1–24, 2001.

[3] S. R. Fluhrer and D. A. McGrew, "Statistical analysis of the alleged rc4 keystream generator.," in *FSE*, pp. 19–30, 2000.

[4] J. Postel, "RFC 793: Transmission control protocol," Sept. 1981. See also STD0007. Status: STANDARD.

[5] V. Jacobson, R. Braden, and D. Borman, "RFC 1323: TCP extensions for high performance," May 1992. Obsoletes RFC1072, RFC1185. Status: PROPOSED STANDARD.

[6] D. Mills, "RFC 2030: Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI," Oct. 1996. Obsoletes RFC1769. Status: INFORMATIONAL.

[7] D. L. Mills, "RFC 1305: Network time protocol (version 3) specification, implementation," Mar. 1992. Obsoletes RFC0958, RFC1059, RFC1119 . Status: DRAFT STANDARD.

Complete Hard Disk Encryption Using FreeBSD's GEOM Framework

Marc Schiesser
m.schiesser [at] quantentunnel.de

October 20th 2005

Abstract

Most technologies and techniques intended for securing digital data focus on protection while the machine is turned *on* – mostly by defending against remote attacks. An attacker with *physical* access to the machine, however, can easily circumvent these defenses by reading out the contents of the storage medium on a different, fully accessible system or even compromise program code on it in order to leak encrypted information.

Especially for mobile users, that threat is *real*. And for those carrying around sensitive data, the risk is most likely *high*.

This paper describes a method of mitigating that particular risk by protecting not only the data through encryption, but also the applications and the operating system from being compromised while the machine is turned *off*.

The platform of choice will be FreeBSD, as its GEOM framework provides the flexibility to accomplish this task. The solution does not involve programming, but merely relies on the tools already provided by FreeBSD.

Table of Contents

1 Background & motivation....2
2 Partial disk encryption....3
2.1 File-based encryption....4
2.2 Partition-based encryption....5
2.3 The leakage risk....5
2.4 New attack vectors....6
3 Complete disk encryption....6
3.1 Tools provided by FreeBSD....6
3.2 The problem with complete disk encryption....7
3.3 Requirements....8
3.4 Complete hard disk encryption using GBDE....8
3.4.1 Erasing previously stored data....8
3.4.2 Initialization & the lockfile....9
3.4.3 Attaching the encrypted medium....9
3.4.4 Partitioning....10
3.4.5 Creating the filesystem....11
3.4.6 Installing FreeBSD....11
3.4.7 Preparing the removable medium....12
3.4.8 The kernel modules....12
3.4.9 The problem with GBDE....13
3.4.10 The memory disk....13
3.4.11 Populating the memory disk filesystem....14
3.4.12 The booting process....14
3.4.13 Creating the symlinks....15
3.4.14 Integrating the memory disk image....15
3.4.15 The swap partition....16
3.4.16 Post-installation issues....16
3.5 Complete hard disk encryption using GELI....16
3.5.1 Readying the hard disk....17
3.5.2 Improvements and new problems with GELI....17
3.5.3 Initialization, attachment and partitioning....18
3.5.4 Filesystem creation and system installation....19
3.5.5 The removable medium....19
3.5.6 Mounting the encrypted partition....19
4 Complete hard disk encryption in context....20
4.1 New defenses & new attack vectors – again....20
4.2 Trade-offs....22
4.3 GBDE vs. GELI....23
5 Conclusion....23
References & further reading....24

1 Background & motivation

As more and more data enters the digital world, appropriate measures must be taken in order to protect it.

Considering the ever-increasing number of networked devices and the inherent exponential growth of the Internet, it is imperative that a large amount of effort go into securing devices against remote attacks. Common technologies and techniques include firewalls, intrusion detection systems (IDS), encryption of all kinds of network transmissions as well as hardening network stacks and fixing buffer overflows.

At the same time, we are witnessing increasingly sophisticated and complex *mobile* devices such as PDAs, smartphones and cell phones becoming pervasive and assuming all kinds of important tasks. Between the general-purpose laptop and the (once) special-purpose cell phone, pretty much anything in between is available.

As people use these devices, they also generate data – either explicitly or implicitly. Explicitly stored data might for example include: entering a meeting into the electronic schedule, storing a telephone number and associating a name with it, or saving an email message draft in order to finish it later.

But then there is also the data which is stored implicitly. Examples include the history of the telephone numbers called or received, browser caches, recently accessed files, silently by the software backed-up data such as email messages, log files and so on.

Even if the user remembers to delete the explicitly stored files after they are no longer needed, it is possible to trace a lot of his or her activity on the device by looking at the aforementioned, implicitly stored data. The more sophisticated the device is, the more such data will usually be generated, mostly without the user's knowledge.

In terms of performance, laptop computers hardly lag behind their desktop counterparts – enabling them to run the same powerful and complex software. It also means that the users tend to generate far more data – both explicitly and implicitly – than on simpler devices.

In addition to being exposed to remote attacks, laptop users are also faced with an increased exposure of the machine itself.

While stationary computers are physically accessible by usually only a limited number of people, a laptop computer is *intended* to be used anywhere and anytime.

This paper does not try to provide any solutions to mitigating the risks of remote attacks. Instead, it concentrates on the risks posed by attackers with *physical* access to the device. An attacker with physical access to a machine can either:

- boot his own operating system, thus overriding any of the restrictions put in place by the defender (login procedures, filesystem and network access control, sandboxes etc.)
- or remove the hard drive from the defender's machine and install it in a system which is under the control of the attacker – in case the target's booting sequence is protected (e.g. by a BIOS password)

Unfortunately, however, most people and companies take quite lax an approach when it comes to protecting their data *in-storage*, while the machine is turned off. The following pieces of news illustrate just how serious a problem the lack of in-storage encryption can become:

- "Thieves stole computer equipment from Fort Carson containing soldiers' Social Security numbers and other personal records, the Army said ..." [Sarche, 2005]

- "Personal devices "are carrying incredibly sensitive information," said Joel Yarmon, who, as technology director for the staff of Sen. Ted Stevens (R-Alaska), had to scramble over a weekend last month after a colleague lost one of the office's wireless messaging devices. In this case, the data included "personal phone numbers of leaders of Congress. . . . If that were to leak, that would be very embarrassing," Yarmon said." [Noguchi, 2005]
- "A customer database and the current access codes to the supposedly secure Intranet of one of Europe's largest financial services group was left on a hard disk offered for sale on eBay." [Leyden, 2004]
- " ... Citigroup said computer tapes containing account data on 3.9 million customers, including Social Security numbers, were lost by United Parcel Service." [Reuters, 2005]
- "Earlier this year, a laptop computer containing the names and Social Security numbers of 16,500 current and former MCI Inc. employees was stolen from the car of an MCI financial analyst in Colorado. In another case, a former Morgan Stanley employee sold a used BlackBerry on the online auction site eBay with confidential information still stored on the device. And in yet another incident, personal information for 665 families in Japan was recently stolen along with a handheld device belonging to a Japanese power-company employee." [Noguchi, 2005]
- " ... trading firm Ameritrade acknowledged that the company that handles its backup data had lost a tape containing information on about 200,000 customers. " [Lemos, 2005]
- "MCI last month lost a laptop that stores Social Security numbers of 16,500 current and former employees. Iron Mountain, an outside data manager for Time Warner, also lost tapes holding information on 600,000 current and former Time Warner workers." [Reuters, 2005]

Even though the number of press articles reporting damage due to stolen mobile computers – or more specifically: storage media – does not reach the amount of publicity that remotely attacked and compromised machines provoke, it must also be taken into account that data on a laptop does not face as much exposure as it does on an Internet server.

A laptop computer can be insured and data regularly be backed up in order to limit the damage in case of *loss or theft*; but protecting *the data from unauthorized access* requires a different approach.

2 Partial disk encryption

Encryption of in-storage data (as opposed to in-transmission) is not a completely new idea, though. There have been several tools around for encrypting *individual files* for quite some time. Examples include the famous PGP (Pretty Good Privacy) as well as its free equivalent GnuPG and the somewhat less known tools AESCrypt[1] and NCrypt[2].

More sophisticated approaches aim towards encrypting *entire partitions.* The

1 http://aescrypt.sourceforge.net/
2 http://ncrypt.sourceforge.net/

vncrypt[3] project is an example that takes this approach.

2.1 File-based encryption

The idea is that the user can decide for each file individually, whether and how it is to be encrypted. This has the following implications:

- CPU cycles can be saved on data that the user decides is not worth the effort. This is an advantage, since encryption requires a lot of processing power. It also allows the user to choose different keys for different files (although reality usually reflects the opposite phenomenon).
- Meta data is not encrypted. Even if the file's contents are sufficiently protected, information such as file name, ownership, creation and modification date, permissions and size are still stored in the clear. This represents a risk which is not to be underestimated.

The usability of in-storage encryption largely depends on how transparent the encryption and decryption process is performed to the user. In order to minimize user interaction, the relevant system calls must be modified to handle the desired cryptography accordingly. That way, neither the user nor the applications must make any additional effort to process encrypted file content, since the kernel will take care of this task.

If system call modification is not going to take place, any program required to process encrypted data must either be modified to perform the necessary cryptographic functions itself or it must rely on an external program for that task. This conversion between *cipher text* and *plain* text – and vice versa – is hardly possible without requiring any user interaction.

Scenario: file-based encryption of huge files

The file might be a database or multimedia container: if cryptography is not performed upon system call invocation, the entire file content must be temporarily stored in plain text – therefore consuming twice the space.

Then the *unencrypted* copy has to be opened by the application. When the file is closed, it obviously has to be encrypted again – unless no modification has taken place. First, the application will therefore save the data in plain text, which must then be encrypted and written out in its cipher text form again – but by a program capable of doing the appropriate encryption.

After encryption, the unencrypted (temporary) copy could of course just be unlinked (removed from the name space), but in that case the unencrypted data would still remain on the medium until physically completely overwritten. So, if one wants to really destroy the temporary copy, several overwrites are required – which can consume a lot of time with large files. Therefore, a lot of unnecessary I/O must be performed.

Scenario: file-based encryption of many files

If one wants to encrypt more than just a small bunch of files, it actually doesn't matter how small or large they are – the procedure described above still must be adhered to. This is going to become a burden very quickly – even more so, if one actually uses *different* passwords for different files.

3 http://sourceforge.net/projects/vncrypt/

The bottom line is that it simply does not scale. In most cases, encryption is therefore either abandoned or the following, more effective and efficient scheme is chosen.

2.2 Partition-based encryption

Obviously, creating a temporary plain text copy of an entire partition each time the data is accessed, is hardly a sane solution. The cryptographic functions must therefore be performed in the kernel, as it has been implemented with GBDE [Kamp, 2003a] and GELI [Dawidek, 2005a] in FreeBSD and `cgd(4)` in NetBSD [Dowdeswell & Ioannidis, 2003], although a few third party add-ons also exist. One example of this is the aforementioned `vncrypt`, which was developed at Sourceforge.

`vncrypt` is, however, in a further sense still file-based, because the encrypted partition is only a mounted *pseudo*-device created via the `vn(4)` facility from a *regular* file. This file holds all the partition's data in encrypted form – including meta data. OpenBSD's `vnconfig(8)` provides a similar feature [OpenBSD, 1993].

One aspect associated with partition-based encryption is that its set-up process is usually more extensive than it is for file-based encryption. But once it has been done, partition-based encryption is far superior to the file-based encryption scheme. All data going to the particular partition is – by default – stored encrypted. As both encryption and decryption is performed *transparently* to the user and *on-the-fly*, it is also feasible to encrypt both large amounts of files and large amounts of data.

But unfortunately, this scheme it not perfect either.

2.3 The leakage risk

As obvious as it may sound, partition-based encryption protects only what goes onto the encrypted partition. The following scenario highlights the particular problem.

<u>Scenario: editing a sensitive document stored on an encrypted partition</u>

A mobile user needs to have a lot of documents (and information in general) at his immediate disposal. Since some information is sensitive, he decides to put it on an encrypted partition.

The problems start as soon as encrypted files are opened with applications that create temporary copies of the files currently being worked on, often in the **/tmp** directory. So unless the user happens to have **/tmp** encrypted, his sensitive data is *leaked* to an *unencrypted* part of the medium. Even if the application deletes the temporary copy afterwards, the data still remains on the medium until it is *physically* overwritten. Meta data such as file name, size and ownership may also have leaked and may therefore remain accessible for some time.

This phenomenon happens equally implicitly with printing. Even if the application itself does not leak any data, the spooler will usually create a Postscript document in a subdirectory of **/var/spool/lpd/**, which is not encrypted unless specifically done so.

Even though it is possible to symlink the "hot" directories such as **/tmp**, **/var/tmp,** as well as the complete **/home** or **/var/spool/lpd/** to directories on the encrypted partition, the leakage risk can never be avoided completely. It is something that users of partition-based encryption just have to be aware of and learn to live with by minimizing the amount of leaked data as much as possible.

The leakage risk is also another reason why file-based encryption is virtually useless. While this issue is certainly a problem for sensitive data, there is a *far* bigger problem, which so far has been quietly ignored.

2.4 New attack vectors

The point of storing data is to be able to retrieve it at some later date. Most of this data is processed by user applications, which in turn require an operating system. So far, everything that was discussed, was based on the assumption that both the OS and the applications were stored *unencrypted* – there is also no point in doing otherwise as long as the data itself is not encrypted.

What follows is the description of one possible way a system can evolve in terms of security:

- if data cannot[4] be destroyed, stolen or modified *remotely*, a dedicated attacker will find a way to gain local (physical) access to the system
- if login procedures, filesystem access control and other restrictions imposed by the OS and applications cannot be defeated or circumvented, the attacker will boot his/her own OS
- if the booting sequence on the machine is protected, the attacker will remove the hard disk and access it from a system under his control
- if the data on the hard disk is encrypted and a brute-force attack is not feasible, then the attacker will most likely[5] target the OS and/or the applications

This situation changes rapidly when the data *is* encrypted: the OS and the applications are now the target. Instead of breaking the encryption, an attacker can try to subvert the kernel or the applications, so they leak the desired data or the encryption key.

The goal is therefore to encrypt the OS and all the applications as well. Just as any security measure that is taken, this scheme involves trade-offs, such as less convenience and decreased performance. These issues will be discussed later. Every user considering this scheme must therefore for him- or herself decide, whether the increase in security is worth the trade-offs.

3 Complete disk encryption

3.1 Tools provided by FreeBSD

The platform of choice here is FreeBSD, because it comes with a modular, very powerful I/O framework called GEOM [Kamp, 2003b] since the release of the 5.x branch. The 5.x branch underwent several major changes compared to the 4.x branch and wasn't declared -STABLE until the 5.3-RELEASE in November 2004. The 5.x branch did,

4 perfect security is not possible; therefore 'cannot' should rather be read as 'cannot easily enough'
5 tampering with the hardware is of course also possible, for example with a hardware keylogger; defending against this kind of attack is not discussed in this paper

however, feature a GEOM class and a corresponding userland utility called GBDE (GEOM Based Disk Encryption) as early as January 2003 when 5.0-RELEASE came out. GBDE was specifically designed to operate on the sector level and is therefore able to encrypt entire partitions and even hard disks or other media.

When the 5.x branch was finally declared -STABLE and therefore ready for production use, 6.x became the new developer branch, carrying all the new, more disruptive features. Into this branch added was also a new module and userland utility called GELI [Dawidek, 2005b]. In addition to containing most of the GBDE features, GELI was designed to enable the kernel to mount the root filesystem (/) from an encrypted partition. GBDE does not allow to do this and therefore requires a "detour" in order to make complete hard disk encryption work.

This paper will discuss the realization of complete hard disk encryption with both tools without having to rely on programming. GELI is a more elegant solution, because it was designed with this application in mind. GBDE, on the other hand, has seen more exposure because it has been available for much longer then GELI and therefore is more likely to have received more testing. Using GBDE for complete hard disk encryption also illustrates some interesting problems inherent with the booting process and how these can be solved.

Which approach is in the end chosen, is left to the user. The following table lists the most important features of GBDE and GELI [Dawidek, 2005b].

	GBDE	***GELI***
First released in FreeBSD	5.0	6.0
Cryptographic algorithms	AES	AES, Blowfish, 3DES
Variable key length	No	Yes
Allows kernel to mount encrypted root partition	No	Yes
Dedicated hardware encryption acceleration	No	Yes, crypto(9)
Passphrase easily changeable	Yes	Yes
Filesystem independent	Yes	Yes
Automatic detach on last close	No	Yes

Table 1: the most important GBDE and GELI features

3.2 The problem with complete disk encryption

There are cases in which it is desirable to encrypt the whole hard disk – especially with mobile devices. This also includes the encryption of the kernel and the boot loader.

Today's computers, however, can't boot encrypted code. But if the boot code is not encrypted, it can easily be compromised. The solution is therefore to store all code necessary for booting and then mounting the encrypted hard disk partition on a medium that can be carried around *at all times*.

While virtually any *removable* medium is easier to carry around than a *fixed* one (or even whole laptop), USB memory sticks are currently the best solution. They provide plenty of space at affordable prices, are easily rewritable many times and easy to use since operating systems can handle them like a hard disk. But most importantly, they are small and light.

Obviously, putting the boot code on a removable medium instead of the fixed hard

disk doesn't solve the problem of compromise – the risk is simply shifted toward the removable medium. But since that medium can be looked after a lot more easily, there *is* a considerable benefit to the user.

3.3 Requirements

Independent of whether GBDE or GELI is used, the following things are required:

- A bootable, removable medium. It will carry the boot code as well as the kernel. This medium is preferably a USB memory stick, because it is small, light and offers a lot of easily rewritable space.
- The device intended for complete disk encryption. Is is very important that this device is capable of booting from the removable medium mentioned above. Especially older BIOSes may not be able to boot from USB mass storage. Bootable CDs will probably work on most machines. Although they work equally well (r/w access is *not* a requirement for operation), they are harder to set up and maintain.
- In order to set up and install everything, a basic FreeBSD system is required. The FreeBSD installation discs carry a "live filesystem" – a FreeBSD system which can be booted directly from the CD. It can be accessed via the `sysinstall` menu entry "Fixit".

All following instructions are assumed to be executed from the aforementioned "live filesystem" provided by the FreeBSD installation discs.

Before proceeding any further, the user is strongly urged to back up all data on the media and the devices in question.

Furthermore, it will be assumed that the hard disk to be encrypted is accessible through the device node ***/dev/ad0*** *and the removable (USB) medium through* ***/dev/da0****. These paths* ***must*** *be adjusted to the actual set-up!*

3.4 Complete hard disk encryption using GBDE

3.4.1 Erasing previously stored data

Before a medium is set up to store encrypted data, it is important to completely erase all data previously stored on it. *All* data on it has to be *physically* overwritten – ideally multiple times. Otherwise the data that has previously been stored unencrypted would still be accessible at the sector level of the hard disk until overwritten by new data. There are two ways to wipe a hard disk clean:

```
# dd if=/dev/zero of=/dev/ad0 bs=1m
```

overwrites the entire hard disk space with zero values. The parameter `bs` sets the block size to 1 MB – the default (512 B) would take a very long time with large disks.

```
# dd if=/dev/random of=/dev/ad0 bs=1m
```

does the same thing, but uses entropy instead of zero values to overwrite data. The problem with the first approach is that it is quite obvious which parts of the medium carry data and which ones are unused. Attackers looking for potential clues about the encryption key can often exploit this information.

In most cases, however, this should not be a major risk. The downside of using entropy is that it requires *far* more processing power than simply filling the hard disk space with zero values. The required amount of time may therefore be too great a trade-off for the additional increase in security – especially on older, slower hardware.

3.4.2 Initialization & the lockfile

After the hard disk to be encrypted has been wiped clean, it can be initialized for encryption. This is done using the `gbde(8)` command:

```
# gbde init /dev/ad0 -L /very/safe/place/lockfile
Enter new passphrase:
Reenter new passphrase:
```

The lockfile is very important, as it is needed later to access the master key which is used to encrypt all data. The 16 bytes of data stored in this lockfile could also be saved in first sector of the medium or the partition, respectively. In that case, however, only the passphrase would be required to get access to the data. The passphrase – however strong it is – will face intensive exposure with mobile devices as it must be typed in every time the system is booted up. It therefore cannot be realistically guaranteed that the passphrase remains only known to those authorized to access the protected system and data.

But since an additional medium is needed anyway in order to boot the core OS parts, it might as well be used as a storage area for the lockfile – effectively functioning as a kind of access token.

With this scheme, two things are required to get access to the data: the passphrase *and* the lockfile. *If the lockfile is unavailable (lost or destroyed), even knowledge of the passphrase will not yield access to the data!*

3.4.3 Attaching the encrypted medium

After the initialization is complete, the encrypted hard disk must now be *attached* – meaning that the user has to provide both the passphrase and the lockfile to `gbde`, which in turn provides (or denies) access to the decrypted data.

```
# gbde attach /dev/ad0 -l /very/safe/place/lockfile
Enter passphrase:
```

If the passphrase and the lockfile are valid, `gbde` creates an additional device node in the **/dev** directory. This newly created node carries the name of the just attached device ("ad0") plus the suffix ".bde".

- **/dev/ad0** can be used to access the *actual* contents of the hard disk, in this case the *cipher text*
- **/dev/ad0.bde** is an abstraction created by GBDE and allows *plain text* access to the data

All reads from and writes to the .bde-node are automatically de-/encrypted by GBDE and therefore no user interaction is required once the correct passphrase and lockfile

has been provided.

The **ad0.bde** node acts just like the original **ad0** node: it can be partitioned using `bsdlabel(8)` or sliced with `fdisk(8)`, it can be formated as well as mounted.

It is important to keep in mind that once a storage area has been attached and the corresponding .bde device node for it has been created, it remains that way until it is explicitly *detached* via the `gbde` command or the system is shut down. *In the period between attaching and detaching, there is no additional protection by GBDE.*

3.4.4 Partitioning

The next step is to partition the hard disk. This is usually done using `sysinstall(8)` – which, unfortunately, does not support GBDE partitions and fails to list device nodes with a .bde suffix. Therefore, this work has to be done using the tool `bsdlabel`.

```
# bsdlabel -w /dev/ad0.bde
# bsdlabel -e /dev/ad0.bde
```

First, a standard label is written to the encrypted disk, so that it can be edited afterwards. `bsdlabel` will display the current disk label in a text editor, so it can be modified. In order to make the numbers in the following example easier to read, the disk size is assumed to be 100 MB. The contents of the temporary file generated by `bsdlabel` might look like this:

```
# /dev/ad0.bde:
8 partitions:
#      size  offset   fstype  [fsize bsize bps/cpg]
 a:  198544     16   unused     0   0
 c:  198560      0   unused     0   0     # "raw" part, don't edit
```

Each partition occupies one line. The values have the following meaning:

column	*description*
1	a=boot partition; b=swap partition ; c=whole disk; d, e, f, g, h=freely available
2 and 3	partition size and its offset in sectors
4	filesystem type: 4.2BSD, swap or unused
5, 6 and 7	optional parameters, no changes required

Table 2: bsdlabel(8) file format

After the temporary file has been edited and the editor closed, `bsdlabel` will write the label to the encrypted hard disk – provided no errors have been found (e.g. overlapping partitions).

It is important to understand the device node names of the newly created partitions. The encrypted boot partition (usually assigned the letter “a”), is now accessible via device node **/dev/ad0.bdea**. The swap partition is **ad0.bdeb** and so on. Just as adding a boot partition to an unencrypted disk would result in a **ad0a** device node, adding an encrypted *slice* holding several partitions *inside* would result in **ad0s1.bdea**, **ad0s1.bdeb** and so on.

An easy way to keep the naming concept in mind is to remember that everything written *after* the .bde suffix is encrypted and therefore hidden even to the kernel until the

device is attached.

For example: **ad0s1.bdea** means that the data on the first slice is encrypted – *including* the information that there *is* a boot partition inside that slice. If the slice is not attached, it is only possible to tell that there *is* a slice on the disk – neither the contents of the slice, nor the fact that there *is* at least one partition inside the slice can be unveiled. In fact, the node **ad0s1.bdea** does not even exist until the slice has been successfully attached, because without having the key (and the lockfile), the kernel cannot know that there is a partition inside the encrypted slide.

Scenario: multiple operating systems on the same disk

It is also possible to have multiple operating systems on the same disk – each on its own slice. The slice containing FreeBSD can be encrypted *completely*, hiding even the fact that the FreeBSD slice contains multiple partitions *inside* (boot, swap, etc). This way, all data on the FreeBSD slice remains protected, while the other operating systems on the machine can function normally on their unencrypted slices. In fact, they can't even compromise the data on the FreeBSD slice – even if an attacker manages to get root access to a system residing on an unencrypted slice.

3.4.5 Creating the filesystem

Now that device nodes for the encrypted partitions exist, filesystems can be created on them:

```
# newfs /dev/ad0.bdea
# newfs /dev/ad0.bded
```

etc.

Note that the swap partition does not need a filesystem; the "c" partition represents the entire (encrypted) disk. This partition must *not* be formated or otherwise be modified!

3.4.6 Installing FreeBSD

Now that the filesystems have been created, FreeBSD can be installed on the encrypted hard disk. Usually, this would be done using `sysinstall` again. But just as `sysinstall` cannot partition and format encrypted media, it cannot install the system on them. The distributions that comprise the FreeBSD operating system, therefore have to be installed *manually.*

The FreeBSD installation disc contains a directory that is named after the release version of the system, for example: 5.4-RELEASE, 6.0-BETA etc. In this directory, each distribution – such as `base`, `manpages` or `src` – has its own subdirectory with an `install.sh` script. The `base` distribution is required, all others are optional.

In order to install the distributions, the encrypted boot partition (and others, if used for example for **/usr**) has to be mounted and the environment variable DESTDIR set to the path where the encrypted boot partition has been mounted. Then all distributions can be installed using their respective `install.sh` script.

The following example assumes that the encrypted boot partition **/dev/ad0.bdea** has been mounted on **/fixed** and the FreeBSD installation disc on **/dist** (the "live-filesystem"

default). If the live-filesystem is used, the **/fixed** directory is easy to create because the root (**/**) is a memory disk.

```
# mount /dev/ad0.bdea /fixed
# export DESTDIR=/fixed
# cd /dist/5.4-RELEASE/base && ./install.sh
You are about to extract the base distribution into /fixed - are you SURE
you want to do this over your installed system (y/n)?
```

After all desired distributions have been installed, there is a complete FreeBSD installation on the encrypted disk and the swap partition is also ready. But since this system cannot be booted from the hard disk, it is necessary to set up the removable medium.

3.4.7 Preparing the removable medium

As it has already been discussed, this medium will not be encrypted. This means that the standard tool `sysinstall` can be used. The removable medium needs one partition of at least 7 MB. This is the absolute minimum and provides only space for the kernel, some modules and the utilities required for mounting the encrypted partition(s). All other modules such as third party drivers need to be loaded *after* `init(8)` has been invoked.

If it is desired, that all FreeBSD kernel modules be available on the removable medium and thus are loadable *before* `init` is called, the slice should be at least 25 MB in size.

The removable medium can be sliced using `fdisk` or via "Configure" - "Fdisk" in the `sysinstall` menu. The changes made to the medium can be applied immediately by hitting "W". After that, the slice has to be labeled (`sysinstall` menu "Label"). All the space on the slice can be used for the boot partition, since the swap partition on the encrypted hard disk will be used. The mount point for the boot partition does not matter; this text, however, will assume that it has been mounted on **/removable**.

`sysinstall` then crates the partition, the filesystem on it and also mounts it on the specified location (**/removable**). After that, `sysinstall` can be quit in order to copy the files required for booting from the removable medium. All that is required is the **/boot** directory – it can be copied from the installation on the encrypted hard disk:

```
# cp -Rpv /fixed/boot /removable
```

3.4.8 The kernel modules

User interaction with GBDE is done through the userland tool `gbde(8)`, but most of the work is carried out by the kernel module **geom_bde.ko**. This module must be loaded before the userland utility is called.

Usually, kernel modules are loaded by `loader(8)` based on the contents of the file **/boot/loader.conf** – then control is passed over to the kernel. In order to have the GBDE module loaded before `init` is executed, it must be loaded in advance by `loader`. The following instruction adds the GBDE kernel module to the `loader` configuration file on the removable medium:

```
# echo geom_bde_load=\"YES\">> /removable/boot/loader.conf
```

In case additional kernel modules are needed at boot time, they must be copied to **/boot/kernel/** and appropriate entries must be added to **/boot/loader.conf** (this file

overrides the defaults in **/boot/defaults/loader.conf**).

In order to save space on the removable medium and therefore also to speed up loading, all kernel modules and even the kernel itself can be gzipped.

```
# cd /removable/boot/kernel
# gzip kernel geom_bde.ko acpi.ko
```

Binary code compresses to about half of the original size and thus brings a noticeable decrease in loading time. The modules which won't be used or later will be loaded from the hard disk can be deleted from the removable medium.

It is important, however, that the code on the removable medium (kernel, modules, etc) is kept in sync with the system on the hard disk.

3.4.9 The problem with GBDE

As discussed earlier, GBDE has been designed with the encryption of partitions and even entire media in mind. Unfortunately, however, the **geom_bde.ko** module does *not* allow the kernel to mount an encrypted partition as the root filesystem.

This is because the passphrase must be provided through the utility *in user space* – even though the module obviously operates *in kernel space.* So, by the time the kernel must mount the root filesystem, the user has not even had the possibility of providing the passphrase and attaching the encrypted device.

There are two solutions to this problem:

- The kernel must be modified to allow mounting of an encrypted root filesystem *by asking for the passphrase in kernel space.* This way, the device node which gives access to the decrypted data (the .bde device node) would be available *before* `init` is started and could be specified in **/etc/fstab** as the root filesystem. The new facility – GELI – has implemented this scheme and therefore makes it a lot easier than the second solution.
- The second solution is not really a solution, but more a "hack", as the shortcomings of GBDE are not solved but avoided. The only conclusion is therefore that the root filesystem *cannot* be encrypted and that the filesystem(s) on the hard disk – although encrypted – must be mounted on directories residing in the *unencrypted* root filesystem. Attaching and mounting the encrypted hard disk must be done *after* the kernel has mounted an *unencrypted* root filesystem and started `init` and subsequently `gbde` from it.

3.4.10 The memory disk

Since the contents of the root filesystem will not be encrypted, it is best to store on it only what is needed to mount the encrypted partitions. Mounting the filesystem on the removable medium as the root filesystem means that the removable medium would have to be attached to the computer while the system is in use and therefore face a lot of unnecessary exposure.

The better solution is to store an image of a *memory disk* on the removable medium, which contains just the utilities necessary to mount the encrypted hard disk. The kernel can mount the memory disk as the root filesystem and invoke `init` on it, so that `gbde` can be executed. After the user has provided the passphrase to the encrypted partitions on the hard disk, the utilities on the memory disk can mount the encrypted partitions and then load the rest of the operating system from the encrypted hard disk – including

all applications and user data.

First, an image for the memory disk must be created on the removable medium.

```
# dd if=/dev/zero of=/removable/boot/mfsroot bs=1m count=10
```

Then a device node for the image is needed, so that a filesystem can be created on it and then mounted.

```
# mdconfig -a -t vnode -f /removable/boot/mfsroot
md1
# newfs /dev/md1
# mount /dev/md1 /memdisk
```

If the output of `mdconfig(8)` differs from “md1”, the path in the following instructions must be adjusted. The assumed mounting point for the memory disk will be **/memdisk**.

3.4.11 Populating the memory disk filesystem

Since this filesystem is going to be mounted as the root filesystem, a directory must be created to serve as the mount point for the encrypted boot partition (**/memdisk/safe**).

```
# cd /memdisk
# mkdir safe
```

Some other directories also act as mount points and don't need to be symlinked to the encrypted hard disk. The directory **/etc**, however, is required, because the `rc(8)` script in it will be modified to mount the encrypted partitions.

```
# mkdir cdrom dev dist mnt etc
```

Now, the lockfile, which is needed to access the encrypted data, must be copied onto the removable medium – turning it into a kind of access token, without which the encrypted data cannot be accessed even with the passphrase available.

```
# cp /very/safe/place/lockfile /memdisk/etc/
```

It is important to remember that the lockfile is updated each time the passphrase is changed.

3.4.12 The booting process

After the kernel has been loaded from the removable medium it mounts the memory disk as the root filesystem and then executes `init`, the first process. `init` in turn calls `rc`, a script that controls the automatic boot process. Since `rc` is a text file rather than a binary executable, it can be easily modified to mount the encrypted boot partition *before* the majority of the system startup – which requires a lot of files – takes place. The `rc` script can therefore be copied from the installation on the hard disk and then be edited.

```
# cp /fixed/etc/rc /memdisk/etc/
```

The following commands have to be inserted after the line “export HOME PATH” (in 5.4-RELEASE: line 51) into **/memdisk/etc/rc**:

```
    /rescue/gbde attach /dev/ad0 -l /etc/lockfile && \
```

```
/rescue/mount /dev/ad0.bdea /safe && \
/rescue/rm -R /etc && \
/rescue/ln -s safe/etc /etc
```

The four commands first attach the encrypted boot partition, mount it on **/safe** and then erase the **/etc** directory from the memory disk, so that it can be symlinked to the directory on the encrypted disk.

Obviously, the utilities in the **/rescue** directory need to be on the memory disk. The **/rescue** directory is already part of a FreeBSD default installation and provides *statically linked* executables of the most important tools. Although the size of the **/rescue** directory seems at first glance to be huge (~470 MB!), there is in fact *one* binary which has been *hardlinked* to the various names of the utilities. The **/rescue** directory therefore contains about 130 tools which can be executed *without any dependencies on libraries.* The total size is less than 4 MB. Although this fits easily on the created memory disk, the directory cannot be just copied. The following example uses `tar(1)` in order to preserve the hardlinks.

```
# cd /fixed
# tar -cvf tmp.tar rescue
# cd /memdisk
# tar -xvf /fixed/tmp.tar
# rm /fixed/tmp.tar
```

3.4.13 Creating the symlinks

The files required for mounting the encrypted boot partition are now in place and the `rc` script has also been appropriately modified. But since the encrypted boot partition will not be mounted as the root (/), but in a subdirectory of the memory disk (**/safe**), all of the relevant directories must have entries in the root pointing to the actual directories in **/safe**.

```
# umount /fixed
# mount /dev/ad0.bdea /memdisk/safe
# cd /memdisk
# ln -s safe/* .
```

3.4.14 Integrating the memory disk image

The memory disk image now contains all the necessary data, so it can be unmounted and detached (if the memory disk image was not previously accessible through **/dev/md1**, the third line must be adjusted).

```
# umount /memdisk/safe
# umount /memdisk
# mdconfig -d -u1
```

In order to save space and to speed up the booting process, the memory disk image can also be gzipped, just like the kernel modules and the kernel itself:

```
# gzip /removable/boot/mfsroot
```

If the kernel was compiled with the MD_ROOT option – which is the case with the

GENERIC kernel – it is able to mount the root from a memory disk. The file that holds the image of the memory disk must be loaded by the FreeBSD `loader`. This works almost the same way as with kernel modules, as the image must be listed in the configuration file **/boot/loader.conf**. Compared to executable code however, the memory disk image must be explicitly specified as such in the configuration file, so the kernel knows how to handle the file's contents. The following three lines are required in **/boot/loader.conf**:

```
mfsroot_load="YES"
mfsroot_type="mfs_root"
mfsroot_name="/boot/mfsroot"
```

It is also important to note that there is no need to maintain an extra copy of the **/etc/fstab** file on the removable medium as the kernel automatically mounts the first memory disk that has been preloaded. Although this **/etc/fstab** issue is not a major problem, it is a necessary measure in order to make this scheme work with GELI – which is able to mount an encrypted partition as the root filesystem.

3.4.15 The swap partition

Although the swap partition has already been set up and is ready for use, the operating system does not yet know which device to use. It is therefore necessary to create an entry for it in the file **/etc/fstab**. This file must be stored on the hard disk, *not* the removable medium.

```
# mount /dev/ad0.bdea /fixed
# echo "/dev/ad0.bdeb   none   swap   sw   0   0" > /fixed/etc/fstab
```

Now, the system is finally ready and can be used by booting from the removable medium. The modified `rc` script will ask for the passphrase and then mount the encrypted partition, so that the rest of the system can be loaded.

3.4.16 Post-installation issues

Since the system on the encrypted disk was not installed using `sysinstall`, a few things such as setting the timezone, the keyboard map and the *root password* have not yet been taken care of. These settings can easily be changed by calling `sysinstall` now. Packages such as the X server, which is not part of the system, can be added using `pkg_add(8)`. The system is now fully functional and ready for use.

3.5 Complete hard disk encryption using GELI

This chapter describes the process of setting up complete hard disk encryption using FreeBSD's new GELI facility. GELI is so far only available on the 6.x branch. It is important to note that the memory disk approach as discussed previously with GBDE is also possible with GELI. But since GELI makes it possible to mount an encrypted partition as the root filesystem, the memory disk is not a requirement anymore. This advantage, however, is somewhat weakened by a drawback that the memory disk scheme does not suffer from. This particular issue will be discussed in more detail later

and ultimately it is up to the user to decide which scheme is more appropriate.

As the concept of having a memory disk with GELI is very similar to having one with GBDE, this chapter discusses only how to use GELI to boot *directly* with an encrypted root filesystem – *without* the need for a memory disk.

Many of the steps required to make complete hard disk encryption work with GBDE are also necessary with GELI – regardless of whether a memory disk is used or not. Therefore the description and explanation of some steps will be shortened or omitted completely here. The necessary commands will of course be given, but for a more detailed explanation the respective chapters in the GBDE part are recommended for reference.

3.5.1 Readying the hard disk

As it has already been mentioned in the GBDE chapter, erasure of previously stored data on the medium intended for encryption is strongly recommended. The data can be overwritten by either using the zero or the entropy device as a source.

```
# dd if=/dev/zero of=/dev/ad0 bs=1m
```

-- or --

```
# dd if=/dev/random of=/dev/ad0 bs=1m
```

Their respective advantages and drawbacks were discussed in chapter 3.4.1.

3.5.2 Improvements and new problems with GELI

Just as GBDE, GELI must first initialize the medium intended for encryption. GELI's big advantage over GBDE for the purpose of complete hard disk encryption is that it enables the kernel to mount an encrypted partition as the root filesystem. This works by passing the -b parameter to the `geli(8)` userland tool when the hard disk is initialized. This parameter causes GELI to flag the partition as “ask for passphrase upon discovery”.

When the kernel initializes the various storage media in the system at boot time, it searches the partitions on them for any that have been flagged by the user and then asks for the passphrase of the respective partition. The most important fact is, that this is done in kernel space – the new device node providing access to the plain text (with the suffix .eli, analogous to GBDE's .bde suffix) therefore already exists *before* the kernel mounts the root filesystem.

Furthermore – as it is possible with GBDE – GELI also allows the key material to be retrieved from additional sources besides the passphrase. While GBDE uses the 16-byte *lockfile* for this purpose, GELI supports the specification of a *keyfile* with the -K parameter. The size of this keyfile is not hardcoded into GELI and can be chosen freely by the user; if '–' instead of a file name is given, GELI will read the contents of the keyfile from the standard input.

This way it is even possible to concatenate several files and feed them to GELI's standard input through a pipeline. The individual files would then each hold a part of the key and the key would therefore be distributed across several (physical, if chosen) places.

Unfortunately, however, the keyfile *cannot* be used with partitions which have been flagged for “ask for passphrase upon discovery”. Using a passphrase and a keyfile to grant access to the encrypted data would require that a parameter be passed to the

kernel – specifying the path to the keyfile. This path could of course also be hardcoded into the kernel, for example that the keyfile must be located at **/boot/geli.keys/<device>**.

Unfortunately, this functionality does not yet exist in GELI. The ability to mount an encrypted partition as the root filesystem comes therefore at the price of having to rely only on the passphrase to protect the data. The memory disk approach that was discussed in order to make complete hard disk encryption work with GBDE also works with GELI. Although it is harder to set up and maintain, it combines the advantages of "something you know" and "something you have", namely a passphrase *and* a lockfile/keyfile. Especially on mobile devices it is risky to rely only on a passphrase, since it will face intensive exposure as it must be typed in each time the system is booted up.

The choice between better usability and increased security is therefore left to user.

3.5.3 Initialization, attachment and partitioning

Initializing the hard disk with GELI works similarly as it does with GBDE – except that the partition must be flagged as "ask for passphrase upon discovery" and therefore cannot (yet) use a keyfile.

```
# geli init -b /dev/ad0
Enter new passphrase:
Reenter new passphrase:
```

Very important here is to specify the -b parameter, which causes the **geom_eli.ko** kernel module to ask for the passphrase if a GELI encrypted partition has been found. The -a parameter can (optionally) be used to specify the encryption algorithm: AES, Blowfish or 3DES.

Attaching the hard disk is also largely the same, again except that the keyfile parameter must be omitted from the command.

```
# geli attach /dev/ad0
Enter passphrase:
```

Upon successful attachment, a new device node will be created in the **/dev** directory which carries the name of the specified device plus a ".eli" suffix. Just like the ".bde" device node created by GBDE, this node provides access to the plain text. The output of `geli` after successful attachment looks something like this (details depend on the parameters used and the available hardware):

```
GEOM_ELI: Device ad0.eli created.
GEOM_ELI:     Cipher: AES
GEOM_ELI: Key length: 128
GEOM_ELI:     Crypto: software
```

Since `sysinstall` cannot read GELI encrypted partitions either, the partitioning must be done using the `bsdlabel` tool.

```
# bsdlabel -w /dev/ad0.eli
# bsdlabel -e /dev/ad0.eli
```

Partition management was discussed in more detail in chapter 3.4.4.

3.5.4 Filesystem creation and system installation

Now that the partition layout has been set, the filesystem(s) can be created, so FreeBSD can be installed.

```
# newfs /dev/ad0.elia
# newfs /dev/ad0.elid
```

etc.

The actual installation of the system on the encrypted hard disk must also be done manually, since `sysinstall` does not support GELI encrypted partitions.

```
# mount /dev/ad0.elia /fixed
# export DESTDIR=/fixed
# cd /dist/5.4-RELEASE/base && ./install.sh
You are about to extract the base distribution into /fixed - are you SURE
you want to do this over your installed system (y/n)?
```

3.5.5 The removable medium

Since this medium is not going to be encrypted, it can be sliced and partitioned with `sysinstall`. The size requirements are largely the same as for GBDE – the minimum is even a bit lower because there is no need to store the image of the memory disk. With a customized kernel, this minimum may be as low as 4 MB.

In order to boot the kernel from the removable medium (**/removable**), it is necessary to copy the **/boot** directory from the encrypted hard disk (mounted on **/fixed**).

```
# cp -Rpv /fixed/boot /removable
```

All kernel modules except **geom_eli.ko** (and **acpi.ko**, if used) can be deleted if space is a problem. Further, all modules and even the kernel can be gzipped. This saves not only space, but also reduces loading time.

```
# cd /removable/boot/kernel
# gzip kernel geom_eli.ko acpi.ko
```

Just as it is the case with GBDE, GELI also needs its kernel module **geom_eli.ko** loaded by `loader(8)` in order to ask for the passphrase before the root filesystem is mounted. The following command adds the appropriate entry to **/boot/loader.conf**.

```
# echo geom_eli_load=\"YES\">> /removable/boot/loader.conf
```

3.5.6 Mounting the encrypted partition

Because of GELI's ability to mount encrypted partitions as the root filesystem the *entire* workaround with the memory disk can be avoided. So far, however, the kernel does not know which partition it must mount as the root filesystem – even if the device node to the plain text of the encrypted hard disk has been created by GELI. The memory disk approach, which is necessary to make complete hard disk encryption work with GBDE, has the advantage that the kernel will automatically mount the memory disk as the root filesystem if an image has been preloaded.

In this case, however, it is necessary to create an entry in **/etc/fstab**, so the kernel knows which partition to mount as the root filesystem.

```
# mkdir /removable/etc
```

```
# echo "/dev/ad0.elia   /      ufs    rw   1      1" >> /removable/etc/fstab
```

It is important to note that this file must be stored on the *removable* medium and serves only the purpose of specifying the device for the root filesystem. As soon as the kernel has read out the contents of the file, it will mount the specified device as the root filesystem and the files on the removable medium (including **fstab**) will be *outside* of the filesystem name space. This means that the removable medium must first be mounted before the files on it can be accessed through the filesystem name space. It also means, however, that the removable medium can actually be *removed* after the root filesystem has been mounted from the encrypted hard disk – thus reducing unnecessary exposure. It is crucial that the removable medium be always in the possession of the user, because the whole concept of complete hard disk encryption relies on the assumption that the boot medium – therefore the *removable* medium, not the hard disk – is uncompromised and its contents are trusted.

If any other partitions need to be mounted in order to boot up the system – for example **/dev/ad0.elid** for **/usr** – they must be specified in **/etc/fstab** as well. Since most installations use at least one swap partition, the command for adding the appropriate entry to **/etc/fstab** is given below.

```
# echo "/dev/ad0.elib   none   swap   sw   0   0" > /fixed/etc/fstab
```

The system is now ready for use and can be booted from the removable medium. As the different storage devices in the system are found, GELI searches them for any partitions that were initialized with the `geli init -b` parameter and asks for the passphrase. If the correct one has been provided, GELI will create new device nodes for plain text access to the hard disk and the partitions on it (e.g. **/dev/ad0.elia**), which then can then be mounted as specified in **/etc/fstab**.

After that, the rest of the system is loaded. `sysinstall` can then be used in order to adjust the various settings that could not be set during the installation procedure – such as timezone, keyboard map and especially the root password!

4 Complete hard disk encryption in context

4.1 New defenses & new attack vectors – again

Any user seriously thinking about using complete hard disk encryption should be aware of what it actually protects and what it does not.

Since encryption requires a lot of processing power and can therefore have a noticeable impact on performance, it is usually not enabled by default. FreeBSD marks no exception here. Although it provides strong encryption algorithms and two powerful tools for encrypting storage media, it is up to the user to discover and apply this functionality.

This paper gave instructions on how to encrypt an entire hard disk while most of the operating system is still stored and loaded from it. It is important to remember, however, that FreeBSD – or any other software component for that matter – will not warn the user if the encrypted data on the hard disk is leaked (see chapter 2.3) or intentionally copied to another, unencrypted medium, such as an external drive or a smart media card. It is the responsibility of the user to encrypt these media as well.

This responsibility applies equally well to data *in transit*. Network transmissions are

in most cases not encrypted by default either. Since all encryption and decryption of the data on the hard disk is done transparently to the user once the passphrase has been provided, it is easy to forget that some directories might contain data which is stored on a different machine and made available through NFS, for example – in which case the data is transferred *in the clear* over the network, unless *explicitly* set up otherwise.

The mounting facility in UNIX is very powerful; but it also makes it difficult to keep track of which medium actually holds what data.

The network poses of course an additional threat, because of an attacker's ability to target the machine remotely. The problem has already been discussed in chapter 1. If a particular machine is easier to attack remotely than locally, any reasonable attacker will not even bother with getting physical access to the machine. In that chase it would make no sense to use complete hard disk encryption, because it does not eliminate the weakest link (the network connectivity).

If, on the other hand, not the network, but the unencrypted or not fully encrypted hard disk is the weakest link and the attacker is also capable of getting physical access to the machine (for reasons discussed in chapter 2.4), then complete hard disk encryption makes sense.

As soon as complete hard disk encryption is in use, it is quite possible that the weakest link is now the network connectivity again – because compromising the operating system or the applications in order to leak encrypted data is likely to be much harder than for example exploiting a buffer overflow in a server.

A key point to remember is that as long as a particular storage area is attached, the data residing on it is not protected any more than any other data accessible to the system. This applies to both GBDE and GELI; even unmounting an encrypted storage area will not protect the data from compromise since the corresponding device node providing access to the plain text still exists. In order to remove this plain text device node, the storage area in question must be *detached.* With GBDE this must be done manually, GELI has a feature that allows for automatic detachment on the last close – but this option must be *explicitly* specified.

Since the partition holding the operating system must always be attached and mounted, its contents are also vulnerable during the entire time the system is up. This means that remotely or even locally introduced viruses, worms and trojans can compromise the system in the same way they can do it on a system *without* complete hard disk encryption.

Another way to attack the system would be by compromising the hardware itself , for example by installing a hardware keylogger. This kind of attack is very hard to defend against and this paper makes no attempt to solve this issue.

What complete hard disk encryption *does* protect against, is attacks which aim at either accessing data by reading out the contents of the hard disk on a different system in order to defeat the defenses on the original system or by compromising the system stored on the hard disk, so the encryption key or the data itself can be leaked. Encryption does *not,* however, prevent the data from being destroyed, both accidentally and intentionally.

If it is chosen that the encrypted partition is mounted directly as the root filesystem – without the need for a memory disk, then it is crucial that a strong passphrase be chosen, because that will be the only thing required to access the encrypted data. Choosing the memory disk approach makes for a more resilient security model, since it enables the user to use a lockfile (GBDE) or a keyfile (GELI) – in order to get access to the data.

While all these previously mentioned conditions and precautions matter, it is absolutely crucial to understand that the concept of complete hard disk encryption

depends upon the assumption that the data on the removable medium is *trusted*. The removable medium must be used because most of the hardware is not capable of booting encrypted code. Since the kernel and all the other code necessary for mounting the encrypted partition(s) must be stored in the clear on the removable medium, the problem of critical code getting compromised has, in fact, not really been solved. The most efficient way to attack a system like this would most likely be by compromising the code on the removable medium.

It is therefore crucial that the user keep the removable medium with him or her at all times. If there is the slightest reason to believe that the data on it may have been compromised, its contents must be erased and reconstructed according to the instructions in the respective GBDE or GELI chapters.

If the removable medium has been lost or stolen *and* there was a keyfile or lockfile stored on it, then two issue must be taken into account:

- The user will not be able to access to encrypted data even with the passphrase. It is therefore strongly recommended that a backup of the keyfile/lockfile be made and kept in a secure place – preferably without network connectivity.
- The second possibility can be equally devastating, since the keyfile/lockfile could fall into the hands of someone who is determined to break into the system. In that case, all the attacker needs is the passphrase – which can be very hard to keep secret for a mobile device. It is therefore recommended that both the passphrase *and* the keyfile/lockfile are changed in the event of a removable medium loss or theft.

4.2 Trade-offs

Complete hard disk encryption offers protection against specific attacks as discussed in chapter 4.1. This additional protection, however, comes at a cost – which is usually why security measures are not enabled by default. In the case of complete hard disk encryption, the trade-offs worth mentioning the most are the following:

- Performance. Encryption and decryption consume a lot of processing power. Since each I/O operation on the encrypted hard disk requires heavy computation, the throughput is usually limited by power of the CPU(s) and not the bandwidth of the storage medium. Especially write operations, which must be encrypted, are noticeably slower than read operations, where decryption is performed. Systems which must frequently swap out data to secondary storage and therefore usually to the encrypted hard disk can suffer from an enormous performance penalty. In cases where performance becomes too big a problem it is suggested that dedicated hardware be used for cryptographic operations. GELI supports this by using the crypto(9) framework, GBDE unfortunately does so far not allow for dedicated hardware to be used and must therefore rely on the CPU(s) instead.
- Convenience. Each time the system is booted, the user is required to attach or insert the removable medium and enter the passphrase. Booting off a removable medium is usually slower than booting from a hard disk and the passphrase introduces an additional delay.
- Administrative work. Obviously the whole scheme must first be set up before it can be used. The majority of this quite lengthy process must also be repeated

with each system upgrade as the code on the removable medium must not get out of sync with the code on the hard disk. As this set-up or upgrade process is also prone to errors such as typos, it may be considered an additional risk to the data stored on the device.

This list is by no means exhaustive and every user thinking about using complete hard disk encryption is strongly encouraged to carefully evaluate its benefits and drawbacks.

4.3 GBDE vs. GELI

FreeBSD provides two tools for encrypting partitions, GBDE and GELI. Both can be used to make complete hard disk encryption work. If GBDE is chosen, the memory disk approach *must* be used, as GBDE does not allow the kernel to mount an encrypted partition as the root filesystem. The advantage is that it is possible to use a lockfile in addition to a passphrase. This makes for a more robust security model and should compensate for the administrative "overhead" caused by the memory disk.

GELI not only makes it possible to use a memory disk too, it also allows the user to choose from different cryptographic algorithms and key lengths. In addition to that it also offers support for dedicated cryptographic hardware devices and of course eliminates the need for a memory disk by being able to directly mount the encrypted boot partition. The drawback of mounting the root directly from an encrypted partition is that GELI so far does not allow for a keyfile to be used and therefore the security of the encrypted data depends solely on the passphrase chosen.

Looking at the features of the two tools, it may seem as though GELI would be the better choice in *any* situation. It should be noted, however, that GBDE has been around for much longer than GELI and therefore is more likely to have received more testing and review.

5 Conclusion

Mobile devices are intended to be used anywhere and anytime. As these devices get increasingly sophisticated, they allow the users to store massive amounts of data – a lot of which may often be sensitive. Encrypting individual files simply does not scale and on top of that does nothing to prevent the data from leaking to other places. Partition-based encryption scales much better but still, a lot of information can be compiled from unencrypted sources such as system log files, temporary working copies of opened files or the swap partition. In addition to that, both schemes do nothing to protect the operating system or the applications from being compromised.

In order to defend against this kind of attack, it is necessary to encrypt the operating system and the applications as well and boot the core parts such as the kernel from a removable medium. Since the boot code must be stored unencrypted in order to be loaded, it must be kept on a medium that can easily be looked after.

FreeBSD provides two tools capable of encrypting disks: GBDE and GELI. Complete hard disk encryption can be accomplished by using either a memory disk as the root filesystem and then mount the encrypted hard disk in a subdirectory or by directly mounting the encrypted hard disk as the root filesystem.

The first approach can be done with both GBDE and GELI and has the advantage that a lockfile or keyfile can be used in addition to the passphrase, therefore providing

more robust security. The second approach omits the memory disk and therefore saves some administrative work. It works only with GELI, however, and does not allow for a keyfile to be used – therefore requiring a trade-off between better usability/maintainability and security.

Under no circumstances does complete hard disk encryption solve all problems related to security or protect against any kind of attack. What it *does* protect against, is attacks which are aimed at accessing data by reading out the contents of the particular hard disk on a different system in order to defeat the original defenses or to compromise the operating system or applications in order to leak the encryption key or the encrypted data itself.

As with any security measure, complete hard disk encryption requires the users to make trade-offs. The increase in security comes at the cost of decreased performance, less convenience and more administrative work.

Complete hard disk encryption makes sense if an unencrypted or partially encrypted hard disk is the weakest link to a particular kind of attack.

References & further reading

Dawidek, 2005a
P. J. Dawidek, *geli – control utility for cryptographic GEOM class*
FreeBSD manual page
April 11, 2005

Dawidek, 2005b
P. J. Dawidek, *GELI - disk encryption GEOM class committed*
http://lists.freebsd.org/pipermail/freebsd-current/2005-July/053449.html
posted on the 'freebsd-current' mailing list
July 28, 2005

Dowdeswell & Ioannidis, 2003
R. C. Dowdeswell & J. Ioannidis, *The CryptoGraphic Disk Driver*
http://www.usenix.org/events/usenix03/tech/freenix03/full_papers/dowdeswell/dowdeswell.pdf
June 2003

Kamp, 2003a
P.-H. Kamp, *GBDE – GEOM Based Disk Encryption*
http://phk.freebsd.dk/pubs/bsdcon-03.gbde.paper.pdf
July 7, 2003

Kamp, 2003b
P.-H. Kamp, *GEOM Tutorial*
http://phk.freebsd.dk/pubs/bsdcon-03.slides.geom-tutorial.pdf
August 19, 2003

Lemos, 2005
R. Lemos, *Backups tapes a backdoor for identity thieves*
http://www.securityfocus.com/news/11048
April 28, 2005

Leyden, 2004
J. Leyden, *Oops! Firm accidentally eBays customer database*
http://www.theregister.co.uk/2004/06/07/hdd_wipe_shortcomings/
June 7, 2004

Noguchi, 2005
Y. Noguchi, *Lost a BlackBerry? Data Could Open A Security Breach*
http://www.washingtonpost.com/wp-dyn/content/article/2005/07/24/AR2005072401135.html
July 25, 2005

OpenBSD, 1993
vnconfig - configure vnode disks for file swapping or pseudo file systems
OpenBSD manual page
http://www.openbsd.org/cgi-bin/man.cgi?query=vnconfig&sektion=8&arch=i386&apropos=0&manpath=OpenBSD+Current
July 8, 1993

Reuters, 2005
Reuters, *Stolen PCs contained Motorola staff records*
http://news.zdnet.co.uk/internet/security/0,39020375,39203514,00.htm
June 13, 2005

Sarche, 2005
J. Sarche, *Hackers hit U.S. Army computers,*
http://www.globetechnology.com/servlet/story/RTGAM.20050913.gtarmysep13/BNStory/Technology/
September 13, 2005

Improving TCP/IP security through randomization without sacrificing interoperability

Michael James Silbersack
The FreeBSD Project

Introduction

Over the years, many security problems have been found in the TCP and IP protocols. This is not surprising; the authors of these protocols probably did not anticipate their creations being used on open, chaotic networks like today's internet. Had they envisioned our present reality, they most certainly would have included provisions to prevent spoofing, modification, and interception of data.

In the face of attackers that can intercept packets, not much can be done to improve TCP/IP without moving to IPSec or other protocols which encrypt the entire packet. However, in the face of spoofing attacks where the attacker has only partial information about the target connection, some improvements can be made.

Over the past few years, FreeBSD has moved slowly to make changes to our TCP/IP stack when security issues that required a change in network visible behavior were announced. There is a simple reason for this – almost every time we have made a reactionary change to the TCP/IP stack, users have reported compatibility problems.

This paper aims to describe the changes that FreeBSD has made to improve network security without sacrificing compatibility, and also to propose some new changes that will increase network security even further.

Four major topics are covered: TCP Initial Sequence Numbers, TCP Timestamps, IP ID values, and ephemeral port randomization.

TCP Initial Sequence Numbers

The topic of TCP initial sequence numbers has been written about many times. The Morris worm made news in 1988, Kevin Mitnick's spoofing attack on Tsutomu Shimomura made news in 1995, "The Problem with Random Increments" appeared in 2001, and Paul Watson's "Slipping in the Window" made the news in 2004. Despite these events, and the publishing of many excellent papers on the topic such as [Zal01] and [Zal02], this is still a topic worth discussing for one main reason: Every operating system still uses a different method of ISN generation!

This divergence is seemingly due to the fact that there is no RFC recommendation on initial sequence numbers that takes all security and compatibility issues into account. Additionally, no paper has reexamined all operating systems to see how effective the response to "Slipping in the Window" has been.

The importance of unpredictable TCP initial sequence numbers

The TCP protocol uses a 32-bit sequence number to track the current state of a connection; this sequence number is incremented for each octet of data sent over the connection, and in response to packets with the SYN or FIN flags. TCP connections are bidirectional, so there are

effectively two sequence numbers that must be tracked per connection, although each is effectively independent.

There are three categories of attacks that can be performed if an attacker can guess the current sequence number of a connection: Connection spoofing, connection resetting, and data injection.

Connection spoofing is potentially the most serious of the attacks, and was described first in [Mor88]. In order to spoof a connection, one must send a fabricated SYN packet with a false source IP address, then guess the sequence number than the destination system will respond with in its SYN-ACK packet. If this value can be guessed and put into a fabricated ACK packet, the server will believe that it has established a connection with the false source IP address. While the attacker can not receive data from the victim in this scenario, he can send data. This is a very dangerous attack when services that can grant permissions based on IP addresses, such as rlogin, are attacked.

As connection spoofing requires an exact guess in order for success to occur, increasing each initial sequence number by a random positive increment over the previously used ISN provides moderate protection from the attack. If a random positive increment in the range of one to one million is used, the attacker must send on average five hundred thousand packets to successfully spoof a single connection. Due to this difficulty and also due to the removal of IP-based authentication in most programs today, connection spoofing is not any longer a serious threat.

Connection resetting attacks have the modest goal of interrupting an existing connection between two hosts. As pointed out in [Wat04], a spoofed RST packet need not be exact, and must only have a value within the TCP sliding window. With many operating systems using a 64K or larger window, this means that a brute force attack on the entire sequence space of a connection would only require 2^32 / 2^16 (65536) packets. When random positive increments are used, and the general range of a connections sequence numbers can be narrowed down, the attack becomes trivial.

Data injection attacks take advantage of the same TCP flaw/feature, but instead of sending a RST packet, they send a data payload. In the case of encrypted connections like SSH/SSL, this will merely result in the connection being terminated. In the case of more free-form protocols such as telnet, commands could probably be successfully injected.

Although most connections are too short lived and unimportant to be worth resetting / injecting data into, [Wat04] points out that BGP sessions are valuable enough to be targeted.

As the result of [Wat04], many improvements to TCP which would make these attacks less likely by reducing the range of sequence values accepted were proposed. Also suggested was the randomization of ephemeral ports to add an additional barrier to the attack. Unfortunately, the complexity, potential compatibility issues, and legal issues surrounding the proposed fixes have caused many operating systems (including FreeBSD) to only partially implement them.

Initial Sequence Number requirements

The original TCP document, RFC 793 states:

```
RFC 793, page 27:
To avoid confusion we must prevent
segments from one incarnation of a
connection from being used while
the same sequence numbers may still
be present in the network from an
earlier incarnation.  We want to
assure this, even if a TCP crashes
and loses all knowledge of the
sequence numbers it has been using.
When new connections are created,
an initial sequence number (ISN)
generator is employed which selects
a new 32 bit ISN.  The generator is
bound to a (possibly fictitious) 32
bit clock whose low order bit is
```

```
incremented roughly every 4
microseconds.  Thus, the ISN cycles
approximately every 4.55 hours.
Since we assume that segments will
stay in the network no more than
the Maximum Segment Lifetime (MSL)
and that the MSL is less than 4.55
hours we can reasonably assume that
ISN's will be unique.
```

Other than stating that the sequence numbers of connections which share the same IP/port tuple should have non-overlapping sequence numbers within the same MSL, which is defined to be 2 minutes, no other requirements are stated.

The goal of having monotonically increasing initial sequence numbers of course only matters when an IP/port tuple is reused within a short period of time. A system reboot (or NAT machine reboot) is one reason this can occur.

Another situation in which port reuse will occur is when a client machine makes frequent connections to a server, going through its entire ephemeral port range in the process. If the client quickly runs through this range and reuses the first ephemeral port, the SYN packet reaching the server will find a socket still in the TIME_WAIT state. In order to maintain the "quiet time" of the TIME_WAIT state, but to still allow new connections on that IP/port tuple to be accepted, the authors of the 4.2BSD TCP/IP stack added a simple check. If the ISN of a SYN packet coming in was greater than the value of the last sequence number used in the connection that previously occupied that IP/port tuple, the old socket would be discarded and a new connection would be established. Given the mod 1 arithmetic used in sequence number calculations and the 32-bit size of sequence numbers, this means that any value up to 31 bits in size greater than the previously used value would be accepted, and any value up to 31 bits less in size would be ignored until the TIME_WAIT socket timed out on its own.

This sequence number check, although originally a quick hack, made its way into many TCP/IP stacks over the years. An operating system that ignores this rule and attempts to send out SYN packets with random ISN values will find that roughly 50% of connections will fail in situations where TIME_WAIT recycling comes into play.

An attempt to emulate the monotonic increase algorithm from RFC 793 while making sequence number prediction hard is what presumably led to the random positive increment algorithms used by many operating systems in the 1990s.

In 2001, as a result of the information in [New01], this author did an ad-hoc survey of the ways in which open source operating systems validated initial sequence numbers, and determined that the BSD-derived TIME_WAIT check is the only actual requirement imposed on a TCP/IP stack author. SYN-ACK packets should exhibit a sequence value greater than the one used by the previous incarnation of a connection on that port, but no known operating system actually checks. Additionally, there is no requirement that an operating system use the same sequence space for SYN and SYN-ACK packets.

An improvement: RFC 1948

In RFC 1948, Steven Bellovin proposed a ISN generation algorithm that would create monotonically increasing ISN values that differed per IP/port tuple:

```
RFC 1948, page 2 - 3
...Instead, we use the current 4
microsecond timer M and set

ISN = M + F(localhost, localport,
remotehost, remoteport).

   It is vital that F not be
computable from the outside, or an
attacker could still guess at
sequence numbers from the initial
sequence number used for some other
connection.  We therefore suggest
that F be a cryptographic hash
function of the connection-id and
some secret data.  MD5 [9] is a
```

```
good choice, since the code is
widely available.
   The secret data can either be a
true random number [10], or it can
be the combination of some per-host
secret and the boot time of the
machine...
```

This algorithm performs exactly as expected, creating a unique value for each IP/port tuple that is then incremented at a constant rate by the system time. Unfortunately, there is one property of this algorithm that precludes it from being used as is. Since the time component increases at a constant rate and the hash component is constant, all future ISNs for a IP/port-tuple may be perfectly predicted once a single value has been observed. This flaw was noted in [Zal01], but no specific improvement was suggested.

Therefore, the following attack (inspired by comments from Robert Watson) is possible when RFC 1948 is used for generating the ISNs sent out in SYN-ACK packets. A spammer with a T1 connection he wishes to keep secret obtains a dial-up connection from an ISP that does not block connections to port 25. The spammer than makes connections from his dynamically assigned IP address to port 25 on each of his intended spam targets, logging the ISN returned and the OS fingerprint detected. Next, the spammer disconnects his modem, and puts the observed data into his mass mailing software. This software then proceeds to use the obtained data to forge connections to each of the target mail servers, causing spam to appear to originate from the dial-up IP address.

RFC 1948 usage in FreeBSD

In the spring of 2001, the results of [New01] showed that the random positive increments algorithm FreeBSD was still using was insecure. As a quick fix, the algorithm used by OpenBSD was ported over. Unfortunately, that algorithm created non-monotonic sequence numbers in SYN packets, and problems with TIME_WAIT recycling were quickly reported by users.

As a result, this author decided to start from scratch, and on August 22nd, 2001 the following ISN generation algorithm was committed to the FreeBSD TCP/IP stack:

ISN values in SYN-ACK packets are given random values, as returned by arc4random().

ISN values in SYN packets are generated by the RFC 1948 algorithm:

ISN = Time + MD5(remoteport, localport, remotehost, localhost, secret)

Time increments at 1MB/second and the secret is a 128-bit system-wide secret value that is seeded when the first outbound connection establishment occurs.

Two user-adjustable values are present:

net.inet.tcp.strict_rfc1948 – When set to 0 (the default), SYN-ACK values are filled with random data. If set to 1, the ISNs of SYN-ACK packets would be generated by the RFC 1948 algorithm.

net.inet.tcp.isn_reseed_interval – This determines the number of seconds between reseeding of the system-wide secret value. If left at 0 (the default), no reseeding will ever occur.

Not using RFC 1948 for SYN-ACK packets was motivated by the predictability issue described in the section above. Ad-hoc research of how other operating systems generated and interpreted SYN-ACK ISNs showed that no common operating system actually cared about monotonicity, so the most secure option was adopted – purely random ISNs.

On the other hand, SYN ISNs needed to be monotonic, due to the TIME_WAIT recycling sequence number requirement discussed above. No secure algorithm other than RFC 1948 could be found that satisfied this requirement. After some consideration, it was determined that prediction of SYN ISNs would not be a commonly abused problem. Such prediction would not allow

for connection spoofing, but would only allow for connection reset/data injection. However, the only connections vulnerable to this would be ones made from a static-IP server to the dynamic IP address which the attacker had previously occupied.

In order to see the results of this implementation, see the graphs in Appendix A.

In the four years since this algorithm was added to FreeBSD, the only major change that has occurred is the addition of TCP syncookies by Jonathan Lemon in December of 2001, as described in [Lem01]. When enabled, as they are by default, syncookies replace the random value in SYN-ACK packets. The algorithm used in SYN packets remains unchanged.

While the use of syn cookies remains controversial (no other operating system uses syn cookies by default), the randomness of the resulting sequence numbers is not in question, as shown by the graphs in Appendix A.

As far as anyone in the FreeBSD project is aware, there have been no reports of compatibility problems caused by this method of ISN generation.

During an audit of the syn cookie code in 2003, one security flaw was found. The secret value used when generating syn cookies was only 32 bits in length, allowing an attacker with a fast processor to perform a brute force hash attack and find out the secret. Once found, the secret could be used to perform perfect connection spoofing attacks against the victim until the secret expired (for about 60 seconds.) The flaw was fixed by simply increasing the secret size to 128 bits, making the attack infeasible. There have been no reports of this flaw being exploited in the wild.

A minor improvement to the FreeBSD algorithm

One oversight in the algorithm currently used by FreeBSD to generate SYN-ACK packets is that it tries to be too random. Specifically, when TIME_WAIT recycling occurs on a socket, a totally new ISN value is chosen. While this works properly under normal circumstances, it means that with certain values of ISN and certain old duplicate packets on the network, old data can be injected into the new connection.

As can be seen in the graph in Appendix B, the proposed change to SYN-ACK generation uses the existing scheme for ISN generation, except when a socket in the TIME_WAIT state is being recycled. In that case, a random positive increment from the previously used sequence number is used. However, if the TIME_WAIT socket expires, as occurs in the 130 second idle time shown above, a fresh ISN is once again chosen. Note that a few ports change their sequence numbers over the 30 second idle period; this appears to be the result of the case where TIME_WAIT sockets are created and expired on the client side of the connection rather than on the server side. This will be examined more thoroughly before the final implementation is completed.

Improving RFC 1948

During a discussion with Jeffery Hsu, which unfortunately can not be located, the idea for an improvement upon RFC 1948 was developed. The basic idea was this: Use two MD5 hashes in RFC 1948, slowly switching between them by averaging these values. This would allow the resulting sequence value to be monotonic, yet unpredictable over long periods of time. Such an algorithm would look like this:

```
md5_1 = MD5(remoteport, localport,
remotehost, localhost, secret1)
md5_2 = MD5(remoteport, localport,
remotehost, localhost, secret2)
ISN = Time +
(md5_1 * (reseed_interval -
elapsed_time)) / (reseed_interval)
+ (md5_2 * (elapsed_time)) /
(reseed_interval)
```

In order for this to work, a few additional

parameters must be specified. The rate of increase of time must be greater than the maximum rate of decrease from md5_1 to md5_2. If this premise is violated, ISNs will actually decrease when certain large values of md5_1 and small values of md5_2 occur.

Reseeding is straightforward in this algorithm. When the elapsed time catches up to reseed_interval, 100% of the value will be from md5_2, and 0% of the value will be from md5_1. At this point, the contents of secret2 should be transferred to secret1, secret2 should be filled with a new random value, and elapsed time should be reset to zero.

To see a visual representation of the ISN values generated by this dual hash algorithm, see Appendix B.

Although this algorithm is an improvement over RFC 1948, it is still predictable until the next reseed occurs. The possibility of using a non-linear function to transition between the two hash values is being investigated.

TCP Timestamps

TCP Timestamp values, as specified in RFC 1323, are intended to improve the performance of TCP by increasing the accuracy of RTT measurement, especially in the case of lost packets, and allow systems to determine if a wrapped sequence number is the result of an old packet or a new connection.

The simplest way to implement TCP timestamps is to use a single global time value for all connections. This is the basic implementation that FreeBSD and most other operating systems use. Unfortunately, this global counter leaks information in two ways. First, as this counter is derived from system uptime, it allows an attacker to know how long the system has been up. Such uptime information could be abused in a variety of ways. A simple scan of a network reveals which systems have long uptimes – and are therefore probably behind on security patches. A more patient attacker who logs this data over a long period of time could learn that a company performs weekly restarts and use this information as part of a timed attack.

The second piece of information leaked by a global counter is a system's identity. Given an range of IP addresses, an attacker looking at timestamp values will be able to determine which IP addresses belong to independent systems, and which IP addresses are aliases belonging to a single machine. This information could be very useful for an attacker – if no obvious security holes are found on one IP address of a machine, he could search all the other IP addresses of the machine for weaknesses, confident that he is still investigating the target machine and not wasting time on a honeypot or some other diversion.

There are three simple solutions to these information leakage problems. First, uptime monitoring may be partially foiled by initializing the global counter to a random value at boot time. Unfortunately, this will be ineffective if an attacker simply probes once per day and records his results. As such, it is an almost useless change.

Solution number two is to switch to using separate timestamp counters for each TCP connection, and to initialize a new connection's timestamp value to 0. This prevents an adversary from learning about a system's uptime, or determining if two IP addresses are hosted by the same computer.

Solution number three differs from number two in that the connection counter is initialized to a random value instead of to zero each time. This change is intended to prevent future attacks which might rely on predicting timestamp values.

While changes number two and three defeat the information leaks listed above, they also go against the spirit of RFC 1323, and may cause problems in certain situations. Section 4 of the RFC discusses how timestamps can be used for PAWS – Protection Against Wrapped Sequence numbers.

Section 1.2 of RFC 1323 describes a case where PAWS would ideally come into play:

```
(2)  Earlier incarnation of the
connection

Suppose that a connection
terminates, either by a proper
close sequence or due to a host
crash, and the same connection
(i.e., using the same pair of
sockets) is immediately reopened.
A delayed segment from the
terminated connection could fall
within the current window for the
new incarnation and be accepted as
valid.
```

If timestamps are generated from a global counter, the PAWS mechanism would have no problem determining that timestamps on packets delayed in the network are old. However, if each connection starts with the timestamp counter at 0, PAWS will be totally foiled, unable to tell new from old packets. In the case that random timestamp initialization is used, PAWS might work in some cases, but be fooled in others – the effects would be unpredictable.

Zeroing or randomizing timestamp values also causes a neat trick used by the Linux TCP/IP stack to break. In Linux, the TIME_WAIT sequence number check has been improved to allow a port to be recycled if the ISN is greater than the previously used value or if the timestamp is greater than the previously used value. This check allows operating systems that used randomized ISN values in SYN packets with a standard timestamp implementation to still recycle ports. However, an operating system that has modified ISN values and timestamps will be out of luck.

The unfortunate part about these changes is that the incompatibilities they cause might not be noticed except under carefully crafted test conditions. While the occurrence of these problems in actual usage is unlikely, the probability is that the problem will occur for some users at some time, which is why these changes have not been implemented in FreeBSD.

Using RFC1948 to improve timestamps

Luckily, there is one potential method of retaining compatibility with the PAWS mechanism while still defeating the information leaks discussed previously. The solution is simple – use the algorithm described in RFC 1948 to generate per-connection timestamps!

Using RFC 1948's algorithm to generate timestamps is not a perfect solution; as with its use in ISNs, it suffers from the issue that it is perfectly predictable to someone who can reconnect with the same IP and port pair. Therefore a service like netcraft, which probes on a regular basis, could determine uptime simply by looking for discontinuities in timestamp values. Someone attempting to determine if two IP addresses were hosted on a single computer could look for matching discontinuities to determine that a reboot of that single machine occurred.

The dual hash improvement on RFC 1948 unfortunately can not be used with timestamps. The differing slopes of each connection would make time measurement more difficult, and the extra math required to generate each timestamp would slow overall throughput.

One additional caveat when implementing RFC 1948 style TCP timestamps is that at least one heuristic in the Linux TCP stack compares the timestamp value of an incoming packet to the timestamp value of other packets to determine if that packet is legitimate when a syn flood is in progress. Assuming that other systems make similar assumptions, perhaps instead of using timestamps that are unique per IP/port pair it would be better to use timestamps that are just unique per IP.

Using timestamps to resist data reset/injection attacks

If TCP Timestamps are made per-tuple unique using the RFC1948 algorithm or

simply randomized at connection start time, using timestamps to greatly improve resistance to blind reset/injection attacks becomes simple to implement. RFC 1323 specifies in section 4.2.2 that timestamps are monotonically incremented at a constant rate between 1 and 1000 ticks per second. This allows a receiver to interpret the sender's timestamps, and use them as additional spoof protection.

Assuming that the sender is following RFC 1323, all a receiver must due in order to make blind spoofing connections on timestamped connections very difficult is to ensure that the following is true for each received packet:

(idle_seconds < 30) && (abs(TScurrent – TSlast) < 32 * 1024)

This still allows any legitimate packet that is up to 30 seconds late in arriving in, while blocking spoofed packets that do not fall into this window. As this algorithm accepts a window of 65536 timestamps out of a possible 2^32 at any point in time, an attacker who attempts to try a brute force reset/injection attack would be required to send an additional 2^16 times as many packets. This increases the difficult of any such attack significantly.

Note that this technique is perfectly compatible with senders using system-wide timestamps and timestamps zeroed at connection start time, but will provide very little added security in those cases.

Unfortunately, the timestamp check must be skipped on idle connections due to the possibility of a host rebooting, losing its timestamp counter, and attempting to reestablish a connection on the same ip/port tuple.

IP ID issues

The problems of sequential IP ID values were described first in [San98] and later in [Fyo] and other places. As of now, FreeBSD has not yet implemented any changes due to the perceived lack of importance of this issue and due to the performance penalties that would be incurred by some of the solutions.

Three main solutions have been implemented in different operating systems to solve the problems of predictable IP ID values.

The simplest option, implemented in Linux, was to use an IP ID value of zero for all packets that had the DF (Don't Fragment) bit set. Unfortunately, while this idea would work if all network devices were RFC compliant, it was discovered that certain network devices would fragment DF packets anyway, leading to a stream of fragments, all with an ID of 0. As a result of such misbehaving devices, the idea of zeroing the IP ID field has been abandoned.

A second solution, now implemented in both Linux and Solaris is to track per-IP state, setting up a seperate IP ID counter for each IP the system communicates with. Unfortunately, this solution would be expensive to implement in FreeBSD; FreeBSD has moved away from looking up per-IP state on every packet reception and transmission. The TCP hostcache, which now stores per-IP information such as MTU, RTT, and other information could be used for this purpose, but it would reduce performance.

A third solution was chosen by the authors of OpenBSD's IP stack. They use a linear congruential generator (LCG) to generate sequences of IP ID values that repeat only after the entire sequence has been cycled. So that the LCG may be reseeded after each cycle without causing a quick reuse of any value, the 16 bit space is split into two 15 bit spaces; the space used is toggled after each cycle. This system will defeat idlescan detection, but may not be as effective at masking packet transmit rate or masking if two IP addresses are hosted by the same machine. If one watches how often a system cycles between the two 15-bit addresses spaces, rough estimates on traffic rates can be gathered. If one notices that two IP addresses always switch IP ID spaces

simultaneously, then they are probably running on the same machine.

One common goal of all of these solutions is to make the time before an IP ID is reused as great as possible. This ideal is mentioned in many documents discussing the topic of IP ID abuse. Fyodor mentions in [Fyo], "This is difficult to get right -- be sure the sequence does not repeat and that individual numbers will not be used twice in a short period."

Despite the pervasiveness of Fyodor's belief, there is in fact no reason why quick recycling of IP ID values is a serious problem.

If two fragmented packets with the same IP ID value are put onto the wire at the same time, there are two possibilities that can occur.

The first possibility is that packet #1 will arrive intact at the destination before packet #2. When this occurs, packet #1 will be reassembled successfully, the reassembly queue will be cleared out, packet #2 will arrive, and it too will be reassembled successfully.

The second possibility is that one of the fragments of packet #1 is lost in transit, and/or the fragments of packet #1 and #2 arrive in some jumbled order. If any of these problems occur, the reassembly process will create a reassembled packet that contains portions of both packets. This corrupt packet will then be handed up to either the TCP or UDP layer, where its checksum will fail verification, and the packet will be discarded. The only way a corrupt packet could be reassembled and passed to an application is if two fragments happen to have the same checksum or if the receiving operating system fails to verify the checksum.

What this means is that in the case where two packets to the same destination are sent with identical IP ID values, the loss of one of the fragments of the first packet will effectively result in the loss of the second packet as well.

Therefore, using a PRNG to generate IP ID values may cause a few extra packet drops in certain unlucky situations where packet loss already exists. These extra packet drops can be considered just like any packet loss – a nuisance, but nothing that TCP and UDP can't handle.

On the positive side, using a PRNG to generate IP ID values totally eliminates any possibility of using a machine as an idlescan drone, estimating traffic rate, or determining how many IP addresses belong to a single host.

Ephemeral Port randomization

In order for a blind spoof attack on a TCP connection to be successful, one of the pieces of information that the attacker must guess is the ephemeral source port used by the client end of the connection. As most operating systems sequentially allocate ephemeral port numbers, narrowing the port used by a recently established connection is relatively easy. All the attacker must do is cause the client to connect to the attacker's machine and determine the ephemeral port used. If the client is running services that perform ident checks, this will be easy to trigger. Other methods of inducing a connection may include sending a message that will bounce to a SMTP server running qmail, connecting to a ftp server using passive mode, or forcing the DNS server on the client machine to perform a TCP DNS lookup.

Randomizing the order of ephemeral port allocation is an obvious method of improving the difficulty of a blind attack. Due to the randomization, the attacker will now have to spoof packets from all ports in the ephemeral port range, rather than just the last 5 to 10. In the case of OSes using the classic ephemeral port range (1024 to 5000), this makes the attack 500 times more difficult, assuming an attack range of 10 before randomization. Operating systems that use large ranges of ephemeral ports

(possibly as large as 1024 to 65535) will require an even greater number of packets to be sent.

Paul Watson's paper "Slipping in the Window" led to a quick implementation of port randomization in FreeBSD. This changed seemed safe, as OpenBSD has randomized ephemeral ports since July of 1996 (revision 1.6 of in_pcb.c.) Unfortunately, a few users started reporting problems soon after the change was made to FreeBSD.

The problem reported was that an accelerating webcache that had been upgraded to include port randomization was suddenly seeing failed connections to the backend web server it connected to. One of the failed connections can be seen in Appendix C. Both the webcache and the webserver were running an up to date version of FreeBSD, and no problems were experienced once the sysctl to disable port randomization was toggled off, eliminating the possibility of an unrelated change that broke the system.

This failure case was not seen prior to port randomization because sequential allocation of ephemeral ports leaves a noticeable amount of time before a port is reused. Randomization, due to its nature, will sometimes cause a port to be reused much more quickly - less than a hundreth of a second in the trace shown here.

While the issue shown here is not directly port randomization's fault – something clearly went wrong in the webserver's TCP state machine – it is also true that just an additional second or two before the port in question was reused would probably have avoided the problem.

The number of TCP stack interactions that will see similar problems to the one captured here is unknown, but these results indicate that if one were to magically add simple port randomization to every machine on the planet at once, many breakage situations such as the one here would be seen.

In order to reduce the likelihood of this problem while retaining the security benefits of port randomization, a method to randomize port use but to ensure that ports are not reused too quickly is needed. Unfortunately, using a linear congruential generator to choose ephemeral ports would not be effective – the length for which a connection stays open is not constant, so a port could still be reused quickly if the previous connection is terminated just before the LCG cycles through all other ports and returns to it.

At present, FreeBSD attempts to avoid this quick port recycling problem by falling back to sequential port allocation whenever the machine is making more than 10 outbound connections per second. This solution is more of a hack than anything, and has been slated to be replaced as soon as a better method can be found.

In discussions with Brooks Davis at BSDCan 2005, a workable system of ensuring that ports would not be recycled too quickly was sketched out. The basic concept is to allocate an array with one slot per ephemeral port. At the time that a connection is terminated, the current time and an amount of buffer time (10 seconds) would be added and stored in the slot for that port. This timestamp would make the first time at which the port could be reused. Port allocation would occur randomly at all times, skipping ports which were marked as not yet ready to reuse. One drawback to this solution is that it would not allow hosts to use the same ephemeral port on two different local IPs simultaneously. As a result, a more creative solution may need to be found.

This system has not yet been implemented. Once it has been implemented and passed preliminary testing, the owner of the troubled accelerator proxy will be one of the first users asked to test the change.

Future Work

Preliminary analysis of the TCP ISN

generation systems of other open source operating systems indicates that they may not meet the security and compatibility criteria set forth in this paper. Research into how these operating systems can be improved will take additional time, and unfortunately can not be put into this edition of the paper.

Also, many of the proposed algorithms in this paper have only had proof of concept implementations, and are not ready for inclusion in the FreeBSD source tree yet. After this paper is presented to wider peer review at EuroBSDCon, work on incorporating the changes can proceed.

Finally, the attacks discussed in [Wat04] and [Gont05] have not been addressed in all operating systems equally, and in some cases have not been addressed at all. A test suite similar that can perform all the described attacks should be created and all operating systems should be put to the test, including FreeBSD.

Once additional work is completed, an updated copy of this paper will be posted at http://www.silby.com/eurobsdcon05/

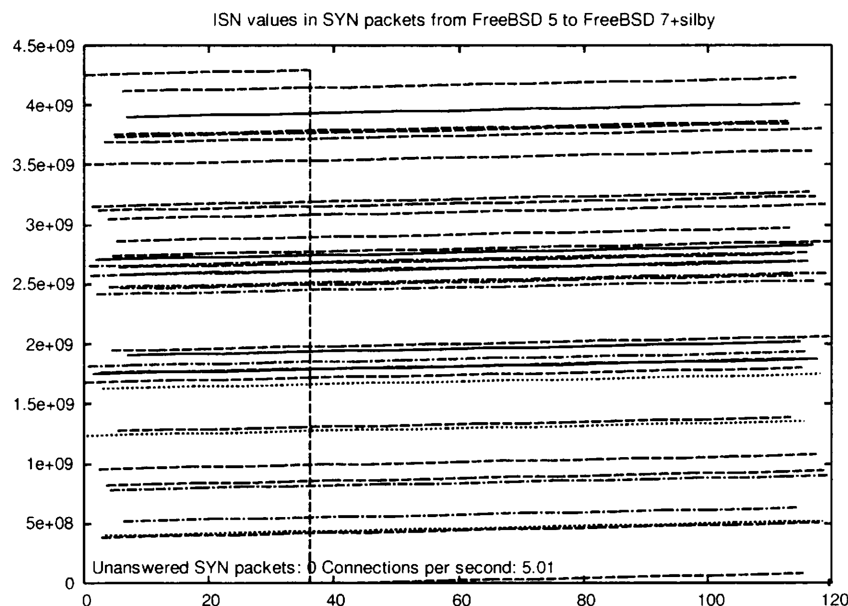
Conclusion

This paper has demonstrated that untested changes to the TCP/IP stack of an operating system can often cause unexpected compatibility issues. However, careful analysis can solve almost any problem, leading to security improvements which do not reduce interoperability.

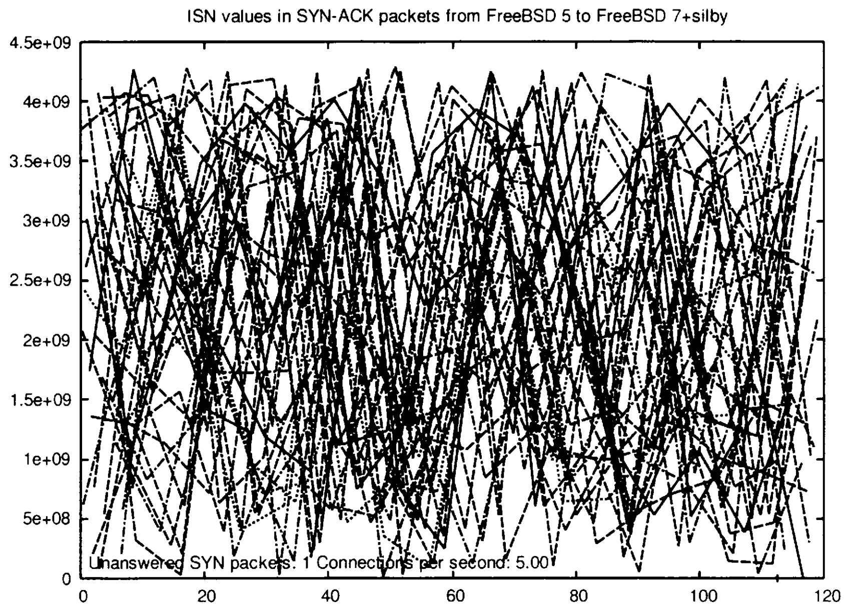
While the new algorithms proposed here have not yet been tested under a wide range of circumstances, it is hoped that the release of this paper will spark a broad discussion on the topic of TCP/IP security, hopefully leading to a new round of standardization that has been sorely lacking in the past few years.

Appendix A: Graph views of FreeBSD 5.4 ISN values

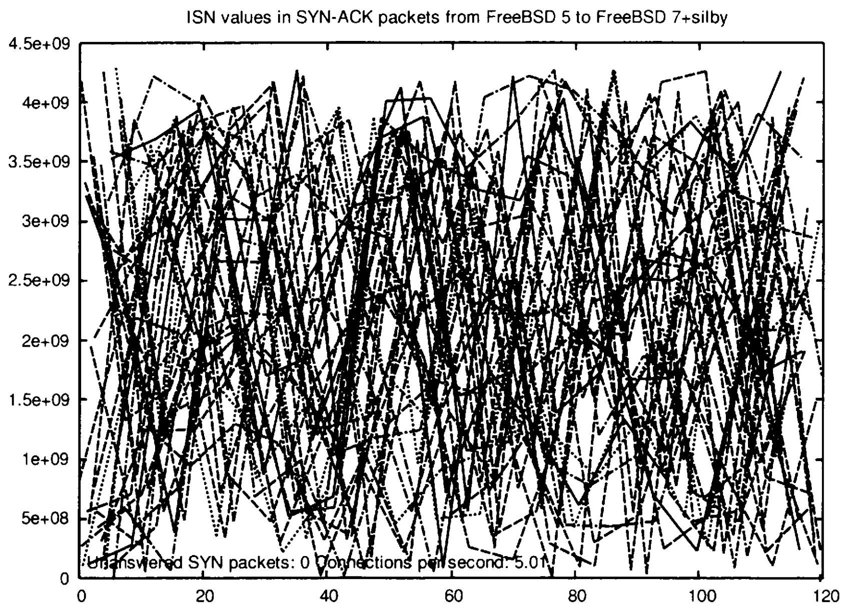
The graphs in Appendices A and B were generated by running a web server on the machine acting as the server in each test, and a http benchmarking tool on the client. To force TIME_WAIT recycling to occur so that the ISN values can be seen per port, the ephemeral port range on the clients was set from 65535 to 65550. The http benchmarking tool was then sent to request a very short HTML page roughly 5 times a second, thereby creating a set of datapoints which was fed into gnuplot. Points are connected together with lines, which is why the graphs of psuedorandom data appear as a graph of haphazard lines rather than a cloud of dots.



ISN values in SYN packets sent from a FreeBSD 5.4 client to a modified FreeBSD 7 server. Notice how each port has a distinct offset from other ports, but how all have the same rate of increase.

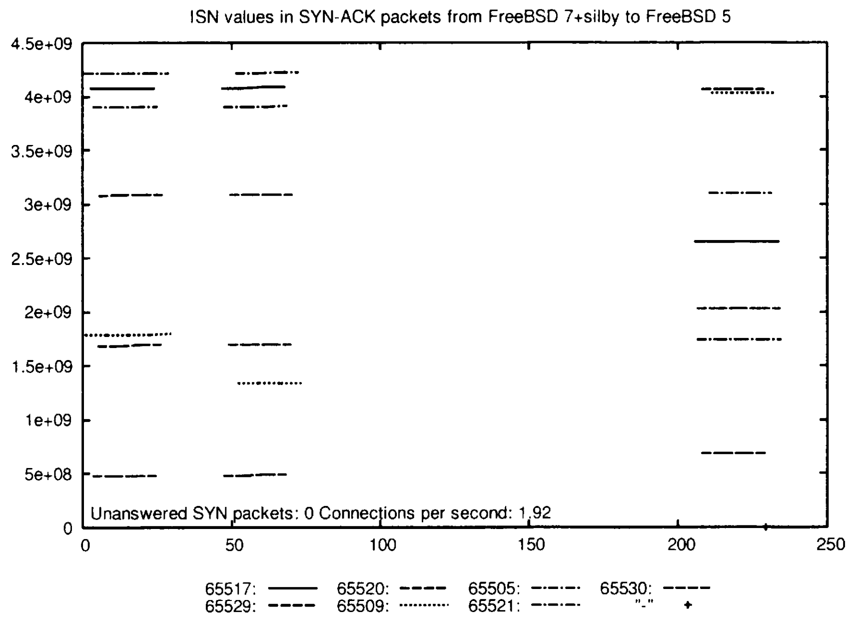


The ISN values in SYN-ACK packets sent by a FreeBSD 5.4 server with net.inet.tcp.syncookies=0. Arc4random is working properly, and prediction of sequence numbers is not possible.

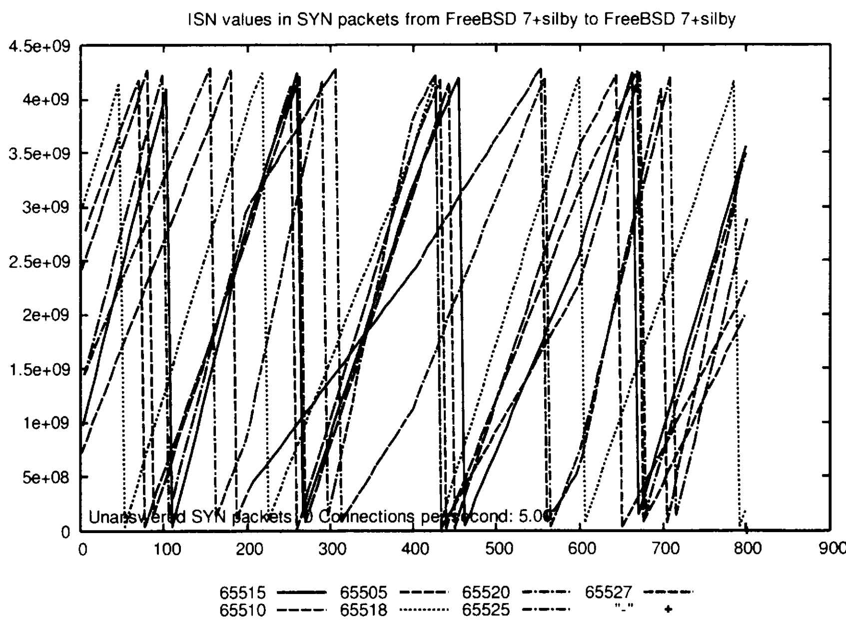


The ISN values in SYN-ACK packets sent by a FreeBSD 5.4 server with net.inet.tcp.syncookies=1. Although syn cookies intentionally create predictability in the short run, it is evident that the long-term effect is similar to randomization.

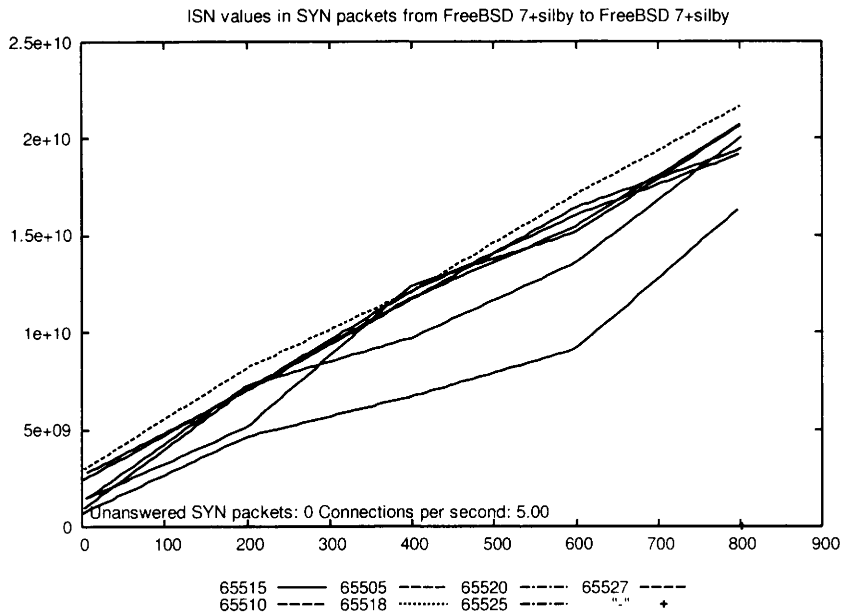
Appendix B: Graphs of proposed changes to FreeBSD's ISN generation schemes



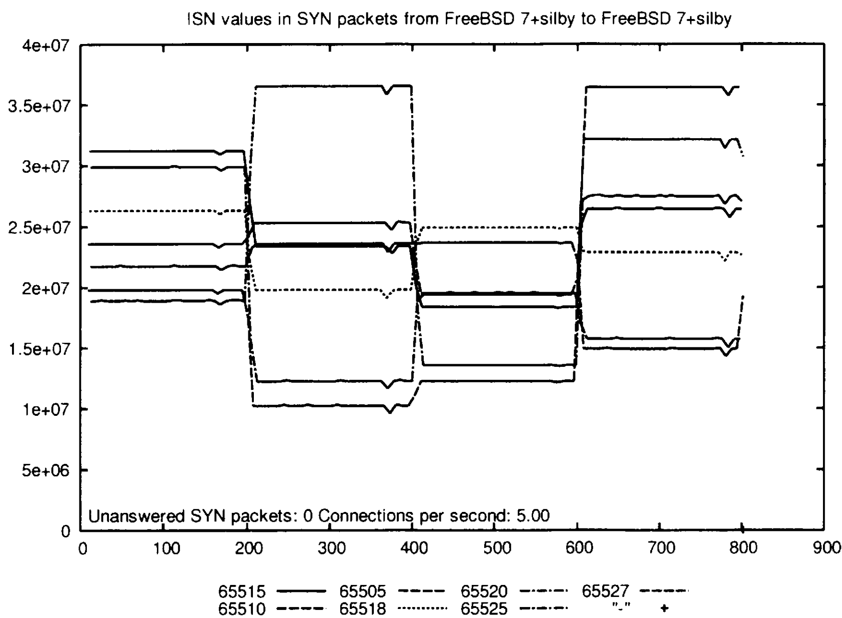
The proposed modification to FreeBSD's SYN-ACK generation is shown. Notice how sequence numbers are the same across the 30 second idle time, but change completely after the 130 second idle time.



A graph of the SYN ISN values from an implementation of the dual hash variant of RFC 1948 using a 200 second reseed interval.



A modification to the dual hash graph so that sequence numbers do not wrap at the 32-bit mark allows for a better view of how the slopes of each port are distinctly different.



A third way of looking at the results of the dual hash algorithm; the first derivative of the ISN values for each port is shown. The slight dips noticeable are due to a glitch in the callout-incremented global time counter.

Appendix C: A failed connection partially due to overly fast ephemeral port recycling

```
17:31:15.372512 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: S 4253937160:4253937160(0) win 8192 <mss
1460,nop,wscale 0,nop,nop,timestamp 152193511 0> (DF)
17:31:15.372642 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: S 1547679919:1547679919(0) ack 4253937161
win 57344 <mss 1460,nop,wscale 0,nop,nop,timestamp 295129972 152193511> (DF)
17:31:15.372656 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: . ack 1547679920 win 8688
<nop,nop,timestamp 152193512 295129972> (DF)
17:31:15.372665 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: P 4253937161:4253937378(217) ack 1547679920
win 8688 <nop,nop,timestamp 152193512 295129972> (DF)
17:31:15.374152 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: . 1547679920:1547681368(1448) ack
4253937378 win 57920 <nop,nop,timestamp 295129972 152193512> (DF)
17:31:15.374243 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: P 1547681368:1547682422(1054) ack
4253937378 win 57920 <nop,nop,timestamp 295129972 152193512> (DF)
17:31:15.374248 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: . ack 1547682422 win 7634
<nop,nop,timestamp 152193515 295129972> (DF)
17:31:15.374253 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: F 1547682422:1547682422(0) ack 4253937378
win 57920 <nop,nop,timestamp 295129972 152193512> (DF)
17:31:15.374257 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: . ack 1547682423 win 8688
<nop,nop,timestamp 152193515 295129972> (DF)
17:31:15.374266 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: F 4253937378:4253937378(0) ack 1547682423
win 8688 <nop,nop,timestamp 152193515 295129972> (DF)
17:31:15.374537 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: . ack 4253937379 win 57920
<nop,nop,timestamp 295129972 152193515> (DF)
17:31:15.389416 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: S 4253971599:4253971599(0) win 8192 <mss
1460,nop,wscale 0,nop,nop,timestamp 152193545 0> (DF)
17:31:15.389598 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: R 1547682423:1547682423(0) ack 4253937379
win 57920 (DF)
17:31:15.389604 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: R 0:0(0) ack 4253971600 win 0 (DF)
```

References:

[RFC793] "RFC 793: Transmission Control Protocol", 1981

[RFC1323] Bellovin, Steven "RFC 1948: Defending Against Sequence Number Attacks", 1996

[Free03] FreeBSD Security Advisory 3:03 – Brute force attack on SYN cookies

[Fyo] Fyodor, "Idle Scanning and Related IPID games"

[Gont05] Gont, F., "ICMP attacks against TCP", September 2005, Internet Draft

[RFC1323] Jacobson, Braden, & Borman "RFC 1323: TCP Extensions for High Performance", 1992

[Lem01] Lemon, Jonathan "Resisting SYN flood DoS attacks with a SYN cache", 2001

[Mor88] Morris, Robert "A Weakness in the 4.2BSD Unix TCP/IP Software, 1985

[New01] Newsham, Timothy "The Problem with Random Increments", 2001

[San98] Sanfilippo, Salvatore Bugtraq posting: "new tcp scan method", 1998

[Wat04] Watson, Paul "Slipping in the window: TCP reset attacks", 2003

[Zal01] Zalewski, Michal "Strange Attractors and TCP/IP Sequence Number Analysis", 2001

[Zal02] Zalewski, Michal "Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later", 2002

A Machine-Independent Port of the MPD Language Run Time System to NetBSD Operating System

Ignatios Souvatzis
University of Bonn, Computer Science Department, Chair V
<ignatios@cs.uni-bonn.de>

20th October 2005

1 Introduction

MPD (presented in Gregory Andrews' book about Foundations of Multithreaded, Parallel, and Distributed Programming[1]) is the successor of SR[2] ("synchronizing resources"), a PASCAL-style language enhanced with constructs for concurrent programming developed at the University of Arizona in the late 1980s[3].

MPD as implemented provides the same language primitives as SR with a different syntax which is closer to C.

The run-time system (in theory, identical) of both languages provides the illusion of a multiprocessor machine on a single single- or multi-CPU Unix-like system or a (local area) network of Unix-like machines.

Chair V of the Computer Science Department of the University of Bonn is operating a laboratory for a practical course in parallel programming consisting of computing nodes running NetBSD/arm, normally used via PVM, MPI, etc.

We are considering to offer SR and MPD for this, too. As the original language distributions are only targeted at a few commercial Unix systems, some porting effort is needed, outlined in the SR porting guide[7] and also applicable to MPD.

The integrated POSIX threads support of NetBSD-2.0 enables us to use library primitives provided for NetBSD's pthread system to implement the primitives needed by the SR and MPD run-time systems, thus implementing 13 target CPUs with a one-time effort; once implemented, symmetric multiprocessing (SMP) would automatically be used on any multiprocessor machine with VAX, Alpha, PowerPC, Sparc, 32-bit Intel and 64 bit AMD CPUs.

This paper describes mainly the MPD port. Porting SR was started earlier and partially described in [6] (Assembler and SVR4 cases) while only preliminary results for our new approach could be presented at the conference.

Most of the differences between our changes to SR and to MPD could be done by mechanically replacing **mpd_** by **sr_** in the code; because of this, and because the

test machine	A	B
architecture	i386	arm
CPU	Pentium 4	SA-110
clock	1600 MHz	233 MHz
cache	2 MB	16kB I + 16 kB D

Table 1: *Test machines*

Implementation	A	B
assembler	0.013 μs	n/a
...context_u libary calls	0.138 μs	0.237 μs
SVR4 system calls	1.453 μs	9.649 μs

Table 2: *Raw context switch times*

machine-independent parts of the SR and MPD run-time support are identical (according to the authors) all results (especially timing results) equally apply to the SR port. (This has been verified.)

2 Generic Porting Problems

Despite the age of SR, the latest version (2.3.3) had been changed to use `<stdarg.h>` instead of `<varargs.h>`, thus cutting the number of patches needed for NetBSD 2.0 and later by half compared to the original porting effort described in [6]. MPD 1.0.1 contains no traces of `<varargs.h>`.

The only patches – outside of implementing the context switching routines – were for 64 bit cleanliness (see also [5]).

3 Verification methods

MPD itself provides a verification suite for the whole system; also a small basic test for the context switching primitives. There is no split between the basic and the extended verification suite, as in SR.

3.1 Context Switch Primitives

The context switch primitives can be independently tested by running `make` in the subdirectory `csw/` of the distribution; this builds and runs the `cstest` program, which implements a small multithreaded program and checks for detection of stack overflows, stack underflows, correct context switching etc.[7] This test is automatically run when building the whole system.

3.2 Overall System

When the context switch primitives seem to work individually, they need to be tested integrated into the run-time system. The SR and MPD authors provide a verification suite in the `vsuite/` subdirectory of the distributions to achieve this, as well as testing the the building system used to build MPD, and the `mpd` compiler, `mpdl` linker, etc.

It is run by calling the driver script `mpdv/mpdv`, which provides options for selecting normal vs. verbose output, as well as selecting the installed vs. the freshly compiled MPD system.

For all porting methods described below (assembler primitives, SVR4 system calls and NetBSD pthread library calls), the full verification suite has been run and any reported problem has been fixed.

Test description	i386 ASM	...context_u	SVR4 s.c.
loop control overhead	0.002 μs	0.002 μs	0.002 μs
local call, optimised	0.011 μs	0.011 μs	0.011 μs
interresource call, no new process	0.270 μs	0.260 μs	0.250 μs
interresource call, new process	0.650 μs	4.200 μs	4.350 μs
process create/destroy	0.540 μs	4.020 μs	4.280 μs
semaphore P only	0.011 μs	0.011 μs	0.011 μs
semaphore V only	0.008 μs	0.008 μs	0.008 μs
semaphore pair	0.019 μs	0.019 μs	0.019 μs
semaphore requiring context switch	0.110 μs	0.220 μs	1.550 μs
asynchronous send/receive	0.300 μs	0.290 μs	0.300 μs
message passing requiring context switch	0.400 μs	0.560 μs	1.920 μs
rendezvous	0.600 μs	0.850 μs	4.200 μs

Table 3: *Run time system performance, system A (Pentium 4, 1600 MHz). The median times reported by the MPD script* `vsuite/timings/report.sh` *are shown.*

4 Performance evaluation

MPD comes with two performance evaluation packages. The first, for the context switching primitives, is in the `csw/` subdirectory of the source distribution; after `make csloop` you can start `./csloop N` where N is the number of seconds the test will run approximately.

Tests of the language primitives used for multithreading are in the `vsuite/timings/` subdirectory of the source tree enhanced with the verification suite. They are run by three shell scripts used to compile them, executed them, and summarize the results in a table.

5 Establishing a baseline

There are two extremes possible when implementing the context switch primitives needed for MPD: implementing each CPU manually in assembler code (what the MPD implementation does normally) and using the SVR4-style functions `getcontext()`, `setcontext()` and `swapcontext()` which operate on `struct ucontext`; these are provided as experimental code in the file `csw/svr4.c` of the MPD distribution.

The first tests were done by using the provided i386 assembler context switch routines. After verifying correctness and noting the times (see tables 2 and 3), the same was done using the SVR4 module instead of the assembler module.

These tests were done on a Pentium 4 machine running at 1600 MHz with 2 megabytes of secondary cache, and 1 GB of main memory, running NetBSD-3.0_BETA as of end of October 2005.

The SVR4 tests were redone on a DNARD system (for its ARM cpu, no assembler stubs are provided in either the SR or MPD distributions).

Table 3 shows a factor-of-about-ten performance hit for the operations that require context switches; note, however, that the absolute values for all such operations are still smaller than 5 μs on 1600 MHz machine and will likely not be noticeable if a parallelized program is run on a LAN-coupled cluster: on the switched LAN connected to the test machine, the time for an ICMP echo request to return is about 200 μs.

Test description	ARM ASM	...context_u	SVR4 s.c.
loop control overhead	n/a	0.057 μs	0.056 μs
local call, optimised	n/a	0.376 μs	0.355 μs
interresource call, no new process	n/a	4.300 μs	4.080 μs
interresource call, new process	n/a	27.250 μs	55.900 μs
process create/destroy	n/a	25.240 μs	58.780 μs
semaphore P only	n/a	0.304 μs	0.301 μs
semaphore V only	n/a	0.254 μs	0.249 μs
semaphore pair	n/a	0.506 μs	0.487 μs
semaphore requiring context switch	n/a	1.570 μs	11.180 μs
asynchronous send/receive	n/a	5.550 μs	5.190 μs
message passing requiring context switch	n/a	6.740 μs	30.140 μs
rendezvous	n/a	9.600 μs	54.000 μs

Table 4: *Run time system performance, system B (StrongARM SA-110, 233 MHz). The median times reported by the MPD script* `vsuite/timings/report.sh` *are shown.*

6 Improvements using NetBSD library calls

While using the system calls `getcontext` and `setcontext`, as the `svr4` module does, should not unduly penalize an application distributed across a LAN, it might be noticeable with local applications.

However, we should be able to do better than the `svr4` module without writing our own assembler modules, since NetBSD 2.0 (and later) contains its own set of them for the benefit of its native Posix threads library (`libpthread`), which does lots of context switches within a kernel provided light weight process[8]. The primitives provided to `libpthread` by its machine dependent part are the three functions `_getcontext_u`, `_setcontext_u` and `_swapcontext_u` with similar signatures as the SVR4-style system calls `getcontext`, `setcontext` and `swapcontext`.

There were a few difficulties that arose while pursuing this.

First, on one architecture (i386) `_setcontext_u` and `_getcontext_u` are implemented by calling through a function pointer which is initialized depending on the FPU / CPU extension mode available on the particular CPU used (8087-mode vs. XMM). On this architecture, `_setcontext_u` and `_getcontext_u` are defined as macros in a private header file not installed. The developer in charge of the code has indicated that he might implement public wrappers; until then, we have to check all available NetBSD architectures and copy the relevant code to our module `csw/netbsd.c`.

Second, we need to extract the relevant object modules from the threading library for static linking (`libpthread.a`) without resolving any other symbols, because normal libpthread is overloading some system calls thus causing failure of applications not properly initializing it.

Again, this set of context switch code has been verified by running `cstest` and the full verification suite.

The low-level as well as the high-level timings with the new context switch package have again been collected in tables 2, 3 and 1.

To ease installation, a package for the NetBSD package system has been built for SR and MPD, available in the `lang/sr` and `lang/mpd` subdirectories of the pkgsrc root.

As the NetBSD package system is available for more operating systems than NetBSD[4], a lot more work would be needed to make the packages universal; thus they are restricted to be built on NetBSD 2.0 and later.

7 Discussion

Our new approach has raw context switch times that are only 10% of the SVR4 system call ones. Compared to the assembler routines, they are only slower by a factor of 10 (see table 2).

Table 3 shows three classes of high level operations.

1. Non-context switching operations have the same speed independent of the context switch primitives used, as expected.

2. The two operations measured requiring a process creation (in the MPD language sense) are about as fast as in the SVR4-system-call case. This was expected, as the process creation primitive does a system call internally.

3. Context switching operations which do not create a new process (in the MPD language sense) are slower than in the assembler case, but faster than in the SVR4-style case, by an amount roughly equivalent to one (semaphore operation, message passing) or two (rendezvous) context switching primitive times.

The same classification can be done for the 233 MHz ARM CPU (table 1). However, SVR4 process creation, destruction and the rendezvous need about one third of the LAN two-way network latency, thus cannot be neglected anymore. We conclude that for machines in the 300 MHz range and below, using assembler implementation (where available) or at least our new implementation of the context switching primitives is a necessity. This is also expected for even slower machines.

MPD can be compiled in a mode where it will make use of multiple threads provided by the underlying OS, so that it can use more than one CPU of a single machine. This has not been implemented yet for NetBSD, but should be.

8 Summary

A method for porting SR and MPD to NetBSD has been shown, for which only preliminary results, and only for SR, were presented earlier.

The SR porting effort was easily adopted for the MPD case. In fact, the run time system (library and srx/mprx) could probably be factored out into a common run-time system package.

The new port was verified using the SR and MPD verification suites.

As discussed above, the SVR4-system-call approach, while feasible, creates an overhead that is clearly visible for non-networked operation of a distributed program; on our Pentium machine, high level context switching operations are slower by a factor between 7 and 11 (the raw context switch primitives are slower by a factor of 110). Even for networked operation, for a 233 MHz StrongArm CPU or slower machines, context switch latency exceeds one third of the network latency.

The approach using the libpthread primitives is much faster for all but the process creation/destruction case and should thus be adequate for about any application in the networked case, and for any in the single-machine case that does not do excessive amounts of implicit or explicit process creation.

For highly communication-bound problems on a single machine, using the assembler primitives might show a visible speedup, where available.

References

[1] Gregory R. Andrews. *Foundations of Multithreaded. Parallel. and Distributed Programming.* Addison-Wesley. 2000 (ISBN 0-201-35752-6)

[2] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings. 1993

[3] Gregory R. Andrews, Ronald A. Olsson. Michael H. Coffin. Irving Elshoff. Kelvin D. Nilsen. Titus Purdin and Gregg M. Townsend. *An Overview of the SR Language and Implementation*. 1988. ACM TOPLAS Vol. 10.1. pp. 51 - 86

[4] Alistair G. Crooks. *A portable package system.* in: Proceedings of the 3rd European BSD Conference. Karlsruhe (Germany) 2004

[5] Martin Husemann. *Fighting the Lemmings*. in: Proceedings of the 3rd European BSD Conference. Karlsruhe (Germany) 2004. http://www.feyrer.de/PGC/Fighting_the_Lemmings.pdf

[6] Ignatios Souvatzis. *A machine-independent port of the SR language run time system to NetBSD.* in: Proceedings of the 3rd European BSD Conference. Karlsruhe (Germany) 2004. http://www.arxiv.org/abs/cs.DC/0411028

[7] Gregg Townsend. Dave Bakken, *Porting the SR Programming Language*. 1994. Department of Computer Science, The University of Arizona

[8] Nathan J. Williams. *An Implementation of Scheduler Activations on the NetBSD Operating System.* in: Proceedings of the FREENIX Track. 2002 Usenix Annual Technical Conference. Monterey. CA. USA. http://www.usenix.org/events/usenix02/tech/freenix/williams.html

New Evolutions in the X Window System

Matthieu Herrb* and Matthias Hopf†

October 2005

Abstract

This paper presents an overview of recent and on-going evolutions in the X window system. First, the state of some features will be presented that are already available for several months in the X server, but not yet widely used in applications. Then some ongoing and future evolutions will be shown: on the short term, the new EXA acceleration framework and the new modularized build system. The last part will focus on a longer term project: the new Xgl server architecture, based on the OpenGL technology for both 2D and 3D acceleration.

Introduction

The X window system celebrated its twentieth birthday last year. After some quick evolution in its early years, its development slowed down during the nineties, because the system had acquired a level of maturity that made it fit most of the needs of the users of graphical work stations. But after all this time, pushed by the competition with other systems (Mac OS X and Microsoft Windows) the need for more advanced graphics features triggered new developments.

The first part of this paper is going to describe some of these features that are already available (and have been used) for a couple of years but not necessarily known by users of the X window system. A second part will address some on-going work that will be part of the X11R7 release: a new 2D acceleration architecture and the modularization of the source tree.

In the third part, a complete redesign of the device dependent layer of the X server, based on OpenGL, will be presented. It will allow a better integration of accelerated 2D and 3D graphics and make it possible to take advantage of the powerful 3D acceleration engines available today even in low end graphics adapters.

*CNRS-LAAS
†SUSE Labs

1 Already available new features

This section aims to remind a couple of the already available features, used by some toolkits to get better user experience with X: client-side font rendering, including anti-aliasing using Xft2 and fontconfig as well as rendering improvements based on the Render and Damage extensions.

It also presents the Composite extension and explains how the composite manager can be used to achieve various effects (transparency, shadows,...), taking advantage of the Render code already present in the existing X server.

1.1 The Render extension

The original X protocol provides a display model based on traditional boolean operations between source and destination. The Render extension was designed to enhance this model by adding image compositing operations. Image compositing was formalized by T. Porter and T. Duff [6], and implemented in the Plan 9 window system by R. Pike and R. Cox. Keith Packard designed and implemented the Render extension in XFree86 [3]. Porter-Duff compositing adds a pixel opacity value called "alpha" to its color attributes. This opacity value can be used to represent two different effects: translucency and anti-aliasing. The effect of translucency is created when all pixels of an object have their color computed as a combination of the intrinsic color of the object and the existing background values. Anti-aliasing is achieved by taking into account partial occlusion of the background by the boundaries of an object.

The Render extension implements new primitives for the display of images and polygons, as well as the basis for a new font rendering system that takes advantage of the image compositing features to render anti-aliased text.

Some of the core X applications have been extended to be able to use the X render extension: for instance `xclock` in analog mode now draws anti-aliased and translucent clock hands.

The Render extension was developed initially in XFree86 (now in X.org). It has been adopted by many commercial X providers too and thus can be assumed as a standard for modern applications.

1.2 Client-side font rendering

When X was originally designed, more than twenty years ago, it was decided that text display would be done by the X server. In the traditional X world, fonts are a server-side resource and applications depend on the fonts available in the server.

This approach had the advantage of limiting the amount of data that text-based applications have to send to the server, but it also caused lots of frustration among application developers. PDF or Postscript viewers for instance need to be able to render fonts that are embedded in the document they are displaying.

Moreover applications need access to more than just bitmaps in the fonts specifications for precise rendering. Attempts to extend the server-based font rendering model have all failed to solve all problems.

So, together with the introduction of Render, a radical decision has been taken to move font rendering from the server to client applications. To achieve this, a new text-rendering library has been designed; it is now at its second revision: Xft2. A companion library, fontconfig provides support to all font naming, installation and caching issues.

Fontconfig can, by the way, be used on a broader spectrum of applications than just X. It could be used by TEX like publishing applications, printer drivers and so on. Fontconfig uses XML formatted configuration files, located in the `/etc/fonts` directory.

Xft2 is based on the Freetype library. It can render several font formats: the traditional bitmap-based PCF format from the legacy server-side font system, Postscript Type 1 and True Type fonts.

Xft2 also provides some enhancements in the font encoding management, among others it is possible to use UTF-8 encoded text directly with Xft2.

Measurements have shown that the new client-side font rendering scheme has little to no impact on the overall performance. In many cases, it reduces the number of round-trips between the application and the server and thus even greatly improves application startup times.

Like Render, the Xft and fontconfig libraries have been embraced by more than XFree86 and X.Org. They are now the standard way of displaying text for X toolkits and applications. The legacy server-side mechanism is obsolete and should not be used by new developments anymore.

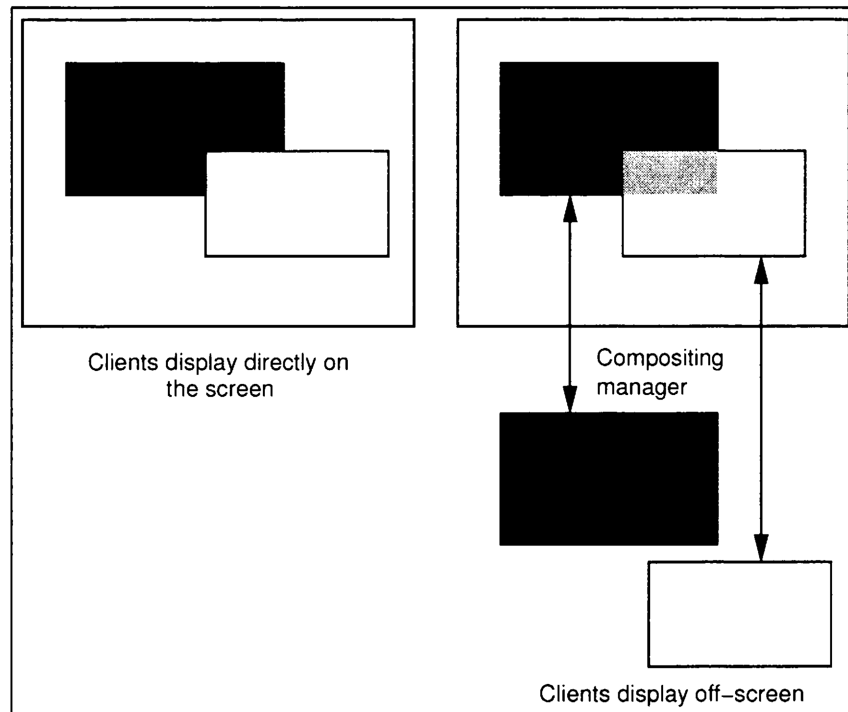
1.3 Composite, Damage, Xfixes extensions and the composite manager

To take the full advantage of the image compositing model provided by the X render extension, for example to provide translucent windows or to have a window manager add drop shadows to the windows, there are some bits missing. In the traditional X model, each application draws its window independently and doesn't take care of underlying or overlaying windows.

To be able to implement those eye candies, applications should be redirected to use off-screen windows, and a specific application, the *composite manager*, will work with the window manager and the new Composite extension to compute the screen's contents, doing compositing operations to produce translucency, shadows and anti-aliased polygons and texts.

To make this work, this application needs a bit more information than before about what is happening on the screen, and it needs this information in an efficient manner. This is the goal of the the Damage extension: it provides an efficient way to notify an application of damage done to a region of the screen by another application.

Figure 1: Composite manager principle: *traditional direct on-screen drawing on the left, vs off-screen drawing & compositing on the right.*



The Xfixes extension provides a general framework to extend the X protocol in order to work around some limitations in the core protocol. It currently contains five fixes. The more important ones allow better manipulation of the application cursor and export the region objects from the server to the clients.

`xcompmgr` is a sample composite manager that can be used with any existing window manager to provide some eye candy. Figure 2 shows the default OpenBSD desktop enhanced with xcompmgr and anti-aliased fonts in xterm and firefox.

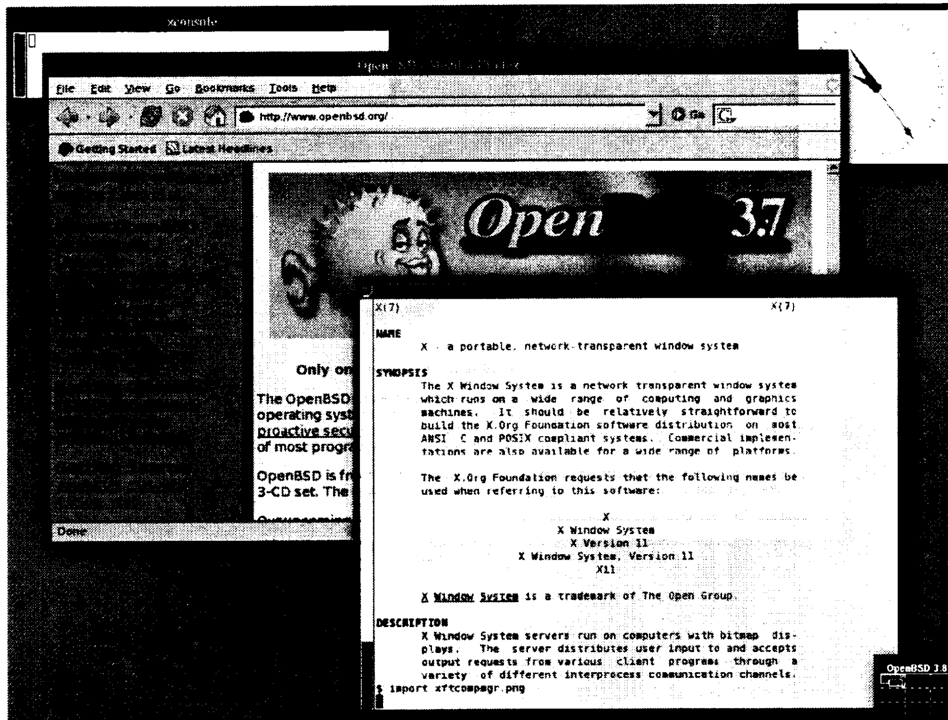
KDE provides its own composite manager, `kompmgr`, while some window managers (Luminocity for instance) are integrating this functionality.

1.4 Cairo

Another important evolution in graphical user interfaces is the growth of vector-based graphics, as opposed to existing bitmap-based graphics. Vectors offer a better representation of screen contents, independent of the actual resolution, allow producing a perfect-looking printed version of an on-screen document, use less space for storage, and provide a better base for anti-aliased graphics.

With the introduction of the Render extension, X now has the ability of producing high-quality graphics based on vector representation.

Figure 2: Adding eye-candy to the default OpenBSD desktop



Cairo[1] is a library that implements vector based graphics with support for multiple output devices. Existing back-ends include X with the Render extension, and image buffers [4]. Experimental drivers include OpenGL (through the glitz library) and PDF files.

The Gtk+ toolkit as well as some existing applications already started to base most of their graphics on the Cairo library.

The OpenGL back-end offers some interesting features: on systems with accelerated OpenGL it provides the toolkits and application with a way to do accelerated 2D graphics that almost completely bypass the X libraries and server. However, this doesn't provide a solution for the global desktop compositing acceleration mentioned above.

2 Ongoing work

The current Xserver is divided in an architecture independent (DIX) and an architecture dependent (DDX) layer, which in turn loads the relevant hardware driver for rendering into the framebuffer. In order to accelerate drawing operations, the hardware drivers offer functions that implement certain operations using the graphics

[1] `http://www.cairographics.org`

hardware. The traditional interface for this is the Xserver Acceleration Architecture (XAA), which mainly focuses on accelerating core X protocol requests.

In contrast to the features described in the previous section most ongoing and future developments focus on the Xserver framework. These changes will not affect the programmer's API, as e.g. the Render and Composite extensions did, so all applications can immediately benefit from their implementation.

This section describes on-going work, which is available in the X11R6.9 / X11R7 release: the new 2D acceleration architecture EXA and the modularization effort. The EXA architecture is aimed at replacing XAA in drivers, focusing on accelerating primitives used by modern applications based on the render extension. The modularization effort will be described from an architectural point of view.

2.1 A new acceleration architecture for Render: EXA

Without composite manager, the performance of the Render extension is decent on reasonably recent hardware, that doesn't even deserve the "fast" qualifier. However, most of of Render computations are done on the main CPU and take little advantage of GPU acceleration. Render takes advantage of MMX or SSE instructions when they're available, and there have been some work done to add basic hardware acceleration for Render in the radeon driver.

When the composite manager is involved, things get worse, performance-wise. Even today's "fast" hardware can feel slow with compositing enabled. It is thus mandatory to rework the acceleration framework so that Render and Composite can be accelerated much better.

The currently used acceleration architecture in Xorg (XAA) is unsuitable for modern desktop usage. As a result of heavily using the card's 2D engine to accelerate mostly rarely used operations (like pattern fills and Bresenham lines) it invalidates any backing store that the X server might have on a region. Furthermore accelerating the Render extension using XAA is rather complicated and severely limited by its memory manager.

EXA (for EXcellent Architecture or Ex-kaa aXeleration Architecture or whatever) aims to extend the life of the venerable XFree86 video drivers by introducing hooks that they can implement to more efficiently accelerate the X Render extension: solid fills, blits within screen memory and to and from system memory, and Porter-Duff compositing and transform operations. It has been implemented by Zack Ruskin in X.org.

A couple of existing drivers have already been converted to use the new EXA acceleration framework if requested: the i810 driver for Intel graphics card adapters, the radeon driver for ATI Radeon cards, the sis driver and the i128 drivers.

2.2 Source tree modularization

One of the big tasks in the lastest X release has been a complete rework of the X build system. The existing source tree, built using the imake build system, was considered as a big monolithic thing in which most developers found themselves uncomfortable. The need for global releases, updating all drivers at once every six month or so doesn't really fit the market of graphics cards that can produce new models more often than that timeframe.

Based on the experiences of other software projects, it was decided to switch to a more modular organization of the project, with more or less independent components [8]. This new organization will allow drivers maintainers (or others) to make independent releases, whenever they are needed.

It was decided that the best tools to manage the build of this new modularized source tree are the GNU auto-tools. They have an existing large user and developer base, and thus feel easier to use by the majority of developers. Being maintained outside of the X.Org project is supposed to lower the maintenance burden on the X developers which are now free to concentrate on their code.

The existing source tree has been split in several components, and each of them is composed of independent packages. The main components are:

- *xproto* which holds all the header files describing the actual X protocol and extensions. There is one package for the core X protocol and one package per X extension (Shape, MIT-SHM, Render, etc.),
- *libs* which holds all the libraries, one package per library (X11, Xext, Xrender, etc.),
- *data* which holds several data files (bitmaps and icons, XKB data files, X cursors),
- *apps* which holds the sample applications provided by X.Org (twm, xcalc, xedit, xlogo, xman, xwd, etc),
- *xserver* which provides the different X servers (Xorg, Xnest, Xprint, Xvfb),
- *drivers* which provides the graphics cards drivers, each one in an independent package,
- *fonts* which provides several fonts packages,
- *doc* for the existing documentation that doesn't fit a a specific package,
- *utils* various utilities that help the modular infrastructure, including an autotoolized version of `imake`, for use with third party applications that still depend on it.

Dependencies and configuration of the new packages is heavily based on the `pkgconfig`[2] tool.

To make the transition smoother, X11R6.9 and X11R7 share the same source base, and as far as possible produce the same set of binaries. X11R6.9 is the last version of the monolithic tree, while X11R7 is the first version based on the new modular tree. Both releases should be equivalent feature-wise.

Future work will be done in the modularized tree only. Only patches and bug fixes will be done in the X11R6.9 branch.

3 The future: Xgl

This last section will present the ideas, the rationale and the work already done to move to a new X server rendering model, based on OpenGL and glitz. This Xserver, called Xgl[3], is mainly developed by David Reveman. Currently it has to be run on top of a regular Xserver, comparable to Xnest, but first steps have been made to use Embedded OpenGL (EGL) extensions to make it run stand-alone [7].

3.1 Why use OpenGL

When looking at the current state of the Xserver architecture, several shortcomings are getting obvious, which we will analyze in detail now:

- XAA does not match current rendering use and is difficult to extend,
- the X server is a mix of high level code (window management etc.) and low level code (drivers),
- there are little to no ideas how to support modern graphics hardware features like pixel shaders,
- the driver API is used by Xserver only,
- the driver API is basically 2D only,
- drivers are difficult to maintain outside of main development tree,
- future graphics hardware won't have a 2D acceleration core any more.

The current acceleration architecture, XAA, has pretty much reached the end of its productive life, as it is difficult to implement and maintain, and modern applications don't use many core X requests for rendering any more. Many new features like the Render extension have to be implemented and tested for each driver, which is a tedious and troublesome work.

[2] `http://pkgconfig.freedesktop.org/wiki/`

[3] `http://http://www.freedesktop.org/Software/Xgl`

EXA is one alternative that has already been discussed in the previous section, but it still has the disadvantage that it keeps the driver code inside the Xserver, while it would be a worthy goal to have a real driver abstraction layer.

Both acceleration architectures are 2D only APIs, that are used by the Xserver alone and not by other programs. APIs that are used by only a small number of programs tend to be less stable and flexible than APIs used by many programs. While using 2D for a windowing system makes basically sense, there are several ideas how 3D user interfaces could enhance productivity in the long-term future, for instance with the project Looking Glass[4].

On the X.Org developer's conference 2005 all attendants agreed that using the industry standard 3D graphics interface OpenGL is a worthy investigation for a driver abstraction layer. David Reveman showed an early version of his Xgl prototype, which since then has matured and supports OpenGL based implementations of most important drawing operations in the Xserver. Additional features have been contributed by the community, for instance Xegl (Dave Airlie, Adam Jackson, John Smirl) and XVideo (Matthias Hopf).

Basically, Xgl has shown even in its early state, that using OpenGL for the drawing operations needed for an Xserver is a viable option, which additionally allows for more advanced compositing operations as it will be shown in Subsect. 3.3. It also gives easy access to modern features of graphics hardware like vertex and pixel shaders, and as the API continues to evolve we will see future capabilites exposed as well. Furthermore, hardware vendors use a lot more transistors and invest more in the design for the 3D core, so it is very likely to be faster than the 2D acceleration core.

The most important advantage is, however, that finally the Xserver has got rid of its hardware drivers, which can now be maintained outside the Xserver tree. E.g. Render is accelerated on every graphics hardware with OpenGL drivers, not just on the ones that actually implement the required acceleration interface. Having drivers removed from the Xserver core is especially important with future graphics hardware, which won't have 2D acceleration any more, and for which only closed-source OpenGL drivers exist. While vendors can (and do) implement their 2D drivers themselves as well, having a stable interface abstraction using a standard API will certainly improve the driver quality.

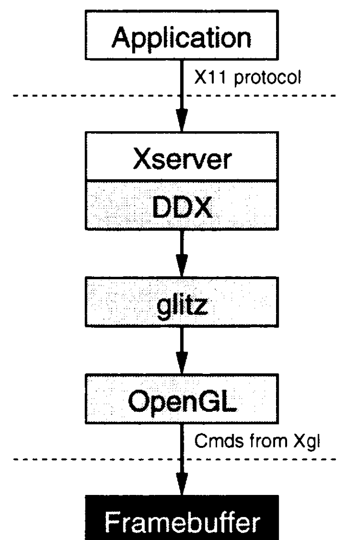
3.2 The architecture of Xgl

Figure 3 shows an overview over the Xgl architecture. At the moment Xgl is one additional DDX in the kdrive Xserver, which is an experimental Xserver mostly written by Keith Packard. After the Xorg modularization is finished, Xgl will slowly be integrated into the main stream Xorg server as an additional DDX as well.

OpenGL is still a relatively low-level API, so it made sense to create an abstrac-

[4] `https://lg3d.dev.java.net/`

Figure 3: Xgl architecture overview



tion layer that covers the most common graphics operations. As many X operations are pixmap oriented, texture handling is of particular importance.

Before working on Xgl David Reveman implemented an OpenGL based backend for Cairo [2]. The semantic of this backend, named glitz, closely resembles the Render protocol, and thus was the perfect abstraction layer for Xgl. Basically, glitz is an OpenGL image compositing library, which provides Porter-Duff compositing of images with implicit mask generation for geometric primitives. This includes, but is not limited to, alpha blending and affine transformations, and it has support for additional features like convolution filters and color gradients, which are not needed for Cairo. It also abstracts general texture use and the different sorts of OpenGL buffers.

There are no software fallbacks in glitz, if the hardware isn't capable of implementing a certain operation, glitz will just report the failure.

Certain operations of glitz require modern OpenGL features, for instance convolution filters or color space conversion and resampling for YUV textures both need pixel shaders. If the hardware isn't capable of these operations, a general software fallback inside glitz would result in poor performance, while the upper layer can easily implement this particular feature (e.g. color conversion) in software in an optimized way.

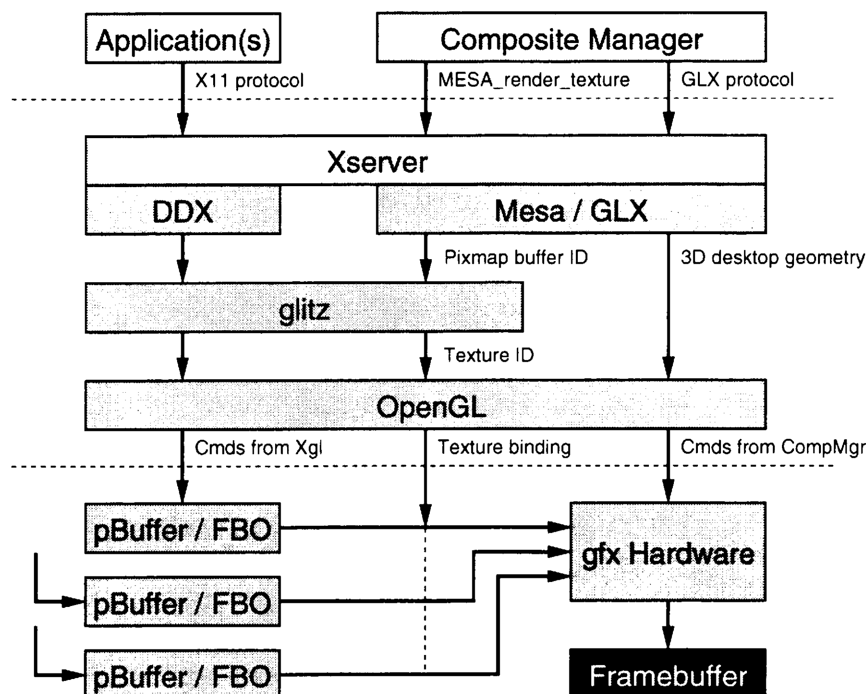
Applications that want to use OpenGL for drawing have to share the drawing space with the Xserver. As currently there is no way to share textures or framebuffers between applications, they currently have to use indirect rendering, i.e. the Xserver is doing the actual OpenGL calls it gets via the GLX protocol from the application. On one hand, this can be significantly slower for applications doing a

lot of memory transfer (video textures or geometry with high primitive count), on the other hand Xgl is now one of the few X servers capable of doing hardware accelerated indirect rendering, for example for running OpenGL programs remotely, which isn't implemented in Xorg yet.

3.3 Composite managers using OpenGL

As already described in Subsect. 1.3, all windows are rendered to off-screen pixmaps when the Composite extension is active. In the OpenGL case, this means the Xserver must render to an off-screen framebuffer, which can be provided by either the pBuffer or the more modern Frame Buffer Object (FBO) extension. Unfortunately, pBuffers are not yet widely supported, and implementation of FBOs is even less common and unstable. In these cases Xgl has to do all rendering to client windows in software and download the window contents to textures afterwards, which surprisingly is still quite usable.

Figure 4: Xgl in combination with a composite manager

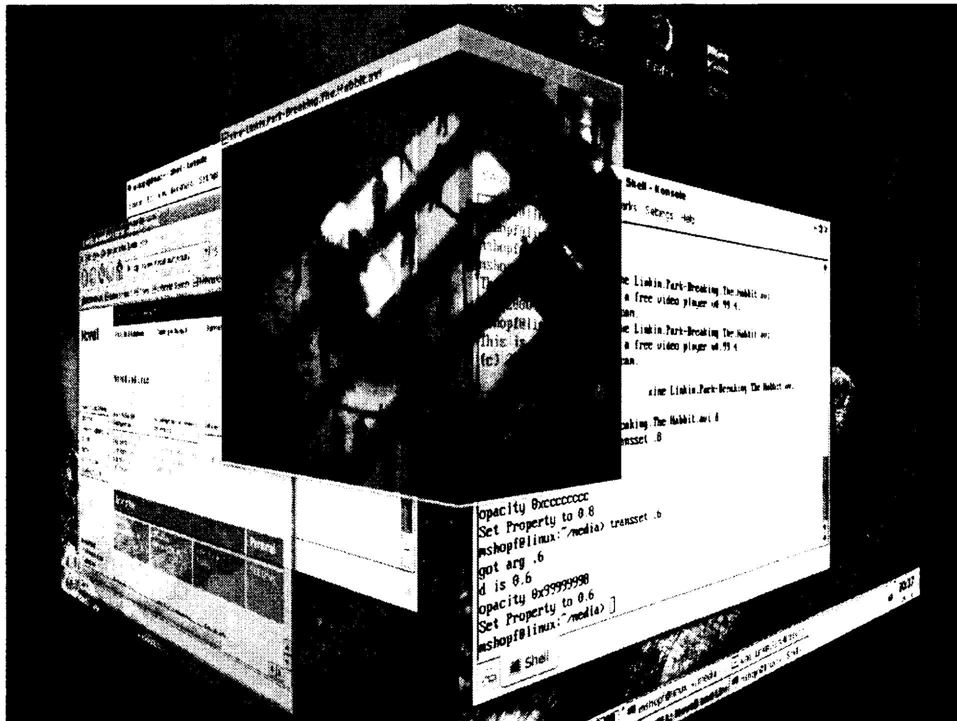


The pixmaps with the window contents can afterwards be composed using the Composite extension. An alternative to this is to use GLX to do the compositing with indirect OpenGL rendering. For this the composite manager has to be able to bind off-screen pixmaps to textures, which is done with the `GLX_MESA_render_texture` extension from Xgl. Figure 4 provides the com-

plete architectural overview over a session using an OpenGL based composite manager.

With this type of composite manager windows can be arbitrarily placed in 3D, which leads to pretty exciting rendering possibilities (see Fig. 5). Note that the pixmaps stay on the graphics hardware all the time, and only the geometry to be rendered has to be transfered from the composite manager to Xgl.

Figure 5: Fancy desktop switching with GLX based composite manager



3.4 Caveats and pitfalls

Currently Xgl is working best when run on top of a regular Xserver, comparable to Xnest. OpenGL provides neither facilities for creating a displayable framebuffer, nor for changing display modes. Both issues are addressed by the experimental `EGL_MESA_screen_surface` extension, which uses the buffer management ideas incorporated in Embedded OpenGL (EGL). The extension is close to being submitted to the Embedded OpenGL ARB for review. Right now there exists an early implementation in Mesa, named Xegl [5], for the R100 and R200 based Radeon chips from ATI, but several hardware vendors want to provide all extensions needed for Xgl to run in the future.

[5] `http://www.freedesktop.org/wiki/Xegl`

However, for full functionality, extensions for creating a hardware mouse pointer, getting monitor information, and setting drivers for different output plugs are needed as well. These extensions are not specified yet.

One major drawback of Xgl right now is that applications cannot do direct OpenGL rendering at all. For this an extension for sharing textures between address spaces is needed, as the application, Xgl, and the composite manager are all running in different address spaces. This is the subject of current discussions, but nothing is specified yet.

During the implementation phase of Xgl, several pitfalls have been encountered, but for most of them a reasonable solution has been found. First, with many OpenGL drivers one can easily get namespace collisions, as Xgl needs to be linked against a software rendering Mesa library for fallback and GLX handling as well as against the current OpenGL library on the host system. This can be solved by loading the OpenGL backend dynamically, which also allows for the Xegl backend to be loaded upon availability automatically. Then, frame buffer objects have turned out to be pretty unstable for many operations, so the code path using pBuffers will stay around longer than anticipated.

One source for major headaches in the open source community is of course the lack of open source drivers for modern graphics hardware, which are often only covered by binary only OpenGL drivers. One notable exception here is Intel, which has committed itself to providing open source drivers for future chips as well. Currently, their drivers are not yet equivalent to their competitors with respect to implemented features, but they are advancing steadily.

3.5 Implementation on BSD systems

Xgl currently runs on any system providing OpenGL, but it is unusable without hardware acceleration (i.e. without DRI support).

For the longer term, Xegl needs a console driver that provides a graphical mode with EGL drivers. The first implementations will be done on the Linux framebuffer driver. NetBSD and OpenBSD share the *wscons* console driver, on which some level of support for graphical console is already available. FreeBSD has his own console driver, *syscons*, that doesn't provides a graphical mode yet, as far as we know.

The integration of these graphical modes with hardware OpenGL acceleration (and DRI) is required to provide an EGL capable console with support for the necessary hardware setup extensions.

BSD developers will have to work with Linux DRI developers to make sure that the direct rendering infrastructure is kept in sync with the Linux DRI with respect to features like the proposed EGL extensions. A good way to help here would be to discuss the new extensions on the `dri` and `dri-egl` mailing lists so that no requirements are missed.

In the short term, with these extensions not completely specified, some more low-level hardware access might be necessary inside the Xserver, and BSD and

X.org should work closely together as soon as more development efforts are concentrated upon Xegl.

Conclusion

After a couple of years of relative stagnation in the world of the X window system, development has resumed, with the goal of providing rich enough functionalities for desktop environments which want to provide eye-candy on par with other Desktop OSs. The wide availability of cheap OpenGL-capable graphics cards makes such a new project realistic, although the lack of support for open source systems by most of the hardware vendors darkens the bright sky of this new technology.

References

[1] S. Nickell. Design fu: Xshots. `http://www.gnome.org/~seth/blog/xshots`, March 2005.

[2] P. Nilsson and D. Reveman. Glitz: Hardware Accelerated Image Compositing Using Opengl. In *Usenix 2004 Annual Technical Conference, Freenix Track*, pages 29–40, June 2004.

[3] K. Packard. Design and Implementation of the X Rendering Extension. In *Usenix Technical Conference, Boston*, June 2001.

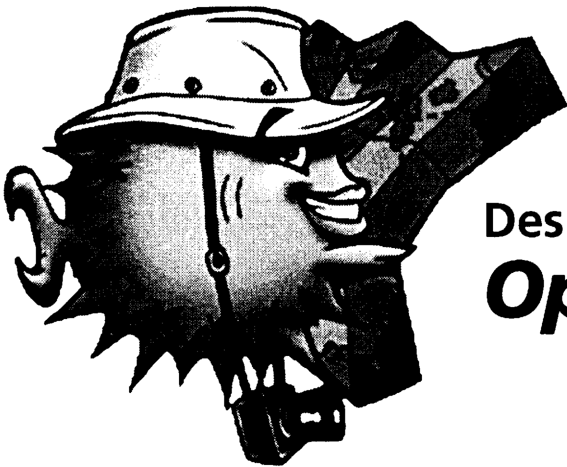
[4] K. Packard. Cairo status. `http://keithp.com/~keithp/talks/cairo-exdc2005/`, June 2005. European X.Org developpers Meeting, Karlsruhe.

[5] K. Packard. X Status Report. `http://keithp.com/~keithp/talks/x-rearch-lca2005`, April 2005. Linux.conf.au.

[6] T. Porter and T. Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[7] J. Smirl. The state of linux graphics. `http://www.freedesktop.org/~jonsmirl/graphics.html`, September 2005.

[8] D. Stone. X.org modularization. "Where to from here?". `http://people.freedesktop.org/~daniels/exdctalk/`, June 2005. European X.Org developpers Meeting, Karlsruhe.



Design and Implementation of *Open*OSPFD

by Claudio Jeker <claudio@openbsd.org>
Internet Business Solutions AG

Abstract

OpenOSPFD is a free and secure implementation of the Open Shortest Path First protocol. It allows ordinary machines to be used as routers exchanging and calculating routes within an OSPF cloud.

OpenOSPFD is the next major step after OpenBGPD for full router capabilities in OpenBSD and other BSDs. Together with OpenBGPD it is possible to re-route traffic in case of link loss resulting in a higher-level of availability.

Overview

1.1 Routing Protocols

The Internet is split into regions called Autonomous Systems (AS). Each AS is under the control of a single administrative entity – for example a university or an ISP. The edge routers of these AS use an Exterior Gateway Protocol (EGP) to exchange routing information between AS. Currently BGP4, the Border Gateway Protocol is the only EGP in widespread use. Routers within an AS use an Interior Gateway Protocol to exchange routing information. There are different IGPs. OSPF, IS-IS, and RIP are the most commonly used. It is possible and common to have multiple IGPs running inside one AS.
The Routing Information Protocol (RIP) is a legacy protocol that is often found on appliances. It is not suitable for larger networks because the distance vector algorithm used by RIP converges slowly. Especially in the face of certain network failures (count to infinity). OSPF and IS-IS on the other hand are both link-state protocols. The Intermediate System to Intermediate System (IS-IS) protocol was developed for the OSI protocol suite under the lead of the ITU.

Why not use one protocol for everything, EGP and IGP? The requirements for an IGP differ from those of a an EGP. For an IGP it is important to recalculate the routing table quickly when the network changes. Another factor is automatic neighbor discovery. On the other hand the most important feature of an EGP is the ability to express routing policies. The resulting routing table is normally cost optimised.

1.2 Algorithms

There are two main concepts to exchange routing information. These algorithms are working in a totally different ways.

1.2.1 Distance Vector Algorithms

Distance vector algorithms got their name from the form of the routing updates: a vector of metrics.
In a distance vector algorithm every router exchanges its routing table with all his neighbors. The neighbors then walk through the list and compare if their current route entry is better or not. If not the route is replaced and redistributed again.
In case of RIP the list of routes and their metric is exchanged every 30 seconds. This results in a slow convergence because an update propagates only one hop every 30 seconds. On the other hand the protocol is simple and robust because every router cares only about his own neighbors. In other words the information about the network topology is distributed. This results in one of the biggest weaknesses of RIP – the count to infinity problem – resulting in slow convergence and routing loops if a network becomes unavailable. There are some countermeasures against this. The simplest is to pass the full routing path instead of only the metric. This path distance vector algorithm is used by BGP. It is easy to implement routing policies on distance vector algorithms.

1.2.2 Link-State Algorithms

In a link-state protocol every router or node sends out his current link-states. The link-state advertisements are distributed to all nodes in the network. The resulting replicated distributed database represents the entire network topology. Every node uses this connectivity map to calculate the shortest path to every other router. Link-state protocols have good convergence properties. The biggest weakness of link-state protocols is the replicated distributed database. If the database gets out of sync non optimal routes are used and in worst case routing loops are created. Link-state protocols are more complicated than distance vector protocols.

OSPF – the protocol

The OSPF routing protocol was developed within the IETF. The work started in 1987. The current version (OSPFv2) of the specification was published in 1998 as RFC 2328.

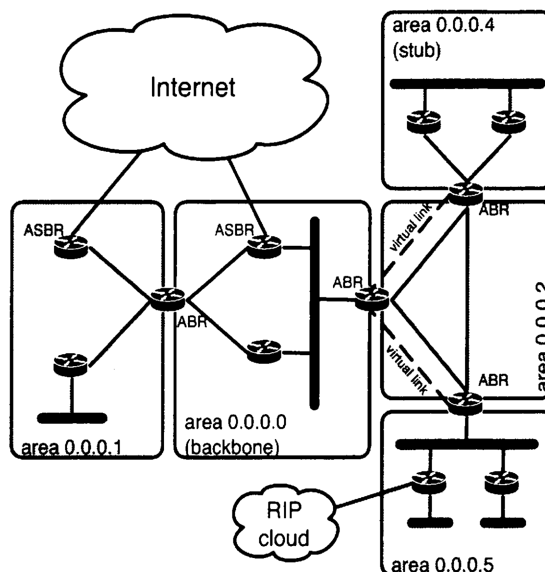


Figure 1: Sample OSPF network

2.1 Architecture

The Open Shortest Path First (OSPF) protocol is a link-state, hierarchical routing protocol. It is probably the most used IGP in the world. It is capable of doing neighbor discovery on different types of networks with minimal need for configuration. OSPF encapsulates its routing messages directly on top of IP as its own protocol type (89). TCP connections are not used because the link-state flooding algorithm already includes its own way for reliable communications – adding to OSPF's complexity. Most obvious the massive use of IP multicast in OSPF makes TCP infeasible.

2.1.1 Networks

An OSPF router discovers neighbors by periodically sending OSPF Hello packets out on all configured interfaces. Depending of the interface type different methods are used. The flooding algorithm depends on the interface type as well.
The simplest interface type is a point-to-point interface. Neighbor discovery is easy – there is only one neighbor on the other side of the link – and no special link-state flooding enhancement is required.
For ethernet and other broadcast networks OSPF uses multicast to find all neighbors on the segment. The link-state updates are flooded via multicast as well. To make the thing even more complicated a designated router (DR) was introduced. The DR has the duty to enforce the reliable flooding for all other routers connected to the same LAN. A backup designated router (BDR) was introduced to take over in case of a DR failure.

Additionally more flooding procedures where defined for other important network types like NBMA (non-broadcast multiple-access) or point-to-multipoint networks. Examples include X.25, Frame Relay, or ATM using full mesh or switched virtual circuits. OpenOSPFD does not support these exotic networks mostly because of lack of support by the OS and missing infrastructure.

2.1.2 Database synchronisation and reliable flooding

Database synchronisation in a link-state protocol is crucial. The routing calculation ensures a loop-free routing as long as the database remains perfectly synchronised. It is no wonder that this is the most fragile part of the specification. Especially with all the additional complexity added by multicasting of updates and the presence of DR and BDR routers. A reliable and robust flooding procedure is very important because a little inadvertence can result in a major network "melt down" where only a full reset of all routers cures the situation.

Database synchronisation takes two forms. First there is the initial database synchronisation. Following it the distributed copies of the database need to be kept in sync by reliably flooding updates to all routers in the network.
The initial database exchange is done when two routers build an adjacency. First a request list is built up through a TFTP like database exchange phase. In the exchange phase one of the two neighbors is elected as master of that session. This router sends a Database Description packet to the slave and waits for an answer. If none is received within some amount of time the packet is retransmitted. A sequence number identifies duplicates. At any given point in time only one packet can be outstanding. Afterwards Link-State Requests are sent between the two routers. The other side then sends the requested link-state announcement (LSA) back to the requesting router. A full adjacency has been set up when the request list is empty. Now reliable flooding needs to ensure that the databases remain perfectly synchronised. Every time a link changes state or after a 30 minute timeout a LSA needs to be reflooded. A LS update received on one interface needs to be sent out on all other interfaces. This simple rule is unfortunately not sufficient because the flooding would never stop. So the router checks his database to see if the update was already received on a different path. In that case the update does not need to get reflooded. It is also necessary to acknowledge the updates because an non reliable transport layer was chosen. Additionally implicit acknowledgements and timeouts, throttling the generated LS updates, help to make the flooding more robust and the implementation more complex, yet again.

2.1.3 Areas

One problem of a link-state protocol is the computation cost bourn by every router, particularly in large networks. Many routers have an underpowered CPU and so OSPF areas where invented to divide a large network into smaller pieces. Every area is connected to a special backbone area. In most cases inter-area routing goes via the backbone. Routers that are connected to multiple areas are area border routers (ABR) and are always connected to the backbone area. If no direct connection to the backbone is possible, a virtual-link has to be established to at least one backbone router. Areas where no transit traffic is exchanged can be converted into stub areas, reducing the routing table to a bare minimum. Stub areas are useful to connect routers with minimal memory configurations to large OSPF clouds.

LSAs are flooded only inside an area. The ABR has the duty to reflood the other areas with special summary-LSAs to inform them of available prefixes inside the originating area.

2.1.4 Border routers

Besides ABRs another kind of boarder router exists. A router is automatically an AS border router (ASBR) if it imports routes from external sources into the link-state database. External sources are other routing protocols or manually configured static routes. These routers are on the boarder of the OSPF cloud but are not necessary on the real AS border. The external routes redistributed by a ASBR are special as they are flooded through the full OSPF cloud instead of per area as all other LSAs. Only stub areas are left out to avoid overloading those poor little routers in them.

2.2 Packets

There are five different packet types defined. Every packet starts with a common 24 byte OSPF header. This header includes all necessary information for the recipient to determine if it should be accepted and processed or ignored and dropped.

Version #	Type	Packet Length	
Router ID			
Area ID			
Checksum		Authentication Type	
Authentication Data			
Authentication Data			

Figure 2: Common OSPF header

The standard IP CRC checksum is used to validate packet integrity. Multiple authentication procedures are defined but only one can be considered useful. Only the cryptographic authentication is enough strong to protect OSPF traffic. Only cryptographic authentication can prevent spoofing and replay attacks. After the verification the payload of the packet is examined.
The following packet types are defined:

Table 1: OSPF packet types

1	Hello
2	Database Description
3	Link-State Request
4	Link-State Update
5	Link-State Acknowledgement

2.2.1 Hello

Version #	1	Packet Length	
Router ID			
Area ID			
Checksum		Authentication Type	
Authentication Data			
Authentication Data			
Network Mask			
Hello Interval		Options	Router Priority
Router Dead Interval			
Designated Router			
Backup Designated Router			
Neighbor			
...			

Figure 3: Hello Header

Hello packets are sent periodically in order to establish and maintain neighbor relationships. Hello packets are sent to a multicast group to enable dynamic discovery of neighboring routers. All routers to a common network must agree on certain parameters. The most important part of the hello packet is the neighbor list at the end. The router ID of each router from which a valid Hello packet has recently been received is added to that list. Only after the own router ID is seen in a neighbors Hello packet an adjacency can be formed.

2.2.2 Database Description

Version #	2	Packet Length	
Router ID			
Area ID			
Checksum		Authentication Type	
Authentication Data			
Authentication Data			
Interface MTU		Options	Flags
DD Sequence Number			
LSA Header			
...			

Figure 4: Database Description Header

These packets are exchanged when an adjacency is initialised. They describe the contents of the link-state database. The initial database exchange is done similar to the TFTP protocol. For that reason a sequence number is included in the header.

Additionally the MTU of the outgoing interface is included to detect possible forwarding issues with large packets. The rest of the packet consists of a list of LSA headers. A LSA header contains all information to uniquely identify a LSA.

2.2.3 Link-State Request

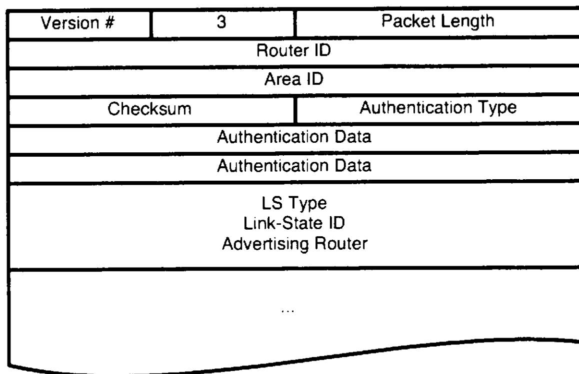


Figure 5: Link-State Request Header

After exchanging Database Description packets with the neighboring router, Link-State Request packets request pieces of the neighbors LS database that are more up-to-date. Each LSA requested is specified by its LS type, Link-State ID, and Advertising Router. This uniquely identifies the LSA, but not its instance. Link-State Request packets are understood to be requests for the most recent instance. It is possible to request multiple LSA with one LS request packet.

2.2.4 Link-State Update

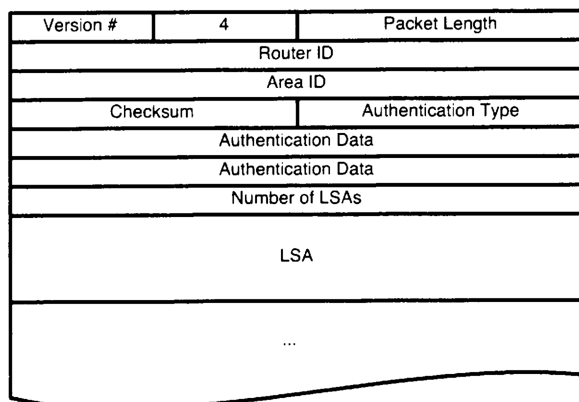


Figure 6: Link-State Update Header

These packets implement the flooding of LSAs. Each Link-State Update packet carries a collection of LSAs one hop further from their origin. Several LSAs may be included in a single packet. The body of the Link-State Update packet consists of a list of LSAs.

2.2.5 Link-State Acknowledgement

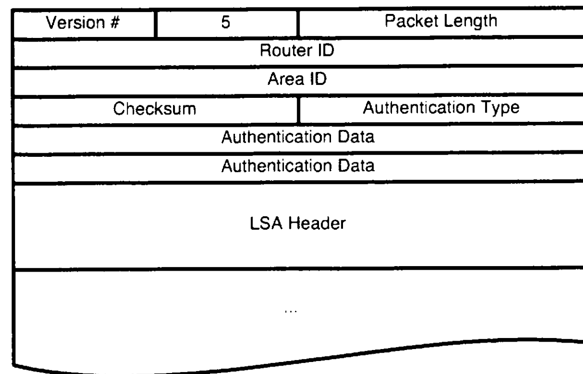


Figure 7: Link-State Acknowledgement Header

In order to make the flooding procedure reliable, flooded LSAs are acknowledged in Link-State Acknowledgement packets. Multiple LSAs can be acknowledged in a single Link-State Acknowledgement packet. The format of this packet is similar to that of the Data Description packet. The body of both packets is simply a list of LSA headers.

2.2.6 Link-State Advertisements Header

Each LSA begins with a common 20 byte header. This header is enough to uniquely identify a LSA. So it is enough to use the LSA header in LS acknowledgements and Database Description packets. LSAs are identified by the LS type, Link-State ID, and Advertising Router triple. Additionally a LS sequence number and LS age are included to determine which instance is more recent. The LS checksum protects the integrity of LSAs. Instead of the known CRC algorithm specified in many IP protocols a ISO checksum algorithm – also known as Fletcher Checksum – is employed.

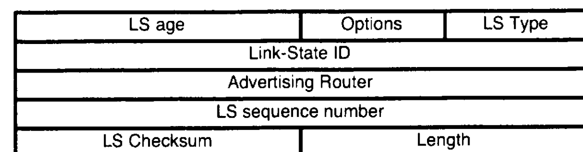


Figure 8: Link-State Advertisements Header

Each LSA type has a separate advertisement format. The LS types defined in the OSPF standard are as follows:

Table 2: LS types

1	Hello
2	Database Description
3	Link-State Request
4	Link-State Update
5	Link-State Acknowledgement

Router- and Network-LSA describe the network inside an area. Summary-LSA are injected by area border routers (ABRs) and describe inter-area destinations. AS-external-LSAs are originated by ASBRs to describe destinations external to the OSPF routing domain.

Design

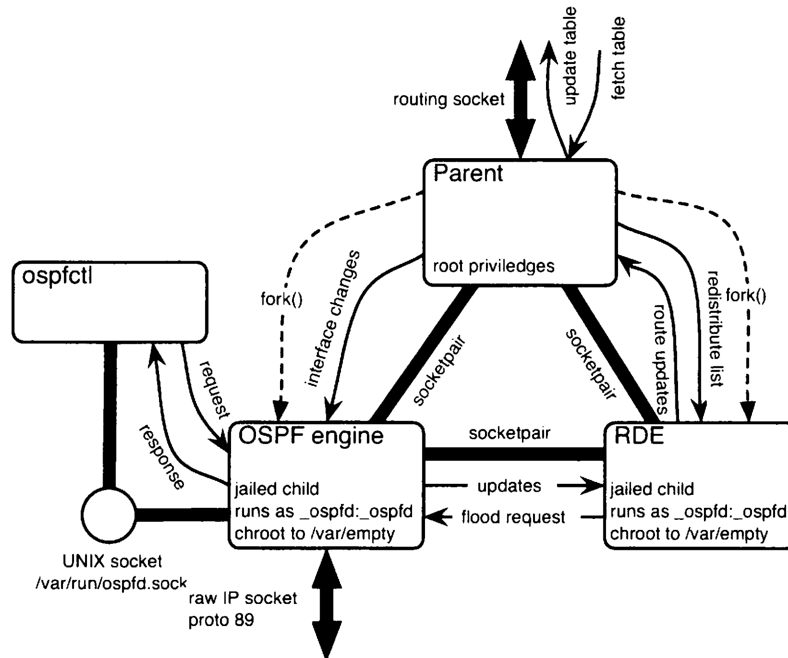


Figure 9: Design of OpenOSPFD

The design of OpenOSPFD is based on the one in OpenBGPD. The routing daemon is split into three processes. The privileged parent process handles the kernel routing table updates. The OSPF engine handles all incoming packets and the state machines with all the necessary periodic events and timeouts. Finally the route decision engine stores the LS database, calculates the SPF tree and the resulting routing table. This separation into three processes does not only enhance the security but also the stability. Even a large database recomputation in the RDE will not hold up the keep alive packets sent out by the OSPF engine. The Inter-Process Communication (IPC) system is almost the same as in OpenBGPD. The only major difference is the use of libevent for timers and file descriptor polling instead of poll(2). The basic imsg framework is still the same. OpenOSPFD switched to libevent mostly because of the OSPF engine. The engine is mostly event driven with many concurrent timers running. OpenOSPFD can be controlled and monitored via ospfctl. It works very similar to bgpctl for OpenBGPD.

3.1 Processes

3.1.1 ospfd parent

The ospfd parent process is the only one running with root privileges. This is necessary to update the kernel routing table. This process listens on a routing socket for changes and updates and distributes that information to the OSPF engine or the RDE. At a later time config-file reloads will be handled by the parent process too.

3.1.2 OSPF engine

The OSPF engine listens to the network and processes the OSPF packets. Both the interface and the neighbor finite state machine are implemented in the OSPF engine. This includes the DR/BDR election process. Additionally the reliable flooding of LS updates with retransmission and acknowledgement is done by the engine.

3.1.3 Route Decision Engine

The RDE stores the LS database, calculates the SPF tree, and informs the parent process about changes in the resulting routing table. Premature LSA aging is done by the RDE as well. Additionally redistribution of networks is handled by the process. The RDE synchronises multiple areas if the router is acting as ABR and refloods summary-LSA into the different areas if necessary.

3.1.4 ospfctl

ospfctl is the tool to control and monitor OpenOSPFD. It uses a UNIX local socket to communicate with ospfd. Over this socket imsgs are passed which encapsulate the information. There is no command line interface to OpenOSPFD because it doesn't make sense to write a clumsy CLI on a UNIX system shipping with very powerful shells and many tools to manipulate the status output. ospfctl is mostly an adapted bgpctl.

Implementation

OpenOSPFD currently consist of around 12'000 lines of C code. For comparison OpenBGPD is currently a bit under 20'000 lines. Zebra/Quagga ospfd has almost 40'000 lines of code. And that is just the ospfd directory, not including the 35'000 lines in lib and the 15'000 lines for the zebra daemon.

Lets start with a short overview of the source files.

Table 3: Overview of source files

area.c	Area handling which is actually very simple.
auth.c	Implementing all OSPF authentication extensions. Nobody wants to run a OSPF network without using cryptographic authentication.
buffer.c	buffer handling mostly for the imsg framework but also used to generate outgoing packets.
control.c	ospfctl session management and message verification.
database.c	Code for the initial database exchange. This is not related the LS database that is managed by the RDE.
hello.c	Generating and parsing of Hello packets is done here.
imsg.c	imsg framework mostly copied from OpenBGPD.
in_cksum.c	Implementation of the CRC16 checksum of the TCP/IP standards.
interface.c	Interface finite state machine, event handling and interface specific functions.
iso_cksum.c	ISO checksum also known as Fletcher checksum for LSAs.
kroute.c	Kernel routing socket handling including the FIB table.
log.c	Various logging functions mostly adapted from OpenBGPD.
lsack.c	Link-State Acknowledgement construction and parsing.
lsreq.c	Link-State Request construction and parsing, including the request list functions.
lsupdate.c	Link-State Updates construction and parsing, including the flooding function and retransmission lists.
neighbor.c	Neighbor finite state machine and event handling.
ospfd.c	Parent process, home of main().
ospfe.c	OSPF engine main event loop plus functions for self originated LSAs.
packet.c	Packet reception and sending.
parse.y	Configuration parser.
printconf.c	Configuration dumping used by the -n switch.
rde.c	RDE main event loop plus other RDE specific functions.
rde_lsdb.c	LS database code.
rde_spf.c	SPF algorithm and RIB calculation.

4.1 Important datastructures

There are four main datastructures in OpenOSPFD. It is important to know what such a structure represents to understand the code. Most of the time when the term interface is used, the actual `struct iface` of that interface is meant. Ditto for neighbor or area.

4.1.1 ospfd_conf

This is the main config of the router. It holds the parameters like the router ID, `spf_delay` or `redistribute_flags`. The `lsa_tree` and `cand_list` are used in the RDE by the LS database and SPF algorithm. The `area_list` holds all configured areas. Finally there is one event handler used for polling the raw socket or implementing the SPF timer depending on the process it is used in.

Code snip 1: struct ospfd_conf

```
struct ospfd_conf {
	struct event		ev;
	struct in_addr		rtr_id;
	struct lsa_tree		lsa_tree;
	LIST_HEAD(, area)	area_list;
	LIST_HEAD(, vertex)	cand_list;
	u_int32_t		opts;
	u_int32_t		spf_delay;
	u_int32_t		spf_hold_time;
	int			spf_state;
	int			ospf_socket;
	int			flags;
	int			redistribute_flags;
	int			options; /* OSPF options */
	u_int8_t		rfc1583compat;
	u_int8_t		border;
};
```

4.1.2 area

Area specific configurations are stored in the area descriptor. There are many parameters that are mostly used by the OSPF engine. Exclusively for the RDE are `lsa_tree` and the `nbr_list`. The first stores the per area LS database. The second is a list of all active neighbors from the RDE perspective. The OSPF engine tells the RDE when neighbors are created, deleted, or when their state changes. On the other hand `active` is only used by

the OSPF engine. active tracks the number of neighbors which are in state *FULL*. If the number is zero the area is considered inactive. This counter is used to determine if a router is an area border router.

Code snip 2: struct area

```
struct area {
        LIST_ENTRY(area)        entry;
        struct in_addr          id;
        struct lsa_tree         lsa_tree;
        LIST_HEAD(, iface)      iface_list;
        LIST_HEAD(, rde_nbr)    nbr_list;
        u_int32_t               stub_default_cost;
        u_int32_t               num_spf_calc;
        u_int32_t               dead_interval;
        int                     active;
        u_int16_t               transmit_delay;
        u_int16_t               hello_interval;
        u_int16_t               rxmt_interval;
        u_int16_t               metric;
        u_int8_t                priority;
        u_int8_t                transit;
        u_int8_t                stub;
};
```

4.1.3 interface

Every configured interface is represented by a struct iface. It stores values like the link_state, baudrate, MTU, and interface type. There are some additional OSPF specific parameters like the auth_type, list of keys used for cryptographic authentication (auth_md_list), interface metric and interface state. Lets have a look at the neighbor list and the three neighbor pointers dr, bdr, and self. dr and bdr are pointers to the active DR or BDR neighbor or NULL if there is none. self is used for a dummy neighbor structure that represents the router himself. Using this dummy neighbor simplifies many cases but additional care needs to be taken to not remove it by accident or doing some other stupid action with it. A back pointer to the parent area this interface is part of is also included. An interface can have up to three concurrent timers running and therefore three different event structures are needed.

Code snip 3: struct iface

```
struct iface {
        LIST_ENTRY(iface)       entry;
        struct event            hello_timer;
        struct event            wait_timer;
        struct event            lsack_tx_timer;

        LIST_HEAD(, nbr)        nbr_list;
        TAILQ_HEAD(, auth_md)   auth_md_list;
        struct lsa_head         ls_ack_list;

        char                    name[IF_NAMESIZE];
        struct in_addr          addr;
        struct in_addr          dst;
        struct in_addr          mask;
        struct in_addr          abr_id;
        char                    *auth_key;
        struct nbr              *dr;
        struct nbr              *bdr;
        struct nbr              *self;
        struct area             *area;

        u_int32_t               baudrate;
        u_int32_t               dead_interval;
        u_int32_t               ls_ack_cnt;
        u_int32_t               crypt_seq_num;
        unsigned int            ifindex;
        int                     fd;
        int                     state;
        int                     mtu;
        u_int16_t               flags;
        u_int16_t               transmit_delay;
        u_int16_t               hello_interval;
        u_int16_t               rxmt_interval;
        u_int16_t               metric;
        enum iface_type         type;
        enum auth_type          auth_type;
        u_int8_t                auth_keyid;
        u_int8_t                linkstate;
        u_int8_t                priority;
        u_int8_t                passive;
};
```

4.1.4 neighbor

struct neighbor represents the neighbor relationship from the local point of view. To maintain a session successfully a LS retransmission and request list is required plus a list for the database snapshot. Then a few values – dd_seq_num, dd_pending, last_rx_options, last_rx_bits, and master – are only used in the *EXCHANGE* phase when Database Description packets are transmitted. peerid is a unique ID used in all three processes. The peerid is used in imsgs to tell the recipient of the message which neighbor is guilty for the just received message. The interface, over which this neighbor is reached, is stored in iface. The neighbor structure is per interface so if two routers are connected via two different networks two different neighbor structures will be created for the same router but the structures are added to different interfaces.

Code snip 4: struct nbr

```
struct nbr {
        LIST_ENTRY(nbr)  entry, hash;
        struct event     inactivity_timer;
        struct event     db_tx_timer;
        struct event     lsreq_tx_timer;
        struct event     ls_retrans_timer;
        struct event     adj_timer;

        struct nbr_stats stats;
        struct lsa_head  ls_retrans_list;
        struct lsa_head  db_sum_list;
        struct lsa_head  ls_req_list;

        struct in_addr   addr;
        struct in_addr   id;
        struct in_addr   dr;     /* designated router */
        struct in_addr   bdr;    /* backup DR */

        struct iface    *iface;
        struct lsa_entry*ls_req;
        struct lsa_entry*dd_end;

        u_int32_t        dd_seq_num;
        u_int32_t        dd_pending;
        u_int32_t        peerid;/* unique ID in DB */
        u_int32_t        ls_req_cnt;
        u_int32_t        crypt_seq_num;

        int              state;
        u_int8_t         priority;
        u_int8_t         options;
        u_int8_t         last_rx_options;
        u_int8_t         last_rx_bits;
        u_int8_t         master;
};
```

4.2 Parent Process

4.2.1 Start-up

On start-up ospfd first initialises the log subsystem and fetches the list of available interfaces. This list is required for the next step, the configuration file parsing. The yacc parser used by ospfd is based on bgpds parser which in turn has his origin in the pf parser. Explaining

the parser goes beyond the scope of this paper. Important to know is that the configuration is parsed into a hierarchy of structures.
The configuration consists of a list of areas and every area holds a list of interfaces that are part of this area. Last but not least every interface has a list of neighbors that is dynamically created as soon as a valid Hello packet is received from an other OSPF router on that interface.
After the file got parsed ospfd daemonises and starts the child processes. Beforehand a set of socketpairs – a special sort of pipes – are created. Finally the event handlers are set up, rest of the kroute structures is initialised and the parent reports ready for service.
Meanwhile both children have started. First of all both *chroot(2)* to */var/empty* and drop privileges by switching to the special user *_ospfd*. Before doing that the OSPF engine creates a UNIX local socket for ospfctl and opens the raw IP socket to receive and send packets to the network. After dropping privileges the OSPF engine initialises the different subsystems, sets the event handlers and starts the actual work by kicking the interface finite state machine. The RDE start-up is even simpler as it just has to initialise the internal structures and event handlers.

4.2.2 Routing socket and FIB

The main purpose of the parent process is to maintain the Forward Information Base (FIB) and keep the information in sync with the kernel routing table. This synchronisation is to be done in both directions. Additionally link-state changes and arrival or departure of interfaces are handled via the routing socket as well.
The kroute code maintains two primary data structures. A prefix tree (`kroute`) and an interface tree (`kif`). These two trees are kept in sync with the kernel through the routing socket. On start-up `fetchtable()` loads the kroute tree and `fetchifs()` does the same for the kif tree. Routing changes are tracked by `dispatch_rtmsg()` which handles kroute changes directly but off-loads interface specific messages to `if_change()` and `if_announce()`. To modify the kernel routing table `send_rtmsg()` is used. `send_rtmsg()` translates a `struct kroute` into a `rt_msg` structure expected by the routing socket. The parent process uses `kr_change()` to add or modify routes and `kr_delete()` to remove routes. These changes are propagated to the kernel routing table if needed.

Both the kroute and kif tree are implemented as red-black trees – a balanced binary tree. An API to find, insert and remove nodes is specified to simplify the tree manipulation.
Everytime a route is added or removed to the kroute tree `kr_redistribute()` is called. This function transmits possible candidates for redistribution to the RDE. In the RDE `kif_validate()` verifies that the nexthop is actually reachable. This is a work a round that should be fixed later as it is currently not possible to track and handle newly arriving network interfaces at runtime. Last but not least `kr_show_route()` and `kr_ifinfo()` pass information about kroutes or interfaces to ospfctl.

4.3 OSPF Engine

The finite state machines implemented in ospfd are simple table driven state machines. Any state transition may result in an specific action to be run. The resulting next state can either be a result of the action or is fixed and pre-determined.

4.3.1 Interface state machine

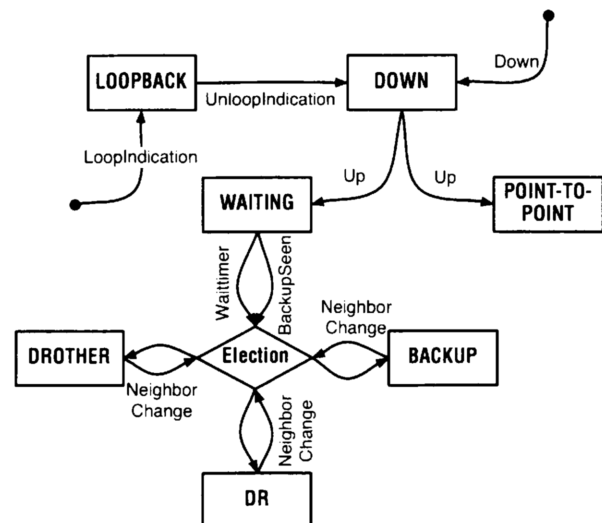


Figure 10: Interface FSM

DOWN

In this state, the lower-level protocols have indicated that the interface is unusable. No protocol traffic at all will be sent or received on such an interface.

LOOPBACK

In this state, the router's interface to the network is looped back. Loopback interfaces are advertised in router-LSAs as single host routes, whose destination is the interface IP address.

POINT-TO-POINT

Point-to-point networks or virtual links enter this state as soon as the interface is operational.

WAITING

Broadcast or NBMA interfaces enter this state when the interface gets operational. While in this state no DR/BDR election is allowed. Receiving and sending of Hello packets is allowed and is used to try to determine the identity of the DR/BDR routers.

DROTHER

The router is neither DR nor BDR on the connected network. In this state the router will only form adjacencies to both the DR and the BDR. All other neighbors will stay in neighbor state *2-WAY*.

BACKUP

The router is the BDR on the connected network segment. If the DR fails it will promote itself to be the new DR. The router forms adjacencies to all neighbors in the network segment.

DR

The router is the DR on the connected network segment. Adjacencies are established to all neighbors in the network segment. Additional duties are origination of a network-LSA for the network node and flooding of LS updates on behalf of all other neighbors.

Only a few events are needed. The events *UP*, *DOWN*, *LOOP*, *UNLOOP* are obvious. The other events *WAITTIMER*, *BACKUPSEEN* and *NEIGHBORCHANGE* are restricted to broadcast and NBMA networks. *WAITTIMER* and *BACKUPSEEN* are used to move out of state *WAITING* by running the election process. The *NEIGHBORCHANGE* event is issued when there is a change in the set of the bidirectional neighbors. This event will force a re-election of the DR and BDR.
The most important actions are `if_act_start()` and `if_act_elect()`. `if_act_start()` sets the correct next state (*POINT-TO-POINT* or *WAITING*), initialises the interface and starts the hello timer to begin with the neighbor discovery process. `if_act_elect()` elects a DR and BDR for a network. This function caused major problems because of subtle bugs and a sloppy written RFC.
First a backup designated router has to be elected.

Code snip 5: BDR election

```
/* elect backup designated router */
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
	if (nbr->priority == 0 ||	/* not electable */
	    nbr->state & NBR_STA_PRELIM ||
					/* not available */
	    nbr->dr.s_addr == nbr->addr.s_addr ||
	    nbr == dr)			/* don't elect DR */
		continue;
	if (bdr != NULL) {
		/*
		 * routers announcing themselves as BDR
		 * have higher precedence over those
		 * routers announcing a different BDR.
		 */
		if (nbr->bdr.s_addr == nbr->addr.s_addr) {
			if (bdr->bdr.s_addr ==
			    bdr->addr.s_addr)
				bdr = if_elect(bdr, nbr);
			else
				bdr = nbr;
		} else if (bdr->bdr.s_addr !=
		    bdr->addr.s_addr)
			bdr = if_elect(bdr, nbr);
	} else
		bdr = nbr;
}
```

Every neighbor is evaluated, neighbors with a priority of 0 are skipped. Additionally all neighbors that are not in state *2-WAY* or higher plus possible DRs are skipped. From the remaining set a BDR is selected. Routers announcing themselves as BDR have higher precedence so the code checks if the current neighbor is announcing himself BDR. The same thing is done with the current candidate. If both are announcing themselves as BDR or both are not announcing themselves as BDR `if_elect()` elects a new candidate. The helper function `if_elect()` compares two neighbors and returns the preferred one. In the other two cases no additional comparison needs to be done as the next candidate is known.

Code snip 6: DR election

```
/* elect designated router */
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
	if (nbr->priority == 0 ||
	    nbr->state & NBR_STA_PRELIM ||
	    (nbr != dr &&
	    nbr->dr.s_addr != nbr->addr.s_addr))
	    /* only DR may be elected check priority too */
		continue;
	if (dr == NULL)
		dr = nbr;
	else
		dr = if_elect(dr, nbr);
}

if (dr == NULL) {
	/* no designate router found use backup DR */
	dr = bdr;
	bdr = NULL;
}
```

Almost the same process is done for electing a DR. Neighbors that are neither in state *2-WAY* or higher or have a priority of 0 are skipped again. Additionally all neighbors that don't announce themselves as DR are skipped as well, with the only exception of the current DR itself. This is done because the election process can be restarted with the current candidates. If no DR was elected the current BDR is promoted DR. If the router is involved in the election it has to redo the election.

Code snip 7: final step of election

```
/*
 * if we are involved in the election (e.g. new DR or no
 * longer BDR) redo the election
 */
if (round == 0 &&
    ((iface->self == dr && iface->self != iface->dr) ||
    (iface->self != dr && iface->self == iface->dr) ||
    (iface->self == bdr && iface->self != iface->bdr) ||
    (iface->self != bdr && iface->self == iface->bdr))) {
	/*
	 * Reset announced DR/BDR to calculated one, so
	 * that we may get elected in the second round.
	 * This is needed to drop from a DR to a BDR.
	 */
	iface->self->dr.s_addr = dr->addr.s_addr;
	if (bdr)
		iface->self->bdr.s_addr = bdr->addr.s_addr;
	round = 1;
	goto start;
}
```

Before doing that we set the current candidates in our own structure so that the second round will actually modify the behaviour. It is well possible that some checks are unnecessary or to complex but this current implementation seems to behave correctly and so we keep it as is.

After the election process a bit of housekeeping has to be performed. If the DR or BDR changed, all neighbors have to be checked if the adjacency is still OK. Additionally it may be necessary to join or leave the `AllDRouters` multicast group. In case the router was or is now the DR an updated network-LSA needs to be reflooded.
Getting the DR/BDR election right was one of the most difficult parts of the development. Often unexpected behaviours where found because of small mistakes here and in `recv_hello()`. It took multiple retries and many debugging sessions to get that code where it is now. The poorly written RFC doesn't help much in clarifying the issues.

4.3.2 Neighbor state machine

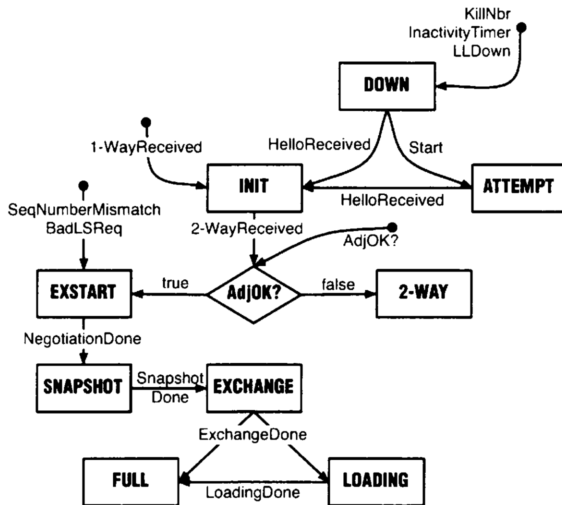


Figure 11: Neighbor FSM

DOWN

A neighbor is considered down if no hello has been received for more than router-dead-time seconds. This is also the initial state of a neighbor.

ATTEMPT

This state is only valid for neighbors attached to NBMA networks. Therefore it is currently unused.

INIT

In this state, a Hello packet has recently been seen from the neighbor. However, bidirectional communication has not yet been established.

2-WAY

The communication between the neighbor and the router is bidirectional. Neighbors will remain in this state if both the router itself and the neighbor are neither DR nor BDR.

EXSTART

This is the first step in creating an adjacency between the two routers. In this state the initial DD sequence number and the master is selected for the upcoming database exchange phase.

SNAPSHOT

This state is actually an extension of the state machine defined by the RFC. Because the LS database is stored in the RDE, a current snapshot of all LSA headers have to be requested by the OSPF engine. The database exchange will start after the snapshot is done.

EXCHANGE

This is the database exchange phase. Additionally all neighbors in state *EXCHANGE* or higher (*LOADING*, *FULL*) participate in the flooding procedure. Starting from this state all packet types can be received inclusive flooded LS updates.

LOADING

The state is only entered if the Link-State Request list is not empty. In that case Link-State Request packets are sent out to fetch the more recent LSAs from the neighbors LS database.

FULL

The two routers are now fully adjacent. The connection will now appear in router-LSAs and network-LSAs. Only in this state real traffic will be routed between the two routers.

4.3.3 Packet reception

The OSPF engine uses the `recv_packet()` libevent handler to receive packets from the raw IP socket. The packet is validated via `ip_hdr_sanity_check()` and `ospf_hdr_sanity_check()`. Some additional length checks are done to ensure that no access outside of the packet is done. It is currently not possible in OpenBSD 3.8 to get the incoming interface via *recvfrom(2)* so we need to find the interface the hard way. `find_iface()` does this job by walking through all configured interfaces and comparing the source address of the incoming packet with the interface address. This is not optimal and will be changed soon. The next step is looking up the neighbor and afterwards the OSPF authentication is run. `nbr_find_id()` takes the unique router ID to get the neighbor structure with all information needed. This is done before `auth_validate()` because the cryptographic authentication method uses a per neighbor specific sequence number to immunize against replay attacks. If necessary `auth_validate()` does the CRC

checksumming of the packet. Finally the packet is passed on according to its packet type to one of the following functions.

recv_hello()

Every hello-interval seconds a Hello packet is sent to all neighbors. On broadcast networks this is done with one multicast packet. The Hello packet is used for neighbor discovery and to maintain neighbor relationships. As first step all the common options need to be compared. If one of hello-interval, router-dead-time, or the stub area flag differs the packet is not accepted. So all routers on a common network must have the same configuration for these values.

Code snip 8: neighbor look up

```
switch (iface->type) {
case IF_TYPE_POINTOPOINT:
case IF_TYPE_VIRTUALLINK:
        /* match router-id */
        LIST_FOREACH(nbr, &iface->nbr_list, entry) {
                if (nbr == iface->self)
                        continue;
                if (nbr->id.s_addr == rtr_id)
                        break;
        }
        break;
case IF_TYPE_BROADCAST:
case IF_TYPE_NBMA:
case IF_TYPE_POINTOMULTIPOINT:
        /* match src IP */
        LIST_FOREACH(nbr, &iface->nbr_list, entry) {
                if (nbr == iface->self)
                        continue;
                if (nbr->addr.s_addr == src.s_addr)
                        break;
        }
        break;
default:
        fatalx("recv_hello: unknown interface type");
}

if (!nbr) {
        nbr = nbr_new(rtr_id, iface, 0);
        /* set neighbor parameters */
        nbr->dr.s_addr = hello.d_rtr;
        nbr->bdr.s_addr = hello.bd_rtr;
        nbr->priority = hello.rtr_priority;
        nbr_change = 1;
}
```

The packet is now accepted and the neighbor is looked up. Depending on the interface type either by router ID or by interface address. If no neighbor could be found a new one is created. A new neighbor is considered a *NEIGHBORCHANGE* and the nbr_change flag is set that an interface neighbor change event can be issued later.

Code snip 9: bidirectional or not

```
nbr_fsm(nbr, NBR_EVT_HELLO_RCVD);

while (len >= sizeof(nbr_id)) {
        memcpy(&nbr_id, buf, sizeof(nbr_id));
        if (nbr_id == ospfe_router_id()) {
                /* seen myself */
                if (nbr->state & NBR_STA_PRELIM)
                        nbr_fsm(nbr, NBR_EVT_2_WAY_RCVD);
                break;
        }
        buf += sizeof(nbr_id);
        len -= sizeof(nbr_id);
}

if (len == 0) {
        nbr_fsm(nbr, NBR_EVT_1_WAY_RCVD);
        /* set neighbor parameters */
        nbr->dr.s_addr = hello.d_rtr;
        nbr->bdr.s_addr = hello.bd_rtr;
        nbr->priority = hello.rtr_priority;
        return;
}
```

Multiple neighbor events have to be generated. First of all is the hello received event. Next it is checked if there is already bidirectional communication between the routers. This is done by walking through the list of neighbors in the hello packet and compared it with the own router ID. If no match was found a *1-WAY* received event gets issued. If the match is done the first time – the neighbor is in an embryonic state like *INIT* – a *2-WAY* received event is generated.

Now the scariest part of OpenOSPFD is coming. Handling fast start-ups and the famous interface event *BACKUPSEEN*. This part of the Hello protocol was rewritten multiple times and the result was always some other obscure problem in the election process. In the end OpenOSPFD had to violate the RFC a bit. The RFC is not very clear about how to handle the event *BACKUPSEEN* correctly.

From the RFC:

- *If the neighbor is both declaring itself to be Designated Router (Hello Packet's Designated Router field = Neighbor IP address) and the Backup Designated Router field in the packet is equal to 0.0.0.0 and the receiving interface is in state Waiting, the receiving interface's state machine is scheduled with the event BACKUPSEEN. ...*
- *If the neighbor is declaring itself to be Backup Designated Router (Hello Packet's Backup Designated Router field = Neighbor IP address) and the receiving interface is in state Waiting, the receiving interface's state machine is scheduled with the event BACKUPSEEN. ...*

Now this sounds simple but it isn't. The first case is not problematic but the second one is. Why? Because it is not known in which order hello packets are received. What does happen if we start an election process and the actual DR neighbor is still in state *1-WAY*? A major confusion is the result. The election process evaluates the BDR as DR and himself as BDR or something like this and the result is a network with too many DR / BDR routers.

Code snip 10: scary fast start-ups

```
if (iface->state & IF_STA_WAITING &&
    hello.d_rtr == nbr->addr.s_addr && hello.bd_rtr == 0)
        if_fsm(iface, IF_EVT_BACKUP_SEEN);

if (iface->state & IF_STA_WAITING &&
    hello.bd_rtr == nbr->addr.s_addr) {
        /*
         * In case we see the BDR make sure that the DR is
         * around with a bidirectional connection
         */
        LIST_FOREACH(dr, &iface->nbr_list, entry)
                if (hello.d_rtr == dr->addr.s_addr &&
                    dr->state & NBR_STA_BIDIR)
                        if_fsm(iface, IF_EVT_BACKUP_SEEN);
}
```

To clear up the situation OpenOSPFD does an additional check. It verifies that the DR has a bidirectional connection to the router and only if that is true a backup seen event is issued. The result is that it may take a bit longer to establish an adjacency and that some initial Database Description packet are dropped. But the confusion of too many DR/BDRs is avoided. The rest of `recv_hello()` is simply here to issue the possible neighbor change events that were detected earlier.

recv_db_description()

While the `send_db_description()` function ended up pretty simple `recv_db_description()` turned out to be more problematic. Usual sanity checking is done first. Afterwards additional checks are performed to verify the MTU and detect possible duplicates because of retransmissions. The MTU check is required by the RFC, the problem is that some OSPF implementations are lying about their MTU and so only bigger MTUs are considered a problem.

The code path is dependent on the neighbor state. Packets received from neighbors in unexpected states are just ignored. This includes state *SNAPSHOT* because during the time the LSA snapshot is done we cannot respond to a received packet. Funnily it is allowed to get Database Description packets in state *INIT*. In that case some kind of super fast start-up needs to be done. It looks like it was simpler to fix the RFC than to fix someone's OSPF implementation. So both the interface and neighbor FSM are kicked and afterwards the new neighbor state has to be checked again. If it is now in state *EXSTART* a fall-through into the next case can be done.

In case *EXSTART* there are two possible scenarios. The first is the reception of a Christmas packet – one with all flags turned on. This is the initial packet and OpenOSPFD has to evaluate if it is master or slave of the database exchange phase. The slave will issue a negotiation done event and sends back a packet with just the *M* bit set.

Code snip 11: EXSTART scenario 1

```
/*
 * check bits: either I,M,MS or only M
 */
if (dd_hdr.bits == (OSPF_DBD_I | OSPF_DBD_M |
    OSPF_DBD_MS)) {
        /* if nbr Router ID is larger than own -> slave */
        if ((ntohl(nbr->id.s_addr)) >
            ntohl(ospfe_router_id())) {
                /* slave */
                nbr->master = 0;
                nbr->dd_seq_num = ntohl(dd_hdr.dd_seq_num);

                /* event negotiation done */
                nbr_fsm(nbr, NBR_EVT_NEG_DONE);
        }
```

The second scenario – a packet with just the *M* bit set, is received. The *M* bit stands for "more" as in more data. The master will finally issue the negotiation done event. So the slave is actually sending valid data ahead of the master. This is a bit strange but we are used to it.

Code snip 12: EXSTART scenario 2

```
} else if (!(dd_hdr.bits & (OSPF_DBD_I | OSPF_DBD_MS))) {
        /* M only case: we are master */
        if (ntohl(dd_hdr.dd_seq_num) != nbr->dd_seq_num) {
                log_warnx("recv_db_description: invalid "
                    "seq num, mine %x his %x",
                    nbr->dd_seq_num,
                    ntohl(dd_hdr.dd_seq_num));
                nbr_fsm(nbr, NBR_EVT_SEQ_NUM_MIS);
                return;
        }
        nbr->dd_seq_num++;

        /* packet may already have data so pass it on */
        if (len > 0) {
                nbr->dd_pending++;
                ospfe_imsg_compose_rde(IMSG_DD,
                    nbr->peerid, 0, buf, len);
        }

        /* event negotiation done */
        nbr_fsm(nbr, NBR_EVT_NEG_DONE);

}
```

Afterwards the actual transfer starts or continues. First of all, packets with invalid flags and options result in a reset of the session (sequence number mismatch event). If the slave receives a duplicate packet it has to resend the last packet. The master does not care about duplicate packets. Actually the master should never see a duplicate – the slave will never send a packet by its own. If the neighbor state is either *LOADING* or *FULL* the only packets received should be duplicates. Anything else is considered an error and the session is reset. Side effect of this is that sending a packet with the Initialise (*I*) bit set can be used to reset a neighbor relationship. Now the sequence number is checked. Only the master is increasing the number so the slave receives packets with the current sequence number plus one. In case of the master the sequence numbers are equal on receive and afterwards the sequence number is increased. Our first implementation was a bit buggy and it took some debugging to find all the small issues like forgetting to bump the sequence number in a specific case.

Code snip 13: synchronising part 1

```
/* forward to RDE and let it decide which LSAs to request
 */
if (len > 0) {
        nbr->dd_pending++;
        ospfe_imsg_compose_rde(IMSG_DD, nbr->peerid, 0,
            buf, len);
}
```

The received LSA headers have to be sent to the RDE where they are compared with the LS database. This resulted in an interesting issue: if the RDE was busy the OSPF engine could move forward and suddenly think that no LSAs have to be requested and move the neighbor directly into state *FULL*. Afterwards the RDE would send some LSAs to request to the OSPF engine but it was too late. To solve this race condition the `dd_pending` counter was added. It gets increased for each sent database description packet.

Code snip 14: synchronising part 2 ospfe_dispatch_rde()

```
nbr->dd_pending--;
if (nbr->dd_pending == 0 && nbr->state & NBR_STA_LOAD) {
	if (ls_req_list_empty(nbr))
		nbr_fsm(nbr, NBR_EVT_LOAD_DONE);
	else
		start_ls_req_tx_timer(nbr);
}
```

When an `IMSG_DD_END` message arrives from the RDE the counter gets decremented. If the counter drops to zero no DD packets are pending. In case that the neighbor state is now *LOADING* we actually hit the race condition and so we have to either move to state *FULL* if the request list is empty or start sending out LS requests. Sometimes running a single daemon as three processes needs some additional work to synchronise the processes. This is a nice example. Finally the next packet is prepared for being sent by `send_db_description()`. If there is nothing left to send and the received packet has no *M* bit set then the exchange phase is mostly done. The slave is finished but the master has to ensure that at least one packet without the *M* bit has been sent and acknowledged. The result is that the slave will always change state before the master. Why should the end of the exchange be less strange than the beginning?

recv_ls_req()

Link-State Requests are simply passed to the RDE but only if the neighbor state is *EXCHANGE* or higher. In all other states Link-State Request packets are ignored.

recv_ls_update()

Link-State Updates are simply dropped if the neighbor is not in state *EXCHANGE* or higher. Otherwise all LSAs are extracted from the packet and sent to the RDE one after the other. While doing that additional length checks are done to guard against buffer overflows.

recv_ls_ack()

Link-State Acknowledgements are only accepted in neighbor state *EXCHANGE* or higher. Otherwise the packet is dropped. Every LSA header included in the packet needs to be roughly validated with `lsa_hdr_check()` and then possibly deleted from the retransmission list. In case the interface is in state *DROTHER* `ls_retrans_list_del()` will be called twice. First it deletes LSAs from the global retransmission list of updates sent to the `AllDRouters` multicast address. Second the per-neighbor queue is purged in case the interface state changed lately.

4.3.4 Packet delivery

send_hello()

`send_hello()` is called by the `if_hello_timer()` function that is run every hello-interval seconds if an interface is not in state *DOWN*. Sending hellos is pretty simple so it is a good example how the buffer framework is used in OpenOSPFD.

Code snip 15: Allocate dynamic buffer

```
/* XXX READ_BUF_SIZE */
if ((buf = buf_dynamic(PKG_DEF_SIZE,
    READ_BUF_SIZE)) == NULL)
	fatal("send_hello");
```

First a dynamic buffer is allocated. Currently a fixed size of `PKG_DEF_SIZE` bytes is used but the buffer is allowed to grow till `READ_BUF_SIZE`. This is not optimal as packets should not be fragmented by OSPF. For Hello packets this is not a big issue because the embedded data is often very small. Other send functions use a different approach by limiting the resulting packet size to the MTU of the corresponding interface.

Code snip 16: Set correct destination

```
dst.sin_family = AF_INET;
dst.sin_len = sizeof(struct sockaddr_in);

switch (iface->type) {
case IF_TYPE_POINTOPOINT:
case IF_TYPE_BROADCAST:
	inet_aton(AllSPFRouters, &dst.sin_addr);
	break;
case IF_TYPE_NBMA:
case IF_TYPE_POINTOMULTIPOINT:
	/* XXX not supported */
	break;
case IF_TYPE_VIRTUALLINK:
	dst.sin_addr = iface->dst;
	break;
default:
	fatalx("send_hello: unknown interface type");
}
```

The outgoing address needs to be determined. For broadcast and point-to-point networks this is the multicast address `AllSPFRouters`. Virtual links are sent as unicast. NBMA and point-to-multipoint are special and currently not supported. For NBMA and point-to-multipoint the packet has to be sent to all neighbors directly and `send_packet()` would be called for every neighbor once.

Code snip 17: create Hello packet

```
/* OSPF header */
if (gen_ospf_hdr(buf, iface, PACKET_TYPE_HELLO))
	goto fail;

/* hello header */
hello.mask = iface->mask.s_addr;
hello.hello_interval = htons(iface->hello_interval);
hello.opts = oeconf->options;
hello.rtr_priority = iface->priority;
hello.rtr_dead_interval = htonl(iface->dead_interval);

if (iface->dr) {
	hello.d_rtr = iface->dr->addr.s_addr;
	iface->self->dr.s_addr = iface->dr->addr.s_addr;
} else
	hello.d_rtr = 0;
```

```
if (iface->bdr) {
        hello.bd_rtr = iface->bdr->addr.s_addr;
        iface->self->bdr.s_addr = iface->bdr->addr.s_addr;
} else
        hello.bd_rtr = 0;

if (buf_add(buf, &hello, sizeof(hello)))
        goto fail;
```

Finally the packet is constructed. First of all the common OSPF header is added. This is done for every packet type and so a helper function `gen_ospf_hdr()` is used. The Hello specific contents are filled in afterwards and added with `buf_add()`.

Code snip 18: Add active neighbors

```
/* active neighbor(s) */
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
        if ((nbr->state >= NBR_STA_INIT) &&
            (nbr != iface->self))
                if (buf_add(buf, &nbr->id,
                    sizeof(nbr->id)))
                        goto fail;
}
```

The Hello packets include a list of all bidirectional neighbors (state *2-WAY* or higher). Again the neighbor IDs are added directly with `buf_add()`. The neighbor ID is stored in network byte order or `htonl()` is used to correctly switch byte order.

Code snip 19: Final step

```
        /* update authentication and calculate checksum */
        if (auth_gen(buf, iface))
                goto fail;

        ret = send_packet(iface, buf->buf, buf->wpos,
            &dst);
        buf_free(buf);
        return (ret);
fail:
        log_warn("send_hello");
        buf_free(buf);
        return (-1);
```

Last is updating authentication and checksum of the outgoing packet. The interface pointer is passed to `auth_gen()` to get the necessary keys and sequence number for the simple and cryptographic authentication. The packet gets sent out via `send_packet()`. Before sending the packet it is necessary to set the outgoing interface for multicast traffic. This is done by `if_set_mcast()` inside of `send_packet()`. Finally the no longer needed buffer is freed.

send_db_description()

`send_db_description()` implements the sending part of the database exchange. It sends out the initial Database Description packet when moving the neighbor state to *EXSTART*.

Code snip 20: Allocate fixed buffer

```
if ((buf = buf_open(nbr->iface->mtu - sizeof(struct ip)))
    == NULL)
        fatal("send_db_description");

/* OSPF header */
if (gen_ospf_hdr(buf, nbr->iface, PACKET_TYPE_DD))
        goto fail;

/* reserve space for database description header */
if (buf_reserve(buf, sizeof(dd_hdr)) == NULL)
        goto fail;
```

Obvious differences to `send_hello()` are the use of `buf_open()` instead of `buf_dynamic()`. `Buf_open()` allocates a fixed size buffer of size `nbr->iface->mtu - sizeof(struct ip)` – which is the maximum packet size that does not get fragmented. Later `buf_reserve()` is used on that buffer to reserve `sizeof(dd_hdr)` bytes. The rest of the packet can be added and later `buf_seek()` can be used to write into the reserved space like this:

Code snip 21: Usage of buf_seek()

```
memcpy(buf_seek(buf, sizeof(struct ospf_hdr),
    sizeof(dd_hdr)), &dd_hdr, sizeof(dd_hdr));
```

The remainder of the function sets up the Database Description header with its bit fields and sequence number. If in state *EXCHANGE*, as many LSA headers as possible are appended. While appending LSA headers one must keep in mind that the cryptographic authentication will append `MD5_DIGEST_LENGTH` bytes to the end of the packet.

send_ls_req()

`send_ls_req()` uses like `send_db_description()` `buf_open()` to get a buffer that doesn't get fragmented. While filling in the requested LSA headers some additional space gets reserved for the possible MD5 sum.

Code snip 22: Filling packet with requests

```
/* LSA header(s), keep space for a possible md5 sum */
for (le = TAILQ_FIRST(&nbr->ls_req_list); le != NULL &&
    buf->wpos + sizeof(struct ls_req_hdr) < buf->max -
    MD5_DIGEST_LENGTH; le = nle) {
        nbr->ls_req = nle = TAILQ_NEXT(le, entry);
        ls_req_hdr.type = htonl(le->le_lsa->type);
        ls_req_hdr.ls_id = le->le_lsa->ls_id;
        ls_req_hdr.adv_rtr = le->le_lsa->adv_rtr;
        if (buf_add(buf, &ls_req_hdr, sizeof(ls_req_hdr)))
                goto fail;
}
```

The rest is straight forward and mostly the same as in `send_hello()`.

send_ls_ack()

Actually we have to start in `ls_ack_tx_timer()` because `send_ls_ack()` is just the last step to send out an ack. `send_ls_ack()` will add the common OSPF header and add the data passed to the function to the packet. The list of acknowledgements is created by `ls_ack_tx_timer()` in a not so nice way and therefore it should not be used as example for other code. Especially as it will be rewritten soon.

send_ls_update()

Sending out LS updates is easy but the retransmission list and flooding procedure are a bit tricky. `send_ls_update()` will just add a LSA to a buffer together with a common OSPF header and send the

results out. But there is one thing that must to be done with the LSA first. It has to be aged with the value of transmit-delay.

Code snip 23: LSA aging

```
pos = buf->wpos;
if (buf_add(buf, data, len))
        goto fail;

/* age LSA before sending it out */
memcpy(&age, data, sizeof(age));
age = ntohs(age);
if ((age += iface->transmit_delay) >= MAX_AGE)
        age = MAX_AGE;
age = htons(age);
memcpy(buf_seek(buf, pos, sizeof(age)), &age, sizeof(age));
```

First the current write position is stored and the LSA is added to the buffer. The LS Age is stored in the first two bytes of the LSA. The `memcpy()` extracts the age because a direct memory access could end on unaligned memory. Then the LSA is aged and written into the buffer with the help of `buf_seek()` and the previously stored position.

4.3.5 Control handling

The handling of control sessions is actually a small UNIX local socket server. There is a listener event (`control_listen()`) that accepts (`control_accept()`) connections and creates a per control connection structure. `control_dispatch_imsg()` reads the request from ospfctl. First the per connection structure are retrieved and then the imsg's sent are extracted. They get either forwarded to the parent, the RDE, or directly answered. Messages forwarded to the other processes will often require a response that needs to be relayed to ospfctl because neither the RDE nor the parent process have access to the socket. Relaying is done by `control_imsg_relay()`. It has to be called for those imsgs that need to get forwarded. This is done in the imsg dispatch functions `ospfe_dispatch_main()` and `ospfe_dispatch_rde()`.

4.4 Route Decision Engine

4.4.1 LS Database

The LS database is implemented as a red-black tree – actually multiple trees exist – one per area and a global one for AS-external-LSAs. The key is the LS-type LS-ID advertising router triple. The LSA is part of a `vertex` that builds a node of the network connectivity graph.

Code snip 24: struct vertex

```
struct vertex {
        RB_ENTRY(vertex) entry;
        TAILQ_ENTRY(vertex) cand;
        struct event     ev;
        struct in_addr   nexthop;
        struct vertex   *prev;
        struct rde_nbr  *nbr;
        struct lsa      *lsa;
        time_t           changed;
        time_t           stamp;
        u_int32_t        cost;
        u_int32_t        ls_id;
        u_int32_t        adv_rtr;
        u_int8_t         type;
        u_int8_t         flooded;
};
```

The vertex contains all necessary information not only for the LS Database but for the SPF calculation too. `entry` and `cand` are used to put the vertex into the red-black tree or into the candidate list respectively. The event `ev` is for a per-LSA entry timeout for aging. Additionally `stamp` is used for aging as well. `changed` is set to the time the last modification was done to the LSA. `ls_id`, `adv_rtr` and `type` are shorthands for the actual values that are stored inside of `lsa`. These are used by the tree search routine. The `flooded` flag should indicate that a LSA was received as part of a flooding. Flooded LSA are locked for `MIN_LS_ARRIVAL` seconds whereas requested LSA are not. `nbr` represents the neighbor from which the LSA was received. `nbr` has nothing to do with the actual originator of the LSA. This is only done to correctly flood out LSAs and sending an acknowledgement back to the neighbor. `prev` is the parent vertex in the SPF tree. It is possible to construct the actual path through the network by following all `prev` pointers. This is used to calculate the `nexthop`. The `nexthop` is the address for forwarding packets to that destination. It is normally the address of the last router-LSA before the root node.

4.4.2 LSA aging

Before using a LSA that is in the DB it normally needs to be aged. This is done by `lsa_age()` with help of the vertex time stamp.

Code snip 25: LSA aging

```
now = time(NULL);
d = now - v->stamp;
/* set stamp so that at least new calls work */
v->stamp = now;

if (d < 0) {
        log_warnx("lsa_age: time went backwards");
        return;
}

age = ntohs(v->lsa->hdr.age);
if (age + d > MAX_AGE)
        age = MAX_AGE;
else
        age += d;

v->lsa->hdr.age = htons(age);
```

Normally it is enough to just add the difference of the current time and stamp. Nonetheless some additional care is needed. First of all `time()` returns the system time and this can be modified by the user. I remember a complete network outage at an ISP because the UNIX time got changed on a Zebra/Quagga router. Afterwards Zebra/Quagga was no longer working until a reboot on the changed machines was performed. So by checking whether the difference is positive it is at least possible to fail in a save way. The other case that needs to be considered is that a LSA may never get older than `MAX_AGE` (1 hour).

4.4.3 Comparing LSA

There are two functions to compare LSA. `lsa_equal()` is similar to a `memcmp()` but compares a bit more. One thing is important to note: LSA with age `MAX_AGE` are never considered equal. This comes from the fact that `lsa_equal()` is mostly used to determine if a recalculation of the SPF tree is required or for similar situations. In that context LSAs with an age of `MAX_AGE` are always special and it is OK to force an update.
The other compare function is `lsa_newer()` and implements the RFC specification of newer, equal and older LSA. It works similar to other compare functions by returning -1 if the first LSA is older, 1 if newer and 0 if equal to the second LSA passed. The function compares the sequence number, the LS checksum, and the LS age. Once again a bit care needs to be taken when comparing ages.

Code snip 26: Comparing ages

```
a16 = ntohs(a->age);
b16 = ntohs(b->age);

if (a16 >= MAX_AGE && b16 >= MAX_AGE)
        return (0);
if (b16 >= MAX_AGE)
        return (-1);
if (a16 >= MAX_AGE)
        return (1);

i = b16 - a16;
if (abs(i) > MAX_AGE_DIFF)
        return (i > 0 ? 1 : -1);

return (0);
```

If both LSA are at age `MAX_AGE` they are considered equal. If only one has age `MAX_AGE` that one is newer and last but not least the LS ages need to be at least `MAX_AGE_DIFF` (15 minutes) apart to be not considered equal.

4.4.4 LSA refresh

All `LS_REFRESH_TIME` seconds a LSA needs to be refreshed by its originator. The age is reset to the initial value and the sequence number is bumped. After modifying the LSA the checksum has to be recalculated. The LSA is flooded and a new timeout event is registered. Non self originated LSA have the same timer running but with `MAX_AGE` instead of `LS_REFRESH_TIME`. If the timer fires the LSA will be deleted from the LS database by flooding it out with age `MAX_AGE`. How to delete LSA will be explained later as it is fairly complex.

4.4.5 LSA merging

If a self originated LSA changes, for example because a neighbor relationship is established or lost, an updated LSA needs to be reflooded. `lsa_merge()` takes care of replacing the LSA in the database with the new one and sets the LS sequence number of the new LSA to the current used number.

Code snip 27: First set sequence number

```
if (v == NULL) {
        lsa_add(nbr, lsa);
        rde_imsg_compose_ospfe(IMSG_LS_FLOOD, nbr->peerid,
            0, lsa, ntohs(lsa->hdr.len));
        return;
}

/*
 * set the seq_num to the current one.
 * lsa_refresh() will do the ++
 */
lsa->hdr.seq_num = v->lsa->hdr.seq_num;
/* recalculate checksum */
len = ntohs(lsa->hdr.len);
lsa->hdr.ls_chksum = 0;
lsa->hdr.ls_chksum = htons(iso_cksum(lsa, len,
    LS_CKSUM_OFFSET));
```

Sure if there was no LSA in the database in the first place there is no need to merge. It is enough to just add and flood the LSA. When changing the sequence number the checksum has to be recalculated. The sequence number is only set to the current value because there is no need to increase it already. Especially if `lsa_merge()` is used to remove a self originated LSA from the database there is no need to rise the sequence number, it is sufficient to set the age to `MAX_AGE`.

Code snip 28: Then overwrite and reflood if necessary

```
/*
 * compare LSA; most header fields are equal
 * so don't check them
 */
if (lsa_equal(lsa, v->lsa)) {
        free(lsa);
        return;
}

/* overwrite the lsa all other fields are unaffected */
free(v->lsa);
v->lsa = lsa;
start_spf_timer();

/* set correct timeout for reflooding the LSA */
now = time(NULL);
timerclear(&tv);
if (v->changed + MIN_LS_INTERVAL >= now)
        tv.tv_sec = MIN_LS_INTERVAL;
evtimer_add(&v->ev, &tv);
```

Now `lsa_equal()` is used to determine whether to actually reflood the LSA. If the LSA did not change there is nothing to modify and we're done. Otherwise the LSAs are exchanged and a SPF recalculation is issued. Finally the reflooding is prepared. This is done via a timer because it is not allowed to send out updates faster than `MIN_LS_INTERVAL` (5) seconds.

4.4.6 lsa_self()

Identifying self originated LSA is an important task. This comes from the fact that if a router leaves the network the other routers will not remove the LSAs of this router until the LS age hits `MAX_AGE`. If the router joins the network again – after a reboot for example – the old LSAs are still floating around. So it is the routers duty to detect those old self originated LSAs and renew them or remove them from the database. This task is done by `lsa_self()`.

Code snip 29: Detect self originated LSA

```
if (nbr->self)
        return (0);

if (rde_router_id() == new->hdr.adv_rtr)
        goto self;

if (new->hdr.type == LSA_TYPE_NETWORK)
        LIST_FOREACH(iface, &nbr->area->iface_list, entry)
            if (iface->addr.s_addr == new->hdr.ls_id)
                    goto self;
return (0);
```

First of all the newly received LSA (new) gets classified. If the router ID is the same or if an interface address matches the LS ID of a network-LSA the LSA is considered self originated.

Code snip 30: Remove or update

```
self:
if (v == NULL) {
        /*
         * LSA is no longer announced, remove by premature
         * aging. The problem is that new may not be
         * altered so a copy needs to be added to the LSA
         * DB first.
         */
        if ((dummy = malloc(ntohs(new->hdr.len))) == NULL)
                fatal("lsa_self");
        memcpy(dummy, new, ntohs(new->hdr.len));
        dummy->hdr.age = htons(MAX_AGE);
        /*
         * The clue is that by using the remote nbr as
         * originator the dummy LSA will be reflooded via
         * the default timeout handler.
         */
        lsa_add(rde_nbr_self(nbr->area), dummy);
        return (1);
}

/*
 * LSA is still originated, just reflood it. But we need to
 * create a new instance by setting the LSA sequence number
 * equal to the one of new and calling lsa_refresh().
 * Flooding will be done by the caller.
 */
v->lsa->hdr.seq_num = new->hdr.seq_num;
lsa_refresh(v);
return (1);
```

In case of a self originated LSA there are two cases. The first one is that the LSA is no longer announced. In that case the LSA gets added to the Database with a LS age of `MAX_AGE`. The database code will then reflood the LSA as soon as possible and by doing that removing it from the database. There is no other way in doing this because removing LSAs is a complex task that only works if the LSA is in the database. The other case is much simpler because there is already a self originated LSA in the local database but the sequence number is lower then the new one. In this case the sequence number is bumped like in the `lsa_merge()` case and `lsa_refresh()` is called to flood the LSA.

4.4.7 LSA check

Before even accepting a LS update the embedded LSA has to be verified. Once again lengths are compared and especially the ISO checksum is verified. Additionally the LS age and sequence number are checked to be in a valid range. Per LS type checks follow the generic ones. It is verified that the packet has the right size for this type and that values like the metric – which is a 24bit value stored as 32bit integer is in the correct range. AS-external-LSAs that are sent to stub areas get silently discarded. At the end the LS age is checked and if it is `MAX_AGE` some special care needs to be taken.

Code snip 31: MAX_AGE handling

```
if (lsa->hdr.age == htons(MAX_AGE) &&
    !nbr->self && lsa_find(area, lsa->hdr.type,
    lsa->hdr.ls_id, lsa->hdr.adv_rtr) == NULL &&
    !rde_nbr_loading(area)) {
        /*
         * if no neighbor in state Exchange or Loading
         * ack LSA but don't add it. Needs to be a direct
         * ack.
         */
        rde_imsg_compose_ospfe(IMSG_LS_ACK, nbr->peerid, 0,
            &lsa->hdr, sizeof(struct lsa_hdr));
        return (0);
}
```

If the LS age is `MAX_AGE` and the LSA is not in the database there is actually no need to add the LSA to the database. However this is a fallacy, there are some additional checks required. The RFC mentions that if a neighbor is currently establishing an adjacency – state *EXCHANGE* or *LOADING* – no short-cuts are allowed. Additionally self originated LSA generated by the OSPF engine have to be passed. Therefore `nbr->self` is tested. If all conditions are met the LSA will not be added. Instead only a direct acknowledgement is sent back.

4.4.8 Deleting LSA

Deleting something from a replicated distributed database is not a trivial task. Especially if there is no LS remove packet type. Removing is done via the LS age. LSA with LS age `MAX_AGE` are ready to be removed from the database. Especially for OpenOSPFD removing LSAs is even more complicated. To remove a LSA it first has to be reflooded and all neighbors have to acknowledge the reception before removing it from the database. In OpenOSPFD the database and the retransmission logic are in two different processes so additional IPC is needed. If the RDE tries to delete the LSA either because it exceeds the `MAX_AGE` age or because of premature aging – used to clean the database from no longer valid LSAs – it simply sets the age to `MAX_AGE` and sends a flood request to the OSPF engine. The OSPF engine will then start the flooding procedure. The LSA is added to the LSA cache and the different retransmission lists refer to the cached LSA. If the last reference to the cached object drops the following happens:

Code snip 32: lsa_cache_put()

```
void
lsa_cache_put(struct lsa_ref *ref, struct nbr *nbr)
{
        if (--ref->refcnt > 0)
                return;

        if (ntohs(ref->hdr.age) >= MAX_AGE)
                ospfe_imsg_compose_rde(IMSG_LS_MAXAGE,
                    nbr->peerid, 0, ref->data,
                    sizeof(struct lsa_hdr));

        free(ref->data);
        LIST_REMOVE(ref, entry);
        free(ref);
}
```

The LS age is compared with `MAX_AGE` and if true a `IMSG_LS_MAXAGE` is sent back to the RDE. In the RDE the message is received and verified. If something is incorrect the RDE bombs out.

Code snip 33: IMSG_LS_MAXAGE handling

```
case IMSG_LS_MAXAGE:
	nbr = rde_nbr_find(imsg.hdr.peerid);
	if (nbr == NULL)
		fatalx("rde_dispatch_imsg: "
		    "neighbor does not exist");

	if (imsg.hdr.len != IMSG_HEADER_SIZE +
	    sizeof(struct lsa_hdr))
		fatalx("invalid size of OE request");
	memcpy(&lsa_hdr, imsg.data, sizeof(lsa_hdr));

	if (rde_nbr_loading(nbr->area))
		break;

	v = lsa_find(nbr->area, lsa_hdr.type,
	    lsa_hdr.ls_id, lsa_hdr.adv_rtr);
	if (v == NULL)
		db_hdr = NULL;
	else
		db_hdr = &v->lsa->hdr;

	/*
	 * only delete LSA if the one in the db isn't newer
	 */
	if (lsa_newer(db_hdr, &lsa_hdr) <= 0)
		lsa_del(nbr, &lsa_hdr);
	break;
```

If there is still a neighbor in state *EXCHANGE* or *LOADING* the LSA may not be removed. It is possible that the neighbor may request that LSA just a bit later. Now the LSA is searched in the database and the entry of the database is compared with the LSA that should be removed. If the database entry is newer the entry will not be removed else it would get finally removed from the database and freed.

4.4.9 SPF and RIB calculation

The SPF calculation is still a large construction area. The code should be split up as some steps are not necessary in all cases. Especially on ABRs this is not optimal and creates a lot of superfluous load. Worth knowing: RIB and FIB are terms from BGP and got inherited into OpenOSPFD. RIB is the Routing Information Base and FIB is the Forwarding Information Base. The FIB is mostly the kernel routing table and is stripped from unneeded ballast whereas the RIB contains all additional protocol specific informations.
To calculate the routing table three calculations are performed. First the SPF tree gets built. Then the local LSAs are added to the RIB and finally the AS-external-LSAs are inserted.

Code snip 34: SPF calculation

```
/* calculate SPF tree */
do {
	/* loop links */
	for (i = 0; i < lsa_num_links(v); i++) {
		switch (v->type) {
		case LSA_TYPE_ROUTER:
			rtr_link = get_rtr_link(v, i);
			switch (rtr_link->type) {
			case LINK_TYPE_STUB_NET:
				/* skip */
				continue;
			case LINK_TYPE_POINTTOPOINT:
			case LINK_TYPE_VIRTUAL:
				/* find router LSA */
				w = lsa_find(area,
				    LSA_TYPE_ROUTER,
				    rtr_link->id,
				    rtr_link->id);
				break;
			case LINK_TYPE_TRANSIT_NET:
				/* find network LSA */
				w = lsa_find_net(area,
				    rtr_link->id);
				break;
			default:
				fatalx("spf_calc: "
				    "invalid link type");
			}
			break;
		case LSA_TYPE_NETWORK:
			net_link = get_net_link(v, i);
			/* find router LSA */
			w = lsa_find(area, LSA_TYPE_ROUTER,
			    net_link->att_rtr,
			    net_link->att_rtr);
			break;
		default:
			fatalx("spf_calc: "
			    "invalid LSA type");
		}

		...
		cand_list_add(w);
	}
	/* get next vertex */
	v = cand_list_pop();
	w = NULL;
} while (v != NULL);
```

The loops starts at the root vertex and moves through one vertex after another. After a vertex is selected all next vertices that are connected to this vertex are extracted and added to the candidate list. After all vertices are added the one with the lowest cost is popped from the list and the loops starts over with this vertex. Before a vertex is added to the candidate list it is verified that the connection is still valid.

Code snip 35: the three dots in the previous snippet

```
if (w == NULL)
	continue;

if (w->lsa->hdr.age == MAX_AGE)
	continue;

if (!linked(w, v))
	continue;

if (v->type == LSA_TYPE_ROUTER)
	d = v->cost + ntohs(rtr_link->metric);
else
	d = v->cost;

if (cand_list_present(w)) {
	if (d > w->cost)
		continue;

	if (d < w->cost) {
		w->cost = d;
		w->prev = v;
		calc_next_hop(w, v);
		/*
		 * need to readd to candidate list
		 * because the list is sorted
		 */
		TAILQ_REMOVE(&cand_list, w, cand);
	}
} else if (w->cost == LS_INFINITY && d < LS_INFINITY) {
	w->cost = d;
	w->prev = v;
	calc_next_hop(w, v);
}
```

On leaf nodes – `w` is `NULL` – there is nothing to do. If the next vertex has an age of `MAX_AGE` it is no longer considered valid and dropped. The connection between the two vertices has to be bidirectional and this is checked by

linked(). The next steps calculate the cost to the new vertex w. There is one important thing to note: only links into a network have a cost but links from the network to the router have no cost. The result is that modifying the cost of an interface will often not change incoming traffic flow only outgoing traffic may be rerouted due to the change. Before adding a vertex to the candidate list it is necessary to check if the vertex is already on the list. If it is, then the calculated cost is compared with the current one. The new path must be shorter than the current selected one. In that case the cost and the prev pointer are modified and the nexthop is recalculated. The vertex is also removed from the candidate list and later added back to keep the list sorted. If the vertex is not on the candidate list then cost and prev pointer are initialised and the nexthop is calculated. Finally the new candidate is added to the list of candidates.

Now the RIB needs to be built. To start the area specific routes are added. First of all, all LSAs with LS age MAX_AGE, a cost of LS_INFINITY, or a zero nexthop address are skipped. They are invalid. All valid network-LSAs are added to the RIB and all router-LSAs for ABRs and ASBRs are added as well. Summary-LSAs are put into the RIB. On ABRs only for area 0. On non ABRs there is no limitation. A summary-LSA is only valid if the ABR was previously added to the RIB. The last step is adding of the AS-external routes to the RIB. This is done only once and not for every area. Similarly to summary-LSAs AS-external-LSAs will do a look-up of the ASBR router and if the router is not found the route is considered invalid. When updating the RIB with rt_update() some order is retained. Intra-area routes (router and network-LSAs) have highest priority, inter-area routers (summary-LSAs) follow and Type1 and Type2 AS-external routes have the lowest priority. So if a network is added multiple times that order will favour intra-area traffic over inter-area or external routes.

4.5 Workflow

4.5.1 Flooding

The flooding and retransmission of LS updates is entirely done in the OSPF engine. The RDE sends a IMSG_LS_FLOOD imsg with the peer ID of the neighbor from which the update was initially received. The OSPF engine uses that information to flood out the LS update to all affected networks.

Code snip 36: flooding part 1

```
ref = lsa_cache_add(imsg.data, 1);

if (lsa_hdr.type == LSA_TYPE_EXTERNAL) {
        /*
         * flood on all areas but stub areas and
         * virtual links
         */
        LIST_FOREACH(area, &oeconf->area_list, entry) {
            if (area->stub)
                    continue;
            LIST_FOREACH(iface, &area->iface_list,
                entry) {
                        noack += lsa_flood(iface, nbr,
                            &lsa_hdr, imsg.data, 1);
            }
        }
} else {
        /*
         * flood on all area interfaces on
         * area 0.0.0.0 include also virtual links.
         */
        area = nbr->iface->area;
        LIST_FOREACH(iface, &area->iface_list, entry) {
                noack += lsa_flood(iface, nbr,
                    &lsa_hdr, imsg.data, 1);
        }
}
```

Before starting the flooding decision process the LS update is added to the LSA cache. Later, if the LSA is added to different retransmission queues, only a reference to the LSA cache is retained. Depending on the LS type it must be flooded to all areas (AS-external-LSA) or only to the current area (all other LSAs). lsa_flood() is doing the per interface specific part of the flooding. More about that a bit later.

Code snip 37: flooding part2

```
/* remove from ls_req_list */
le = ls_req_list_get(nbr, &lsa_hdr);
if (!(nbr->state & NBR_STA_FULL) && le != NULL) {
        ls_req_list_free(nbr, le);
        /*
         * XXX no need to ack requested lsa
         * the problem is that the RFC is very
         * unclear about this.
         */

        noack = 1;
}

if (!noack && nbr->iface != NULL &&
    nbr->iface->self != nbr) {
        if (!(nbr->iface->state & IF_STA_BACKUP) ||
            nbr->iface->dr == nbr) {
                /* delayed ack */
                lhp = lsa_hdr_new();
                memcpy(lhp, &lsa_hdr, sizeof(*lhp));
                ls_ack_list_add(nbr->iface, lhp);
        }
}

lsa_cache_put(ref, nbr);
break;
```

After flooding the LSA out on all affected interfaces an acknowledgement has to be sent back to the initial sender of the LS update. In some cases there is no requirement to send a LS acknowledge back. One of those cases are requested LSAs – sending back a LSA ack to an explicitly requested LSA does not make much sense. However the RFC is not very clear about this fact. So let's be prepared for some broken implementations out there. The last step adds the LSA to the LS acknowledge list so that a, possibly delayed, acknowledge can be sent back. This is only done if an ack is required, the neighbor where the ack is sent to is not ourselves and additionally no acks were sent from the BDR to the DR. Finally the acquired reference of the LSA gets passed back. Reference counting makes careful programming a necessity to avoid missing a reference change somewhere.

lsa_flood()

As mentioned earlier `lsa_flood()` is used for flooding on a per interface scope. In particular it loops over all neighbors and decides if it has to send the update to this neighbor or not.

Code snip 38: neighbor loop part 1

```
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
        if (nbr == iface->self)
                continue;
        if (!(nbr->state & NBR_STA_FLOOD))
                continue;
```

First of all `self` is skipped. Then all neighbors which are not available for flooding – their state is neither *FULL* nor *LOADING* nor *EXCHANGE* – are skipped as well.

Code snip 39: neighbor loop part 2

```
if (iface->state & IF_STA_DROTHER && !queued)
        if ((le = ls_retrans_list_get(iface->self,
            lsa_hdr)))
                ls_retrans_list_free(iface->self, le);

if ((le = ls_retrans_list_get(nbr, lsa_hdr)))
        ls_retrans_list_free(nbr, le);
```

Afterwards the retransmission lists are searched for an older LS update for the same LSA. If an older LSA is found it is removed and replaced later with the new one. A special queue is used for interfaces with state *DROTHER* as explained later on. Because only one queue is used, redoing this check after the LSA got queued once results in unexpected behaviour. So this case is protected by the `!queued` check.

Code snip 40: neighbor loop part 3

```
if (!(nbr->state & NBR_STA_FULL) &&
    (le = ls_req_list_get(nbr, lsa_hdr)) != NULL) {
        r = lsa_newer(lsa_hdr, le->le_lsa);
        if (r > 0) {
                /* to flood LSA is newer than requested */
                ls_req_list_free(nbr, le);
                /* new needs to be flooded */
        } else if (r < 0) {
                /* to flood LSA is older than requested */
                continue;
        } else {
                /* LSA are equal */
                ls_req_list_free(nbr, le);
                continue;
        }
}
```

If the adjacency is not yet full, the LS request list is examined. If a LSA is found we know the exact LSA the neighbor has in his database. So if the LSA in the request list is older than the new one, the requested one is removed and the new one will be flooded. Otherwise if the LSA is older than the requested one, there is no need to flood it to the neighbor and the request list is left alone so that the newer LSA of that neighbor is requested later. In case both LSAs are equal there is no need to request the LSA anymore. There is also no need to flood the LSA to that neighbor.

Code snip 41: neighbor loop part 4

```
if (nbr == originator) {
        dont_ack++;
        continue;
}

/* non DR or BDR router keep all lsa in one retrans list */
if (iface->state & IF_STA_DROTHER) {
        if (!queued)
                ls_retrans_list_add(iface->self, data);
        queued = 1;
} else {
        ls_retrans_list_add(nbr, data);
        queued = 1;
}
```

If the current neighbor is the initial sender of this LS update there is high chances that no ack has to be sent back. This decision is done later. At least there is also no need to flood the LS update back to this router.

Finally the LS update or actually a reference to the LS update is added to the retransmission queue. Depending on the interface state, different queues are chosen. If the interface is not in state *DROTHER* it will be added to the neighbor retransmission list. In case of *DROTHER* only one global queue is used because all updates go to the `AllDRouters` address. For this special case `iface->self` is "abused". Because only one queue is used it is important to protect the queue from multiple adds. Currently there is a known feature in the queuing behaviour of OpenOSPFD that needs to be solved. In case of the router being BDR it will queue the update to all neighbors on that interface including the DR. The DR therefore is required to send an acknowledge to the BDR. This will not happen and so one retransmission is done from the BDR to the DR and the DR will then answer with a direct acknowledge. This is unnecessary and no updates to the DR should be queued unless they are self originated or from a different interface.

Code snip 42: sending LS update

```
if (!queued)
        return (0);

if (iface == originator->iface &&
    iface->self != originator) {
        if (iface->dr == originator ||
            iface->bdr == originator)
                return (0);
        if (iface->state & IF_STA_BACKUP)
                return (0);
        dont_ack++;
}

/* flood LSA but first set correct destination */
switch (iface->type) {
case IF_TYPE_POINTOPOINT:
        inet_aton(AllSPFRouters, &addr);
        send_ls_update(iface, addr, data, len);
        break;
case IF_TYPE_BROADCAST:
        if (iface->state & IF_STA_DRORBDR)
                inet_aton(AllSPFRouters, &addr);
        else
                inet_aton(AllDRouters, &addr);
        send_ls_update(iface, addr, data, len);
        break;
...
}

return (dont_ack == 2);
```

After inspecting every neighbor and adding LSA references to the retransmission lists an initial flooding gets sent out. If nothing got queued there is no reason to send the LSA, do a return. In the other cases we send the update to the correct address. For point-to-point links it is always `AllSPFRouters`. For broadcast networks it is either `AllSPFRouters` or `AllDRouters` to multicast the update to the correct group. All other interface types use unicast to send the updates. Before sending out the LS update a special check is done mostly for broadcast and NBMA networks. In case the originator of the initial LS update is on the now outgoing interface more checks have to be done. First of all if the originator is DR or BDR there is no need to send an update. The actual flooding was already done by the DR respectively BDR. Additionally if the router itself is BDR there is no need to flood the network. This will be done by the DR. If none of these two tests where true it is now clear that no acknowledgement needs to be sent back. Therefore `dont_ack` is bumped a second time and so `lsa_flood()` will return true.

4.5.2 Retransmission Lists and LSA Cache

Now lets have a look at the retransmission lists. All other lists – acknowledge, request, and database descriptor list – are implemented in a similar way. The retransmission list is a bit more complex because of the LSA cache. To add a LS update to the request list `ls_retrans_list_add()` is used.

Code snip 43: ls_retrans_list_add()

```
if ((ref = lsa_cache_get(lsa)) == NULL)
        fatalx("King Bula sez: somebody forgot to
lsa_cache_add");

if ((le = calloc(1, sizeof(*le))) == NULL)
        fatal("ls_retrans_list_add");

le->le_ref = ref;
TAILQ_INSERT_TAIL(&nbr->ls_retrans_list, le, entry);

if (!evtimer_pending(&nbr->ls_retrans_timer, NULL)) {
        timerclear(&tv);
        tv.tv_sec = nbr->iface->rxmt_interval;

        if (evtimer_add(&nbr->ls_retrans_timer, &tv) == -1)
                log_warn("ls_retrans_list_add: evtimer_add
failed");
}
```

First of all a LSA cache reference is acquired via `lsa_cache_get()`. If this call fails we have an internal program error and the OSPF engine has no way to recover from that. The reference is added to a list element that in turn is added to the retransmission list. And if there is no timer pending a new retransmission timer is started.

Removing works in a similar way. First the correct entry is searched with the help of `ls_retrans_list_get()` and afterwards it gets freed if the LSA was the same. `ls_retrans_list_get()` uses the known LSA triple to identify a LSA.

Code snip 44: ls_retrans_list_free()

```
void
ls_retrans_list_free(struct nbr *nbr, struct lsa_entry *le)
{
        TAILQ_REMOVE(&nbr->ls_retrans_list, le, entry);

        lsa_cache_put(le->le_ref, nbr);
        free(le);
}
```

`ls_retrans_list_free()` will not only unlink the LSA from the request list but hands the LSA cache reference back by calling `lsa_cache_put()`. Again it is important to take care of those references.

How does this LSA cache work?

The LSA cache is nothing more than a hash list. A simple hash is built over the LSA header and used to find the correct hash bucket. In the LSA cache a LSA is identified not only by LS type, LS ID, and advertising router. The sequence number and LS checksum is compared as well. To find a LSA in the cache the internal `lsa_cache_look()` function is used.

`lsa_cache_get()` returns a new reference to an existing LSA.

Code snip 45: lsa_cache_get()

```
struct lsa_ref *
lsa_cache_get(struct lsa_hdr *lsa_hdr)
{
        struct lsa_ref  *ref;

        ref = lsa_cache_look(lsa_hdr);
        if (ref)
                ref->refcnt++;

        return (ref);
}
```

This function is very simple and the only important step is not to forget to bump the reference count.

`lsa_cache_add()` works very similar to `lsa_cache_get()`. Again `lsa_cache_look()` is used to find already added LSAs. In that case a bump of the reference count is enough. Else a new reference object gets allocated and filled in. There is a timestamp included to age the LSA when it is sent out. The initial reference count is set to one because a reference is immediately returned to the caller.

Code snip 46: lsa_cache_add()

```
struct lsa_ref *
lsa_cache_add(void *data, u_int16_t len)
{
        struct lsa_cache_head*head;
        struct lsa_ref  *ref, *old;

        if ((ref = calloc(1, sizeof(*ref))) == NULL)
                fatal("lsa_cache_add");
        memcpy(&ref->hdr, data, sizeof(ref->hdr));

        if ((old = lsa_cache_look(&ref->hdr))) {
                free(ref);
                old->refcnt++;
                return (old);
        }

        if ((ref->data = malloc(len)) == NULL)
                fatal("lsa_cache_add");
        memcpy(ref->data, data, len);
        ref->stamp = time(NULL);
        ref->len = len;
        ref->refcnt = 1;
```

```
        head = lsa_cache_hash(&ref->hdr);
        LIST_INSERT_HEAD(head, ref, entry);
        return (ref);
}
```

lsa_cache_put() was only roughly explained in the MAX_AGE handling. First the reference count is decreased and if it hits zero the cache is no longer referenced and can be freed. Now the known MAX_AGE dance comes. Sending back an IMSG_LS_MAXAGE if the LSA has an age of MAX_AGE to make it possible to remove the LSA from the LS DB. Afterwards the cache object is cleaned and removed.

Code snip 47: lsa_cache_put()

```
void
lsa_cache_put(struct lsa_ref *ref, struct nbr *nbr)
{
        if (--ref->refcnt > 0)
                return;

        if (ntohs(ref->hdr.age) >= MAX_AGE)
                ospfe_imsg_compose_rde(IMSG_LS_MAXAGE,
                    nbr->peerid, 0, ref->data,
                    sizeof(struct lsa_hdr));

        free(ref->data);
        LIST_REMOVE(ref, entry);
        free(ref);
}
```

4.5.3 Self originated LSA

There are three kinds of self originated LSAs. First router and network-LSAs – those are generated in the OSPF engine. Then AS-external-LSAs which are generated in the RDE with the help of the parent process. Finally on ABRs summary-LSAs are generated – this happens in the RDE as well.
To create a self originated LSA in the OSPF engine and commit it to the LS DB in the RDE is a bit tricky. Let's have a look at orig_net_lsa() because it is a lot simpler than orig_rtr_lsa().

Code snip 48: originate network-LSA

```
if ((buf = buf_dynamic(sizeof(lsa_hdr), READ_BUF_SIZE)) ==
    NULL)
        fatal("orig_net_lsa");

/* reserve space for LSA header and LSA Router header */
if (buf_reserve(buf, sizeof(lsa_hdr)) == NULL)
        fatal("orig_net_lsa: buf_reserve failed");

/* LSA net mask and then all fully adjacent routers */
if (buf_add(buf, &iface->mask, sizeof(iface->mask)))
        fatal("orig_net_lsa: buf_add failed");

/* fully adjacent neighbors + self */
LIST_FOREACH(nbr, &iface->nbr_list, entry)
        if (nbr->state & NBR_STA_FULL) {
                if (buf_add(buf, &nbr->id,
                    sizeof(nbr->id)))
                        fatal("orig_net_lsa: "
                            "buf_add failed");
                num_rtr++;
        }

if (num_rtr == 1) {
        /*
         * non transit net therefor no need to generate
         * a net lsa
         */
        buf_free(buf);
        return;
}

/* LSA header */
if (iface->state & IF_STA_DR)
        lsa_hdr.age = htons(DEFAULT_AGE);
else
        lsa_hdr.age = htons(MAX_AGE);

lsa_hdr.opts = oeconf->options;/* XXX */
lsa_hdr.type = LSA_TYPE_NETWORK;
lsa_hdr.ls_id = iface->addr.s_addr;
lsa_hdr.adv_rtr = oeconf->rtr_id.s_addr;
lsa_hdr.seq_num = htonl(INIT_SEQ_NUM);
lsa_hdr.len = htons(buf->wpos);
lsa_hdr.ls_chksum = 0;/* updated later */
memcpy(buf_seek(buf, 0, sizeof(lsa_hdr)), &lsa_hdr,
    sizeof(lsa_hdr));

chksum = htons(iso_cksum(buf->buf, buf->wpos,
    LS_CKSUM_OFFSET));
memcpy(buf_seek(buf, LS_CKSUM_OFFSET, sizeof(chksum)),
    &chksum, sizeof(chksum));

imsg_compose(ibuf_rde, IMSG_LS_UPD, iface->self->peerid, 0,
    -1, buf->buf, buf->wpos);

buf_free(buf);
```

Once again the buf API is used. First space for the header is reserved then the network mask is added and finally a list of all fully adjacent routers is added. The router itself needs to be added as well but this is no problem because of the special self neighbor. If there is no other OSPF router on the network it is not necessary to create a network-LSA. A stub network entry in the router-LSA will do the job. In that case the buffer gets freed and the function returns. Otherwise the LSA header has to be built. First the correct age is set. To remove a network-LSA the age is set to MAX_AGE else the initial DEFAULT_AGE is used. Other important fields are LS type, LS ID and advertising router. Also the sequence number has to be set but the correct instance number is only known by the RDE. The RDE uses lsa_merge() later on to merge this LSA into the database and lsa_merge() will take care of the sequence number – so here we set it just to the initial value. Copy the header into the buffer, calculate the checksum and finally send this self originated LSA with the peerid of the special neighbor self to the RDE.
Originating a router-LSA is done in a similar way. It is just more complex because many additional informations are added in the router-LSA. One tricky part is setting the correct router flags.

Code snip 49: originate router-LSA

```
/* LSA router header */
lsa_rtr.flags = 0;
/*
 * Set the E bit as soon as an as-ext lsa may be
 * redistributed, only setting it in case we redistribute
 * something is not worth the fuss.
 */
if (oeconf->redistribute_flags &&
    (oeconf->options & OSPF_OPTION_E))
        lsa_rtr.flags |= OSPF_RTR_E;

border = area_border_router(oeconf);

if (border != oeconf->border) {
        oeconf->border = border;
        orig_rtr_lsa_all(area);
}

if (oeconf->border)
        lsa_rtr.flags |= OSPF_RTR_B;
if (virtual)
        lsa_rtr.flags |= OSPF_RTR_V;
```

There are three bits that have to be set. The *E* bit indicates that the router is an AS border router and will announce AS-external routes. The *E* bit is used in the SPF calculation and for summary-LSAs. In the SPF calculation routers with *E* bit set are added to the RIB. Without setting the *E* bit all AS-external routes using this router as advertising router are considered invalid because the router is not present in the RIB. Similar happens for summary-LSAs. On ABRs router summary-LSAs will be generated for every router with *E* bit set. OpenOSPFD tricks a bit with the *E* bit by setting the bit as soon as it is possible that a AS-external route is redistributed and not when the router actually redistributes a route. Other implementations have the same sloppy behaviour. Even more complex is setting the *B* bit, which is used to mark ABRs. As soon as a router is part of two active areas the *B* bit has to be set on all router-LSA. `area_border_router()` returns true if there are two or more active areas. If the state of the ABR changes all self originated router-LSAs in all areas have to be updated. This is done via `orig_rtr_lsa_all()` which in turn calls `orig_rtr_lsa()` for all areas but the current one. Afterwards setting the *B* bit is no longer a problem. The last bit that can be set is the *V* bit. It is used to mark interfaces where a virtual link is terminated. Areas where one router has a *V* bit set are transit areas. Transit areas need some special handling in the SPF calculation as example it is not allowed to send aggregated summary routing information into a transit area.

4.5.4 ABR and summary-LSA

The code handling ABRs and summary-LSAs is still in some flux. There are to many work a rounds and some stuff is still missing. Lets have a look at it anyway. It actually starts in the SPF calculation. The code that recalculates the RIB looks currently like this:

Code snip 50: SPF timer

```
rt_invalidate();

LIST_FOREACH(area, &conf->area_list, entry)
        spf_calc(area);

RB_FOREACH(r, rt_tree, &rt) {
        LIST_FOREACH(area, &conf->area_list, entry)
                rde_summary_update(r, area);

        if (r->d_type != DT_NET)
                continue;

        if (r->invalid)
                rde_send_delete_kroute(r);
        else
                rde_send_change_kroute(r);
}

LIST_FOREACH(area, &conf->area_list, entry)
        lsa_remove_invalid_sums(area);

start_spf_holdtimer(conf);
```

First the RIB is invalidated by flagging routes as invalid. While doing that old invalid routes are removed from the tree. Afterwards the SPF calculation is run for every area. This is one of the things that should be changed. There is no need to recalculate an area if there was no changes in that area. In the next step a walk over the RIB is done. By calling `rde_summary_update()` for every area and any route all required summary informations are generated. Afterwards the kernel routing table is updated by sending change or delete messages to the parent process. This is only done for routes that describe networks. After that old invalid summary-LSAs get removed from all areas. Finally the hold timer is started. This is specified in the RFC so that the SPF calculation does not kill the underpowered routers.
`rde_summary_update()` does the decision if it necessary to create a summary-LSA.

Code snip 51: Is summary-LSA needed?

```
/* first check if we actually need to announce this route
*/
if (!(rte->d_type == DT_NET || rte->flags & OSPF_RTR_E))
        return;
/* never create summaries for as-ext LSA */
if (rte->p_type == PT_TYPE1_EXT || rte->p_type ==
PT_TYPE2_EXT)
        return;
/* no need for summary LSA in the originating area */
if (rte->area.s_addr == area->id.s_addr)
        return;
/* TODO nexthop check, nexthop part of area -> no summary
*/
if (rte->cost >= LS_INFINITY)
        return;
/* TODO AS border router specific checks */
/* TODO inter-area network route stuff */
/* TODO intra-area stuff -- condense LSA ??? */
```

First of all only network routes or router routes where the *E* bit is set are summarised into other areas. The *E* bit is the same as the one in router-LSAs specifying that the router is an ASBR. An ASBR has to be added to other areas so that they can validate the AS-external-LSAs. As AS-external routes are flooded through all areas there is no need to create summaries for those networks. The originating area and all invalid routes are skipped. Finally there are some other minor but very complicated things left out for now.

Code snip 52: update summary-LSA

```
/* update lsa but only if it was changed */
if (rte->d_type == DT_NET) {
        type = LSA_TYPE_SUM_NETWORK;
        v = lsa_find(area, type, rte->prefix.s_addr,
            rde_router_id());
} else if (rte->d_type == DT_RTR) {
        type = LSA_TYPE_SUM_ROUTER;
        v = lsa_find(area, type, rte->adv_rtr.s_addr,
            rde_router_id());
} else
        fatalx("orig_sum_lsa: unknown route type");

lsa = orig_sum_lsa(rte, type);
lsa_merge(rde_nbr_self(area), lsa, v);

if (v == NULL) {
        if (rte->d_type == DT_NET)
                v = lsa_find(area, type,
                    rte->prefix.s_addr, rde_router_id());
        else
                v = lsa_find(area, type,
                    rte->adv_rtr.s_addr, rde_router_id());
}
v->cost = rte->cost;
```

To update the LS DB `lsa_merge()` is used. Before it is possible to call `lsa_merge()` two things have to be done. First the current database version of the LSA has to be found. Secondly a new LSA is generated by `orig_sum_lsa()`. After merging the LSA it is necessary

to update the cost of the vertex so that a later call to `lsa_remove_invalid_sums()` sees that this vertex is still in use. In case the LSA was newly added the previous `lsa_find()` returned `NULL` so the search has to be repeated to get a valid vertex.
`lsa_remove_invalid_sums()` does nothing more than a tree walk looking for summary-LSAs with a cost of `LS_INFINITY` and removes those by setting their age to `MAX_AGE` and calling `lsa_timeout()` to flood them out.

4.5.5 Originating AS-external-LSA

To redistribute AS-external-LSA the parent process sends a list of candidates to the RDE. The RDE uses `rde_asext_get()` to convert the kroute into a LSA and with the help of `lsa_find()` and `lsa_merge()` the LSA is added to the database. Similarly on remove `rde_asext_put()` is used to get the no longer needed LSA and again `lsa_find()` and `lsa_merge()` do the actual job.
`rde_asext_put()` has a more or less simple job. Find the kroute, remove it from the list and create a LSA with LS age `MAX_AGE` if the LSA was used.

Code snip 53: rde_asext_put()

```
LIST_FOREACH(ae, &rde_asext_list, entry)
        if (kr->prefix.s_addr == ae->kr.prefix.s_addr &&
            kr->prefixlen == ae->kr.prefixlen) {
                LIST_REMOVE(ae, entry);
                used = ae->used;
                free(ae);
                if (used)
                        return (orig_asext_lsa(kr,
                            MAX_AGE));
                break;
        }
return (NULL);
```

On the other hand `rde_asext_get()` has a bit more to do. It first looks if the route was added already before. In that case the route needs to be updated, else a new one is created.

Code snip 54: rde_asext_get() part 1

```
LIST_FOREACH(ae, &rde_asext_list, entry)
        if (kr->prefix.s_addr == ae->kr.prefix.s_addr &&
            kr->prefixlen == ae->kr.prefixlen)
                break;

if (ae == NULL) {
        if ((ae = calloc(1, sizeof(*ae))) == NULL)
                fatal("rde_asext_get");
        LIST_INSERT_HEAD(&rde_asext_list, ae, entry);
}

memcpy(&ae->kr, kr, sizeof(ae->kr));

wasused = ae->used;
ae->used = rde_redistribute(kr);
```

Next task is to find out if the route should be redistributed. The actual logic is in `rde_redistribute()` and so lets have a look at that.

Code snip 55: rde_redistribute()

```
int
rde_redistribute(struct kroute *kr)
{
        struct area*area;
        struct iface*iface;
        int             rv = 0;

        if (!(kr->flags & F_KERNEL))
                return (0);

        if ((rdeconf->options & OSPF_OPTION_E) == 0)
                return (0);

        if ((rdeconf->redistribute_flags &
            REDISTRIBUTE_DEFAULT) &&
            (kr->prefix.s_addr == INADDR_ANY &&
            kr->prefixlen == 0))
                return (1);

        /* only allow 0.0.0.0/0 if REDISTRIBUTE_DEFAULT */
        if (kr->prefix.s_addr == INADDR_ANY &&
            kr->prefixlen == 0)
                return (0);

        if ((rdeconf->redistribute_flags &
            REDISTRIBUTE_STATIC) &&
            (kr->flags & F_STATIC))
                rv = 1;
        if ((rdeconf->redistribute_flags &
            REDISTRIBUTE_CONNECTED) &&
            (kr->flags & F_CONNECTED))
                rv = 1;

        /*
         * interface is not up and running so don't
         * announce
         */
        if (kif_validate(kr->ifindex) == 0)
                return (0);

        LIST_FOREACH(area, &rdeconf->area_list, entry)
                LIST_FOREACH(iface, &area->iface_list,
                    entry) {
                        if ((iface->addr.s_addr &
                            iface->mask.s_addr) ==
                            kr->prefix.s_addr &&
                            iface->mask.s_addr ==
                            prefixlen2mask(kr->prefixlen))
                                /* already announced
                                 * as net LSA */
                                rv = 0;
                }

        return (rv);
}
```

First it is checked if we have to redistribute anything. Afterwards the default route gets handled. The default route is only redistributed if explicitly enforced via "*redistribute default*". Dependent on the flags it is now decided if routes gets redistributed. The interface state is checked and finally all configured interfaces are inspected to see if the route is not already part of a network-LSA or is announced as a stub network.
After the `rde_redistribute()` call it is now clear what remains to be done.

Code snip 56: rde_asext_get() part 2

```
if (ae->used)
        /* update of seqnum is done by lsa_merge */
        return (orig_asext_lsa(kr, DEFAULT_AGE));
else if (wasused)
        /*
         * lsa_merge will take care of removing the
         * lsa from the db
         */
        return (orig_asext_lsa(kr, MAX_AGE));
else
        /* not in lsdb, superseded by a net lsa */
        return (NULL);
```

If the route has to be redistributed a LSA with the initial LS age is generated and returned. If it is no longer used a LSA with LS age `MAX_AGE` is generated and returned.

Otherwise the work is completed and function returns.
In case an interface state changes, `rde_update_redistribute()` is called and all routes that depend on this interface are recalculated very similar to the presented code here. Again going through `rde_redistribute()`, `orig_asext_lsa()`, `lsa_find()`, and `lsa_merge()`.

4.6 Issues and other stuff

There are still some problems in OpenOSPFD that have to be solved. Some features are incomplete and so there is still a lot of work to be done. Lets look back at the solved problems. The first problem encountered was probably the privilege separation because a clever splitting had to be done. This is still sometimes an issue – for example the current redistribute code is partially done in the wrong place. The result is massive overhead if the router does "*redistribute static*" with a full view in the routing table. All ~170'000 routes are passed to the RDE and evaluated there. It works but is inefficient. Other problems with privsep were solved like the `MAX_AGE` or the database exchange problems explained earlier. A good example of a work a round is the multicast handling. A real fix for this problem is in progress but some kernel patches are required to make it fly. At least many issues and bugs were identified and fixed in the flooding and database exchange phase – the most important part of the protocol.
Things that remain to be fixed include the redistribute code or the missing support for interface aliases. The ABR code is still not optimal and is not as good tested as the normal case. Virtual links still need a lot of work to get them flying – a lot of code is around but some important bits are missing. Interface handling should be improved, like supporting aliases and dynamic interfaces. Last but not least there are all those supercool new features planned but that's a different paper. :)

Bibliography

[1] Moy, J. *OSPF version 2*. RFC 2328, April 1998.

[2] Moy, J. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, September 1998

[3] OpenBSD, *http://www.openbsd.org/*

[4] OpenBGPD, *http://www.openbgpd.org/*

[5] OpenOSPFD source code, *http://www.openbsd.org/cgi-bin/cvsweb/src/usr.sbin/ospfd/*

Remote user access VPN with IPsec

Emmanuel Dreyfus

October 24, 2005

Abstract

IPsec is a set of Internet Protocol (IP) extensions used to bring secure communication to the network level. IPsec can be used in various Virtual Private Network (VPN) scenarios such as bridging private networks or user remote access to a private network.

Remote User access VPN is an area where the the IPsec tools available on BSD systems were a bit frustrating. In this paper, we describe some requirements for a remote user access solution and how the existing solutions based on IPsec did not fully satisfy the requirements.

We then have a look to the IPsec extensions implemented by other vendors and how they would match our goals if we had them. The end of the paper tells how these extensions have been added to NetBSD IPsec stack, and how it led NetBSD to integrate software from the ipsec-tools project.

It is assumed that the reader is knowledgeable with TCP/IP networking.

1 Virtual Private Networks

A Virtual Private Network (VPN) is a link between two private networks, which are usually connected through the Internet.

The link is secured so that no one can easily eavesdrop or alter the traffic. This is why the VPN is said to be private. It also maintains the illusion of a single private network, without the Internet in between, and this is why we speak about a virtual network.

Maintaining the security of the VPN is not trivial. Of course the network traffic must be encrypted and checked against modifications, but the endpoints must also authenticate each other. If this mutual authentication is not done, the VPN is left vulnerable to Man in the Middle (MiM) attacks, where an attacker will impersonate each VPN endpoint and will be able to tamper with the network traffic.

The VPN can be a set of links between various sites of the same enterprise. In this situation the network administrator only has to deal with the secure communication between the border VPN gateways. This situation is quite comfortable, since it only deals with mutually authenticating machines. Many tools

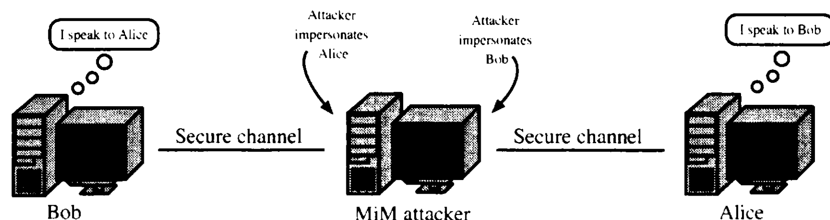


Figure 1: MiM attack

are available in BSD systems to get the work done.

The remote user access VPN is another scenario where one of the VPN end-points is not a network but a single machine. It can be a road warrior accessing the private network from home or from a conference. In this situation, the network administrator is not likely to be the mobile machine administrator, so this is not just a problem of authenticating a machine – we need to authenticate the remote user instead.

2 Requirements for a remote user access VPN

Let us define the requirement we could have for a remote user access VPN. We want our remote user access VPN to

- be secure
- use login and passwords for user authentication
- be as simple as possible for users to configure
- use free software on the server
- be compatible with as many client Operating Systems (OSes) as possible

Here is a deeper review of the requirements.

2.1 Security

We want a secure VPN. We do not want attackers to eavesdrop or modify the traffic, and we do not want an unauthorized user to gain access to our private network.

2.2 Login and password authentication

There are a few ways of authenticating users. The weakest way is using a group password, one which is shared by all users. This is lower security because a shared password does not remain secret very long, as the administrator has no way to know who disclosed it if it gets disclosed.

Login and password is a bit better, because at least they are personal. We get a better idea of who is doing what. This is not highest security because passwords can be guessed or cracked.

A better user authentication is to use digital certificates. This is the highest security, but it may be impossible to manage in the real world. Certificate enrollment is not a trivial task and users may not be knowledgeable enough to manage their digital certificates.

Depending on the situation, login and passwords may be the best balance between security and usability. We will assume this is the situation here. We target a security level equivalent to SSHv2 (Secure SHell version 2) using password authentication: passwords cannot be eavesdropped, but they could be guessed, so we need to regularly check that user passwords are strong enough.

2.3 User friendliness

We want a solution as easy as possible to manage for users. In an ideal world, the user would only have to configure a VPN gateway IP, a login and a password, and the VPN connection should be made.

2.4 Free software on the server

For a lot of good reasons, we have a strong bias for using free software. It is easier to debug, it can be more easily enhanced, and it is free as in beer. So we would like our VPN gateway to run on a free OS.

2.5 Compatible with as many client systems as possible

We cannot rule out proprietary software on the mobile host, as we assumed that the network administrator was not managing it. So our solution must be compatible with Windows systems, for instance. But we also want to avoid locking out users of free OSes.

3 A short survey of remote user access VPN techniques

3.1 PPP over SSH

Point to Point Protocol (PPP) over SSH is about running a PPP session over an SSH tunnel. Security is handled by SSH, and PPP allows us to build a virtual network.

This solution meets the security and login authentication requirement. It is easy to implement on a BSD system as all the tools are available in the base system. It still has major drawbacks.

The first drawback is performance. The network stack when a web page is fetched through this kind of VPN would be

HTTP
TCP
IP
PPP
SSH
TCP
IP
layer 2

We can see that Transfer Control Protocol (TCP) is used twice in the stack. The two TCP layers will fight each other when trying to deal with network bandwidth, resulting in very poor performances.

Second, it is not easy at all to configure for a client running a Unix-like system, and third, if it is even possible at all, it would be very difficult to configure for a client running a Windows system.

So PPP over SSH looks like a quick and dirty solution for knowledgeable user, but it does not fulfill our requirements.

3.2 PPTP

Point to Point Tunneling Protocol (PPTP) is a VPN protocol designed by Microsoft. Built-in support for it has been available since Windows NT 4.0, and a few free-software implementation are available for Unix-like systems. Apple also provides a built-in PPTP capability since MacOS X.3.

PPTP has a long and scary history of security flaws [PPTP], and Microsoft seems to be adopting Layer 2 Tunneling Protocol (L2TP) over IPsec now. PPTP does not appear to be a good candidate to build a new VPN solution today.

3.3 Tunneling over SSL

Secure Socket Layer (SSL) is a security layer used on the top of TCP to secure application layer traffic. It uses digital certificate for mutual authentication and has provisions for the situation where only the server side uses a certificate. In that situation, a secure communication can take place where the client knows it speaks to the server, and the server does not know who it is speaking with.

At that stage, the client can authenticate itself through the secure channel. This may be done with login and passwords, and the password cannot be eavesdropped easily.

OpenVPN [OpenVPN] is a free software project that proposes a VPN solution using tunneling over SSL. It supports authentication using login and passwords, while the server uses a certificate. This meets our security objective. It is reasonably easy to use and even has a graphical user interface for Windows and MacOS X. It also runs on a lot of different Operating Systems.

Finally, OpenVPN uses User Datagram Protocol (UDP) as its transport layer, so it does not suffers the performance issue we described for PPP over SSH.

OpenVPN fulfills our requirements, but we would like some alternatives. Let us look in the direction of IPsec.

3.4 Plain IPsec

As we said before, IPsec is a set of IP extensions to bring security to the network layer. On the network protocol front, here is what it introduces:

- An Authentication Header (AH) for IP packets. This IP option is used to authenticate the host that sent an IP packet and to guarantee the data and IP header integrity.
- the Encapsulating Security Payload (ESP). This is a layer 4 protocol to carry encrypted data. The sending host is authenticated and the data integrity is guaranteed. ESP can be used in transport mode, where it encapsulates TCP or UDP, or in tunnel mode, where it encapsulates IP. For a VPN setup, the tunnel mode is used in order to build a virtual network. Because ESP in tunnel mode guarantee the integrity of the inner IP header, IPsec VPN do not need AH.
- The IP compression protocol (IPcomp). This is another layer 4 protocol used to reduce the overload of encryption.

In order to do their cryptographic job, ESP and AH need a shared key on the sending and receiving host. This key, and the various informations needed to use it (algorithms, key length...) are known as an IPsec Security Association (SA). It is stored inside the kernel, in the Security Association DataBase (SAD). The SAD can be set up manually (using the setkey(8) command for instance), or key exchange can be handled by a userland daemon. In that situation, the who hosts willing to establish an IPsec SA needs to have key exchange daemons that speak the same protocol. Today's protocol for IPsec key exchange is known as Internet Key Exchange (IKE) protocol, and it is defined in RFC 2409 [IKE].

IKE defines two phases. In phase 1, the peers authenticate each other and set up a phase 1 Security Association (also known as an ISAKMP SA). The phase 1 SA is a shared key stored in memory by the IKE daemons. It is used to periodically run a phase 2, which uses the phase 1 SA to produce IPsec keys (also known as IPsec SA or phase 2 SA). This two-layer system enables periodical re-keying, where the IPsec SA can be periodically renewed in order to make the attacker's job more difficult.

Static keys may seem easier to manage but if we use them, we lose the re-keying feature. It is much better to use IKE, and this is what we are going to do. Let us now focus on IKE phase 1 authentication.

IKE phase 1 authentication can be done by two ways: shared secret or digital certificate. Both peers have to use the same method. We said that we did not want user digital certificates, and a shared secret is not the same thing as a login/password pair. The lack of user credential management tools seems to always drive network administrators to use a group password as the

shared secret. The real problem is that IKE phase 1 was mostly designed to authenticate hosts and not users.

An additional problem is that the configuration on the client side is not easy, as proper routing entries shall be created to get the private traffic going through the ESP tunnel.

For those reasons, plain IPsec does not seem to meet our goals.

3.5 L2TP over IPsec

Layer 2 Tunneling Protocol (L2TP) provides the ability to perform a user authentication through a login/password, and it is able to carry several point to point links on the top of ESP in transport mode. The multiple links can be used to carry a multi-protocol VPN, doing both IP and AppleTalk, for instance. L2TP is defined by RFC 3931 [L2TP].

L2TP over IPsec has a login/password authentication, and we have built-in clients in Windows XP and MacOS X.3. It looks like a nice solution to our problem but a closer look will show it is not.

The problem is that the L2TP completely relies on IPsec for securing the user authentication. If the IPsec SA is insecure, then authentication credentials may be stolen by an attacker. And the IPsec SA is secure only if the IKE phase 1 SA is secure. And the IKE phase 1 SA is secured by an authentication which can only use a shared secret or certificates.

Therefore even if L2TP has a user authentication using a login/password, it can only be made secure using certificates for users. This is not what we want.

4 Some Ipsec extensions

The temporary conclusion to our remote user access VPN survey is that there is no IPsec solution that meet our requirements. Let us see what other vendors have done to improve IPsec so that it can get the job done.

4.1 Xauth

Xauth (which has nothing to do with X Window Xauth) is an IKE extension that introduces a user authentication step between IKE phase 1 and IKE phase 2. The user authentication can be done through a login and a password. This solution has exactly the same drawback as L2TP over IPsec: the user authentication is secured by IKE phase 1, and IKE phase 1 can only be secured by a shared secret or certificates.

It is worth noting that a lot of vendors recommend using Xauth (or L2TP over IPsec) with a group password as the IKE phase 1 shared key because this is easier to manage. Such a setup leads to a weak security: anyone that knows the shared secret can eavesdrop other users' traffic and steal their user credentials.

Xauth adds a user authentication to IKE, but it does it in a way where a user certificate is still required in order to have a decent level of security. That is not very helpful.

Xauth is documented in a dead IETF draft [Xauth], but it is used in various VPN solutions, such as Cisco VPN.

4.2 Hybrid authentication

Hybrid authentication is another IKE extension that makes the phase 1 asymmetric: the VPN gateway authenticate to the mobile host by using a certificate, and the mobile host does not authenticate in phase 1.

At the end of phase 1 we get a secure channel where the VPN gateway does not know who it is speaking with, and the mobile host knows it speaks with the VPN gateway. In this secured channel, a user authentication through Xauth can safely take place.

Hybrid authentication with Xauth gives us the security level of SSHv2 with passwords, and this is what we were looking for. We only have to manage a server certificate, which is easy to do, and users will authenticate using login and passwords: no user certificate, no group password.

Hybrid authentication is documented in an IETF draft [Hybrid].

4.3 ISAKMP mode config

ISAKMP mode config is another IKE extension used by a mobile host to pull the network configuration from the VPN gateway. It can also be used by the VPN gateway to push the network configuration. This extension makes the user's life much more easier, as it enable VPN auto-configuration.

Without ISAKMP mode config, the peers must agree some way on the private addresses to use for tunneling. This leaves no other choice than manual IP configuration for each VPN user, which is not very convenient.

ISAKMP mode config mechanism is also used by Xauth for requesting and submitting the user credentials.

ISAKMP mode config is documented in an IETF draft [mode-cfg].

4.4 Nat-Traversal

IPsec VPNs are based on ESP, and ESP has trouble going through firewalls and Network Address Translators (NAT). There are two problems: first most network administrators block any traffic and allow only things they know about. ESP is not widespread enough to be allowed everywhere, and in fact it is blocked nearly anywhere.

The other problem is that ESP has no ports like TCP and UDP, which makes it difficult to handle for a network address translator. Most NAT will just not let it get through.

The solution to this is NAT-Traversal, a set IPsec extensions used to encapsulate ESP in UDP datagrams.

There have been many NAT-Traversal drafts, but the final RFC works that way: IKE starts on standard UDP port 500. After the first exchange, port 4500 is used. After IKE phase 2 is done, the ESP packets are encapsulated in UDP

packets. The UDP ports used for IKE are used for ESP over UDP. This ensures that NAT state installed by the IKE exchange can reused by ESP over UDP.

NAT-Traversal also require that keep-alive packets gets transmitted regularly to avoid NAT state timeout.

NAT-Traversal is defined by RFC 3947 [NAT-T] and RFC 3948 [ESP-over-UDP]. Microsoft claimed to hold a US patent on it [MS-IPR], but it is not clear that NAT-Traversal is really encumbered.

4.5 Dead Peer Detection (DPD)

Remote users accessing through a VPN may experience dangling connections, and it is important to detect when a remote user gets disconnected. Unfortunately, IPsec has no good built-in mechanism to discover that the remote host crashed or has gone off-line. Such an event is only detected at re-keying time. If phase 2 cannot complete, the IPsec SA gets killed. But it is not convenient to use a very short phase 2 lifetime just to detect dead peers.

DPD is yet another IKE extension used to monitor the peer and quickly detect when it gets unreachable. It works by exchanging probe packets, and if the peer does not answer for some time, the security associations are killed.

DPD is documented by RFC 3706 [DPD].

4.6 IKE fragmentation

Many network appliances such as DSL routers or home firewalls consider that UDP is only for DNS and NTP. Big UDP packets are not expected and are not allowed to pass through. This is a problem for our VPN since IKE tend to produce big UDP packets.

Of course we could use IP fragmentation to send smaller packets, but unfortunately some broken appliance will also consider UDP fragment as an evil thing that must be blocked.

IKE fragmentation is one more IKE extension used to split a big IKE packets into smaller fragments, so that they can pass through any network appliance.

IKE fragmentation seems to be a proprietary extension from Cisco and it seems it is not documented anywhere.

4.7 ESP fragmentation

ESP over UDP suffers exactly the same problem as IKE with network appliances. This can be fixed without any IPsec extensions, by using a simple trick. For our VPN, we use ESP in tunnel mode. The packets look this way on the wire:

```
MAC header:IP:UDP:ESP:IP:TCP:HTTP
```

And fragmented packets look like this:

```
MAC header:frag(IP:UDP:ESP:IP:TCP:HTTP)
MAC header:frag(IP:UDP:ESP:IP:TCP:HTTP)
```

We know that some network appliance will block big UDP packets or fragmented UDP packets. The idea is to fragment packets at the IP level before ESP encapsulation takes place. The fragmented packets look like this on the wire:

```
MAC header:IP:UDP:ESP:frag(IP:TCP:HTTP)
MAC header:IP:UDP:ESP:frag(IP:TCP:HTTP)
```

Because the fragmentation takes place inside ESP payload, network devices in between the mobile host and the VPN gateway have no way to see that the packet was fragmented.

ESP fragmentation ensures that packets of any size can be sent through the VPN. There might still be some problems with TCP being unable to perform Path Maximum Transmission Unit (PMTU). This is addressed by using a well known fix known as Maximum Segment Size (MSS) clamping.

5 Implementing what we need

With all these new features, we get closer to having what we are looking for. Using hybrid auth, we get our login and password authentication, and it is as secure as SSHv2 using login and passwords. Using ISAKMP mode config and DPD, we get something easy to use for the end-user. NAT-Traversal, IKE fragmentation and ESP fragmentation make the VPN usable when users connect from behind a NAT.

But now we want some real software. On the client side, Cisco makes Cisco VPN client, which implement all the nice IPsec extension we talked about. It works on Windows and MacOS X, and Cisco gives it for free if you buy a Cisco VPN 3000, a network device that does the VPN server side.

The Cisco VPN 3000 configuration is something really heavyweight that relies on a web interface or text-base menus. We said we wanted a VPN gateway running free software for various reasons (which may also include not using the VPN 3000 bloated web interface anymore). So let us now have a look on how NetBSD was enhanced to replace a Cisco VPN 3000.

5.1 The KAME project's IPsec stack

NetBSD and FreeBSD IPsec stacks were obtained from the KAME project. KAME's goal was to provide an IPv6 stack for BSD systems. Because IPv6 contains IPsec, KAME also provided IPsec.

The IPsec stack is split into a few components:

- incoming and outgoing packet handing in the kernel
- key management in the kernel.
- an IKE daemon in userland called racoon
- the setkey command used to manipulate SAD and SPD manually

- the libipsec library, which is a layer between racoon and the kernel. The userland/kernel key management interface is done through special sockets of type PF_KEY. The interface itself is also called PF_KEY, and it is defined in RFC 2367 [PF_KEY].

Implementing ISAKMP mode config, Xauth, and hybrid authentication in racoon was not very difficult. All the protocols are documented by IETF drafts, so most of the work was done by trying to connect to racoon using the Cisco VPN client, and filling the gaps each time something was causing a failure.

IKE fragmentation has been more difficult because it was not documented. Some reverse engineering was required. Fortunately the protocol was rather simple: each IKE fragment had a small header containing the fragment length, the fragment index, and a flag for the last fragment.

Once the Cisco VPN client was able to establish an IPsec SA with a NetBSD machine, it was obvious that the project was going somewhere, and, therefore, the question of integrating these racoon changes raised. NetBSD racoon was only an import of KAME racoon, and hence the goal was to integrate the changes in KAME and import the newer racoon in NetBSD.

Unfortunately, the KAME project has not really been interested by this work. It was impossible to get it integrated, so the time had come to think about doing a fork. But managing a fork is never a pleasant plan, because it means a lot of work, merging future fixes from the original project with code added in the forked version.

Fortunately, a KAME fork already existed: ipsec-tools

5.2 IPsec-tools

Some time ago, it has been decided that KAME racoon was the way to go for Linux. Some contributions where done to KAME racoon for adding Linux support, but that turned into a racoon fork known as ipsec-tools.

ipsec-tools is only a fork of the userland part of the KAME IPsec stack: it contains setkey, libipsec and racoon. The project accepted the contributions to restore NetBSD build, and to add ISAKMP mode config, Xauth, Hybrid authentication, and IKE fragmentation.

ipsec-tools was later further improved to better fit in the remote user VPN scenario. Support was added to behave as a VPN client for hybrid authentication, and on the server side, things such as RADIUS and PAM authentication for Xauth logins were added. Another contributor added DPD.

For a documentation about how to configure a NetBSD system as a VPN gateway or a VPN client using hybrid authentication, see the how-to in the NetBSD documentation [HOW-TO].

5.3 NAT-Traversal in NetBSD

ipsec-tools has support for NAT-Traversal in tunnel mode, but that requires kernel support which was only available on Linux. Some work had to be done in the NetBSD kernel to get NAT-Traversal working on NetBSD.

The new code is ifdef'ed by the `IPSEC_NAT_T` option in the NetBSD kernel. It sits in four locations:

- in the setsockopt(2), where the IKE daemon tells the kernel that a given socket can be used for ESP over UDP.
- in the PF_KEY interface: the IKE daemon gives the UDP port number that is being used by IKE, and that will be used for ESP over UDP.
- in the UDP input function: for sockets tagged as using ESP over UDP, the UDP payload is transmitted to the ESP input function or to the IKE daemon in userland. A non-IKE marker is used at the beginning of the UDP payload to distinguish ESP and IKE packets
- in the ESP output function, data that is to be handled by an IPsec SA using ESP over UDP is sent to the UDP output function.

The original NAT-Traversal support written for Linux was not able to cope with the situation where IPsec SA were installed with multiples peers behind a NAT. This was because the code only used the IP addresses in SPD and SAD. Because the port information was missing, there was no way of distinguishing the traffic coming from different machines. NAT-Traversal support in racoon and in the NetBSD kernel was improved to keep track of port information in order to fix that.

5.4 Switching from KAME to ipsec-tools

At that time it was clear that ipsec-tools had much more activity than KAME racoon. Many new features were being integrated into ipsec-tools, while KAME was too busy on other problems to integrate anything. This led to the decision to integrate ipsec-tools into NetBSD instead of KAME racoon. The switch was done in April of 2005. It caused a few minor regressions such as TCP-MD5 lossage and transport mode without NAT-Traversal being broken, but those problems have been fixed.

At the end of April 2005, the KAME project dropped support for racoon and advised users to use ipsec-tools racoon instead [KAME]. This decision was made because the primary goal of KAME is to develop IPv6 and not an IKE daemon, and because there is no point having two projects for racoon.

Conclusions

The need for a particular set of requirement has driven the integration of many new features in NetBSD IPsec stack. The solution described here is an average security solution, where some security is traded for usability. But stronger security setups, where digital certificates are used for user authentication, also benefited from the new features that were added, such as NAT-Traversal, DPD, ISAKMP mode config, or IKE fragmentation.

Racoon was also improved on some fronts unrelated to network communications. For instance, it is now able to run as an unprivileged user and within a chroot environment.

In the future, racoon should keep evolving towards being compatible with more IPsec VPN implementations. There are also some missing features that are frequently required: NAT-Traversal in transport mode and IKEv2 integration. As users frustration increase, the odd gets better that someone will implement these missing features as well

NAT-Traversal in transport mode will require support in the NetBSD kernel. There is also another point on the road map for NetBSD IPsec: `FAST_IPSEC`. This alternative in-kernel IPsec stack is designed to make cryptographic operation asynchronous so that hardware accelerators can be used. It has not been modified for NAT-Traversal, and this is another gap that has to be filled.

Thanks to John R. Shannon, Greg Troxel, Greg Oster, and D'Arcy J.M. Cain for reviewing this paper. Thanks also to Florence Henry for the help with LaTeX.

References

[PPTP] http://www.schneier.com/paper-pptpv2.pdf

[OpenVPN] http://openvpn.net

[IKE] http://www.ietf.org/rfc/rfc2409.txt

[L2TP] http://www.ietf.org/rfc/rfc3931.txt

[Xauth] http://cvs.sourceforge.net/viewcvs.py/*checkout*/ipsec-tools/ipsec-tools/src/racoon/rfc/draft-beaulieu-ike-xauth-02.txt

[Hybrid] http://cvs.sourceforge.net/viewcvs.py/*checkout*/ipsec-tools/ipsec-tools/src/racoon/rfc/draft-ietf-ipsec-isakmp-hybrid-auth-05.txt

[mode-cfg] http://cvs.sourceforge.net/viewcvs.py/*checkout*/ipsec-tools/ipsec-tools/src/racoon/rfc/draft-dukes-ike-mode-cfg-02.txt

[NAT-T] http://www.ietf.org/rfc/rfc3947.txt

[ESP-over-UDP] http://www.ietf.org/rfc/rfc3948.txt

[MS-IPR] https://datatracker.ietf.org/public/ipr_detail_show.cgi?&ipr_id=78

[DPD] http://www.ietf.org/rfc/rfc3706.txt

[PF_KEY] http://www.ietf.org/rfc/rfc2367.txt

[HOW-TO] http://www.netbsd.org/Documentation/network/ipsec/rasvpn.html

[KAME] http://www.atm.tut.fi/list-archive/snap-users-2005/msg00105.html

Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack

Robert N. M. Watson
rwatson@FreeBSD.org

Computer Laboratory
University of Cambridge

Abstract

The FreeBSD SMPng Project has spent the past five years redesigning and reimplementing SMP support for the FreeBSD operating system, moving from a Giant-locked kernel to a fine-grained locking implementation with greater kernel threading and parallelism. This paper introduces the FreeBSD SMPng Project, its architectural goals and implementation approach. It then explores the impact of SMPng on the FreeBSD network stack, including strategies for integrating SMP support into the network stack, locking approaches, optimizations, and challenges.

1 Introduction

The FreeBSD operating system [4] has a long-standing reputation as providing both high performance network facilities and high levels of stability, especially under high load. The FreeBSD kernel has supported multiprocessing systems since FreeBSD 3.x; however, this support was radically changed for FreeBSD 5.x and later revisions as a result of the SMPng Project [1] [6] [7] [8].

This paper provides an introduction to multiprocessing, multiprocessor operating systems, the FreeBSD SMPng Project, and the implications of SMPng on kernel architecture. It then introduces the FreeBSD network stack, and discusses design choices and trade-offs in applying SMPng approaches to the network stack. Collectively, the adaption of the network stack to the new SMP architecture is referred to as the Netperf Project [5].

2 Introduction to Multiprocessors and Multiprocessing Operating Systems

The fundamental goal of multiprocessing is the improvement of performance through the introduction of additional processors. This performance improvement is measured in terms of "speedup", which relates changes in performance on a workload to the number of CPUs available to perform the work. Ideally, speedup is greater than 1, indicating that as CPUs are added to the configuration, performance on the workload improves. However, due to the complexities of concurrency, properties of workload, limitations of software (application and system), and limitations of hardware, accomplishing useful speedup is often challenging despite the availability of additional computational resources. In fact, a significant challenge is to prevent the degradation of performance for workloads that cannot benefit from additional parallelism.

Architectural changes relating to multiprocessing are fraught with trade-offs, which are best illustrated through an example. Figure 1 shows performance results from the Supersmack MySQL benchmark [2] on a quad-processor AMD64 system, showing predicted and measured transactions per second in a variety of kernel configurations:

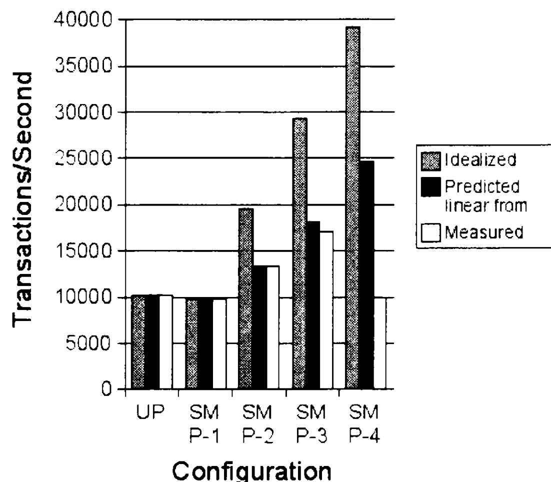


Figure 1: Speedup as Processors Increase for MySQL Select Query Micro-Benchmark

A number of observations can be made from these performance results:

- There is a small but observable decrease in performance in moving from a UP kernel to an SMP

kernel, even with the SMP kernel compiled to run only on a single CPU. This is due to the increased overhead of locked instructions required for SMP operation; the degree to which this is significant depends on the workload.

- An "optimal" performance figure in these results is extrapolated by predicting linear improvement from the single-processor case: i.e., with the addition of each processor, we predict an improvement in performance based on each new CPU accomplishing the amount of work performed in the single processor case. This would require that the hardware and OS support increased parallelism without increased overhead, that the work performed by the application be improved linearly through added parallelism, and that the application itself be implemented to use available parallelism effectively. As suggested by the graph, speedups of less than 1 are quite common.

- The graph also includes a predicted speedup based on linear improvement at the rate measured when going from a one-CPU to a two-CPU configuration. In the graph, the move from two to three processors accomplishes close to predicted; however, when going from three to four processors, a marked decrease in performance occurs. One possible source of reduced performance is the saturation of resources shared by all processors, such as bus or memory resources. Another possible source of reduced performance is in application and operating system structure: that certain costs increase as the number of processors increases, such as TLB invalidation IPIs and data structure sizes, resulting in increased overhead as CPUs are added.

This benchmark illustrates a number of important principles, the most important of which is that multiprocessing is a complex tool that can hurt as much as it helps. Improving performance through parallelism requires awareness and proper utilization of parallelism at all layers of the system and application stack, as well as careful attention to the overheads of introducing parallelism.

2.1 What do we want from MP systems?

Multithreading and multiprocessing often requires significant changes in programming model in order to be used effectively. However, where these changes are exposed is an important consideration: the SMP model selected in earlier FreeBSD releases was selected on the basis of minimal changes to the current kernel model, and minimal complexity. Adopting new structures and programming approaches offers performance benefits with greater software changes. The same design choice applies to the APIs exposed to user applications: in both earlier work on FreeBSD's SMP implementation and the more recent SMPng work, the goal has been to maintain standard UNIX APIs and services for applications, rather than introducing entirely new application programming models.

In particular, the design choice has been made to offer a Single System Image (SSI), in which user processes are offered services consistent with executing on a single UNIX system. This design choice is often weakened in the creation of clustered computing systems with slower interconnects, and requires significant application adaptation. For the purposes of the SMPng Project, the goal has been to minimize the requirement for application modification while offering improved performance. In FreeBSD 5.x and later, multiprocessor parallelism is exposed to applications through the use of multiple processes or multiple threads.

2.2 What is shared in an MP System?

The principle behind current multiprocessing systems is that computations requiring large amounts of CPU resources often have data dependencies that make performing the computation with easy sharing between parts of the computation cost effective. Typical alternatives to multiprocessing in SMP systems include large scale cluster systems, in which computations are performed in parallel under the assumption that a computation can be broken up into many relatively independent parts. As such, multiprocessor computers are about providing facilities for the rapid sharing of data between parts of a computation, and are typically structured around shared memory and I/O channels.

Shared	Not Shared
System memory PCI buses I/O channels ...	CPU (register context, TLB, on-CPU cache, ...) Local APIC timer ...

This model is complicated by several sources of asymmetry. For example, recent Intel systems make use of Hyper-Threading (HTT), in which logical cores share a single physical core, including some computation resources and caches. Another source of asymmetry has to do with CPUs having inconsistent performance in accessing regions of system memory.

2.3 Symmetric Memory Access

The term "symmetric" in Symmetric Multiprocessing (SMP) refers to the uniformity of performance for memory access across CPUs. In SMP systems, all CPUs are able to access all available memory with roughly consistent performance. CPUs may maintain local caches, but when servicing a cache miss, no piece of memory is particularly favorable to access over any other piece of memory. Whether or not memory access

is symmetric is primarily a property of memory bus architecture: memory may be physically or topologically closer to one CPU than another.

Environments in which uniform memory access is not present are referred to as Non-Uniform Memory Access (NUMA) architectures. NUMA architectures become necessary as the number of processors increases beyond the capacity that a simple memory bus, such as a crossbar, can handle, or when the speed of the memory bus becomes a point of significant performance contention due to the increase in CPUs outstepping the performance of the memory that drives them. Strategies for making effective use of NUMA are necessarily more refined, as making appropriate use of memory topology is difficult.

Traditional two, four, and even eight processor Intel-based hardware has been almost entirely SMP-based. Until relatively recently, all low-end server and desktop systems were SMP, and NUMA was largely found in high-end multiprocessing systems, such as supercomputers. However, with the introduction of the AMD64 hardware platform, NUMA multiprocessor systems are now available on the desktop and server.

2.4 Inter-Processor Communication

As suggested earlier, communication between processors in multiprocessing systems is often based on the use of shared memory between those processors. For threaded applications, this may mean memory shared between threads executing on different CPUs; for other applications, it may mean explicitly set up shared memory regions or shared memory used to implement message passing. Issues of memory architecture, and in particular, memory consistency and cache behavior, are key to both correctness and performance in multiprocessing systems. Significant variations exist in how CPU and system architectures handle the ordering of memory write-back and cache consistency.

Also important in multiprocessor systems is the inter-process interrupt (IPI), which allows CPUs to generate notifications to other CPUs, such as to notify another CPU of the need to invalidate TLB entries for a shared region, or to request termination, signalling, or rescheduling of a thread executing on the remote CPU.

3 SMPng

Support for Symmetric Multi-Processing (SMP) has been a compile-time option for the FreeBSD kernel since FreeBSD 3.x. The pre-SMPng implementation is based on a single Giant lock that protects the entire kernel. This approach exposes parallelism to user applications, but does not require significant adaptation of the kernel to the multiprocessor environment as the kernel runs only on a single CPU at a time.

The Giant lock approach offers relative simplicity of implementation, as it maintains (with minimal modification) the synchronization model present in the uniprocessor kernel which is concerned largely with synchronizing between the kernel and interrupts operating on the same CPU. This permits user applications to exploit parallism to improve performance, but only in circumstances where the benefits of application parallelism outweight the costs of multiprocessor overhead, such as cache and lock contention.

The FreeBSD SMPng Project, begun in June, 2000, has been a long-running development project to modify the underlying kernel architecture to support increased threading and substitute a series of more fine-grained data locks for the single Giant lock. The goal of this work is to improve the scalability of the kernel on multiprocessor systems by reducing contention on the Giant lock, resulting in improved performance through kernel parallelism.

The first release of a kernel using the new kernel architecture was FreeBSD 5.0 which offered the removal of the Giant lock from a number of infrastructural components of the kernel as well as some IPC primitives. Successive FreeBSD 5.x releases removed Giant from additional parts of the kernel such as the network stack, device drivers, and the majority of remaining IPC primitives. The recently released FreeBSD 6.0 also removes Giant from the UFS file system and refines the SMPng architecture resulting in significantly improved performance.

SMPng was originally targetted solely at SMP class systems, but with the increased relevance of NUMA systems, investigation of less symmetric memory architectures has become more important for the FreeBSD SMPng Project. Figures 2 and 3 illustrate prototypical Quad Xeon (SMP) and Quad AMD64 (NUMA) hardware layouts relevant to the FreeBSD SMPng Project. Figures 4 and 5 illustrate Graphical Processing Unit (GPU) and cluster architectures not considered as part of this work.

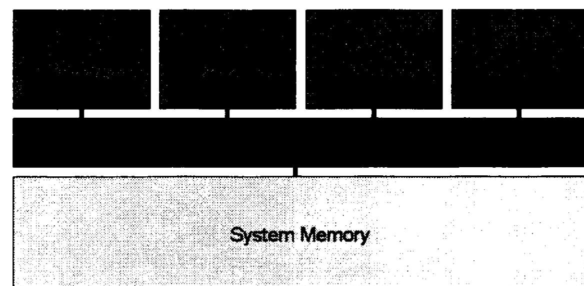


Figure 2: SMP Architecture: Quad-Processor Intel Xeon

3.1 Giant Locked Kernels

Support for multiprocessing in operating systems is either designed in from inception or retrofitted into an existing non-multiprocessing kernel. In the case of

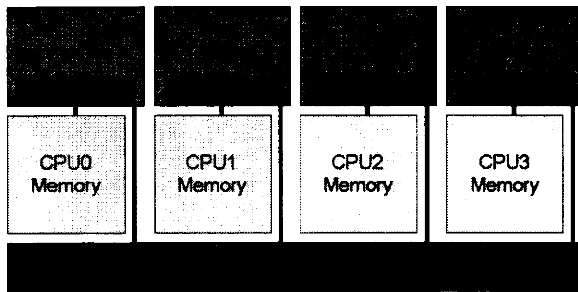


Figure 3: NUMA Architecture: Quad-Processor AMD64

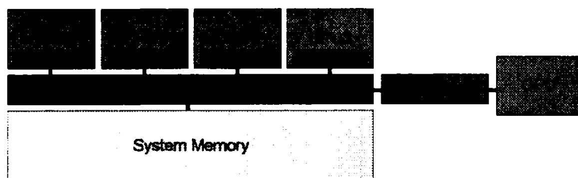


Figure 4: GPU Architecture: External Graphics Processor

most UNIX systems, multiprocessing support has been an after-thought, although the degree of redesign and reimplementation has varied significantly by product and version. The level of change has varied from low levels of change (using a Giant lock to maintain single-CPU synchronization properties and hence single-CPU kernel architecture), to complete reimplementation of the operating system based on a Mach microkernel and/or message passing.

The most straight forward approach to introducing multiprocessing in a uniprocessor operating system without performing a significant rewrite of the system is the Giant lock kernel approach. This approach maintains the property that most kernel synchronization occurs between the "top" and "bottom" halves – i.e., between system call driven operation and device driver interrupt handlers, and can be synchronized using critical sections or interrupt protection levels. In a Giant lock kernel, a single spinlock is placed around the entire kernel, in essence restoring the assumption that the kernel will execute on a single processor.

The FreeBSD 3.x and 4.x kernel series make use of a Giant spinlock which ensures mutual exclusion whenever the kernel is running. While the approach is simple, there are some important details: when a process attempts to enter the kernel, even the process scheduler, it must acquire the Giant lock. This results in lock contention when more than one processor tries to enter the kernel at a time (a common occurence with kernel-intensive workloads, such as network- or storage- centric loads common on FreeBSD). In the FreeBSD 4.x kernel, interrupts are able to preempt running kernel code. However, if an interrupt arrives on a CPU while the kernel is running on another CPU, it must be forwarded to the CPU where the kernel is running using an inter-processor interrupt (IPI).

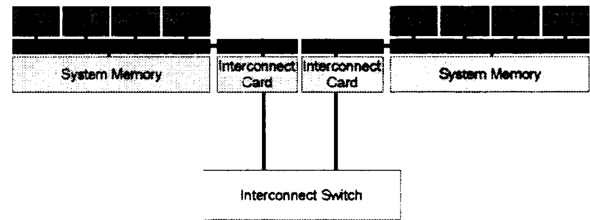


Figure 5: Cluster Architecture: Non-Uniform Memory via Complex Interconnect

3.2 Giant Contention

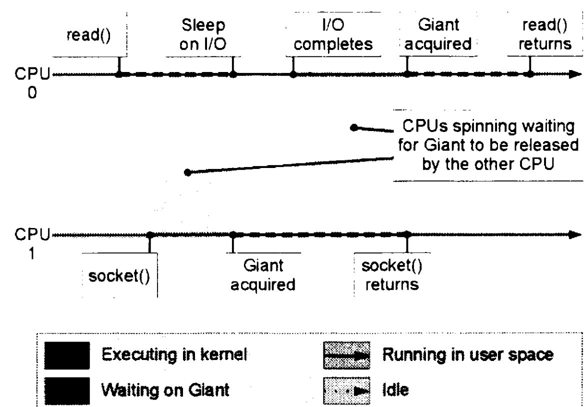


Figure 6: Impact of the Contention of a Giant Lock on Socket IPC

On systems with small numbers of CPUs, Giant Lock kernel contention is primarily visible when the workload includes large volumes of network traffic, inter-process communication (IPC), and file system activity. These are workloads in which the kernel must perform a significant amount of computation, resulting in increased delays and wasted CPU resources as other CPUs wait to enter the kernel, not to mention a failure to use available CPU resources to perform kernel work in parallel. On systems with larger numbers of CPUs, even relatively kernel non-intensive workloads can experience significant contention on the kernel lock, resulting in rapidly reduced scalability as the number of CPUs increases.

3.3 Finer Grained Locking

The primary goal of the SMPng Project has been to improve kernel performance on SMP systems by replacing the single Giant kernel lock with a series of smaller locks, each with more limited scope. This allows the kernel to usefully execute in parallel on multiple CPUs, potentially allowing more effective use of available

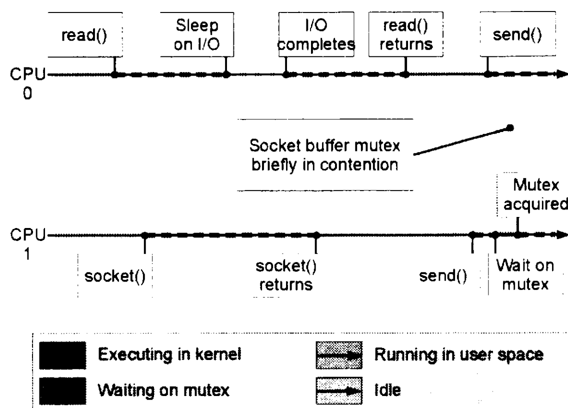


Figure 7: Reduced Lock Contention with Finer Grained Locking

CPUs by the kernel on multiprocessor systems. This also has the added benefit of avoiding wasting CPU as a result of Giant lock contention. This goal requires that the "interrupt level" synchronization model be replaced with one oriented around parallelism, not just preemption, resulting in a number of significant kernel architecture changes. For example, as disabling interrupts on a CPU will no longer prevent interrupt code from executing at the same time as system call code, interrupts must now make use of locks to synchronize with the remainder of the kernel. This in turn leads to a decision to execute interrupt handlers in full thread contexts (interrupt threads or ithreads).

This strategy has a number of serious risks:

- The new synchronization approaches must be more mature than the Giant lock approach, as introducing additional locks increases the risk of deadlocks. They must also address issues relating to concurrency and locking, such as priority inversion.
- Kernel synchronization must take into increased consideration the memory ordering properties of the hardware, as it has become a true multithreaded program.
- Inter-processor synchronization typically relies on atomic operations and locked bus operations, which are expensive to perform; by adding additional locking requirements, overhead can add up quickly.
- Race conditions previously visible in the kernel only under high memory pressure are now far more likely to occur.

On the other hand, the architectural goals also have a number of significant benefits:

- In adopting synchronization primitives similar to those exposed by user threading libraries, such as mutexes and condition variables, developers familiar with process threading will be able to get started quickly with the kernel synchronization environment.
- By moving from a model with implicit synchronization properties (automatic acquisition and dropping of Giant) in 3.x/4.x to a model of explicit synchronization, the opportunity is provided for introducing much stronger assertions.
- Adopting a more threaded architecture, such as through the use of ithreads, increases the opportunities for parallelism in the kernel, allowing kernel computation to make better use of CPU resources.

The new SMPng kernel architecture facilitates the use of parallelism in the kernel, including the creation of multiprocessor data pipelines. By adopting an iterative approach to development, removing the dependency for Giant gradually over time, the system was left open to other development work continuing as the SMP implementation was improved.

The next few sections document the general implementation strategy followed during the SMPng Project, taking a "First make it work, then make it fast" strategy:

3.4 SMP Primitives

The first step in the SMPng Project was to introduce new locking primitives capable of handling more mature notions of synchronization, such as priority propagation to avoid priority inversion, and advanced lock monitoring and debugging facilities.

3.5 Scheduler Lock

Efforts to decompose the Giant lock typically begin with breaking out the scheduler lock from Giant, so that code executing without Giant will be able to make use of synchronization primitive that interact with the scheduler. The availability of scheduling facilities is fundamental to the implementation of most kernel services, as most kernel services rely on the the `tsleep()` and `wakeup()` mechanisms to manage long-running events.

Simultaneously, scheduler adaptations to improve scheduling on multiprocessor systems can be considered: IPI's between CPUs to allow the scheduler to communicate explicitly with the kernel running on other processors, scheduler affinity, per-CPU scheduler queues, etc. A variety of such techniques have been introduced via modifications to the existing 4BSD scheduler, and in a new MP-oriented scheduler, ULE [13].

3.6 Interrupt Threads

Next, interrupt handlers are moved into ithreads, allowing them to execute as normal kernel threads on various CPUs and use of kernel synchronization facilities. This has the added benefit that interrupt handlers now gain access to many more kernel service APIs, which previously often could not be invoked from interrupt context.

3.7 Infratructure Dependencies

With basic services such as synchronization and scheduling available without the Giant lock, additional important dependencies are then locked down. Among these are the kernel memory allocator and event timers. This includes both the general memory allocator and specific allocators such as the Mbuf allocator. In FreeBSD 6.x, a single Universal Memory Allocator (UMA) is used to allocate most system memory, rather than using a separate memory allocator for the network stack [12]. This allows the network stack to take advantage of the slab allocation and per-CPU cache facilities of UMA, make use of uniform memory statistics, and interact with global notions of kernel memory pressure.

3.8 Data-Based Locking

In most subsystems, data-based locking is used, combining locks and reference counts to protect the integrity of major data structures. Generally, we have started with coarser-grained locking to avoid introducing overhead without first determining that finer granularity helps with parallelism. Typically, the Virtual Memory system will be an early target as there is almost constant interaction between processes and virtual memory due to the need for multiprocessor operation to invalidate TLBs across processors. In this stage, locking will be applied based on data structures in a relatively naive fashion, in order to provide a first cut of Giant-free operation that can then be refined.

3.9 Slide Giant off Gradually

As Giant becomes unnecessary for subsystems or components and all of their dependencies, remove the Giant lock from covering those paths. This has the effect of reducing general contention on Giant, improving the performance of components still under the Giant lock.

3.10 Synchronization Refinment

Drive refinement of locking based on lock contention vs. lock overhead. Make use of facilities such as mutex profiling and hardware performance counters.

When balancing overhead and contention, there are a number of strategies that can be used. For example, replicating data structures across CPUs can prevent contention on locks, if the cost of maintaining replication is lower than the overhead the contention would cause. Statistics structures are a prime starting point, as they are frequently modified, so reducing writing to the same memory lines will avoid cache invalidations. Statistics can then be coalesced for presentation to the user: this approach is used for a variety of memory allocator statistics.

Likewise, synchronization with data structures accessed only from a specific CPU can often be performed using critical sections, which see lower overhead as they need only prevent preemption, not against parallelism. Another example of this approach is used in the UMA memory allocator: in 5.x, per-CPU caches are protected with mutexes due to accesses to the cache from other CPUs during certain operations. In FreeBSD 6.x, per-CPU caches are protected using critical sections, avoiding cross-CPU synchronization for per-CPU access.

Operating system literature documents a broad range of strategies for inter-CPU synchronization and data structure management, including lockless queues and Read-Copy-Update (RCU). As hardware architectures vary in both performance and semantics, optimization approaches may be specific to hardware configurations.

4 FreeBSD Network Stack

Having reviewed the FreeBSD SMPng kernel architecture, we will now explore how this architecture is implemented in the FreeBSD network stack. The FreeBSD network stack is one of the most complex components of the BSD kernel, consisting of over 400,000 lines of code excluding distributed file systems and device drivers, also large subsystems.

The network stack includes a number of service abstractions, such as network interfaces, communications sockets, event dispatch, remote procedure calls (RPCs), a protocol-independent route table, and user event models. Of particular importance is that data flows rapidly and continuously across many layers of abstraction and implementation, requiring careful consideration of the interactions between components. These software layers of abstraction often, but not always, map to layers in protocol construction.

4.1 Introduction to the Network Stack

The network stack contains many large and complex components:

- "mbuf" memory allocator
- Network interface abstraction, including a number of queueing discplines
- Device drivers implementing network interfaces

- Protocol-independent routing and user event model
- Link layer protocols – Ethernet, FDDI, SLIP, PPP, ATM, etc.
- Network layer protocols – UNIX Domain Sockets, IPv4, IPv6, IPSEC, IPX, EtherTalk/AppleTalk, etc.
- Socket and socket buffer IPC primitives
- Netgraph extension framework
- Many netgraph nodes implementing a broad range of services

System call and socket	kern_send()	kern_recv()
	sosend() sbappend()	soreceive() sbappend()
TCP	tcp_send() tcp_output()	tcp_reass() tcp_input()
IP	ip_output()	ip_input()
Link Layer, Device Driver	ether_output()	ether_input()
	em_start()	em_intr()

Figure 8: FreeBSD Network Stack: Common Dataflow

4.2 Network Stack Concerns

Introducing parallelism and preemption introduces a number of additional concerns:

- Per-packet costs: network stacks may process millions of packets per second – small costs add up quickly if per-packet.
- Ordering: packet ordering must be maintained with respect to flow, as protocols such as TCP are very sensitive to minor misordering.
- Optimizations may conflict: optimizing for latency may damage throughput, or optimizing for local data transfer may damage routing performance.
- When using locking, ordering is important – lock order prevents deadlock, but passage through layers in the network stack is often bi-directional.
- Some amount of parallelism is available by virtue of the current network stack architecture – introducing new parallelism is necessary in order to improve utilization of MP resources, but depends on introducing additional threads, which can increase overhead.

These concerns are discussed in detail as the locking strategy is described.

4.3 Locking Strategy

The SMPng locking strategy for the network stack generally revolves around data-based locking. Using this strategy involves identifying objects and assigning locks to them; the granularity of locking is important as each lock operation introduces overhead. Useful rules of thumb include:

- Don't use finer-grained locking than is required by the UNIX API: for example, parallel send and receive on the same socket has benefit, but parallel send on a stream socket has poorly defined semantics, so not permitting parallelism can avoid unnecessarily complexity.
- Lock references to in-flight packets, not packets themselves. For example, lock queues of packets used to hand off between threads, but use only simple pointe references within a thread.
- Use optimistic concurrency techniques to avoid additional lock overhead – i.e., where it is safe, test a value that can be read atomically without a lock, then only acquire the lock if work is required that may have stronger consistency requirements.
- Avoid operations that may sleep, which can result in multiple acquires of mutexes, as well as unwinding of locks. In general, the network stack is able to tolerate failures through packet loss under low memory situations, so take advantage of this property to lower overhead.

Also important is consideration of layering: as objects may be represented at different layers in the stack by different data structures, decisions must be made both with respect to whether layers share locks, and if they don't share locks, what order locks may be acquired in. Control flow moves both "up" and "down" the stack, as packets are processed in both input and output paths, meaning that if locks are simply acquired as processing occurs, lock order cycles will be introuced as processing occurs in two directions.

The following general strategies have been adopted in the first pass implementation of fine-grained locking for the network stack:

- Low level facilities, such as network memory allocation, route event notification, packet queues, and dispatch queues, generally have leaf locks so that they can be called from any level of the stack including device drivers.

- Protocol locks generally fall before device driver locks in the lock order, so that device drivers may be invoked without releasing stack locks.
- Protocol locks generally fall before socket locks in the lock order, so that protocols can interact with sockets without releasing protocol locks.

Just as asynchronous packet dispatch to the netisr in earlier BSD network stacks allows avoiding of layer recursion and reentrance, it can also be used to avoid lock order issues with an MPSAFE network stack. This technique is used, for example, to avoid recursing into socket buffer code when a routing event notification occurs as the result of a socket event, and prevents deadlock by eliminating the "hold and wait" part of the deadlock recipe. The netisr will processed queued routing socket events asynchonously, delivering them to waiting sockets.

4.4 Global Locks

Global locks are used in two circumstances: where global data structures are referenced, or where data structures are accessed sufficiently infrequently that coalescing locks does not increase contention. The following global locks are a sampling of those added to the network stack to protect global data structures:

Lock	Description
ifnet_lock	Global network interface list
bpf_mtx	Global BPF descriptor list
bridge_list_mtx	Global bridge configuration
if_cloners_mtx	Cloning network interface data
disc_mtx, faith_mtx gif_mtx, gre_mtx, lo_mtx ppp_softc_list_mtx stf_mtx, tapmtx, tun_mtx, ifv_mtx	Synthetic interface lists
pfil_global_lock	Packet filter registration
rawcb_mtx, ddp_list_mtx, igmp_mtx, tcbinfo_mtx, udbinfo_mtx ipxpcb_list_mtx natm_mtx, rtsock_mtx	Per-protocol control block lists
hch_mtx	TCP host cache
ipqlock, ip6qlock	IPv4 and IPv6 fragment queues
aarptab_mtx, nd6_mtx	Link layer address resolution
in_multi_mtx	IPv4 multicast address lists
mfc_mtx, vif_mt mrouter_mtx	IPv4 multicast routing
sptree_lock, sahtree_lock regtree_lock, acq_lock spacq_lock	IPSEC

The following is a sampling of locks have been added to data structures allocated dynamically in the network stack:

Structure	Field	Description
ifnet	if_addr_mtx	Interface address lists
	if_afdata_mtx	Network protocol data
	if_snd.ifq_mtx	Interface send queue
bpf_d	bd_mtx	BPF descriptor
bpf_if	bif_mtx	BPF interface attachment
ifaddr	ifa_mtx	Interface address
socket	so_rcv.sb_mtx	Socket, socket receive buffer
	so_snd.sb_mtx	Socket send buffer
ng_queue	q_mtx	Netgraph node queue
ddpcb	ddp_mtx	netatalk PCB
inpcb	inp_mtx	netinet PCB
ipxpcb	ipxp_mtx	netipx PCB

4.5 Network Stack Parallelism

Parallelism in the FreeBSD kernel is expressed in terms of threads, as they represent both execution and scheduling contexts. In order for one task to occur

in parallel with another task, it must be performed in a different thread from that task. In order for the FreeBSD kernel to make effective use of multiprocessing, work must therefore occur in multiple threads.

A fair amount of parallelism in the network stack is simply from conversion of the existing BSD network stack model to the SMPng architecture:

- Each user thread has an assigned kernel thread for the duration of a system call or fault, which performs work directly associated with the system call or fault. In the transmit direction, the user thread is responsible for executing socket, protocol, and interface portions of the transmit code, which includes the cost of copying data in and out of user space. In the receive direction, the user thread is responsible for primarily for executing the socket code, along with copying data in and out of user space; under some circumstances, calls into the protocol and interface layers may also occur.

- Each interrupt request (IRQ) is assigned its own ithread, which is used to execute the handlers of interrupt sources signaled by that interrupt. As long as devices are assigned different interrupts, their handlers can execute in parallel. By default, this will include execution of the link layer interface code, but a dispatch to the netisr thread for higher stack layers.

- A number of kernel tasks are performed by shared or assigned worker threads, such as callouts and timers running from a shared callout thread, several task queue processing threads for various subsystems, and the netisr thread in the network stack, which is primarily responsible for the protocol layer processing of in-bound packets.

While multithreading is required in order to experience parallelism, multithreading also comes with significant costs, including:

- Cost of context switching, which may include the cost of cache flushes when a thread migrates from one CPU to another and the cost of entering the scheduler.

- Cost of synchronizing access to data between threads: typically, a locked or otherwise synchronized data structure or work queue.

Figure 9 illustrates the UDP send path, and possible parallelism between the user thread sending down the stack layers, and ithread receiving acknowledgements from the network stack in order to recycle packet memory.

Figure 10 illustrates the UDP receive path, and possible parallelism between the user thread interacting with the socket layer, netisr processing the IP and UDP layers, and the ithread receiving packets from the network interface and processing the link layer.

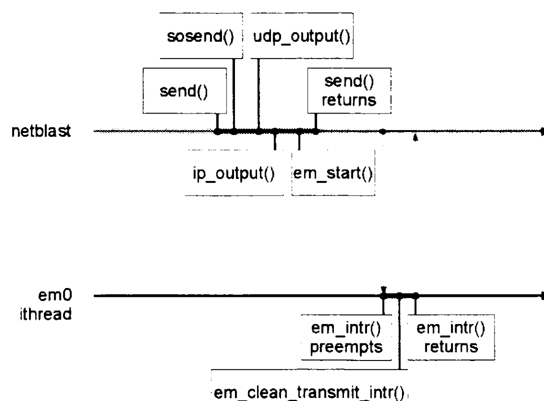


Figure 9: Parallelism in the UDP send path

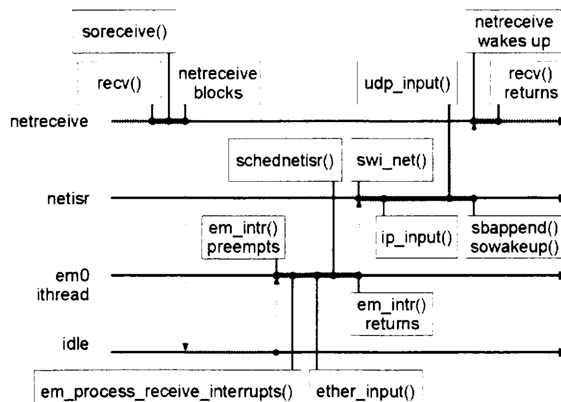


Figure 10: Parallelism in the UDP receive path

5 MP Programming Challenges

Multiprocessing is intended to improve performance by introducing greater CPU resources. However, unlike a number of other hardware-based performance improvements, such as increasing clock speed or cache size, multiprocessor programming requires fundamental changes in programming model. In this section, we consider two important concerns in multiprocessing and multithreading programming and their relationship to the network stack: deadlock, and event serialization.

5.1 Deadlock

Deadlock is a principal concern of systems with synchronous waiting for ownership of locks on objects. Deadlocks occur when two or more simultaneous threads of execution (typically kernel threads) meet the following four conditions:

- Attempt to simultaneously access more than one resource which can be owned, but not shared (mutual exclusion).

- Hold and wait: threads acquire and hold resources in an order.

- No preemption: once acquired, a resource cannot be preempted without agreement of the thread.
- Circular wait: threads acquire and attempt to acquire resources such that a cycle is formed, resulting in indefinite wait.

The above description is intentionally phrased in terms of resources rather than locks, as deadlock can occur in more general circumstances. For example, low memory deadlock is another type of widely experienced deadlock.

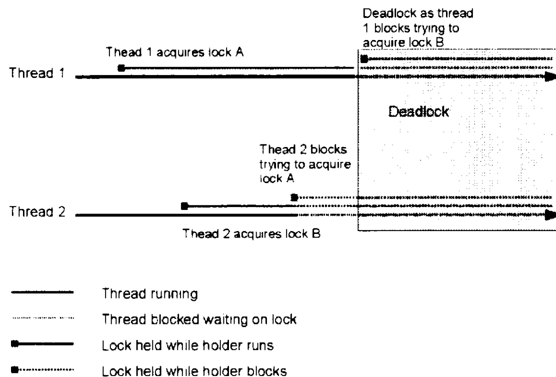


Figure 11: Deadlock: The Deadly Embrace

There is extensive research literature on deadlock avoidance, detection, and management; however, one of the most straight forward and easiest ways to avoid deadlock is simply to follow a strict lock order. Lock orders indicate that, whenever any two locks can be acquired as the same time, they will always be acquired in the same order. This breaks lock order cycles, and thus prevents deadlock, and is a widely used technique.

In order to assist in documenting lock orders and prevent cycles, BSDI created WITNESS, a run-time lock order verifier, which was refined by the FreeBSD Project to support additional lock types and assertion types. WITNESS can be used as both a tool to document a specification for lock interaction through a hard-coded lock order list, and to dynamically discover lock order relationships through run-time monitoring. WITNESS maintains a graph of lock acquisition orders, and provides run-time warnings (along with stack traces and other debugging information), when declared or discovered lock orders are directly or indirectly violated.

WITNESS and other lock-related invariants also detect and report a variety of other lock usage, such as acquiring sleepable locks while holding mutexes or in critical sections, or kernel threads returning to user space while holding locks.

FreeBSD also makes use of other deadlock avoidance techniques, including the use of optimistic concurrency techniques in which attempts are made to acquire locks in the wrong order, and then if this would

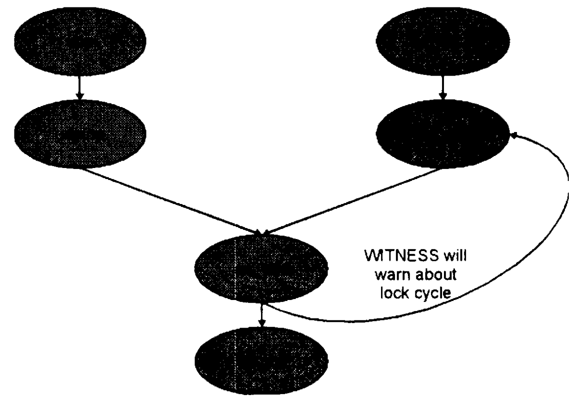


Figure 12: Lock order verification with WITNESS: Cycles in the lock graph are detected and reported using graph algorithms.

result in a deadlock, falling back on the defined order. Another technique used in the kernel is the use of guard locks, acquired before acquiring locks on multiple objects with no defined lock order between them. By serializing attempts at simultaneous acquire behind a lock, the lock order of the objects becomes defined only when they are acquired at once, and no conflicting lock order can be simultenously defined, preventing deadlock.

5.2 Event Serialization

In a singlethreaded programming environment, the order of events is largely a property of programmed order, so maintaining the order associated with a data structure or the processing of data is essentially a problem of ordering of the program. In a multithreaded programming environment, concurrency in code execution means that parallel threads of execution may execute at different rates, and that any ordering of events must occur as a result of planning. If events must occur in a specific order, programmers must either execute them in a single thread (which serialized events into programmed order), or synchronization primitives and communication primitives must be used so that ordering is either maintained during computation, or restored during post-processing after the computation.

This is particularly relevant to the implementation of the network stack, in which discrete units of work, typically represented by packets, are processed in a number of threads. The order of packets can have a significant impact on performance, and so maintaining necessary orders is critically important. For example, out-of-order delivery of TCP packets can result in TCP perceiving packet loss, resulting in a fast (and unnecessary) retransmit of data. Packet ordering must typically be maintained with respect to its flow, where the granularity of the flow might include a stream of packets sourced from a particular network interface, packets between two hosts, or packets in a particular connec-

tion.

In the single-threaded FreeBSD 4.x receive path, ordering is maintained throughout through the use of last-in, first out (LIFO) queues between threads, effectively serializing processing. A single netisr thread processes all inbound packets from the link layer to the network layer. Naively introducing multithreading into a network stack without careful consideration of ordering might be performed by simply introducing additional in-bound packet worker threads (netisrs). Figure 13 illustrates that this might result in misordering of packets in a simple packet forwarding scenario: two packets might be dispatched in one order to different worker threads, and then be forwarded in reversed order due to scheduling of the worker threads.

In FreeBSD 6.x, two modes of operation are documented for packet processing dispatch: queued serialized dispatch with a single netisr thread, or direct dispatch of packet processing from the calling context. In direct dispatch mode, context switches are reduced by performing additional packet processing in the originating thread for a packet, rather than passing all packets to a single worker thread – for example, in the interrupt thread for a network interface driver. This implements a weaker ordering by not committing to an ordered queue, but maintains sufficient ordering. Weakened packet ordering improves the opportunities for parallelism by permitting more concurrency in packet processing, and is an active area of on-going work in the SMPng Project. One downside to direct dispatch in the ithread is reduced opportunity for parallelism, as in-bound processing is now split between two threads: the ithread and the receiving user thread, but not the netisr.

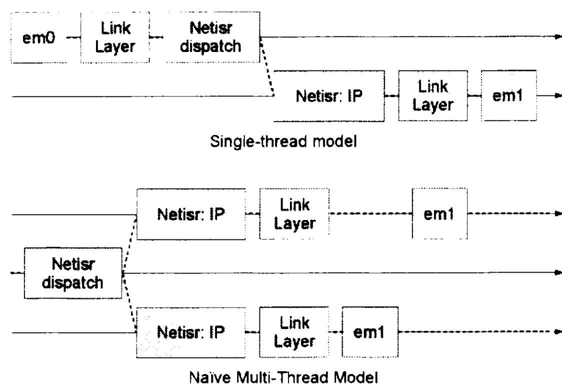


Figure 13: Singlethreaded and naive multithreaded packet processing, in which sufficient ordering is no longer maintained

6 Status of the SMPng Network Stack

As of FreeBSD 6.0, the vast majority of network stack code is run without the Giant lock in the default configuration. This includes most link layer network device drivers and services, such as gigabit ethernet drivers and ethernet bridging, ARP, the routing table, IPv4 input, filtering, and forwarding, FAST_IPSEC, IP multicast, protocol code such as TCP and UDP, and the socket layer. In addition, many non-IP protocols, such as AppleTalk and IPX are also MPSAFE.

Some areas of the network stack continue to require Giant, and can generally be put in two categories:

- Code that requires Giant for correctness (perhaps due to interacting with another part of the kernel that requires Giant), but can be executed with Giant but an otherwise Giant-free network stack.
- Code that requires Giant for correctness, but due to lock orders and construction of the network stack, requires holding Giant over the entire network stack when used.

In the former category lie the KAME IPSEC implementation and ISDN implementation. Giant is required over the entire stack because these code paths can be entered in a variety of situations where other locks (such as socket locks) can already be held, preventing Giant from being acquired when it is discovered the non-MPSAFE code will be entered. Instead, Giant must be acquired in advance unconditionally.

Other areas of the system also continue to require the Giant lock, such as a number of legacy ISA network device drivers and portions of the in-bound IPv6 stack. In both cases, Giant will be conditionally acquired in an asynchronous execution context before invoking the non-MPSAFE code. A number of consumers of the network stack also remain non-MPSAFE, such as the Netware and SMB/CIFS file systems. With the FreeBSD 6.0 VFS now able to support MPSAFE file systems, locking down of these file systems and removal of Giant is now possible; in the mean time, they execute primarily in VFS consumer threads that will already have acquired Giant, and not synchronously from network stack threads that run without Giant, permitting the network stack to operate without Giant.

Components operating with Giant for compatibility continue to see higher lock contention and latency due to asynchronous execution. It is hoped that remaining network stack device drivers and protocols requiring Giant will be made MPSAFE during the 6.x branch lifetime.

7 Related Work

Research and development of multiprocessor systems has been active for over forty years, and has been performed by hundreds of vendors for thousands of products. As such, this section primarily points the reader at a few particularly useful books and sources relevant

to the SMP work on FreeBSD, rather than attempting to capture the scope of prior work in this area.

Curt Schimmel provides a detailed description of multiprocessor synchronization techniques and the application in UNIX system design in *UNIX Systems for Modern Architectures*, including detailed discussion of design trade-offs [14].

Uresh Vahalia provides general discussion of advanced operating system kernel architectures across a number of UNIX systems in *UNIX Internals* [15].

The FreeBSD SMPng architecture has been significantly impacted by the design and implementation strategies of the Solaris operating system, discussed in *Solaris Internals* by Jim Mauro and Richard McDougall [9].

The Design and Implementation of 4.4BSD by Kirk McKusick, et al, describes earlier BSD kernel architecture, and particularly synchronization, in great detail, and makes a useful comparison with *The Design and Implementation of the FreeBSD Operating Systems*, which describes the FreeBSD 5.x architecture [10] [11].

A good general source of information on multiprocessing and multithreading programming techniques, both for userspace and kernel design, are the design and implementation papers relating to the Mach operating system project at Carnegie Mellon [3].

8 Future Work

Remaining work on the SMPng network stack falls primarily into the following areas:

- Complete removal of Giant requirement from all remaining network stack code (device drivers, IPv6 in-bound path, KAME IPSEC).
- Continue to explore improving performance and reducing overhead through refining data structures, lock strategy, and lock granularity, as well as further exploring synchronization models.
- Continue to explore improving performance through analyzing cache footprint, interprocessor cache interactions, and so on.
- Continue to explore how to further introduce useful parallelism into network processing, such as additional parallel execution opportunities in the transmit path and in network interface polling.
- Continue to explore how to reduce latency in processing through reducing queued dispatch, such as via network interface direct dispatch of of the protocol stack.

It is expected that the results of this further work will appear in future FreeBSD 6.x and 7.x releases.

9 Acknowledgments

The SMPng Project has been running for five years now, and has had literally hundreds of contributors, whose contributions to this work have been invaluable. As a result, not all contributors can be acknowledged in the space available, and the list is limited to a subset who have worked actively on the network stack parts of the project.

The author greatfully acknowledges the contributions of BSDI, who contributed prototype reference source code for parts of a finer-grained implementation of the BSD kernel, and specifically, network stack, as well as their early development support for the SMPng Project as a whole. The author also wishes to recognize the significant design, source code development, and testing contributions of the following people without whom the Netperf project would not have been possible: John Baldwin, Antoine Brodin, Jake Burkholder, Brooks Davis, Pawel Dawidek, Julian Elischer, Don Lewis, Brian Feldman, Andrew Gallatin, John-Mark Gurney, Paul Holes, Peter Holm, Jeffrey Hsu, Roman Kurakin, Max Laier, Nate Lawson, Sam Leffler, Jonathan Lemon, Don Lewis, Scott Long, Warner Losh, Rick Macklem, Ed Maste, Bosko Milekic, George Neville-Neil, Andre Oppermann, Alfred Perlstein, Luigi Rizzo, Jeff Roberson, Mike Silberback, Bruce Simpson, Gleb Smirnoff, Mohan Srinivasan, Mike Tancsa, David Xu, Jennifer Yang, and Bjoern Zeeb.

Financial support for portions of the Netperf Project and test hardware was provided by the FreeBSD foundation. The Netperf Cluster, a remotely managed cluster of multiprocessor test systems for use in the Netperf project, has been organized and managed by Sentex Communications, with hardware contributions from FreeBSD Systems, Sentex Communications, IronPort Systems, and George Neville-Neil. Substantial additional testing facilities and assistance have been provided by the Internet Systems Consortium (ISC), Sandvine, Inc., and Yahoo!, Inc. The author particularly wishes to acknowledge Kris Kennaway for his extended hours spent in testing and debugging SMPng and Netperf Project work as part of the FreeBSD package building cluster.

10 Conclusion

The FreeBSD SMPng Project has now been running for five years, and has transformed the architecture of the FreeBSD kernel. The resulting architecture makes extensive use of threading, fine-grained and explicit synchronization, and offers a foundation for a broad range of future work in exploiting new hardware platforms, such as NUMA. The FreeBSD SMPng network stack permits the parallel execution of the network stack on multiple processors, as well as a fully preemptive network stack. In this paper, we've presented

background on MP architectures, an introduction to the SMPng Project and recent work on SMP in FreeBSD, and the design principles and challenges in adapting the network stack to this architecture.

11 Availability

The results of the SMPng Project began appearing in FreeBSD releases beginning with FreeBSD 5.0. The FreeBSD 6.x branch reflects further completion of SMPng tasks, and significant refinement of the work in FreeBSD 5.x. General information on the FreeBSD operating system, as well as releases of FreeBSD may be found on the FreeBSD web page:

`http://www.FreeBSD.org/`

Further information about SMPng may be found at:

`http://www.FreeBSD.org/smp/`

The Netperf project web page may be found at:

`http://www.FreeBSD.org/projects/netperf/`

References

[1] BALDWIN, J. Locking in the Multithreaded FreeBSD Kernel. In *Proceedings of BSDCon'02* (February 2002), USENIX.

[2] BOURKE, T. Super smack. http://vegan.net/tony/supersmack/.

[3] CARNEGIE MELLON UNIVERSITY. The Mach Project Home Page. http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html.

[4] FREEBSD PROJECT. FreeBSD home page. http://www.FreeBSD.org/.

[5] FREEBSD PROJECT. FreeBSD Netperf Project. http://www.FreeBSD.org/project/netperf/.

[6] FREEBSD PROJECT. FreeBSD SMP Project. http://www.freebsd.org/smp/.

[7] HSU, J. Reasoning about SMP in FreeBSD. In *Proceedings of BSDCon'03* (September 2003), USENIX.

[8] LEHEY, G. Improving the FreeBSD SMP Implementation. In *Proceedings of FREENIX Track: 2001 USENIX Annual Technical Conference* (June 2001), USENIX.

[9] MAURO, J., AND MCDOUGALL, R. Solaris Internals: Core Kernel Architecture, 2001.

[10] MCKUSICK, M., BOSTIC, K., KARELS, M., AND QUARTERMAN, J. The Design and Implementation of the 4.4BSD Operating System, 1996.

[11] MCKUSICK, M., AND NEVILLE-NEIL, G. The Design and Implementation of the FreeBSD Operating System, 2005.

[12] MILEKIC, B. Network Buffer Allocation in the FreeBSD Operating System. http://www.bsdcan.org/2004/papers/NetworkBufferAllocation.pdf.

[13] ROBERSON, J. ULE: A Modern Scheduler for FreeBSD. In *Proceedings of BSDCon'03* (September 2003), USENIX.

[14] SCHIMMEL, C. UNIX Systems for Modern Architectures, 1994.

[15] VAHALIA, U. UNIX Internals: The New Frontiers, 1996.

Failover Mechanisms for Filtering Bridges on the BSD's

An overview of the technologies available to make your bridged network fault-tolerant

-----

Massimiliano Stucchi, WillyStudios.com
(stucchi@willystudios.com)

INTRODUCTION

There seems to be much appreciation of bridges when a network topology is already set, since they offer an easy way of integrating a a firewalling or traffic shaping device without compromising what's already in place. We all know that a rule of thumb of the network- or sysadmin is not to touch something that works, so bridges help us in this case.
In this simple work of replicating packets across interfaces, bridges can represent a weak point of failure we put in our topology, and can easily lead to outages, isolating entire network segments in a matter of seconds.

So, what can be done to avoid this in environments where high availability is needed?

INITIAL PROBLEMS

I started thinking about this some time ago, but found no way, so I avoided bridges until I could get a grasp on the issue. Bringing up the topic at a dinner with Luigi Rizzo and Gianmarco Giovannelli also lead to nothing. Rizzo mentioned the Spanning Tree Protocol as a means of achieving this, but he also reported he had a thesis proposal for its implementation in the FreeBSD TCP/IP stack, so it was far away from the real possibility of having it, and the necessary mix of time and knowledge to do so was not there. There were also rumors about a quad-ethernet card which turned itself into a hub when losing power from the CPU, thus losing filtering or traffic shaping capabilities, but preserving network functionalities in case of a hardware failure, but searches on google and to hardware wharehouses didn't expose any product. It seemed as there was no applicable solution to redundancy in the bridging world, excluding complicated solutions which involved BGP and multihoming, which are not affordable for the SMB our services aim at.

THE ACTUAL SITUATION

A few months ago, when having to start a new big project which involved focusing on network communication and security for a large installation, I went back to see if the situation had changed in the meantime, since using bridges was an appealing way of achieving exactly what we needed.

CASE STUDY

The project was about setting up a VoIP infrastructure for a mid-sized telecom company who wanted to get into the internet telephony market, and approached us with the request of implementing an extension to their network, with the idea of switching completely to VoIP in less than a year. Given the budget restrictions, the challenge was made bigger by the fact that we had to use the same infrastructure the telco already had in place for their hosting, housing and colocation services. This involved having to deal with bandwidth usage issues, and finding a way to deal with realtime communications required by VoIP services, tied with the applications already in place.

The idea was to use FreeBSD to host the VoIP services, while we we went out to see what operating system, always from the BSD family, could be adopted for firewalling and filtering purposes.

WHAT IS AVAILABLE

The BSD's, deriving from the same codebase, share large pieces of code, old and new, and software written for one of the variants is often ported to the others in a matter of days. This was the case for PF, the OpenBSD Packet Filter, CARP, if_bridge and many others which are now spawned in every project. There are, however, cases where particular features are hard to port from one flavour to the other, and this is why we had to carefully look at the alternatives offered by the market before choosing the solution to implement.

CARP

CARP Stands for Common Address Redundancy Protocol. As described in its manpage, CARP allows multiple hosts on the same local network to share a set of IP addresses. This means you can have a cluster of machines doing transparent failover using a single IP address, and managing states of the hosts inside the cluster.

CARP is able to do failover and load balancing at the same time, and can be configured to handle situations with a master machine along with some slaves. When the master machine goes down, one of the slaves takes its place and starts answering requests. Working between OSI levels 2 and 3, CARP creates a “group” of hosts which share the same MAC address, thus giving the chance to have more redundant IP's on the same cluster, and being at the same time able to address both IPv4 and IPv6.
Like in other failover management software, the master machine in the cluster

sends out periodic announcements to other machines, using the CARP protocol (IP protocol 112) and multicast. If the backup machines do not hear these announcements, they promote themeselves – currently, only one at a time – to be master. The time that it takes for the backup machines to react to missing announcements, and the priority in which machines have to be promoted, are highly configurable. Communications between machines in the cluster are all encrypted using SHA-1 HMAC, and every host has to know a password to enter the cluster, so CARP is to be considered generally safe to use.
Load balancing, a feature that can be combined with failover, is achieved by means of a configuration keyword to issue at the time of creation of the interface. However, load is not distributed equally accross machines, nor it is done using round-robin assignment, but a hash of the originating ip address is used to determine which host is going to take care fo the communication.

But CARP, in its vanilla incarnation, is not able to handle bridged configuration, having the need for the interface it is enabled on to already have an IP address. For this purpose, the *carpdev* keyword was introduced by Ryan McBride – one of the original authors of CARP - by the end of 2004.

CARPDEV

The carpdev keyword, as stated before, allows for the use of CARP over interfaces without an IP address, as it is common in bridged configurations. While it was introduced some time ago in the OpenBSD tree, this feature is still not present on FreeBSD, nor it seems to be on NetBSD.
I have set up a test environment to look over the possibility of porting the feature to FreeBSD, but time constraints and lack of knowledge are making this task require more time than expected. However, OpenBSD with this feature enabled behaves really well in bridged environments, and constitutes a good choice.

PFSYNC

PFSync is to be considered an add-on for CARP environments, since its goal is to provide a means to synchronize connection states between machines in a failover cluster. It is very useful in accordance to the aforementioned protocol, in order to achieve high levels of redundancy and at the same time giving administrators the certainty that connections will not be dropped in case a machine goes down. PFSync exchanges connection states data between machines in the cluster, and if one of them becomes unreachable, the others can take care of the connections it was handling, without a glitch in the mechanism.

IF_BRIDGE

The bridge interface was originally written by Jason Wright, as an undergraduate independent project at the University of Carolina at Greensboro. It was then imported into NetBSD by Wasabi Systems, where it was renamed if_bridge, and it was recently ported to FreeBSD from thie latter work. It is mainly an interface which provides bridging capabilities along with

some more features than the ones provided from older implementations, especially the bridge facility in FreeBSD, which lacked support for adding filtering mechanisms, and it needed too much tweaking in order to be able to do so. if_bridge allowed FreeBSD to do filtering using PF in bridged environments. At the same time, if_bridge added a failover functionality to the field, giving the chance to use spanning tree on bridged interfaces.

THE SPANNING TREE PROTOCOL

The Spanning Tree Protocol is often used on network equipment to enable link redundancy and failover. It generally works by using a mix of greedy algorithms to learn about the surrounding network, interpreting the data through a shortest path algorithm, and determining possible loops in the network topology. If a link goes down, it is also able to adjust itself and adapt to the new environment transparently.

The empirical difference between Spanning Tree and CARP resides mainly in the time needed for the failover mechanism to come into play and adjust the configuration. While with CARP it can take up to 12 seconds, STP has an average of 25 to 30 seconds to adapt. STP can give you better results by tweaking the source in if_bridge.c, but there is no configuration variable that can be modified without recompiling to influence this behaviour.

TRUNK

The trunk interface was recently added to the OpenBSD tree. Its main goal is to enable the creation of a virtual interface out of multiple links on multiple network interfaces, thus giving the chance to load balance traffic on multiple interfaces.
While not designed with failover features in mind, the trunk interface can provide some sort of failover mechanisms using the failover option. While this can be effective in case of network problems on a switch port or on an interface card, it doesn't add any failover capability to a normal bridged environment, where a hardware fault compromises the topology.

NETGRAPH

A word of mention can also go to the Netgraph framework. Netgraph modules can easily be created out of this framework, and can do nearly anything. In the base FreeBSD tree – to which netgraph is specific – we can find modules that range from bridging (ng_bridge), to vlan management (ng_vlan), to port trunking (ng_one2many), similar to the trunk interface. Although not widely known, netgraph modules can be an easy and effective way to implement network features that are missing or that are needed in particular environments.

OUR CHOICES

After some tests with different implementations, we summed everything, and came to the conclusion that CARP and carpdev on OpenBSD was the solution we needed. However, after some discussions, we all agreed that keeping an homogeneous network would have been better, since all the machines already in place had FreeBSD on them, and we also maintain a local patchset for some ports and some meta-ports with the purpose of installing config files and copying data accross machines.
We finally decided to pursue the FreeBSD way, and to try to develop in.house support for what we needed. The actual implementation relies on if_bridge and STP on EPIA DP10000 boxes for our firewalling needs, and since we haven't had any problem since the day we set them up, we are not putting much effort in bringing the new feature in FreeBSD. We are, however, working on that.

DVCS or a new way to use Version Control Systems for FreeBSD

Ollivier ROBERT
<roberto@FreeBSD.org>

25th October 2005

Abstract

FreeBSD, like many open source projects, uses CVS as its main version control system (VCS), which is an extended history of all modifications made since the beginning of the project in 1993. CVS is a cornerstone of FreeBSD in two ways: not only does it record the history of the project, but it is a fundamental tool for coordinating the development of the FreeBSD operating system.

CVS is built around the concept of centralised repository, which has a number of limitations.

Recently, a new type of VCS has arisen: Distributed VCS, one of the first being BK from BitMover, Inc. Better known from the controversy it generated when Linus Torvalds started using it, it has nonetheless changed the way some people develop software.

This paper explores the area of distributed VCS. We analyse two of them (Arch in its Bazaar[1] incarnation and Mercurial[2] and try to show how such a tool could help further FreeBSD development, both as a tool and as a new development process.

1 Introduction

FreeBSD[3] has been using CVS as its main version control system (VCS) tool for as long as it exists and has now more than 10 years of history. A few years ago, limitations inherent in CVS design became too much to workaround and the project begun using Perforce[4] for projects that needed to change fast without "polluting" the main tree.

It works well but using two VCS instead of one is making merges harder than it needs to be and CVS limitations have become too much even for the main tree itself. The separation of the repository into 4 different ones helped but it is still complicated.

After a bit of history, we will explore how we could solve this problem.

2 A brief history of VCS

2.1 Ancestors

At the beginning, the main tools used to manage different versions of software were pretty primitive but it was enough for most people during early stages of development and large pieces of software were developed using SCCS or later, RCS.

Both SCCS & RCS uses the same basic principles to handle changes with a special directory in the working area, storing the files and the diferents revisions in a special format with a fundamental difference between the two: SCCS uses a special format called "weave" (see [5]) whereas RCS stores separate deltas: always storing the latest revision and storing differences down to the first one.

RCS assumes you will want a fast access to revisions close to what we will call the *HEAD*, the latest checked-in revision in the main line of de-

velopment, the main inconvenience being that as you add branches and tags, the backend storage file gets more complicated and the system will gradually slow down.

AT&T and CSRG at Berkeley both used SCCS to manage whole versions of UNIX^{TM} and one can find in many files the marker for SCCS[1]. The `what(1)` command is used to "reveal" SCCS strings in binaries. The author even used to embed the SCCS marker inside RCS `$Id$` strings to be able to use either `what(1)` or `ident(1)` from RCS on such binaries.

Both SCCS and RCS use a "locking" model where checkout means locking a file before being able to modify it. This model essentially works because it assumes that a given file will be modifed by only one person at some point in time. It is pretty easy to see that it doesn't scale especially with teams in different locations, as neither of them support remote operations. That means that sharing a tree can only be done using NFS or an equivalent sharing file system.

What is actually interesting is that among the currently available VCS (distributed or not), most of them use SCCS or RCS as the base for its design, either by copying the User Interface (UI) – SVN for example – or by extending it in different ways:

- Perforce uses the RCS file format with DB files for metadata
- BK is more or less a rewrite of SCCS to allow cloning of repositories/branches (See the announce posted on the `linux-kernel` list [6]).

There is another area where these venerable VCS don't work efficiently: binary file support. They are fundamentaly designed to cope with text files such as source code; binary files would be stored as "text" with all the potential loss of information and any command that tries to display or merge would generate gibberish on the screen.

2.2 The *Golden Age* of CVS

In 1986, Dick Grune created CVS with two of his students as a set of shell scripts over RCS in order to be able to work on pieces of a compiler independently[7]. His work was then rewritten in C by Brian Berliner and a paper published in 1990 at USENIX Winter Technical Conference[8].

At first, CVS didn't have remote operations and thus, to work on a given CVS tree, you would have to be logged on a machine with "physical" access to the tree (of course it could be through NFS[2]).

The main advantages of CVS apart from being free – a very useful feature in itself of course – at that time were:

- A central repository instead of a collection of small trees each with its own unsharable `RCS` directory
- A "no-locking" mode of operation, allowing concurrent access to the repository through extraction (also known as *checkout*) of a subset of the tree in *sandboxes* – a private workarea from which each developer can commit his work regularely and merge his/her modifications with others.
- A central repository enables the use of custom scripts (for sending commit logs by mail), access control lists and triggers.

For all these reasons and the fact that none others existed, CVS became the VCS of choice for many projects, both proprietary[3] and free ones like all the BSDs[4].

Soon, most if not all FOSS[5] began using CVS with the notable exception of Linux, as the Linus Torvalds disliked CVS so much that he refused to use an imperfect product[6].

Then CVS gained remote operation support (through either RSH, SSH or its own pserver mode) and its adoption by SourceForge[9] and similar *projects repositories* really pushed CVS in

[1] The marker is `@(#)` and the RCS equivalent is `$Header$`

[2] NFS: Network File System

[3] The author knows for a fact that the French telco company Alcatel has built its own VCS on top of CVS.

[4] All 4 of them, not counting TrustedBSD: FreeBSD, NetBSD, OpenBSD and more recently DragonflyBSD

[5] FOSS: Free and Open Source Software

[6] How he managed to not use a VCS for so long (1991 - 2000) keeps on baffling the author of the paper...

the open. The various bits of documentation available first on FTP sites then on the WWW, the books (see [10], [11] and [12]) and the simple but effective UI of CVS also helped a lot to lower the difficulty of using a VCS and thousands of people began using it.

2.3 CVS flaws

Nowadays, many developers confess to stumbling on one or several misfeatures or design problems with CVS. Its flaws are now very well-known:

- Commits are not atomic (i.e. there is no concept of a changeset[7]), the granularity being the directory: in a single directory, you get consistency between the various impacted files through a lock but if a given commit spans multiple directories, you are on your own: that's why the various convertion tools like Tailor[13] or `cscvs`[8] have some difficulties finding all files in a given changeset.

- You can not rename files and directories. The only way to achieve that is either by *delete+add* – which is very wrong and loses history - or manually edit the repository to copy or move the corresponding backend file.

- CVS has no fine grained access control facilities and needs to rely on filesystem permissions for many things although heavy users such as the FreeBSD project have developed customs wrappers and scripts to add per-branch ACL, review workflow and more.

- Directories are not versioned meaning that permissions and ownership is not preserved or kept but also that you can not have an empty directory. That's not a major inconvenient and people have been living without it for a long time but it is desirable.

- Branches are cumbersome to use, especially when you want to merge bits of this and bits of that from another branch. As the main entity versioned is the file and CVS has no memory of branching, you have to keep somewhere the exact revision for each file to be merged or use tags and/or dates to get this information.

- Third-party code integration and maintainance is very cumbersome[9] as it is done through a special branch called the *vendor branch*.

The last two points are critical for projects such as FreeBSD because we have to maintain parallel branches for STABLE/CURRENT developement and releases (and security branches). The weak support for branches is also something that slows down development in general. It makes working on specific sub-projects or features more complicated. That's why a few years ago, FreeBSD made the choice of using a different VCS for such cases: Perforce (see below).

Offline work is possible through *CVSup* (see below) but it is still very limited.

2.4 Enter Subversion

To remove all these limitations, SVN was born.

SVN is often cited as the natural successor to CVS and looking at it, it is easy to see why:

- It is written by former CVS developers
- It is touted as *CVS done right*
- Resembling CVS as much as possible is one of its primary goals (you can alias the `cvs` command to `svn` and get running in no time – ignoring the differences of course)

It has the same properties we all come to like in a centralised VCS with many CVS flaws corrected: atomic commits, easy and fast branching and tagging[10], triggers, ACLs and all that. It has also its own book [14] and the Pragmatic Programmers have written one too [15].

[7] See `http://en.wikipedia.org/wiki/Atomic_commit`

[8] A cvs-to-arch conversion tool. See `http://wiki.gnuarch.org/cscvs`

[9] The author is also maintainer for the ntp codebase integration in FreeBSD and suffers everytime it is time to upgrade...

[10] It is even the same mechanism here, a tag is just a one time branch

You can even have a DVCS on top of that through `svk`[16].

So what are the problems with SVN?

If we ignore the centralised part (we will look at these aspects in section 2.5), SVN has still some important shortcomings (although it gets better over time):

1. It is a big program, with some large dependencies (like Apache2 to get a web interface), `apr` & `apr-util` (both part of Apache2). Apache2 is *not* mandatory though, there are other ways to access a given repository (`svnserve` and `SSH` are available)

2. It used to require Berkeley DB as storage backend for everything (strings contained in the checked-in files, and so on) and experience has shown us that BDB is not stable enough (and in the early days, the DB schema was changing for almost every release which was painful). Current versions use FSFS as default storage method.

3. The way branches and tags are implemented, replication of a given repository would generate copies of entire files...

4. It is snapshot based as opposed to changeset-based (See [17] for a very nice description of both types of VCS) which scale less than the latter ones.

5. In addition to the previous point, it has no memory of what has been merged overtime (like CVS).

While both the first and second ones make integration with FreeBSD rather difficult and the third one surprises me (but is explained in [18]), the last point is the worst one. It complicates merging between branches and makes managing third party code (found in for example `src/contrib` in FreeBSD) more difficult.

SVK[16] could be a way to work around the centralised design of SVN but as it does not change the first and fourth problems and adds a Perl layer to SVN, making integration even more difficult although there are good things coming fromSVK: it is faster and uses less disk space for the working copy and SVK metadata.

However, when FreeBSD started to look at another tool, SVN was not ready and Perforce was choosen and we have been pretty happy with it; it helped a lot getting projects such as SMPng and others up and running (and finally integrated into the main CVS tree of course). We also have a regular export from the Perforce tree in a special CVS tree to allow people to see what is going on on these projects[11].

Perforce is a very nice *centralised* VCS (supports fast and easy branches, is fast and well supported) but in that respect, it is even worse than CVS: all operations are done through the network and one has to be connected to the server to do anything[12].

Another problem, as we will see in more details through section 2.6, is that it is closed-source proprietary software and that creates a questionable dependency for an open source project.

One last bit of information about CVS: after the last round of CVS security advisories, some OpenBSD folks have decided to rewrite CVS completely to get a more secure source code base, see the OpenCVS website[19]. Some of the enhancements planned a long time ago for CVS like atomic commits and rename seem to be forthcoming but it has not been released yet (Oct. 2005).

2.5 Enter the distributed VCS

In 1999-2000, Larry Mc Voy, formerly of Sun, Inc., started his own company called BitMover, Inc. around a new product named BitKeeper. BitKeeper (BK in short) was a distributed version control system based on the venerable SCCS[13],

[11] Note that it allows external people to get the code but does not give an easy way for them to hack on it and contribute back.

[12] The author used to be a Perforce user for 5 years but the lack of easy synchronisation between repositories became too much and he switched to Arch.

[13] The author's guess is that McVoy used it at Sun and felt it was a good starting point.

extending it to cope with repository cloning and merging.

Apart from being close to its ancestor SCCS, it brings the distributed aspect to revision control and with it a whole new way of working and sharing source code.

Up to now as we have seen earlier, developers had to share code either through a common tree (CVS, Perforce, and so on) or through the much more cumbersome way of generating patches. With BK it becomes as simple as cloning a given repository and start hacking on it with pull/push mechanisms to share code and patches.

With its vision of *a repository is a branch*, generating a branch is the same as cloning meaning that you can have as many branches as you want and that:

- They are cheap (so you can throw them away if not needed anymore or a dead end),
- It is easy to merge between all these branches as the system knows where the branch was created from and which changesets are present.

The concept is rather new and we should thank McVoy for pushing the limits for all VCS developers because it was the starting point of what we have now. BK really took the lead of DVCS when Mc Voy, for good or worse, convinced Linus Torvalds to finally start using a VCS for the Linux kernel.

2.6 The BK debacle

It was a big change for Linus (not so much for the developers' community as many of them had started using CVS for their own trees) and it also pushed many people towards using a DVCS.

Many people recognise that BK works well, is reasonably fast and it does the job[14]. These people also generally agree on two points:

- The license is one of the worst I've seen. Not only there are many unacceptable restrictions (like being prevented to work on developing any other VCS during the license validity plus one year and being forbidden to reverse engineer the wire protocol and product – something one can not forbid in the EU) but if you wanted a "free" license, you have to send all your commit logs to them,
- Worse: all the generated metadata that makes it interesting (like who branched what and when and all that) is considered as *proprietary* data by BitMover, Inc. even though it concerns a FOSS project![15]

Add to these the fact that Mc Voy has constantly been saying on several mailing-lists (mainly the `linux-kernel` one) how difficult it was to write such a product, how costly it would be and don't bother trying to reproduce it, it is too difficult and so on[16].

To add insult to injury, he also said that if anyone tried to reverse engineer anything related to BK, he would change the wire protocol and prevent people to do it.

In the end, what had to happen did: In April, 2005, Andrew "tridge" Tridgell, of Samba fame, tried to reverse engineer the wire protocol – which proved to be trivially easy thanks to BK itself – and BitMover decided to revoke all "free" licenses therefore putting Linus and other Linux developers in a difficult position[17].

I will not dive into Linux politics and what happened but we must see that the whole debacle was the driving force behind the current trend of DVCS and spurred development of many systems now available.

People are now aware of the problems and caveats of distributed developement and the solutions behind them. We now have several

[14]The author tends to agree even though there are some questionable things in it like the 65536 limit on the number of changeset – it is now fixed – and the heavy use of `system(3)` throughout the binary.

[15]Note that this is very different from Perforce: both are proprietary software but the format of Perforce's metadata is known and there is even a Perl `p5-VCS` module for it meaning that you are not locked by using Perforce.

[16]It is interesting to note that the development of Mercurial started in March 2005 and is now pretty close to BK featurewise in only six months.

[17]As it is, BitMover is still trying to stiffle the competition by forcing people not to work on free projects; a major contributor to Mercurial has been recently "asked" not to work on it as long as his company has a commercial BK license.

very interesting VCS, some close to Linus' own git[20] (cogito[21] and in some ways Mercurial) and many others, each with its own set of interesting features (Darcs and the theory of patches[22], Monotone[23], and so on).

The second consequence is that people are hopefully convinced that using a proprietary VCS as the main one is a *very* bad idea.

3 The FreeBSD context: figures and processes

The FreeBSD project started in 1993 just after NetBSD using 386BSD as its base tree. Originally planning to be 386BSD 0.1.5, it finally became FreeBSD as both Bill and Lynne Jolitz, the original authors, refused contributions and maintaining the various patches became too cumbersome.[18]

3.1 Figures

The current CVS tree is the second one we have been maintaining. Due to legal restrictions coming from the AT&T/BSDi/CSRG lawsuit, we were forbidden to keep on using and distributing the FreeBSD 1.x repository so a whole new tree was created in 1994 with the import of 4.4BSD-lite.

The whole repository was broken up into four a few years ago to be close to the organisation of committers: we have *src, ports, doc* committers and those who can commit in several or all categories. The *doc* committers includes those working on the www subtree. The following table lists the sizes as of mid-Sept. 2005:

Repository	Size (MB)	Directories	Files
doc	183	1653	6171
ports	903	43490	124338
src	1402	9030	60708
www	112	595	3479

I do not have figures about the number of changesets as the notion doesn't exist in CVS but when P. Wemm did some conversion tests back in 2000 during our evaluation of Perforce, we were already at more than 75000 changes in the Perforce converted tree. I estimate the current tree to have more than 200,000 - 220,000 changesets by now, all repositories considered (more on these figures below in 6.2). .

When using *CVSup*[19], all the repositories can be combined in a single one through symlinks as it is easier to work with. Note that of course, having the entire repository does not allow to commit (or it would completely mess up with the next *CVSup* run). One feature was added to *CVSup* to ignore a special branch and allow for local modifications while syncing the CVS tree but that is only a *hack*.

3.2 Development process

Today, the developement process in FreeBSD is pretty straightforward: committers have access to all repositories, the main difference between types of committers will reflect in the commit log. If a *doc* committer checks in a change in a manpage in src/share/man, the commit message will say at the top that it was done by a *doc* committer.

Committers are strongly advised to *CVSup* the repository on their local machines, edit, compile and test and then push to the real one by overriding the repository path. That way, the network and the CVS machine are not overloaded and we can keep disk space at a reasonnable level. Of course when a commit must be tested on the FreeBSD cluster with different machines and architectures and the committer doesn't have the local resources, local checkouts are allowed.

The central repositories are also responsible for sending the commit logs to the various mailing-lists (cvs-all has everything but there are also broken down for specific subtrees such as cvs-ports and doc); this is an important part of the process so any system aiming to replace CVS must be able to offer and support such features.

In day-to-day operations, we see CVS's flaws in action when we need to move things around (it

[18]The complete history of FreeBSD and its relation to the other BSD can be found on the web, I will not reproduce it here.

[19]CVSup: CVS-aware replication tool – http://www.cvsup.org/

can be because a port was not imported in the right place or in case of code reorganisation); we have some people called *CVSmeisters* that are specifically allowed to manipulate the repository and execute the unfortunately common *repocopy* operation[20]. That way, history is not lost.

It is unfortunate that we have to manually edit the repository fairly often[21] but there is no other way due to CVS's limitations.

The two products we are going to evaluate have ways of replicating a given repository to remote sites but we will keep on exporting all changesets in a CVS repository for easy duplication through *CVSup*, anonymous CVS usage and more generally because it is so well-known even by some non-technical people. The nice thing is that converters to CVS are not difficult to implement or find and it is easier to go from a changeset-based system to CVS than the reverse.

3.3 Release Engineering

The FreeBSD project maintains several branches in parallel to support our notions of STABLE and CURRENT trees. We also have security branches on which only security fixes are applied (this happen to all STABLE versions after they have been released) and they are supported for a limited amount of time (that varies from branch to branch and can be more than 18 months). We have also recently allowed non security fixes in the release branches (`RELENG_*`). To help release builds, we have some period of time during which the trees are either completely frozen or strictly controlled by the Release Engineering team (also known as the `re@` alias) and `portmgr` (for the `ports` tree).

Such freezes happen independently in the *src*, *doc* and *ports* trees but the goal stays the same: to be able to have a *stable* tree to cut a release from.

These procedures are somewhat of a necessary pain because CVS is not as we've said before very helpful with its branch handling (sometimes the trees stay semi-frozen for weeks). This is one of the main reasons not to branch the entire ports tree for each release: It would be taking too much time to tag the tree as every single file needs to be written into and we need to block everyone through various scripts we have developed over CVS.

Switching to another VCS requires these issues to be cleanly handled.

3.4 FreeBSD requirements

From the previous sections, we can extract a set of FreeBSD requirements that we want a future tool to handle.

- Atomic commits to get *real* changesets
- Easy & cheap branches (and merging) and tags to enable parallel lines of development (that is essential for projects like *SMPng* which have a very big impact on many source files)
- Fast system for common operations
- Ability to keep and distribute a "reference" tree, knowing that it should also be exported to CVS
- Ability to rename files within directories without losing history
- Ability to help simplify the way we handle releases (and freezes, slushes, ...) in order to avoid locking the trees.
- Ability to digitally sign revisions or repositories to avoid file corruption and to detect unwanted modifications
- Automated or mechanically assisted merging
- Ability to work *offline* – like on a plane – without requiring too much work: not only being able to list differences but also to commit

Most of these requirements can be met by centralised VCS but the second and last points are those pointing to a non-centralised or distributed VCS.

[20] It is achieved by `cp` the `foo,v` file from the old place to the new one.

[21] On the other hand, manually editing is faisible, which can save your day if you have a repository corruption.

4 Is Arch/bazaar suited to FreeBSD?

In this section, *Arch* is the "protocol" (for lack of a better word) designed by Tom Lord and both tla[24] and Bazaar are implementions of this protocol. Both implementions are compatible with each others (unless you specifically ask at creation-time for a `baz` archive which `tla` can not read) but Bazaar has the backing of a company (Canonical, Ltd.[25]) and is the only one currently maintained. Tom Lord has announced he was stopping all developments on both tla and revc[22].

Arch has some unique features among the DVCS:

- It is both a VCS and a cataloging system:

  Everything is divided into archives, whose name generally contains the email address of the developer like in `lord@emf.net--gnu-arch-2004`.

  Archives contain the equivalent to CVS modules named here *categories*. Whereas most DVCS use as many repositories as you have branches, *Arch* still uses a separate sandbox as workarea.

  Categories are the main work unit in *Arch*; they can be checked out for editing, branched (here it means both as a local branch and as a remote one) and versioned. Branches and versions are specified in the full name of the category, separated by "`--`" like in `calife--pam--3.0`

  The main inconvenience is that you must type a lot more to refer to something managed by *Arch*[23]

- Whereas many modern VCS try to duplicate the well known UI of CVS, tla has a lot (and I really mean *a lot*) of different commands for dealing with archives, categories, revision libraries, branching and merging and so on. Bazaar has tried to redesign the UI to be easier for beginners but still, the output of `baz help | wc -l` shows 187 lines...

- Both tla and Bazaar use weird-looking filenames for temporary subtrees and filenames (with ++ or ,, as prefix). The metadata directory in the sandbox is named `{arch}`. Most VCS use `.something` (and `_darcs` for Darcs[26]) to store that information. That is not a big point but it does confuse newcomers.

- *Arch* eats a lot of diskspace. In addition to the archive which contains directories of changesets, you will need space for the checked out categories and either a "pristine" copy of the sandbox (a gzipped-tar file) inside `{arch}` or a revision library (a complete tree of hardlinked files for most or all checked out/merged revisions).[24]

- *Arch* needs to uniquely identify all files managed by it so there are several ways to generate a unique id and to tell *Arch* what it is:

  names The filename is the *id* itself, it is obviously not the recommended way for normal operations

  explicit It is analogous to how CVS works, you use the `add` command to `baz` to attach an *id* to the file[25]

  tagline This one is special: *Arch* will look into the first and last 1 KB of each file for a special string[26] and use that as unique *id*.

This unique *id* enables *Arch* to track file/directories renames more or less automatically (which is nice) but also, in the *tagline* case, complicates third parties code as you are not really allowed to modify it.

[22] `revc` was supposed to be Arch 2.0 with a whole new storage backend (close to `git`), no more categories/branches/versions and a different archive format along with an heavy use of SHA-1 checksums everywhere.

[23] Fortunately, there are completion modules available for the common shells – `zsh`, `bash` and `tcsh`. Trust me, you can not live without such a completion module.

[24] To be fair with *Arch*, SVN has also a pristine copy of your files inside `.svn`.

[25] It will be stored in a special sub-directory called `.arch-ids`.

[26] `<comment characters>arch-tag: <unique-tag>` (people often use UUID[27] for that purpose: `/* arch-tag: b11c0274-29ee-11da-9b43-000d93c89990 */`)

All of these items makes *Arch* rather complicated to use, especially for beginners (in the VCS world) but really, I have been very happy with it for two years. I would even say that the namespace issue for categories forces users to think a bit more on how to organise things in an archive which is not without value.

If we want to use it for FreeBSD, there are several things that we need to look at, mainly because of the design of *Arch* and the whole category feature. Do we want a single category named `freebsd`, separated into branches (like `freebsd-current` and `freebsd-stable`) eventually with a version number or do we want to use a category per subtree?

This is a big point and one that will have an important impact on *Arch* speed because it tends to walk the whole tree several times during commit and other operations (that is called running an "inventory" in Arch-speak). A given category is pretty much independent, if you want to group categories to form a complete source tree, you have to use a special mechanism called *configs*: you have a category with a special file with all the other categories you want to include (pretty much like the `CVSROOT/modules` does for CVS.) Then you use the `build-config` command to extract all categories and create the tree.

The big problem that comes from configs usage is that as I said before, the work unit is the category. What it implies is that commit also works on categories, not on a source tree built with configs... *Arch* has a way to iterate on all categories coming from a config but:

- It is a bit cumbersome although you will end up with writing a lot of aliases or shell scripts to automate this,
- The changeset is not global either: you will have one commit per category.

The second point is pretty much a killer in my mind. If you want to do a sweeping change in `/usr/ports` for example, you want a changeset of the whole thing, not more than 12000 changesets... You also don't want 12000 mails to be sent to the `cvs-all` mailing-list.

Another subtle characteristic of *Arch*: when merging multiple changesets between archives, on the receiving end, there will be only *one* changeset incorporating all the changesets (named *patch logs* in *Arch*) and users will be able to see only the summary lines for each embedded changeset. If the sending archive is available, full commit messages can be retrieved of course. This is clearly different from BK and Mercurial where every remote changeset is included as-is.

Bazaar satisfies one of our requirements: every commit can be digitally signed with PGP/GPG, this is an important security feature.

Last but not least: Bazaar is rather complicated to build; it does not use the autoconf system but its own home-built system (called *package-framework*) and has dependencies on several external packages such as `gpgme`, `libgpg-error`, `neon` (for http/webdav access) and very recent versions of various GNU utilities (`patch`, `diffutils`). It does complicate its possible inclusion in the main FreeBSD tree. tla is not as complicated – although it does use *package-framework* as well – but tla should now be considered as dead (and probably not worth maintaining due to the above limitations).

4.1 Common operations

We will take `/usr/src`[27] to make most of our tests, knowing that it is a moderately large tree (checkout is around 448 MB) with more than 33000 files inside 3766 directories...

In order to avoid wasting too much disk space among developers, each of them having possibly several checkout copies lying around, we can define a *revision library*. This area will hold hard-linked copies of the checkout files and so only the modified files will take more space between all users. Of course it does eat space (but we hope to reduce the overall diskspace requirement) and all developers must configure their text editor to break hard-links to avoid corrupting this revision library.

At first, we will try to import that as a single category because we want changesets to span the

[27] The main problem is that Bazaar 1.5 keeps on dumping core on my FreeBSD 4.11 system when using `/usr/ports`

whole tree. To have Bazaar work as transparently as possible, we will use the *names* tagging method.

Operation	Time	CVS equiv.	Time
`baz import`	11:21	`cvs import`	4:18
`baz get src`	3:28	`cvs co`	14:43
`baz commit -s`	4:29	`cvs commit`	11:52
`baz status`	6:05	`cvs update`	5:22
`baz status`	3:33	`cvs update`	idem

NOTES:

- The first `baz status` command generated a revision library entry while the second one is just using it.
- The `baz get` command used the revision library to hardlink all files in it.
- For some operations, system limits (see `get/setrlimit(2)`) had to be raised (datasize in particular) or `baz` would dump core.

Bazaar is clearly faster than CVS but not by a large margin and some operations require multiple traversal of the whole tree (the inventory system) which slows it down. `commit` can take an optional list of file names to be considered by the commit itself but on a very large tree such as `/usr/ports` it is really painful to list all modified files. The correct method is generally to have a *wrapper* command around the actual command-line interface (CLI) that builds this list and hands it out to the tool when committing.

Disk space requirements must also be considered: If a given tree is N MB, it will generate N MB as a revision library entry and gzip(N) MB in the archive itself. Commits are stored as compressed changesets so it takes much less space. For each commit, a plain text version of all modified files will be added in the next revision library entry and the rest is hardlinked. Revision libraries must be pruned regularly of course as you'll accumulate revisions you'll probably never extract gain.

The alternative is to avoid using a revision library but then, Bazaar will generate a complete copy of the checkout files – called a *pristine tree* – below `{arch}` which does take as much disk space as the checkout tree...

4.2 NOTE

It must be noted that most of Canonical's effort has been recently concentrated on the next generation of bazaar: `bzr` *aka* bazaar-ng *aka* bazaar 2. Version 0.1 of `bzr` has just been released (Oct., 11th, 2005), incorporating a very important change in repository format: it is now using the *weave*[28] format instead of the full-text one previously used (the same as `git`). It is too early to really test `bzr` as it is pretty young and performance is still lacking but it is very promising.

This is an important change and one that will make Bazaar 2 much more interesting. There will be an upgrade path from Bazaar 1 to Bazaar 2 but they are completely different in design

5 Mercurial to the rescue

While working with *Arch* and trying to see how to overcome the limitations and design problems described in 4, I found *Mercurial*. Following what we have seen in section 2.6 and the appearance of Linus' `git`, exploring what have been started with Darcs and *Monotone*, Matt Mackall announced he had started to write a DVCS[28][29].

Another reason to look at Mercurial is that Bazaar 2 was far from being feature-complete (without even thinking about performance)
What is really nice about Mercurial is not so much its speed – although it is important and impressive – but the fact that in a few months, it has grown into a nearly-mature product, with almost all features you could ask for a DVCS[30].

Add to that:

- A very friendly and open-minded author

[28] See `http://bazaar.canonical.com/BzrWeaveFormat` for a detailed explanation about weaves.
[29] Mercurial was started because of the BK debacle according to the author.
[30] See the roadmap: `http://www.selenic.com/mercurial/wiki/index.cgi/RoadMap`
[31] Mailman interface: `http://www.selenic.com/mailman/listinfo/mercurial/`
[32] See `http://www.freenode.net/`

- A growing community both on the mailing list[31] and on IRC (*#mercurial* on the Freenode[32] network).
- A relatively small and portable system compared to others like Monotone or Arx[29]
- Written in Python[33] without too many external dependencies

You end up with something small, fast and easy to use and setup. As we will see in the timing section 5.2, its handling of large trees is adequate for most usage and it is evolving without breaking too many things from one version to the next (no repository format change is foreseen in the near future for example, something that other VCS have done on a regular basis).

Of course there are a few things that need to be implemented to have a complete system like:

- Better handling of binary files. You can put binaries in a Mercurial repo but you will not be able to use `hg export` to submit; the only way to do it is either to use the `bundle` command that create a binary version of a set of changesets or to use the `push/pull` mechanism.
- Better rename/move support. At the moment, history is preserved by the `copy/rename` operations but it is not available to the user so it appears to be lost[34]
- Better support for managing changesets within a repo: currently, there are different way to revert a changeset or a set of changesets (`undo` only reverts the last one). It means that if you make a mistake, it may become a bit difficult to undo it.
- Support for digital signature of commits (most of the infrastructure is there but needs to be completed and on by default)
- Full permissions are not versioned except for the 'x' bit. Permissions are kept but if you change a file from `600` to `664`, it will not be not taken into account.
- Lack of Internationalisation (i18n) support. It is necessary to lower the entry bar for many people.
- More documentation

All these should be corrected for the *1.0* release around December 2005.

All these reasons made the author choose Mercurial first for his own usage and second to include it in the scope of this paper. The rather fundamental technical differences between *Arch* and Mercurial designs do not have a big impact on section 6 about processes and policies changes that are needed when moving from a centralised to a distributed VCS. These differences will have an impact on the technical side of the migration and setup of course.

These main differences between Mercurial and *Arch* are:

- *Arch* follows the traditional design with one side the archive/repository and on the other side the working trees/sandboxes
- Mercurial does not force a specific namespace on repository and module naming (like *Arch* does in `archive/category--branch--version`)
- In Mercurial, there is no inventory like the one done in *Arch*, no tagline/explicit/implicit/names method of include/exclude files from being versioned.
- The work unit is the tree/branch, not a subset of it

[33]While Python is not the preferred language of the author of this paper, it is easy to understand and thus to contribute [to the project]

[34]This is true as of version 0.7 released on Sept, 16th, 2005

5.1 Technical specifications

Mercurial shares some common characteristics with the other available DVCS:

- A repository is a branch (this is a simplification as you can have several branches within a given repository)
- The working directory is the repository, there is no *sandbox* like in CVS or SVN
- Branches are cheap and the main way to replicate (called *cloning*) repositories
- You can lay down tags on a given revision but with a twist: tags can be either local or global, the latter means that if you clone a repository, you will get the tags along the way.
- You must have a *merging* tool like `kdiff3` or `tkdiff` to handle any conflict during merging. It must be noted that merging is done on a separate branch within the repository first then you merge the result with your own local changes. This approach generally lowers the number of conflicts when dealing with external sources.
- It has an integrated *CVSweb*-like interface, either through a CGI script or through its own `hg serve` command.

It has also an interesting technical feature, shared in principle by *Arch*, the various files in the `.hg` tree are append-only. That means that it is a bit more robust (compared to the RCS file format where everything including tags are stored in a single `,v` file) and that going back to a previous revision is done through simply truncating the file.

The storage method used seems to be pretty efficient, specially when compared to the default `git` backend where full files are stored for a given revision and various tests done by the author ([30] for example) shows the differences. I do not believe that the fact that hard disks are now cheap is a good reason to waste that space.

Something interesting has been available for Mercurial for quite some time: an extention to manage "stacks" of patches has been written. This extention, called `mq` does something similar to `quilt`[35]; it allow to manage a series of patches by keeping track of the changes each patch makes. Patches can be applied, un-applied, refreshed, etc.

5.2 Tests timing

We take the same `/usr/src` tree to make comparisons with *Arch* and CVS.

Let's assume we want to put `/usr/src` under Hg, discarding the previous CVS history for the moment[36].

Operation	Time	CVS equiv.	Time
`hg clone src`	3:09	`cvs co`	14:43
`hg commit -A`	5:12	`cvs import`	4:18+14:43
`hg commit -m`	0:09	`cvs commit`	5:32
`hg status`	0:06	`cvs update`	3:30

NOTES:

- `clone` and `co` don't do the same exact thing as there is no history in `co` case.
- `cvs import` creates the repository but we need a checkout to work; Mercurial doesn't need that phase as the working directory is the repository.
- `cvs update` is not strictly equivalent to `hg status` but `status` is much more verbose.

It is *very* fast. It is fast enough that we don't really care about trying to use sub-trees (see 6.1) as it gets more complicated to submit patches.

As Mercurial can handle the `/usr/ports` tree, here are some timings:

Operation	Time	CVS equiv.	Time
`hg clone ports`	9:18	`cvs co`	16:35
`hg commit -A`	10:34	`cvs import`	4:41+16
`hg commit -m`	0:39	`cvs commit`	11:52
`hg status`	0:52	`cvs update`	5:22

Even on a much larger tree – `/usr/ports` is more than 124,000 files in more than 32300 directories – Mercurial manages to stay fast.

[35] `http://savannah.nongnu.org/projects/quilt`

[36] Due to CVS design and misconception, converting a whole tree is rather complicated and *very* slow.

Repository overhead is small too, although on pathologic cases such as /usr/ports with a lot of very small files, the fact that you have *two* files for each versioned files is showing:

Tree	Size	.hg size
/usr/src	417 MB	227 MB
/usr/ports	430 MB	358 MB

The nice thing is that as the trees will accumulate revisions, the way Mercurial does store changesets is very efficient: if the delta between the next version and the original is bigger than some amount, the new version is stored compressed in its entirety. It ensures than we don't need a huge amount of data to reconstruct *any* version of a given file. Add to that the fact that all files below .hg are append-only, you have a repository that can resist corruption better than others.

The author of this paper would have liked to do more speed comparisons, especially when working with older branches (an area where CVS is rather weak as it needs to go back and forth with in the repository to reconstruct a branch) but the difficulties with repository conversion prevented that. The author would like to point out that due to its design, Mercurial would probably shine in that respect because generating an older branch is consist of merely cloning (something which is fast) the given reference tree...

6 How to get this to work: processes and policies

A tool, however powerful, is not enough to support a whole project running and do it that way in a reasonable form, especially when dealing with volunteers. A project this size (more than 300 people, working around the world with different timezones) has to have some kind of processes and policies.

Since the beginning of the FreeBSD project, everything has been built upon CVS and upon its features and flaws. In particular, almost all the constraints we have now for Release Engineering and the whole set of policies of freezes, slushes and al. have their roots in CVS in one way or another.

It is the opinion of the author of this paper that it is time to review them, classify them as CVS-specific (or not) and see how they would have to evolve if the FreeBSD project was to switch its VCS over to Mercurial[37].

A distributed or decentralised VCS enables a more parallel way of working, facilitates working in different trees and branches without the fear of a complicated merge and without playing with patches. The fact that it enables offline work is a very much needed feature; likewise, merging from offline trees is no different from merging different branches and is as easy.

We should also note that some software would have to be written or changed to adapt to the new VCS as some assumptions coming from a CVS-oriented world are not true anymore: a central server with all the related aspects like pre-commit checks, post-commit triggers and so on.

6.1 FreeBSD environment

If we look at the FreeBSD requirements in 3.4 and try to answer them, we will see that the first three are easily met by Mercurial as they are part of its design. The last one is the key point what we will concentrate on. The point here is not to disturb the developers too much.

Let's see what would be needed to reproduce a CVS-like environment:

- A "reference" tree that people can clone from just like people use *CVSup* now to get the *official* source tree.
- A way to handle either merge requests from the various developers or a way to queue patches sent through various sources (email for example) for integration in the "reference" trees *aka* a patch queue manager[38].
- A way to generate commit messages to be sent to various mailing-lists; if we have the above request satisfied, then the patch

[37] Note that most of what we will say there is applicable to any distributed VCS, the key here is *distributed*.

[38] Like the one used by Canonical – http://mirrors.sourcecontrol.net/robert.collins@canonical.com-general/

queue manager (PQM) is the obvious candidate for this.

When one wants to make some modifications to a given tree, he/she will clone the repository, hack on it and then submit the changesets to the PQM. One easy way to do this is to have a cloned tree that is only updated through `hg pull`, serving as a *local* reference tree and the developer will clone this one at will for specific purposes.

To maintain coherency with these cloned trees, he/she will regularely merge from this reference tree into the other ones. This is where having a fast VCS is interesting because having to wait half an hour just to be able to edit something is not really productive.

Even though Mercurial is fast for cloning big trees, it still takes some time. A possible solution to this problem could be to create sub-trees on demand: when you want to do a small modification, you just go to the sub-tree, `hg init` followed by `hg commit -A` to *import* the sub-tree in a little repository. It is then very fast and easy to generate a diff and submit it to the main tree (then forget the sub-tree with `rm -rf .hg`). Of course, it would be for small modifications which don't require you to keep the repository.

As for the patch queue manager, PQM has already been modified to work with Bazaar 2 and ArX so I think that adding Mercurial support should not be too difficult.

One area where things become easier is *Release Management*: there is no real need for ports/src freezes/slushes as it is just a matter of cloning the "reference" repository into a branch/release one, making it available through the PQM and use different rules for merging.

Likewise, there is no need to manually edit the repository, no more repocopies or tag sliding[39], thus simplifying the whole repository administration.

6.2 Repository conversion

The problem of converting the history of the project is a complicated one as the tools we are coming from and the one we would use are completely different in several ways, the major one being that having a repository for each branch complicates conversion as the tool used for that should be aware of branches and should generate a different repository when it comes across a branch tag. At the moment, none of the available tools support that. *Tailor* can convert whole branches into a given repository but we would have to manually do it for each branch starting at the branch point. It can only follow a given path, not descend in other branches.

Other complicating factors includes encoding of commit messages (do you want to convert everything from ASCII or ISO into UTF-8 or UTF-16?), tags (the notion of tags varies betweeen VCS, ...

To give everybody an idea of what repository conversion is about, the `/usr/ports` tree already mentioned has 138696 changesets (mid-October 2005) which is a *lot*. `/usr/src` is around 117233 changesets. Last time the author tried to convert `/usr/src`, it took 3 hours for less than 700 changesets (estimating the total time is left as an exercise for the reader) and consumed close to 1.2 GB of memory.

7 Conclusions

It is clear for the author that such a migration should be carefully planned over a few months and that the different issues mentioned before should be fixed. Mercurial itself is still lacking features but is evolving quite fast. Other tools were outside the scope of this paper and maybe should be evaluated but at the moment, only Mercurial has enough features and is stable and fast enough for our purposes.

The infrastructure still needs to be written or adapted to Mercurial and the big question on how to import the previous CVS history is something that should worked upon.

The problem with the repository conversion tools may mean that we would have to maintain both CVS and Mercurial as long as we support older branches or finding a way to partially convert them.

[39]Tags are static in CVS and references a specific file revision. When preparing a release, a critical bug can be fixed and the release/branch tag be modified to reference the fixed revision. This is manual intervention in the repository.

A not-so-minor point to consider: Mercurial is not written in a language that we have in core FreeBSD so the question becomes do we want Python in the core OS or can we accept that our main VCS will force people to install Python from ports. From a pure maintainability point of view, ports is easier.[40]

The learning curve of the new tool and the new ways of working are also important. The UI is a big part of that and Mercurial tries to mimic the old but well-known CVS one (where applicable of course).
Meanwhile, there are some advantages coming from the distributed part of the new tool:

- There is no need to have such complicated pre-commit and post-commit tools such as the ones we have now in `CVSROOT`, the PQM will manage all that.
- You don't need a central server with SSH keys, Kerberos or any form of access control; people just clone the "reference" repository and work from that. Access will still be needed at the PQM level of course to distinguish between committers, developers and users.
- The new capabilities of Mercurial could open the way to new working-styles like task-oriented branching and merging (like we do in Perforce now). With a possible link to the bug report database, we could think having a PR automatically closed when a task is done.

It is the wish of the author of this paper to help the FreeBSD Project to start thinking about a possible switch. It will be up to the FreeBSD Project to decide whether this is a worthwhile project to engage ourselves into of course.

8 Thanks

The author would like to thank Phil Regnauld, Mark Murray, Anton Berezin and Robert Watson for reviewing this paper over a rather short period of time. Much appreciated folks!.

References

[1] Various authors at Canonical, Inc., *Bazaar, an Arch implementation.* `http://bazaar.canonical.com/`.

[2] Matt Mackall, *Mercurial, a distributed SCM.* `http://selenic.com/mercurial/`.

[3] The FreeBSD Project, *FreeBSD, The Power to Serve.* `http://www.FreeBSD.org/`.

[4] Inc. Perforce, *Perforce, The Fast Software Configuration Management System.* `http://perforce.com/`.

[5] Mark J. Rochkind. The source code control system. In *IEEE Transactions on Software Engineering (Vol. SE-1, no. 4)*, December 1975.

[6] Larry McVoy, *SCCS & Source mgmt.* 1997. `http://lkml.org/lkml/1997/5/23/105`.

[7] Dick Grune, *Concurrent Versions System CVS.* 1986. `http://www.cs.vu.nl/~dick/CVS.html#History`.

[8] Brian Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[9] Sourceforge team, *Sourceforge software development hosting system.* `http://www.sourceforge.net/`.

[10] Karl Fogel and Moshe Bar. *Open Source Development with CVS.* Number 1-932111-81-6 in ISBN. Paraglyph Press.

[11] Andy Hunt Dave Thomas. *Pragmatic Version Control Using CVS.* Number 0974514004 in ISBN. Pragmatic Programmers, 2003.

[12] Per Cederqvist, *Version Management with CVS ('official' manual).* `http://ximbiot.com/cvs/manual/`.

[13] Lele Gaifax, *Tailor.py, A tool to migrate changesets between VCS.* `http://www.darcs.net/DarcsWiki/Tailor`.

[40] Tcl and Perl were both at some point in time part of core FreeBSD and they were both removed.

[14] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. *Version Control with Subversion*. Number 0-596-00448-6 in ISBN. O'Reilly, 2004.

[15] Mike Mason. *Pragmatic Version Control using Subversion*. Number 0-9745140-6-3 in ISBN. Pragmatic Programmers, 2005.

[16] ChiaLiangKao, *svk, a decentralized version control system, using Subversion*. `http://svk.elixus.org/`.

[17] Martin Pool, *Integrals and derivatives*. July 2004. `http://sourcefrog.net/weblog/software/vc/derivatives.html`.

[18] C. Watson B. Robinson, J. Hess and ISHIKAWA Mutsumi, *Debian X Strike Force Hackers' Guide*. `http://necrotic.deadbeast.net/xsf/XFree86/HACKING.txt`.

[19] OpenBSD, *OpenCVS, a FREE implementation of the Concurrent Versions System*. `http://www.opencvs.org/`.

[20] Linus Torvalds, *Linus' own version control software'*. `http://www.kernel.org/pub/software/scm/git/`.

[21] *Cogito, a version control system layered on top of git*. `http://www.kernel.org/pub/software/scm/cogito/`.

[22] David Roundy, *Theory of patches*. `http://www.abridgegame.org/darcs/manual/node8.html#Patch`.

[23] *Monotone, a free distributed version control system*. `http://venge.net/monotone/`.

[24] Tom Lord, *tla, a revision control system*. `http://www.gnu.org/software/gnu-arch/`.

[25] Ltd. Canonical, *Canonical main web site*. `http://canonical.com/`.

[26] David Roundy, *Darcs, a revision control system*. `http://www.darcs.net/`.

[27] ISO (International Organization for Standardization). *Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)*. ISO organisation, 1996.

[28] Matt Mackall, *Mercurial v0.1 - a minimal scalable distributed SCM*. `http://www.ussg.iu.edu/hypermail/linux/kernel/0504.2/0670.html`.

[29] *ArX, an easy to use distributed revision control system*. `http://www.nongnu.org/arx/`.

[30] Matt Mackall, *Patch: Mercurial 0.3 vs git benchmarks*. `http://lwn.net/Articles/133594/`.

Porting NetBSD/evbarm to the Arcom Viper

Antti Kantee
<pooka@cs.hut.fi>

Helsinki University of Technology

ABSTRACT

NetBSD is best and foremost known for its portability. This paper examines that claim in the light of porting NetBSD to an ARM XScale-based single-board computer. The paper starts with a general discussion on NetBSD code organization and attempts to identify components common to all porting tasks. After that this particular porting effort is investigated in more detail, outlining what needs to be done to add support for new hardware to NetBSD/evbarm and describing the problems and respective solutions in the effort.

1. Introduction

The definition of *port* in NetBSD is a very loose one. Sometimes a port consists of only a single type of machine with self-contained CPU support, such as *NetBSD/pc532*, sometimes it can consists of a single machine with "outsourced" CPU support such as *NetBSD/shark*, or sometimes it is simply a collection of hardware that seems to fit nicely under a common umbrella, say *NetBSD/hpcsh*. In the good old days all machines under one port used to be able to boot a common kernel and use auto-detection to figure out what kind of hardware was available, but in modern times things are different. In "consumer hardware", such as PCs or Macs the ability to have a single distributed kernel for all machines is still important, but in ports featuring embedded systems, such as *evbarm*, a special-purpose kernel for each machine is acceptable.

Adding support for a completely new CPU has been discussed in the AMD64 porting effort [1] and a similar effort of adding machine support where CPU support exists already has been documented in adding support for the JavaStation to NetBSD/sparc [2]. This paper's contribution, in addition to being full of (non-)amusing anecdotes, aims to be to explain the porting effort and outline the involved steps in terms which are hopefully understandable to an audience without experience in kernel hacking.

It should be noted that the discussion is quite specific at some parts, and can be guaranteed to hold only for the evbarm port of NetBSD, and even there sometimes only for the Viper hardware support. Other ports have other conventions as dictated by the hardware and do things differently. Some parts do apply to NetBSD in general, but no attempt is made to separate the specifics from the generics.

1.1. The hardware

The Viper is a single-board computer built around the PXA255 Intel XScale™ RISC processor. Features hardware include a CompactFLASH socket, TFT connector, audio device, serial ports, USB, 10/100BaseTX Ethernet, PC/104 expansions bus and the usual set of GPIO pins. However, at this date NetBSD supports only the necessary hardware for bringing the system up to multiuser in an NFS-root configuration. As is typical for an ARM-based system, there are no fans or other noise-generating components involved. In other words, for home use the system would make a nice MP3-player, even though the physical size is quite large by modern standards. Of course professional use is another completely different story.

1.2. NetBSD/evbarm

The NetBSD port for ARM evaluation boards is quite simply a collection of mostly indepent hardware support for development and prototyping boards which feature some version of the ARM processor[1]. These independent pieces of support code consist of:

- low-level startup. This code is written in the assembly language and takes care of setting up an acceptable memory mapping for the rest of the bootstrapping process.
- machine-dependent C-language initialization routines. It is the responsibility of these routines to set up the console, initialize the memory management information in the CPU-specific pmap and machine-independent UVM ready for prime-time, coerce the CPU into the mode we want it to operate in and initialize machine-dependent vectors.
- device driver frontends. NetBSD has most hardware support readily available in machine-independent format. Only a small amount of glue code is required to attach the MI driver to specific bus behind which the device (or a bus itself) is sitting.

In addition to the machine-specific implementations, an important part of the port is the shared ARM code located under *sys/arch/arm*. It is this shared code that enables to add support for a new machine with relatively minor effort.

2. The Porting Effort

All efforts for writing support for hardware consist more or less of the following tasks:

- Locating documentation for the hardware, reading the documentation and understanding the documentation. As usual, since time is of the essence, there is a great danger of trying skimp at this stage. It will have consequences later on[2].
- Creating a kernel configuration file, and if required the necessary auxiliary files to match the set of hardware that should be supported.
- "Filling in the blanks", i.e. writing the necessary glue code where required.
- Setting up the development environment. This includes building a cross-compiling toolchain, setting up a place where the system can boot from (usually over network), cross-compiling the target system kernel and userland and configuring the system firmware to fetch and execute code from the desired location. These actions will not be discussed in this paper any further.

2.1. Documentation

The documentation for the Viper consists of documentation which deals with the specific hardware [4], generic documentation on the ARM ISA [3] and finally documentation dealing with the XScale [5] processor family. Contrary to the usual situation with modern hardware, the above mentioned documentation is available from the Internet without any need for NDAs or other lawyerly trickeries.

In addition, documentation for various chips on the evaluation board can usually be found from the manufacturers of those chips. A popular trick for accomplishing this is to punch the chip number into Goo^H^H^Ha search engine and see what happens. Usually the documentation will present itself. Sometimes documentation for the original chip might be difficult to find, but it might be possible to find the documentation for another chip which is compatible with the one that documentation is sought for.

Finally, it is a good idea to know how the firmware works to get a kernel loaded and the kernel execution under way. The Viper features RedBoot firmware, for which documentation [6] is available from the Internet. In light of the porting effort it is also nice to provide the relevant information for users in the NetBSD Installation Guide. This way potential users have a consistent set of sources and matching documentation available from one source and do not have to go hunting around the Internet for various bits and pieces to figure out how to get the system up and running.

[1] It can be likened to a support shelter for various homeless pieces of hardware with no other place to go ... if that is a comparison I am allowed to make.

[2] For example, if your question is answered with the words: "read ARM ARM [3] Chapter 1.1.1.", you know you should have read the documentation more carefully. It is left as an exercise to the reader to figure out if this is a purely fictional example.

Entry to NetBSD

```
/*
 * You are standing at the gate to NetBSD. --More--
 * Unspeakable cruelty and harm lurk down there. --More--
 * Are you sure you want to enter?
 */
mov     pc, r8                              /* So be it */
```

2.2. Writing the Configuration File

BSD systems decide what to include in kernels and how to probe the device-tree with the help of a configuration file [7,8]. A good idea for creating a configuration file for some specific machine is to copy a similar configuration file and modify that. In this case the definition of "similar" was "another board based on PXA255".

The only piece of information specific to the Viper in the current supported hardware configuration is the Ethernet controller:

```
# SMC91C111 Ethernet
sm0 at pxaip0 addr 0x08000300 intr 0
```

The above tells the system autoconfiguration that *sm0* can be probed as a child of the *pxaip0* bus at address *0x08000300* and that the device will interrupt at interrupt level *0*. This information will be used by the driver. Note that in this special case, since, as we soon shall see, the networking driver frontend is written specifically for the Viper, providing the address and interrupt in the config file is not strictly necessary. The information could be hardcoded into the driver as well. However, since using the configuration file to contain configuration information is the correct approach as opposed to hardcoding the information into various drivers all around the source tree, we use that approach.

In addition to the configuration file itself, the following files need to be modified:

- *conf/files.machname*
- *conf/mk.machname*
- *conf/std.machname*

The configuration file itself makes sure that versions of the above files specific for this hardware are used by including *std.viper.*

files.viper

This file specifies all the devices and source files specific to the Viper. Currently it tells the system to include the file *viper_machdep.c* in the kernel and, as presented below, informs of the possibility to attach the *sm* driver at the *pxaip* bus:

```
# SMSC LAN91C111
attach sm at pxaip with sm_pxaip
file arch/evbarm/viper/if_sm_pxaip.c \
                                sm_pxaip
```

The information about the *sm* driver is a system-level counterpart of what was written in the configuration file and keeps the config-file author from having to know details about the actual implementation of the driver.

mk.viper

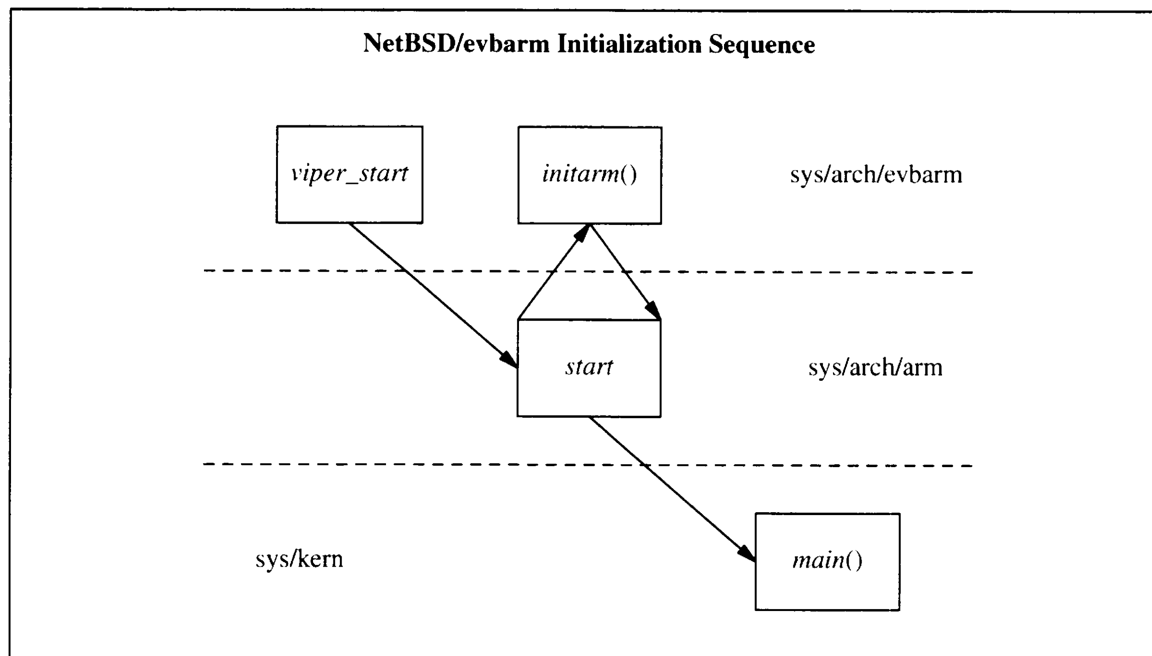
As the name suggests, this file deals with the viles of the kernel Makefile framework. The current significance is specifying the kernel base address and object format copy used in linking the kernel image. In addition, the first object file linked into the kernel image is specified here. This object file should, at the beginning of it, contain the kernel entry point.

std.viper

The file contains the standard config options you cannot live (or run) without. These include for example support for execution of *ELF* binaries and the config option for specifying that the system uses the PXA-specific interrupt handling implementation.

2.3. Low-Level Startup Code (*viper_start.S*)

For a programmer with previous experience in programming XScale CPUs, writing the low-level code is mostly a walk in the park. In case you are not familiar with ARM assembly, the MMU and such, it is more involving, but only slightly. By no means can it be likened to black magic.



The level of ARM assembly programming prowess required for writing the first stage initialization code is not very spectacular. One has to have the knowledge on how to read and write memory, do simple arithmetic-logical instructions, and how to write loops and jumps. In addition one needs to know how to talk to the the MMU. This level of skill, of course, results in non-optimal code, but a microsecond more to the boot time and a few hundred bytes more code are probably not an issue. If they are, feel free to increase your skill levels beyond the description given above.

Most evbarm machines accomplish the low-level init by first turning the MMU off, jumping to the physical memory address of the loaded kernel, and then proceeding to configure the memory map and other CPU information, and finally turning the MMU back on. This is a good way to proceed, since there is no need to worry about protection levels, cache flushing and other complex issues related to the MMU. However, this approach did not work for some reason on the Viper, no matter how careful one tried to be.

The problem, or more specifically the effect, was the system going totally dead after turning the MMU off. This might have been a simple bug in the physical address calculation routine, which made the kernel jump to hyperspace instead of jumping to the physical address of the code as it was supposed to. Or it might have been some really complex interaction between multiple variables. Nevertheless, the author was left perplexed after around 200 attempts to work around the problem. Finally, it was decided to do the initialization with the MMU on.

The decision of doing the initialization with the MMU active morphed the steps of low-level init into the following:

- Build an identity VA mapping for VA == PA. It is easier to copypaste code[3] from other sources later on if this assumption holds.
- Map the system physical memory to `0xc0000000` and up. For 64MB of memory present on the Viper, this spells mapping memory up to `0xc4000000`.
- Map devices used during the bootstrap process.
- Relocate the kernel to the location we want it at.

Building mappings

The XScale MMU deals with memory mappings on multiple levels. The first level, or L1, page table entries are 1MB, or 0x100000 bytes, in size while L2 entries describe a single page of memory. This standard multilevel approach makes it possible to map large chunks of memory easily and with little overhead (both space and coding effort) while still allowing for a

[3] Yes, so we all sin every now and then.

fine-grain per-page description. The page table is described on the XScale by a continuous set of memory, with the first four bytes in the table describing L1-sized chunk of memory at at virtual address `0x0`, the second the virtual address `0x100000`, and so forth. For L1 mappings, each entry contains the significant bits of the physical address in addition to the protection levels of that particular table entry. For L2, the physical memory address of the relevant L2 descriptors is indicated. The mappings are modified by modifying the memory contents of the correct offset in the page table. And of course it is possible to instruct the MMU to switch to use a different mapping table located at a different address.

Per the NetBSD convention, we load the kernel at `0xc0200000`. Per the same convention, physical memory is also mapped starting from `0xc0000000` onward. This leaves room to allocate bootstrap memory from the two megabytes before the kernel load address. Per a different conversion, physical memory is assumed to have an identity mapping during the C-level bootstrap process. So we map also from the virtual address `0xa0000000` onward to physical memory.

To be able to use various devices during the startup sequence, or at any point during execution for that matter, the devices need to be mapped into the memory so that accessing them is possible. During later stages of execution mapping devices in and out is fairly easy, since we have C-level convenience functions available for managing the mappings, but during the very first steps we must manually build the necessary mappings in assembly. Technically it would be enough to map a few simple bytes of memory window to operate the devices. However, since dealing with them is a fuss in assembly, we map an entire L1 entry for each device[4].

Kernel relocation

Kernel relocation sounds much more difficult than it actually is. It simply involves just a size calculation and a load-store loop. The kernel image consists basically of text followed by data. We know that the kernel entry point is at the beginning of the kernel[5] and we also know that the end of the data segment is marked by the symbol `_edata`. The size of the kernel image for copying is a simple operation: end_address - start_address. This is rounded up to the next four bytes, since the load-store loop is done one word at a time.

And now for something C-pish

After having done all machine-specific initialization, *viper_start* calls *start* located under *sys/arch/arm*. This routine is responsible for setting up an initial stack for running C code and calling *initarm*(), which once again is a routine specific to the Viper.

2.4. Low-Level debugging

Another annoying part of writing the low-level init is that no console is available, and one must resort to various forms of trickery for debugging. A popular approach is to blink LEDs attached to the system to give hints on where and how the code is executing. The only downside to this method is that one must be bothered to connect the LEDs to some available ports and also to figure out a way to toggle those ports.

The RedBoot firmware is nice enough to contain a debugger, namely gdb over serial. This means that instead of taping LEDs to the back of the board, most cases can be solved by simply examining the mess at hand in gdb. Because we are not yet running C code, the mess will simply present itself in assembly language. A useful trick is to load idle machine registers with relevant information on system state and upon a crash (or explicit *bkpt* instruction) examine the system state with the gdb command `info registers`.

It should also be noted, for sake of being complete, that you need to run a version of gdb compiled for the target system, not the host system. Luckily NetBSD makes this easy, and you can build a cross-gdb simply by giving the argument `-V MKCROSSGDB=yes` to *build.sh* [9] when building the cross toolchain used for development.

2.5. *initarm*()

After a long struggle with the assembly language low-level init, the platform-dependent C initialization code was almost a piece of cake to

[4] And feel slightly guilty about wasting many megabytes of virtual address space... well, no, not really. We'd much rather feel guilty for eating creme brulee with chocolate sabayon.

[5] That's what we specified in *mk.viper*.

handle.

First of all, the device mapping we generated in *start* must be described to the C code so that the correct mapping can be built later on. This is done by building a table of *pmap_devmap* structures, each entry containing the virtual and physical address, mapping size, protection level and caching attributes; the contents are similar to what we used in the assembly code for building the device mappings.

Second, we want to initialize the console device so that we finally gain the ability to do debugging-by-printf. After having mapped the console serial port to memory, attaching the console is a job of calling the machine independent *comcnattach*() to specify where the console port is found and what its parameters are. If all goes well, the console will work after that. And lo, there was *printf*.

After this, *initarm*() performs memory management initialization. Since that code was refactored in[6], discussion on it will be skipped. The code looks straightforward, but since the code was not written by the author, authorative comments on how straightforward it was to write and get working originally cannot be made.

As its final act, *initarm*() returns the new stack pointer, which is put into use by *start*. Finally, *start* calls the machine independent kernel entry point: *main*(), which takes care of the rest of initialization tasks, such as device autoconfiguration based on information in the config file.

2.6. Networking

Although it would have been entirely possible to include a root disk image in the kernel and therefore accomplish a "full" boot, a system these days is not really usable without networking, so the development direction was adding networking support.

The networking chip in the Viper is an extremely common chip designated "SMC91C111". Support was merely a question of a frontend for the existing driver in *sys/dev/ic/smc91xx.c*.

The frontend driver is divided into two interface functions: a *match* function which tells the system autoconfiguration if the probed device is present and an *attach* function which readies the driver (but not necessarily the device itself) for operation.

[6] Which is just a really fancy way of saying that it was copypasted.

Match & Attach

The easiest way to write a match function for hardware that is always present is to return success in all cases. This is also the common lazy idiom for writing *match*-functions for devices which are non-detachable and somehow non-detachably integrated into a certain system. However, this is possible of course only if *match* is called for only the device in question. Since the PXA interrupt controller probes through all the devices under *pxaip*, matching every caller as the network device is not a good idea; the interrupt controller did not function very well as the NIC. This was fixed making the match check against the device physical address before deciding if it was the right driver for the job. Since the frontend driver is currently specific for the Viper, this is something we are allowed to do, even though it might not be considered as something very pretty to do.

The attach routine contains three parts. First of all, *bus_space_map*() is called for the device to generate a *bus_space_handle* for it. Second, *smc91cxx_intr*() from the MI driver is established as the interrupt handler by calling *pxa2x0_gpio_intr_establish*(). Finally, the frontend is attached to the MI driver by calling *smc91cxx_attach*().

The Trouble with Tuples

As fate usually has it, even though the driver should, according to theory, have been working flawlessly after writing the frontend, NFS mounting root was still not successful. After running tcpdump the problem was revealed: `ff:45:67:64:2e:ef > 01:ff:ff:ff:ff:ff`. The query was not properly broadcast to the Ethernet broadcast address because some mysterious "01" had managed to mangle itself into the middle of the packet. But the mysteriousness of the mystery became much more fathomable once the on-wire Ethernet protocol was recalled after some hours of banging ones head against The Wall: on-wire the destination comes before the source, tcpdump just decides to print them the other way around.

This revelation lead to careful analysis of the Ethernet chip documentation. The length of

an Ethernet frame is specified to the hardware by writing the frame's length in 16 bits to the chip prior to writing the packet contents. The original MI part of the driver did this write in two one-byte pieces. However, due to the 16bit bus on the Viper, the chip got two 16bit values instead of two 8bit values containing the lower and upper bytes. The chip proceeded to interpret the high-order byte of the length as data bound for the network and a chaos was ready to ensue. Changing from two wrong writes to one right write fixed the problem.

Buffer Space, The Final Frontier

Having root on NFS places a fair deal of stress on the networking subsystem right after mountroot. This is because NFS tries to send maximal size UDP[7] packets to transport the binaries to the client system.

Some Ethernet chips have only a tiny amount of buffer space available, such as the 8kB specimen on the Viper. If the buffer is filled before the operating system has a chance to offload frames from the Ethernet chip into operating system memory (there is no DMA), the nature of Ethernet is to lose frames. Getting a full default size 8kB UDP packet through up to the application level without dropping a single one of the Ethernet frames that make up the fragments is something closely akin to winning the lottery[8]: the timings are really critical. If a single frame is dropped, the UDP packet can never be reassembled and therefore the data does not reach its destination. NFS deals with this by requesting the same information again, but it is very likely that the resent data will not reach its destination any better than the original.

A simple workaround for the problem is to set the NFS read and write sizes to a low default by specifying the parameter `options NFS_BOOT_RWSIZE=1024` in the kernel configuration file. Since 1024 bytes is less than the Ethernet frame size, dropped UDP fragments are not a problem. The real solution is immensely more complex involving a soldering iron and some really steady handywork.

[7] Assuming we are using UDP as the transport in NFS, of course.

[8] But, if I could choose, I would rather choose to the win the lottery.

3. Conclusions

For someone, namely the author, who had no previous experience in working with the evbarm port and only a limited number of encounters with the ARM CPU and no real background in writing ARM assembly, porting NetBSD/evbarm to a new platform proved to be extremely easy. Out-of-the-box cross-buildability proved its usefulness once again, since a toolchain for development was available after typing in one command. One of the really big surprises was that after fixing all the bugs in *initarm*(), the kernel managed to bootstrap itself all the way up to *mountroot*() without a single error. The battle preparations for weeding through Viper-induced bugs and glitches in the machine-independent code were completely unnecessary.

Acknowledgements

This paper and the code imported to NetBSD was reviewed by Steve Woodford. The Viper hardware for development was provided by Data Respons OY.

References

1. Frank van der Linden, *Porting NetBSD to the AMD x86-64: a case study in OS portability*, pp. 1-10, Proceedings of BSDCon '02 (2002).
2. Valeriy Ushakov, *Porting NetBSD to JavaStation-NC*, pp. 161-165, Proceedings BSDCon Europe 2002 (2002).
3. *ARM Architecture Reference Manual*, Addison Wesley. ISBN 0-201-73719-1.
4. Arcom, *Viper Technical Manual.*
5. *Intel XScale(R) Microarchitecture for the PXA255 Processor User Manual* (March, 2003). Order number 278796.
6. eCosCentric Limited and Red Hat, Inc., *RedBoot User's Guide.*
7. Chris Torek, *Device Configuration in 4.4BSD* (December 17, 1992).
8. *config -- the autoconfiguration framework "device definition" language.* NetBSD Kernel Developer's Manual.
9. Matthew Green and Luke Mewburn, *build.sh: Cross-building NetBSD*, pp. 47-56, Proceedings of BSDCon '03 (2003).

Appendix 1: Kernel bootlog

```
RedBoot> load -r -b 0x2000000 netbsd.kaesi
Using default protocol (TFTP)
Raw file loaded 0x02000000-0x0224b1eb, assumed entry at 0x02000000
RedBoot> go

NetBSD/evbarm (viper) booting ...
initarm: Configuring system ...
init subsystems: stacks vectors undefined page pmap
Loaded initial symtab at 0xc03dba58, strtab at 0xc0413018, # entries 13244
pmap_postinit: Allocated 35 static L1 descriptor tables
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.

NetBSD 3.99.3 (VIPER) #255: Sun Jun  5 22:07:00 EEST 2005
        pooka@brain-damage.localhost.fi:/sys/arch/evbarm/compile/obj/VIPER
total memory = 65536 KB
avail memory = 59092 KB
mainbus0 (root)
cpu0 at mainbus0: PXA255/26x step A-0 (XScale core)
cpu0: DC enabled IC enabled WB enabled LABT branch prediction enabled
cpu0: 32KB/32B 32-way Instruction cache
cpu0: 32KB/32B 32-way write-back-locking Data cache
pxaip0 at mainbus0: PXA2x0 Onchip Peripheral Bus
pxaip0: CPU clock = 396.361 MHz
pxaintc0 at pxaip0 addr 0x40d00000-0x40d0001f: Interrupt Controller
pxagpio0 at pxaip0 addr 0x40e00000-0x40e0006f: GPIO Controller
sm0 at pxaip0 addr 0x8000300 intr 0
sm0: SMC91C111, revision 1, buffer size: 8192
sm0: MAC address xx:xx:xx:xx:xx:xx, default media MII (internal PHY)
sqphy0 at sm0 phy 0: Seeq 84220 10/100 media interface, rev. 0
sqphy0: using Seeq 84220 isolate/reset hack
sqphy0: 10baseT, 10baseT-FDX, 100baseTX, 100baseTX-FDX, auto
com0 at pxaip0 addr 0x40100000-0x4010001f intr 22: ns16550a, working fifo
com0: console
com1 at pxaip0 addr 0x40200000-0x4020001f intr 21: ns16550a, working fifo
com2 at pxaip0 addr 0x40700000-0x4070001f intr 20: ns16550a, working fifo
saost0 at pxaip0 addr 0x40a00000-0x40a0001f
saost0: SA-11x0 OS Timer
clock: hz=100 stathz = 64
```

Appendix 2: Devices in the configuration file

```
# The main bus device
mainbus0 at root

# The boot CPU
cpu0     at mainbus?

# peripherals
pxaip0        at mainbus0

# interrupt controller & gpio pins
pxaintc0 at pxaip0
pxagpio0 at pxaip0

# serial ports
options  COM_PXA2X0
options  FFUARTCONSOLE
com0 at pxaip0 addr 0x40100000 intr 22  # FFUART
com1 at pxaip0 addr 0x40200000 intr 21  # BTUART
com2 at pxaip0 addr 0x40700000 intr 20

# these two are not hanging off of pxaip, not really tested either
#com3     at pxaip0 addr 0x14300000   # COM5
#com4     at pxaip0 addr 0x14300010   # COM4

# SMC91C111 ethernet
sm0  at pxaip0 addr 0x08000300 intr 0

# MII/PHY support
sqphy*   at mii? phy ?          # Seeq 80220/80221/80223 PHYs
```

New Networking Features in FreeBSD 6.0

André Oppermann
andre@FreeBSD.org

The FreeBSD Project

Abstract

FreeBSD 6 has evolved drastically in the development branch since FreeBSD 5.3 [1] and especially so in the network area. The paper gives an in-depth overview of all network stack related enhancements, changes and new code with a narrative on their rationale.

1 Internal changes – Stuff under the hood

Mbuf UMA

UMA (Universal Memory Allocator) is the FreeBSD kernels primary memory allocator for fixed sized data structures. It is a SLAB type allocator, fully SMP aware and maintains per-CPU caches of frequently used objects. All network data is stored in Mbufs of 256 bytes and Mbuf clusters of 2048 bytes which can be attached to Mbufs and replace their internal data storage. When a cluster is attached the Mbuf serves as descriptor for the cluster containing all associated Mbuf and packet information for the kernel and protocols. To use UMA for efficient Mbuf allocation some enhancements have been made to it. Most important is the packet secondary zone holding pre-combined Mbuf+cluster pairs. This allows protocols to save one memory allocation by directly obtaining a large data structure instead of allocating an Mbuf and then attaching a separately allocated Mbuf cluster. The secondary zone is special as it is only a cache zone and does not have its own backing store. All mbuf+cluster combinations in it come from their own original Mbuf and cluster zones. Mbuf UMA provides good SMP scalability and an accelerated allocation path for frequently used Mbuf+cluster pairs. *For more information see [2], mbuf(9) and uma(9).*

SMP Locking

SMP locking of network related data structures is the main theme of FreeBSD 6. Locking is necessary to prevent two CPUs accessing or manipulating the same data structure at the same time. Locking gives exclusive access to only one CPU at a time and makes them aware of each others work – it prevents CPUs from stomping on each others feet. Generally it is desirable to break down locking into fine-grained portions to avoid lock contention when multiple CPUs want to access related but independent data structures. On the other hand too fine-grained locking is introducing overhead as each locking and unlocking operation has to be reliably propagated to all other CPUs in the system. The first go on fine-grained network locking in FreeBSD 5 has been greatly enhanced and refined for excellent SMP scalability. For single processor machines all performance regressions due to locking overhead have been eleminated and FreeBSD 6 reaches the same speed in heavy duty network streaming as the FreeBSD 4 series. *For more information see [3], mutex(9) and witness(4).*

Socket Buffer Locking

Every active or listening network connection is represented as a socket structure in the kernel. The socket structure contains general bookkeeping on the socket and two socket buffer for transmitted and received packets. Protocols (TCP, UDP, IPX, SPX, etc.) extend the socket structure with their own bookkeeping to track connections state and other vital information. Many of these structures are linked forth and back and among each other. This makes proper locking complicated. Additionally the socket data structure may be accessed and manipulated at any time either from an application writing, reading or closing the socket or from the kernel itself when it has received data, retransmits or error messages for that socket. FreeBSD 6 implements a multi-level locking strategy to efficiently cope with these constrains. Each socket structure has a general lock and two separate send and receive socket buffer locks. Thus sending and receiving may happen concurrently. Any operation that changes the state of the entire socket (ie. connection tear down) has to obtain the general lock. On the protocol side (using TCP as example) two more locks are embedded. One protects the IN and TCP control blocks which contain IP protocol and TCP specific information, such as the addresses of the end points and the state of the TCP connection. The other lock protects all IN control blocks as a whole. Locks with such a global scope are normally frowned upon but here it is necessary to prevent changes in the control blocks while searches and lookup's are performed on it. A search and lookup happens every time a packet is received from the network. While this is not optimal it has shown to express only modest contention.

Protocol Locking

Since early 2005 the entire network stack is running without any global and exclusive lock. All Internet protocols and IPX/SPX have been individually locked and thus made fully SMP aware and scalable.

Network Interface Structure Locking

An area of particular concern for proper locking was the ifnet structure. The ifnet structure contains all information the kernel knows about network interfaces. In FreeBSD network interfaces drivers may be loaded and unloaded any time as KLDs (Kernel Loadable Modules) or may arrive or depart as hot-plug interfaces like PCCARDs in laptops. Allowing these actions to occur in a SMP-safe way has required significant work and re-work of the ifnet structure and its modes of access. For example some fields in the structure were holding flags manipulated by the network stack and the driver. Each of them had to obtain the lock for the full structure to change its own fields and flags. This lead to contention and limitations on parallelism for access to the physical network. Any such unnecessary contention point has been identified and each party has got their own field which they can manipulate independently without stalling the other. *For more information see ifnet(9), netintro(9), [4] and [5].*

2 Netgraph

Netgraph is a concept where a number of small, single-job modules are stringed together to process packets through stages. Many modules may be combined in almost arbitrary ways. Netgraph may be best explained as an assembly line with many little functions along a conveyor belt versus one big opaque machine doing all work in one step. As part of the network stack netgraph has received fine grained locking too. Depending on the function and task of the module it was either locked as whole or every instance of it separately. *For more information see netgraph(4), netgraph(3) and ngctl(8).*

Module ng_netflow

Ng_netflow is a new module for accounting of TCP and UDP flows in ISP (Internet Service Provider) backbones. It accumulates statistics on all TCP and UDP session going through the machine and once one has finished (FIN or RST

for TCP) sends a UDP packet in the Netflow 5 format to a statistics collector for further processing. The node can either run in parallel to the normal IP forwarding and packet processing, in this case it gets a copy of every packet, or all packets are passed through it unmodified. *For more information see ng_netflow(4).*

Module ng_ipfw

Ng_ipfw is a new module providing a way for injecting arbitrary matched packets into netgraph using ipfw. It works very much like an ipfw divert rule diverting the packet to netgraph instead of a divert socket. This allows, for example, to send by ipfw filtered or rejected packets to netgraph for further analysis or to capture certain types of IP packets for further netgraph manipulations. The packet matching capabilities of ipfw are very powerful in this context. *For more information see ng_ipfw(4), ipfw(8) and ipfw(4).*

Module ng_nat

Ng_nat is a new module providing netgraph access to the kernel-level libalias for network address translation. Libalias used to be a userland-only application library but was written with in-kernel use in mind. For ng_nat is got imported into the kernel. *For more information see ng_nat(4) and libalias(3).*

Module ng_tcpmss

Ng_tcpmss is a new module changing the MSS (Maximum Segment Size) optoin of TCP SYN packets. Many broadband users are behind DSL lines with a reduced MTU (Maximum Transmission Unit) of 1492 bytes. The normal size for ethernet is 1500 bytes. If a packet does not fit the MTU of link is has to be fragmented – it gets split into two packets. This is a CPU intensive process and to be avoided if possible. Normally the TCP path MTU discovery mechanism is supposed to automatically detect smaller MTUs along the way but over-zealous firewall administrators often block the ICMP MTU adjustment messages. As workaround a router along the path of the packet scans for TCP SYN packets and manipulates it to reduce the MSS to fit the lower MTU. *For more information see ng_tcpmss(4).*

3 IPv4

DHCP Client

The most visible change is the new DHCP client. It is a port of the OpenBSD dhclient and adapted to FreeBSD specific needs. It has many security features like privilege separation to prevent spoofed DHCP packets from exploiting the machine. Additionally it is network interface link state aware and will re-probe for a new IP address when the link comes back up. This is very convenient for laptop users who may connect to many different networks, be it wired or wireless LANs many times a day. *For more information see dhclient(8), dhclient.conf(5) and dhclient.leases(5).*

IPFW Firewall

IPFW has received many visible and invisible modifications. The most prominent visible changes are IPv6 rule support and ALTQ tagging of packets. The IPv6 support is further discussed in the IPv6 section. ALTQ is an alternative queuing implementation for network interfaces. Whenever an output interface doesn't have enough bandwidth to forward all waiting packets immediately queuing happens. Excess packets have to wait until earlier packets are drained and capacity is available again. Standard queuing strategy is a tail queue – all new packets get appended to the tail of the queue until the queue is full and any further packets get dropped. In many situations this is undesirable and for QoS (Quality of Service) it should treat various types of packets and traffic differently and with different priorities. ALTQ allows to define different queuing strategies on network interfaces to prioritize, for example, TCP ACKs on slow ADSL uplinks or delay and jitter sensitive VoIP (Voice over IP) packets. IPFW

can be used as packet classifier for ALTQ treatment. IPFW has another packet queue manager called DUMMYNET which can perform many of ALTQ function too. However it is more geared towards network simulations in research setting than to general network interface queuing. Under the hood of IPFW the stateful inspection of packet flows has been converted to use UMA zones for flow-state structure allocation. *For more information see ipfw(8), ipfw(4), altq(8), altq(9) and dummynet(4).*

IPDIVERT

The IPDIVERT module is used for NAT (Network address Translation) with IPFW. It is now a loadable module that can loaded into the kernel at runtime. Before it always required a kernel re-compile to make it available. *For more information see divert(4).*

IP Options

IP Options are a sore spot in the entire IPv4 specification. IP Options extend the IP header by a variable size of up to 40 bytes to request and record certain information from routers along the packets path. IP Options are seldom used these days and have essential zero legitimate use other than Record Route perhaps. IP Options handling in the kernel is complicated and was handled through a couple of global variables in the IP code path. Access to these variables had to be locked and it prevented multiple CPUs from working on IP packets in parallel. The global variables have been moved into mtags attached to mbufs containing IP packets with IP Options. This way all CPUs can work on IP packets in parallel without risk of overwriting information and the IP Options information always stays with the packet it belongs to. Even when one CPU hands off the packet to another CPU.

IPFILTER Firewall

IPFILTER 4.1.8 was imported and provides proper locking of its data structures to work in SMP environments. *For more information see ipf(8), ipf(5) and ipf(4).*

NFS Network File System

NFS has been extensively tested and received numerous bug fixes for many edge cases involving file access as well as some network buffer improvements.

ICMP

ICMP Source Quench support has been removed as it is deprecated for a long time now. Source Quench was intended to signal overloaded links along a packet path but it would send one Source Quench message per dropped payload packet and thus increased the network load rather to reduce it. It is not and was never used in the Internet. *For more information see [6].*

ICMP replies can now be sent from the IP address of the interface the packet came into the system. Previously it would always respond with the IP address of the interface on the return path. When the machine is used as a router this could give very misleading error messages and traceroute output. *For more information see icmp(4).*

ARP Address Resolutions Protocol

Many ARP entry manipulation races got fixed. ARP maps an IPv4 address to a hardware (MAC) address used on the ethernet wire. It stores the IP address of each machine on all directly connected subnets as a host route and attaches their MAC address. ARP lookup's and timeouts can happen at any time and may be triggered at any time from other machines on the network. In SMP environments this has led to priority inversions and a couple of race conditions where one CPU was changing parts of an ARP entry when a second CPU tried to do the same. They clashed and stomped on each others work leading to incorrect ARP entries and even crashes sometimes. An extensive rework and locking has been done to make ARP SMP-safe.

IP Multicast

IP Multicast had many races too. Most of them related to changes of IP addresses on network interfaces and disappearing interfaces due to unload or unplug events. Proper locking and ordering of locks has been instituted to make IP Multicast SMP-safe.

IP Sockets

An IP_MINTTL socket option got added. The argument to this socket option is a value between 1 and 255 which specifies the minimum TTL (Time To Live) a packet must have to be accepted on this socket. It can be applied to UDP, TCP and RAW IP sockets. This option is only really useful when set to 255 preventing packets from outside the directly connected networks reaching local listeners on sockets. It allows userland implementation of 'The Generalized TTL Security Mechanism (GTSM)' according to RFC3682. Examples of such use include the Cisco IOS BGP implementation command "neighbor ttl-security". *For more information see ip(4) and RFC3682.*

The IP_DONTFRAG socket option got added. When enabled this socket option sets the Don't Fragment bit in the IP header. It also prevents sending of packets larger than the egress interface MTU with an EMSGSIZE error return value. Previously packets larger than the interface MTU got fragmented on the IP layer and applications didn't have a direct way of ensuring that they send packets fitting into the MTU. It is only implemented for UDP and RAW IP sockets. On TCP sockets the Don't Fragment bit is controlled through the path MTU discovery option. *For more information see ip(4).*

4 TCP Transmission Control Protocol

SACK Selective ACKnowledgements

SACK has received many optimizations and interoperability bug fixes. *For more information see tcp(4).*

T/TCP Transactional TCP

T/TCP support according to RFC1644 has been removed. The associated socket level changes however remain intact and functional. FreeBSD was the only mainstream operating system that ever implemented T/TCP and its intrusive changes to the TCP processing made code maintenance hard. It primary feature was the shortening of the three-way TCP handshake for hosts that knew each other. Unfortunately it did this in a very insecure way that is very prone to session spoofing and packet injection attacks. Use of it was only possible in well secured Intranets. It never enjoyed any widespread use other than on round trip time sensitive satellite links. A replacement is planned for FreeBSD 7.

TCP Sockets

The TCP_INFO socket option allows the retrieval of vital metrics of an active TCP session such as estimated RTT, negotiated MSS and current window sizes. It is supposed to be compatible with a similar Linux socket option but still experimental.

Security Improvements

The tcpdrop utility allows the administrator to drop or disconnect any active TCP connection on the machine. This tool was ported from OpenBSD. *For more information see tcpdrop(8).*

The logic processing of TCP timestamps (RFC1323) has been improved to prevent spoofing attempts.

TCP Path MTU Discovery has been improved to prevent spoofing attacks. It now checks the entire TCP header that is quoted in the ICMP Fragmentation Needed message to ensure it matches to a valid and active TCP connection. *For more information see [5].*

Port Randomization led to some problems when applications with very high connection rates

came close to exhaust the port number range. The randomization function was calculating random ports numbers which were most likely already in use and fell into an almost endless loop as the odds of finding a free port at random dropped constantly. If exhaustion is near it now switches to normal allocation for 45 seconds to make the remaining ports available with little overhead. *For more information see [6].*

UDP

All global variables have been removed to prevent locking contention and allow for parallel processing of packets.

5 IPv6

IPFW Firewall

IPFW now supports IPv6 rules and allows all available actions for IPv6 packets too. The previously separate ipfw6 packet filter is to be retired. The primary advantage of this merge is a single code base and packet flow for IPv4 and IPv6 without duplication or feature differences. *For more information see ipfw(8) and ipfw(4).*

KAME netinet6 Code

Many bugfixes and small improvements have been ported from the KAME codebase.

6 IPX

IPX/SPX is still in use at a non-negligible number of sites and some significant effort has been made to lock SPX data structures and to make them SMP-safe.

7 Interfaces

CARP Common Address Redundancy Protocol

CARP is a special network interface and protocol that allows two or more routers to share the same IP address. Thus for all hosts using that router any fail-over from one to another one is transparent and no service interruption occurs. Routers in a CARP system may do hot-standby with priorities or load-sharing among them. CARP has been ported from OpenBSD and is similar in functionality to VRRP from Cisco. *For more information see carp(4).*

Ethernet Bridge if_bridge

If_bridge is a fully fledged ethernet bridge supporting spanning tree and layer 2 or layer 3 packet filters on bridged packets. If_bridge has been ported from NetBSD and replaces the previous bridge implementation of FreeBSD. Spanning tree is very important in bridged networks because it prevents loops in the topology. Ethernet packets do not have a TTL that is decremented on each hop and all packets in a looped bridge topology would cycle for an infinite amount of time in the network bringing it to a total standstill. *For more information see if_bridge(4).*

IEEE 802.11 Wireless LAN

The Wireless LAN subsystem has been enhanced to support WPA authentication and encryption in addition to WEP. It may be operated in client (Station) mode or AP (Access Point) mode. In both modes it supports the full WPA authentication and encryption set. The availability of the AP mode depends on the wireless LAN chip vendor, obtainable documentation (w/o NDA) and driver implementation. All cited features are implemented in the ath driver for Atheros-based wireless cards which have the best documentation available. *For more information see ieee80211(4), wlan(4), wlan_ccmp(4), wlan_tkip(4), wlan_wep(4), wlan_xauth(4), wpa_supplicant(8), wpa_supplicant(1), hostapd(8), hostapd(1) and ath(4).*

Interface Polling

The network interface polling implementation has been re-implemented to work correctly in SMP environments. Polling is no longer a global configuration variable but enabled or disabled individually per interface if the driver supports it. Most commonly found network drivers support polling. *For more information see polling(4).*

NDIS Compatibility – Project Evil

Binary compatibility with Windows NDIS miniport drivers. The NDIS compatibility layer emulates the Windows XP/Vista kernel network driver interface and allows Windows network card drivers to be run on FreeBSD. It supports wired and wireless LAN cards. Many parts have been rewritten and updated as more Windows drivers could be tested, better documentation became available and a more throughout understanding of the NDIS nits developed. It has been updated to work in SMP systems. While NDIS emulation works well it is only a last resort when all attempts of obtaining network chip documentation have failed. A FreeBSD native drivers is always preferred to using Windows drivers through the NDIS emulation layer. *For more information see ndis(4), ndis_events(8), ndiscvt(8), ndisgen(8).*

Network Driver Locking

Network drivers have to set up and maintain a couple of internal structures. Examples of the structures include send and receive DMA rings and MII information from the PHY. Whenever packets are sent or received the CPU must have exclusive access to the these structures to avoid clashes and confusion. Many drivers had to be re-worked to make them SMP-safe as originally multi-access wasn't a concern. Depending on the network card the driver got a single lock covering all aspects of its operation. Sometimes an even more fine grained approach was taken to have a lock for each send and receive direction plus global state manipulations. Separate send and receive locks provide the best efficiency in SMP systems as two CPU may simultaneously receive and transmit packets.

References:

[1] FreeBSD 5 Network Enhancements, André Oppermann, September 2004, http://people.freebsd.org/~andre/FreeBSD-5.3-Networking.pdf

[2] Network Buffer Allocation in the FreeBSD Operating System, Bosko Milekic, May 2004, http://bmilekic.unixdaemons.com/netbuf_bmilekic.pdf

[3] Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack, Robert N. M. Watson, November 2005

[4] TCP/IP Illustrated, Vol. 2, Gary R. Wright and W. Richard Stevens, Addison-Wesley, ISBN 0-201-63354-X

[5] The Design and Implementation of the FreeBSD Operating System, Marshall Kirk McKusick and George V. Neville-Neil 2004, Addison-Wesley, ISBN 0-201-70245-2

[6] ICMP attacks against TCP, Fernando Gont, October 2005, IETF Internet Draft draft-gont-tcpm-icmp-attacks-05.txt

[6] Improving TCP/IP Security Through Randomization Without Sacrificing Interoperability, Michael J. Silbersack, November 2005

Embedded OpenBSD

Niall O'Higgins <niallo@openbsd.org>
Uwe Stühler <uwe@openbsd.org>

EuroBSDCon, 2005

Abstract

OpenBSD is often overlooked in the embedded computing domain. In this paper we will highlight some of the features which make OpenBSD an excellent choice as an embedded operating system. We will give real-world examples from small i386 systems and the Sharp Zaurus. Finally, we will discuss the technical issues involved in starting a port to a new platform.

1 Advantages of OpenBSD

Before we talk about the advantages of OpenBSD, it is important to acknowledge the limitations and caveats concerning its usage in the embedded space. OpenBSD is a general purpose UNIX-like operating system. It is not designed specifically for embedded systems, and as such lacks certain niche features these systems may require.

1.1 Documentation

It is not unreasonable to assert that OpenBSD lacks some of the kinds of documentation typical to commercial embedded operating systems. For example, there is no single centralised "OpenBSD developer's guide", nor is there a step-by-step guide for embedded systems development using OpenBSD. Commercial embedded operating systems such as QNX Neutrino and VxWorks offer frankly much more comprehensive documentation both on the system overview level and HOWTO-style articles from the most basic "Get started with Hello World!" to more advanced topics such as "Develop a device driver".

However, OpenBSD has documentation in the form of books, manual pages, FAQs, mailing lists, and papers/slides. It should be mentioned that OpenBSD, as a descendent of the original Berkeley Software Distribution, has over 30 years of history and as such has a very mature and widely-understood architecture. Much of the information in books such as "The Design and Implementation of the 4.4BSD Operating System"[?] is still relevant today. As a POSIX-compliant UNIX-like operating system, you can pick up any good UNIX software programming book and start writing programs for OpenBSD. Furthermore, the manual pages are meticulously checked for consistency. Most library function manual pages include concise examples and in some cases where there are security issues they specifically mention improper usage idioms which should be avoided. A good example of this is the ***snprintf(3)*** function. Of course, the ultimate resource available to the programmer is the source code, which is freely available along with the CVS history which can yield insight into the reasons behind certain pieces of code or design decisions.

1.2 Realtime Systems

At this time, because OpenBSD does not preempt processes executing in kernel mode, realtime response to events is dependent upon the amount of time spent in each system activity. Additionally, processes have no way to determine which of

their pages are resident, so they have no way of ensuring that they will be able to execute a sequence of instructions without incurring one or more page faults.[1] Since the system guarantees no upper bounds on the duration of a system activity, OpenBSD is decidedly not a realtime system[1].

Soft realtime systems are those where it is acceptable to miss a deadline occasionally. The most obvious examples are digital video and audio processing systems. It is theoretically possible to add soft realtime capabilities to OpenBSD.

RTMX Incorporated sell addons to OpenBSD which add POSIX realtime extensions for messaging, signals, named and un-named semaphores, shared memory, realtime / memory mapped files, message queues and fixed priority tasking. This source code has in fact been donated to OpenBSD and may be integrated into the official release at some point.

Hard realtime systems are systems where a deadline cannot be missed, or cannot be missed by more than a fixed amount of time. OpenBSD cannot offer hard realtime capabilities. However, this is not necessarily a problem, because these features can be provided by dedicated hardware.

1.3 Filesystems

OpenBSD uses the Berkeley Fast File System (FFS), which is very robust and mature. It uses a notion of 'cylinder groups', comprising one or more consecutive cylinders on a disk. FFS tries to allocate new blocks on the same cylinder as the previous block in the same file - optimally these blocks will also be rotationally well positioned. Thus, cylinder groups minimise seek times and maxmise throughput. To offset the space wasted by the relatively large 4k block size, individual blocks can be split into "fragments" to optimise storage utilisation - particularly important on UNIX-like systems where there are typically large numbers of sub-4k files in existence.[2]

While FFS is very well suited for use on hard disks, it was not designed with flash memory in mind. As such, many of the optimisations and allocation strategies employed by FFS merely constitute overhead when used with flash memory. Additionally, flash memory has some particular qualities, such as long (1 second) erase times in the case of NOR memory or the expectation of bad blocks in the case of NAND memory.

Of special importance is the fact that both types of flash have a limited number of erase/write operations due to wear on the insulating oxide layer around the charge store mechanism used to store data. Typical NAND flash memory wears out after

[1] This problem can be avoided by not specifying a swap device.

1,000,000 erase/write operations. This causes problems on UNIX-like systems because every file has a time of last access associated with it, so even read-only operations incur write operations to flash memory - thus your flash memory could die very quickly. Fortunately, OpenBSD supports the "noatime" ***mount(8)*** option which disables updating the last access time of files unless the last data modification time or last file status change is being changed as well, greatly decreasing the number of writes to the filesystem.

Also of concern to embedded computing environments - where the system can be expected to resume operation very quickly after sudden power loss or unclean shutdown - is the requirement to perform a filesystem integrity check (***fsck(8)***) before mounting a dirty FFS filesystem as writable. This integrity check can be both a time and memory consuming process. Fortunately, there are workarounds. Many embedded systems need only to write data relatively infrequently, for example to update a configuration file every few days. In cases such as these, it is feasible to keep the filesystem mounted read-only, mounting as writable only when write operations are necessary. Thus, the window where sudden powerloss or unclean shutdown could force a filesystem integrity check is reduced to at most a couple of seconds while the write operation completes. In the future, we hope to greatly reduce the time and memory requirements of ***fsck(8)***, possibly through the addition of a mini-journal to our FFS implementation.

In addition to FFS, OpenBSD also has support for the Andrew File System (AFS), Linux (EXT2/EXT3), ISO-9660 (CD-ROM), MS-DOS (FAT/VFAT), Network File System (NFS), NTFS (read-only) and UDF (DVD-ROM) filesystems. Of particular interest to those working with embedded systems is likely the MS-DOS filesystem support. This is because most flash media used by devices such as MP3 players or digital cameras is formatted as MS-DOS for compatibilty purposes. OpenBSD can read, write and format these kinds of partitions natively. The ***newfs(8)*** program can perform this formatting by supplying the command line option *-t msdos*.

1.4 Memory

Like most general purpose UNIX operating systems, OpenBSD requires a CPU with a Memory Management Unit (MMU). For mass-produced or low profit margin applications where you have a choice between different CPUs, this could add unnecessary expenditure. However, modern embedded architectures with MMUs are available at affordable prices. The unit price for a ARM720T CPU with MMU, running at 80 Mhz could well be below 15 € by now, even in small quantities.

As with CPU prices, the total cost of memory can be an issue in some applications.

OpenBSD requires a relatively modest amount of memory for a general-purpose desktop or server operating system. In its default configuration, OpenBSD typically needs 16MB RAM, and the base system requires at least 128MB free space in the filesystem to install, but just about everything can be stripped out (of course, this is no longer an officially supported configuration).

However, memory requirements can be reduced dramatically, at the expense of having to develop and maintain an unsupported system configuration. People use CompactFlash cards with capacities of less than 64MB in Soekris machines to run OpenBSD, and a minimal system that can act as a router can theoretically be fitted on a 1.44MB floppy, using the same compression techniques that are also applied to create the official installation floppy images.

Again, for mass-produced or low profit margin applications, RAM usage can be greatly reduced by stripping out unneeded device drivers and other components from the kernel. For an ARM machine like "cats" we have been able to create a kernel that comprises only about 1MB of read-only code and data, needs only 1MB RAM for its data structures during startup, and runs directly from read-only memory. This configuration included only the needed device drivers, optional generic features such as TCP/IP networking (only IPv4 on purpose), multiple filesystem support, and the kernel debugger ***ddb(4)***.

1.5 Portability

It is one of the expressed goals of the OpenBSD project to support multiple platforms.[3]

Naturally, an operating system supporting different hardware platforms is advantageous to its users. It allows them to switch hardware freely and works against vendor lock-in.

Having multiple platforms is also an advantage for the entire codebase; leading to better machine abstractions and causing bugs to be uncovered in subsystems like ***bus_dma(9)***[2]. Of course, the plethora of endian-ness, 64-bit, pointer length, and alignment issues being found and fixed by having platforms that differ in these areas increase code quality throughout the tree.

OpenBSD is already a very portable operating system. For example, 95% of the kernel source code is machine-independent, even for big ports like "sparc". The port consists of 76,000 lines of code, but more than 1,300,000 lines are currently machine-independent. Many hardware platforms are supported by the existing ports, and new ports are constantly being worked on.[4] Furthermore, because

[2]Machine and bus independent DMA transfer interface

OpenBSD - as a descendant of NetBSD - has much infrastructure in common with that operating system, new ports can still be based on one of the NetBSD ports.

In section 3 of this paper we will dwell a bit on porting.

1.6 Completeness

The BSD centralised development model, which produces a complete working operating system, is inherently advantageous to embedded projects. In addition to a good kernel, you get a complete userland. Not only all the standard utilities you would expect on a UNIX-like system, but many useful daemons and libraries ready out of the box.

Examples of third-party software that come "bundled" with OpenBSD are: Perl, OpenSSH, Sudo, Sendmail as a mail server and filter, BIND DNS server with security-related improvements, Apache (1.3.x) web server with many security-related improvements, and of course the X Window System from X.org.

Whatever else is missing for your application can be installed via ports and packages. Especially if you have to create a very small distribution, the OpenBSD ports tree will help you to create and maintain your own versions of packages. Packages can even be installed on a system where ***perl(1)*** and the ***pkg(1)*** tools are not available due to space constraints.

1.7 Security

OpenBSD is well known for its security, very briefly here is a short list of some of the techniques used:

- On OpenBSD, ***gcc(1)*** comes with the "ProPolice" stack protection extension, which is enabled by default[3]; kernel, userland and just about all packages are compiled with this.
- Randomness is cheap; use it everywhere we can to mix things up for the attacker: randomise address space allocated with ***mmap(2)*** or ***malloc(2)***, use randomness in the TCP/IP stack, load shared libraries in random order.
- W^X memory protection on many platforms - on some (e.g. ARM) it is not possible unfortunately.

[3] See ***gcc-local(1)*** for a list of other OpenBSD-specific modifications.

- Privilege separation in network daemons.
- Source code auditing, e.g. replacement of unsafe string handling functions like ***strcpy(3)*** and ***sprintf(3)*** with safe variants.

Developers of small embedded systems who are not accustomed to using a complete operating system may be in doubt as to whether security features such as ***mmap(2)*** randomisation truly helps their product. Some may argue that randomness makes it harder to track down problems, but random memory adresses combined with strict memory protection in fact greatly improves software quality. This is because it detects out of bounds memory accesses much earlier in the development cycle. Essentially, it forces you to deal with the bug immediately in order to proceed with the development of the program, instead of noticing it much later when it has become orders of magnitude more difficult to track down. Features like this and others such as the new ***malloc(3)*** implementation which crash a buggy program early effectively add a significant degree of "self-monitoring" to the system. Furthermore they encourage fault-tolerant application design and emphasise program correctness in general by making the run-time environment much stricter and less forgiving. It should be noted that unforgiving environments are not uncommon for embedded systems; related fault-tolerant design concerns include the capacity to deal with sudden power loss and maintaining consistent application state. Finally, tools such as ***watchdogd(8)*** are now shipped with OpenBSD which can trigger a reboot if process scheduling fails.

1.8 Licensing

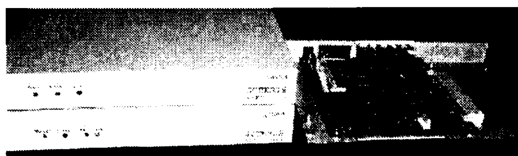
Some of the commercial operating systems are enormously expensive, and the vendors use "inconvenient" license models, to say the least. If the operating system source code is made available at all, then you can usually not afford it.

OpenBSD is extremely strict about the licensing of source code it integrates. We want to make available code that anyone can use for *any purpose*, with no restrictions. Having undergone numerous *license audits*, OpenBSD should add minimal legal headache to your embedded project.[5]

2 Real-world Examples

The machines we describe below are not embedded systems as such, but embedded systems could make use of similar hardware. One of the main differences between Soekris and the Sharp Zaurus is that Soekris appliances can use the exisiting OpenBSD/i386 port whereas the Zaurus required a whole new port. Of course, the Zaurus was able to build upon the existing OpenBSD/cats port, as too would any products based on an ARM processor.

2.1 Soekris



Soekris machines are small i386 architechture computers based on a 100/133 Mhz AMD ElanSC520 486-class CPU or a 233/266 Mhz NSC SC1100 586-class CPU. Soekris Engineering sell various models to accomodate different numbers and types of peripheral devices; eg PCMCIA cards, Mini-PCI cards, etc. However, they all share some fundamental characteristics:

- Hardware watchdog. This is supported by the ***watchdog(4)*** driver; it can be used to automatically reset the machine in case of system overload or freeze.
- Serial BIOS. This removes the necessity for any kind of VGA adapter, monitor or keyboard, making the case much more compact. Remote administration is simplified since the BIOS can be accessed easily along with the bootloader for system upgrades or maintenance.
- PXE boot ROM. This enables diskless booting, useful not least for first-time operating system installation.
- One user-programmable LED. This is accessible through the ***gpio(4)*** framework and is useful for indicating status at a glance. For example, the LED could be used to indicate the connection status of a VPN gateway, or to quickly tell which machine is master in a ***carp(4)*** configuration and which are backups.

- Some number of GPIO pins. These can be connected to simple devices such as more LEDs, thermal sensors, relay controllers, etc - they can even be used to drive SD card readers/writers in SPI mode.

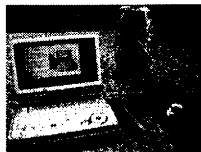
Furthermore, the systems are completely passively cooled removing the need for any fans. Power consumption is low, in the order of 10 or 15 Watts. In fact, since the Soekris takes DC electricity it is actually possible to provide power to it via a battery.

The most obvious use for a Soekris machine is as a router or firewall, using ***pf(4)***. They can be fitted with a power-over-ethernet adapter which is very useful if you are installing the computer in a place where it would be infeasible or even impossible to supply power separately. Thus they are well-suited for use as wireless routers, since such machines typically need to go in out-of-the-way places in order to maximise signal coverage. OpenBSD's inclusion of the ***authpf(8)*** gateway authentication program makes OpenBSD on Soekris an appealing choice for wireless network access provision.

Another example of a role suited to Soekris machines is a low-traffic VPN router with ***ipsec(4)***. Add another machine and some ***carp(4)*** and ***sasyncd(8)*** and you get IPsec failover. The Soekris can be fitted with various hardware encryption accelerators which are supported by OpenBSD and may help performance somewhat, but other important bottlenecks will remain.

However, while Soekris machines are useful in many situations, their capabilities should not be over-estimated. The onboard ***sis(4)*** NICs are not very efficient, generating lots of interrupts and the CPUs are underpowered for dealing with substantial quantites of network traffic. Realistically, they are a low to mid level networking device, on par with proprietary "soho" routers - but much more configurable and flexible of course.

2.2 Zaurus



The Zaurus is effectively a miniature laptop, taking technology from the embedded systems domain - at its core is the XScale PXA27x processor, a SoC design - and scaling it up into a fully featured personal computer, albeit a very small one. Sharp has designed it as a PDA and ships it with Trolltech's Qtopia running

on a custom Linux kernel, plus some additional PDA applications. Yet, many customers who use it more like a mini laptop run an alternative operating system.

The PXA27x processor is a very interesting CPU, with just about everything integrated on-chip. It almost makes the Zaurus a perfect example of a truly embedded system - but not quite. This is because the functionality of the Zaurus has not been constrained as much by its hardware design or resources as is typically the case with embedded systems. The initially targeted C3000 and C3100 models run at 416Mhz, contain 64MB RAM and a 4GB CompactFlash hard drive internally. The biggest weakness is perhaps the lack of a floating-point unit.

Additionally, the Zaurus has many useful interfaces: a high-resolution LCD screen, a touch-sensitive screen surface, an infrared serial port, a regular serial port (with an adapter cable), a USB 1.1-compliant interface for host and client mode (selected via an adapter cable; client mode is not yet supported by OpenBSD), a SDIO card socket (not supported by OpenBSD yet), a CompactFlash socket (regular PCMCIA cards can also be used with an adapter), keypad buttons that can be accessed when the lid is closed and of course a 3.5mm headphone jack.

Except for the touch-screen, keypad and the audio CODEC, *all* of the above mentioned interfaces are essentially controlled directly by the PXA27x. Other on-chip peripherals are used only underneath the surface (memory-, DMA-, GPIO- and interrupt controllers, real-time clock and high-resolution timers), or to communicate with a few supporting chips on the board (I^2C, I^2S and SSP units).

Its capabilities now include the ability to boot multi-user, run the X Window System, act as a USB 1.1 host, play audio, suspend and resume much like a laptop. Many of the thousands of ports also compile and run well, including games like Doom and applications such as Kismet.

3 Porting OpenBSD

Making a port requires, first and foremost, motivation and persistent work. It is important to recognise what is feasible and what is not. A port which kind of, sort of works in a half baked way with a ton of caveats and bugs is not much use to anybody. For a port to become a fully supported OpenBSD port it must meet a certain standard of quality. Full support means that the release install media is known to work, that the architecture can compile itself and that most of the basic tools exist on the architecture. Furthermore, releases with pre-compiled application packages should always exist, and there should be attempts to make snapshots available on a regular basis.[4]

To maintain full support for any architecture, about 20 developers should run machines of that architecture to find, discuss and fix bugs, review and test diffs, build packages, keep documentation up to date and so on. A substatial number of users are also required in order to expose the port to many different machine configurations, usage patterns and to help with development in general.

The rest of this section should give an impression of what steps are involved in the porting process. Much more detail can be found in *Porting BSD UNIX to a New Platform*[6]. It is somewhat outdated, but still valuable reading.

3.1 Preparation

The first step in the porting process is to become familiar with the architecture of the target hardware. The more documentation about the CPU and other components, the better. It is simply not practical to start a port entirely from scratch, one always picks an existing port and works from that.

When choosing a port as the basis for another, it is best to choose one for a machine that has as many things as possible in common with the target machine. One may find that some port's CPU is similar to the target CPU, but some other port supports a machine architecture that is closer to the target machine architecture (with respect to busses, external interfaces, and other subsystems). In such cases, a judgement call is required.

3.2 Development Environment

To create a working development environment for building a kernel - unless there is already a similar machine running OpenBSD which uses the same CPU family as the target machine - it is necessary to use cross-compilation.

Obviously it is not feasible for a project like OpenBSD to develop a complete compiler toolchain. This gap is filled with software from the Free Software Foundation - GNU binutils and the GNU Compiler Collection (GCC). However, this makes it difficult to port OpenBSD to a CPU architecture which is not already supported by GNU binutils and the GCC.[4]

OpenBSD provides the "cross-tools" and "cross-distrib" targets in the top-level Makefile for the purpose of setting up an initial cross-development environment on another OpenBSD system and to build a minimal distribution for the target system. Although they may work like a charm on some combination of host and target machines, these makefile targets are only considered to be porting aids and are not as well supported as native builds are. Since native builds are a good stress test for any new machine and port, OpenBSD developers usually switch to native builds as soon as possible.

3.3 Boot Loader

The boot loader can be as simple as 50 lines of assembly code, or a complete C program. If the CPU supports JTAG[5], it may be convenient during initial development to use it to load the whole kernel image directly into the target RAM and start execution from there.

In the long run, its a good idea to port OpenBSD's ***boot(8)*** program[6] to the machine. It acts as a first- or second-stage boot loader that provides a convenient way to load the kernel and to test the hardware. Generally it initialises console devices, loads kernel images from supported storage or network devices, passes information about the machine to the kernel, and provides an interactive command line. Porting ***boot(8)*** is fairly straightforward - it is not necessary to deal with the full kernel build infrastructure. The source code is spread across the `sys/stand`, `sys/lib/libsa` and `sys/arch/`*`machine`*`/stand` subtrees.

In cases where a feature-rich BIOS or firmware is lacking but an operating system is already running on your machine, one could significantly speed up the process of porting ***boot(8)*** by running it from that other operating system and replacing the running operating system in RAM with a loaded OpenBSD kernel image on the fly. This is what has been done on the Zaurus.

[4]In theory though, a different toolchain could be used.

[5]Joint Test Action Group, IEEE 1149.1

[6]Traditionally it is called the *stand-alone kernel*.

3.4 Building the Kernel

The main kernel configuration files are located in `sys/arch/`*`machine`*`/conf` and other files are included by them. The ***config(8)*** program sets up a kernel build directory given a particular kernel configuration file in the ***files.conf(5)*** format.

The GENERIC configuration defines a kernel that includes all drivers considered to be stable for any hardware that could possibly be present in a particular machine. The RAMDISK configuration may include less device drivers and may use different options to produce less code. In particular, devices which are not essential for a system installation may not be supported in the RAMDISK configuration. A certain amount of space in the resulting kernel image is reserved for embedding the root filesystem containing the system installation tools.

Compiling the RAMDISK configuration is as simple as compiling the GENERIC configuration, but additional steps are required to embed the actual filesystem into the resulting kernel image *bsd*. Programs to be included must be built in a compressed form because the filesystem is small. A filesystem image of the correct size should be created and written into the reserved space within the *bsd* kernel image. Two more build tools are therefore required: *crunchgen(1)* and *crunchide(1)* for building the compressed programs. Either "elfrdsetroot" or "rdsetroot" can be used to patch the filesystem into the kernel image. Only the *crunch tools* must be built and installed manually from the `distrib/crunch` subtree. The Makefile in `distrib/`*`machine`*`/`*`ramdisk`* already takes care of compiling a kernel, preparing the filesystem image, and patching the kernel using the *rdsetroot* program.

3.5 Startup Code

For every machine or CPU architecture there is a routine called *start()* to which the boot loader jumps after loading the kernel. It is the *start()* routine's responsibility to:

- Disable interrupts,
- Bring the CPU into a predictable state (depends on boot loader),
- Initialise the MMU by setting up preliminary page mappings or disabling the MMU (also depends on the boot loader),
- Set up an initial C program stack,
- Pick up boot arguments from the boot loader,

- Initialise the interrupt controller so that device drivers can register their own interrupt handlers,
- Initialise system timers at some point for *delay()* to work,
- Optionally initialise the console device early - it is done again later by consinit(),
- Probe for available RAM,
- Set up and activate the kernel memory maps, including mapping or relocating the kernel image,
- Set up the permanent kernel stack(s),
- Set up a permanent exception vector or trap table.

After *start()* has finished setting up the MMU and the initial stack it may call other C functions, like *initarm()* on machines with an ARM CPU. Finally, it jumps to the C function *main()* defined in `sys/kern/init_main.c`. The machine-independent startup carried out by *main()* again involves several machine-dependent steps: initialising the console with *consinit()*, completing initialisation of the main CPU with *cpu_startup()*, configuring device drivers with *cpu_configure()*, finding the boot disk with *diskconf()* and finally starting secondary processors on multi-processor systems with *cpu_boot_secondary_processors()*.

3.6 Subsequent Work

The next steps after writing the missing device drivers are roughly to boot the machine multi-user, build the compiler toolchain natively and recompile the whole system natively.

Once the port is running natively, plenty of work still remains if it is to be official:

- Fix most annoying bugs,
- Port ***boot(8)***,
- Document the boot process (***boot_zaurus(8)***, ...),
- Document already supported devices (***intro(4)***, ...),
- Build snapshots, announce the port and make it available,

  - Update web pages,
  - Set up a mailing list,

- Write and document new device drivers,
- Fix more bugs,
- Make the ports tree aware of the new platform, e.g. create "plists".

It's very important to get other people involved with the port as soon as possible. This is to polish documentation and find more bugs and corner cases. Some kind of user community is essential to ensuring the long term survival of the port.

4 Conclusion

At the end of the day, using OpenBSD in an embedded system is entirely viable from a technical perspective but developers must be prepared to use a wide range of resources to find the information they need, including reading source code and possibly involving contacting the developers themselves or consulting mailing lists. One can say that much more independent thinking is required versus commercial solutions. Ultimately, however, developers of embedded systems who choose OpenBSD are rewarded with greater indepdendence and flexibility.

References

[1] Marshall Kirk McKusick, Samual J. Leffler, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley (May, 1989), ISBN 0-201-06196-1, pages 90-91.

[2] Marshall Kirk McKusick, William N. Joy, Samual J. Leffler, Robert S. Fabry, *A Fast File System for UNIX*, Computer Systems, Volume 2 number 3 (1984), pages 181-197.

[3] *OpenBSD Project Goals*, ***http://www.openbsd.org/goals.html***.

[4] *OpenBSD Platforms*, ***http://www.openbsd.org/plat.html***.

[5] *OpenBSD Copyright Policy*, ***http://www.openbsd.org/policy.html***.

[6] Lawrence Kesteloot, *Porting BSD UNIX to a New Platform*, Lawrence Kesteloot (June 28, 1995), available online in several formats, ***http://www.teamten.com/lawrence/291.paper.pdf***.

rthreads: A New Thread Implementation for OpenBSD

Ted Unangst

Abstract

The next generation thread library for OpenBSD will be rthreads. Based on the rfork() system call, rthreads improve the performance, robustness, and scalability of OpenBSD's thread support. In contrast to other recent threading models introduced to BSD systems, rthreads is not based on scheduler activations.

The existing userland pthreads has carried us a long way but it's been showing its age recently. As more applications place more demanding requirements on the thread library its shortcomings become more apparent. This paper will explain these problems, highlight how rthreads resolve them, and then continue with an overview of the rthreads implementation.

Threads

Briefly, threading opens up a new programming model for a developer to use, instead of asynchronous I/O or an event loop. While POSIX defines an API for threads, called pthreads, several implementations are possible. The core of any threading implementation needs to provide two fundamentals, concurrency and synchronization. Concurrency allows a programm to accomplish multiple tasks, while providing the programmer with an abstraction that only one task need be addressed at a time. Synchronization permits multiple threads to interact in an orderly manner.

Userland Threads

One way to implement threads is entirely as a userspace library. The userlandapproach has two advantes. First, it works on operating systems which don't natively support threads. Second, for some tasks, it offers good performance. By not involving the kernel, syscall overhead is avoided.

By the same token, however, the kernel is unaware of the thread library's intentions. This means that it is subject to inoppurtune scheduling by the kernel. There's no true concurrency, but the only illusion of concurrency, achieved by replacing potentially blocking I/O calls with nonblocking calls. In practice, however, nonblocking I/O has a tendency to block, notably when reading from the filesystem. select() and poll() will always indicate that data is available, even when it isn't in the buffer cache. If one thread blocks waiting for data from disk, all threads in the same process block. This drawback severely handicaps the ability of any userland thread library to provide concurrency.

Kernel Threads and Scheduler Activations

To alleviate these shortcoming, support for threads was added to many systems' kernels. Now, the kernel can schedule another thread from the same process to run while one thread waits on disk, and a third thread can be running on a different CPU. A userland library still exists to provide the API, but many tasks, such as thread creation or synchronization, are delegated to the kernel through new syscalls.

The next stage of evolution for many thread implementations was a technique called Scheduler Activations. Originally developed by Anderson, et al, SA expands the userland/kernel thread interface to include a message passing system for all scheduler events. Instead of the kernel scheduler selecting a new thread to run when the currently running thread blocks, a message is sent to the library which then performs the task switch. SA were designed

to improve the performace of operations like thread creation by avoiding a syscall, and increase the flexibility of the userland scheduler, by placing it in full control of thread scheduling.

OpenBSD and Threads

At the current time, the only supported thread model for OpenBSD is a userland library. It suffers from the typical set of problems. Anyone who has used a threaded media player on OpenBSD has likely discovered for themselves that when one thread blocks, they all block.

The introduction of SMP support for the i386 and amd64 architectures also highlighted the fact the because libpthread only utilizes one process, and therefore one scheduling entity, it could not take advantage of multiple CPUs. Several applications such as MySQL are written to utilize threads in an attempt to improve performance.

Rthreads

To address these issues, it was clear that kernel support for threads was required. Instead of an approach based on scheduler activations, implementations of which can be found in both FreeBSD and NetBSD, a direct 1:1 mapping of user threads to kernel threads was selected. The already existing rfork() system call provides a means to create multiple processes that share an address space - in effect, threads. In some ways, this is similar to the LinuxThreads library, particularly the FreeBSD port of which used rfork() as well. However, LinuxThreads relied on an extra control thread, and the kernel was unable to properly distinguish threads from processes.

Kernel Modifications

rfork() typically creates full fledged processes, not threads. Building a thread library directly on rfork() with no additional kernel support is possible; however, this leads to artifacts such as every thread appearing independently in the output of utilities such as ps and top. A new flag to rfork() was added, RFORK_THREAD, to indicate to the kernel that the new process should be considered a part of the parent. A linked list is maintained of threads for each process, similarly to the process sibling list. No separate thread structure has been created in the kernel. Threads are just processes with a special flag set.

All threads created so contain a thread parent pointer, which points to the struct proc for the process. The thread parent pointer for non-threaded processes is initialized to point back to itself. In this way, any access to data which particularly needs to address the process can be done through the thread parent pointer.

The advantage of this approach is that the kernel was made "thread-aware" with only changes to a few files - those dealing with process creation and exiting. When a thread of a process calls exit(), the kernel iterates over the list of sibling threads and also calls exit() for them. A new syscall was added to allow a single thread to exit. No other changes were initially necessary. As time goes on, more changes have been and will be made to more naturally integrate thread awareness into the kernel.

One disadvantage of the approach is that some of the struct proc fields are redundant for a thread. Future work will consider the feasibility of restructuring. In the mean time, the lossage even for thousands of threads measures only a few dozen KB.

In contrast to the schedular activation approach, the direct 1:1 mapping simplifies scheduling. Under SA, when a thread blocks in the kernel, a complex dance of interactions and upcalls is performed to find a new thread. This operation also occurs whenever a timeslice expires. Because an rthread is implemented as just another process in the kernel, nothing special need happen when it blocks. A new process is selected to run and the kernel performs its usual context switch operation.

Syscalls

The sys_rfork() system call is not new, but a modified version of sys_exit() needed to be provided so that one thread could exit by itself. The new syscall, sys_threxit(), simply calls exit1() with a special flag set to indicate only this thread intends to stop. Other threads may wait for an exiting thread using wait() like any other process. sys_getthrid() is the equivalent of sys_getpid(), although it does not map all threads to the parent process's pid. To support voluntary yielding, sys_yield() was added.

In order to support userland mutexes and semaphores, it was necessary to add two additional syscalls, sys_thrsleep(long ident, int timeout, void *lock) and sys_thrwakeup(long ident). These functions export the tsleep() and wakeup() kernel functions to userland. sys_thrsleep() is used to inform the kernel that the current thread wants to cease execution for an extended period (extended really only meaning more than a clock tick). The ident value is entered into a list of idents for the current process, and then tsleep() is called on the address of the list node. This enables the userland process to sleep on any address, much as a process on the kernel can block waiting on any address, while assuring that every process has a unique ident space and without requiring the kernel to interpret userland data. sys_thrwakeup() finds the node with the matching ident, then calls wakeup() on its address. A timeout may be specified to sys_thrsleep() to control the maximum sleep time. The final address is intended to be a spinlock currently held by the calling thread. The kernel will release just before calling tsleep(). It can be used to ensure that a second thread doesn't call sys_thrwakeup() before the first thread is fully asleep.

Library Code

Binary Compatible

The rthreads library is binary compatible with the pthreads library it replaces. The design of the original pthreads library was such that all exposed types are really pointers to opaque types. This means that compiled programs are agnostic to the size and organization of such types.

MD Code

The majority of rthreads code is machinde independent. On a per architecture basis, pieces of machine dependent code must be provided. The first is the rfork_thread(int flags, void *stack, void (*fn)(void *), void *arg) function. This function calls rfork(flags) and returns the thread id of the child to the parent. The child does not return. Instead, it has its stack pointer adjusted and jumps immediately to fn, passing it arg.

The second function is _atomic_lock(_spinlock_lock_t *lock) which performs an atomic compare and swap operation. This function is used to implement the userland spinlock functions.

Synchronization

pthreads includes several types of synchronization operations and data structures, such as simple mutexes, reader-writer locks, and condition variables. In rthreads, these are all implemented as layers on top of semaphores.

The semaphores are implemented using a combination of userland and kernel code. Spinlocks are used to protect the counter that indicates whether the semaphore is available. In the simple acquistion case, the count is adjusted and the spinlock released. In other cases, the thread must block until the semaphore becomes available. Blocking requires finding a new thread to schedule, so on rthreads a syscall is involved to inform the kernel. The blocking

thread calls sys_thrsleep(), passing it both the address of the semaphore and the address of the semaphore's spinlock. The kernel then atomically releases the spinlock, and finds a new process to run or enters the idle loop.

The first thread will remain waiting on the kernel's wait list until a second process increases the semaphore count. If the current semaphore count is 0, the thread calls sys_thrwakeup().

Scheduling

One of the advantages often credited to SA is that the scheduling of threads is under control of the process and not subject to the kernel. Unfortunately, this flexibility comes at the cost of considerable complexity. At present, librthread has only limited control over the kernel scheduler. Ideally, some new syscalls can be added to expose more control to userland without undue complexity. Otherwise, it's possible for a running thread to yield the CPU at designated sequence points.

Future Work

Quite simply, signal handling is one the most complicated aspects of threads to get right. I'd also like to explore re-using kernel threads to improve performance, instead of calling threxit() immediately when finished. Some paradigms create a new thread to acomplish every small task and eliminating two or three syscalls will likely be a remarkable improvement.

Conclusion

The majority of the code and complexity with the old pthreads code dealt with trying to fake nonblocking I/O and scheduling. The requirement to perform the first has been eliminiated entirely, and the second task is now the responsiblity of the kernel. For this reason, rthreads is implemented using only a fraction of the amount of code previously required. rthreads is both a better and simpler replacement.

Thanks

Of course, any discussion of libpthread needs to mention John Birrel, its original author, and all the other FreeBSD developers who worked on it. All the OpenBSD developers, especially anyone who has worked on improving libpthread and who now face adapting many of those changes to librthread.

Bibliography

Anderson, et al. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *ACM Transactions on Computer Systems*, 1992.

Drepper, Ulrich and Ingo Molnar. "The Native POSIX Thread Library for Linux."

Evans, Jason and Julian Elischer. "Kernel-Scheduled Entities for FreeBSD"

Williams, Nathan. "An Implementation of Scheduler Activations on the NetBSD Operating System", USENIX 2002.

FreeBSD Jails in depth. An implementation walkthrough and usefulness example

Matteo Riondato

matteo@FreeBSD.org

Copyright © 2005 Matteo Riondato
July 2005

FreeBSD is a registered trademark of the FreeBSD Foundation.
Motif, OSF/1, and UNIX are registered trademarks and IT DialTone and The Open Group are trademarks of The Open Group in the United States and other countries.
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this document, and the Author was aware of the trademark claim, the designations have been followed by the "™" or the "®" symbol.

Jails are probably one of the best known features of FreeBSD, not only between the BSD aficionados, but also between external people. Introduced in FreeBSD 4.0 by Poul-Henning Kamp, they were greatly enhanced in 5.x and 6.x and became an useful and flexible sysadmin tool.

Although jails'use is widespread in a moltitudo of different tasks, the documents describing their features are mainly out-of-date, due to the fast development that jails undertook in the last year. Therefore, the purpose of this paper is to give an insight view of jails' implementation and a complete description of the sysadmin's tools for managing them. At the end, a proof of concept of jails'usefulness is proposed.

1. No jails implies free criminals

UNIX® was born to share resources between users, not to keep them separated from each other. Although this design decision was obvious twenty years ago, when users were trusted and no menace could come from the outside, today the situation is completely different: always more UNIX systems serve just one local user but their being connected to the Internet makes them vulnerable to remote attacks.

The traditional UNIX model for privileges separation doesn't help to improve the security of the system; on the contrary, having an omnipotent user like `root`, can be a serious threat to the security: once an attacker has gained super-user powers through an insecure application running with `root` privileges, he can do everything on the system.

To workaround these design choices, many different solutions were implemented to offer a more granular access control over system resources but most of them resulted in a mixture of access lists and similars so difficult to administer that the time needed to configure them made their advantages insightful.

At the same time, other solutions were created to limit the disastrous effects of a succesful privilege escalation. The chroot(2) system call and its omonymous user-space companion are an example. What `chroot` does, is changing the root directory of a filesystem sub-tree so that processes running in the `chroot`ed environment can not access outside resources. Implemented in 4.2BSD, over the years, `chroot` revealed its weaknesses both in a security and in a flexibility point of view: ways to escape from a `chroot`ed environment were found and the limitations in terms of system administration became evident.

Jails were created not just to unify the advantages of both solutions without taking the disadvantages of them, but to solve the problem of fine-grained security in a simple yet powerful way.

A jail can be thought as a FreeBSD system inside another FreeBSD system: it has its own / directory (heritage of chroot(2)), its own `root` super user, its own IP address. In a sense, the concept of jail is similar to the concept of a virtual machine, although jails does not emulate hardware resources. Due to this and other limitations, the `root` user of the jail has not that illimitated power that makes his corrispondent in the host system so peculiar: he cannot change the IP addess, cannot create device nodes, cannot run some services and, what is more, cannot shutdown his own jail. Anyway, these limitations havJe proved not to restrict jails'usage.

2. *"Go to Jail. Do not pass Go"* (Monopoly)

As many other FreeBSD features, the jail system consists in a kernel part and in an userland one. The kernel part is mainly represented by the jail(2) system call, while the userland part is the jail(8) binary. In the following sections the implementations of both are described, although, for the jail(8) binary we'll speak more about its usage rather than about its implementation in terms of code.

2.1. The jail(8) binary

The jail(8) binary resides in `/usr/bin/` and is the main consumer of the jail(2) syscall, so we start our jail system's implementation description from it. The source for this program resides under `/usr/src/usr.sbin/jail/` and consists in the `jail.c` file, distribuited under Poul-Henning Kamp's Beerware License. The main task of this ~150 lines file is to fill a `jail` struct (included from `<sys/jail.h>`) with the command line arguments provided by the user and to call the jail(2) syscall. The `jail` struct has just four fields:

```
struct jail {
 u_int32_t version;
 char  *path;
 char  *hostname;
 u_int32_t ip_numer;
}
```

The last three fields are self-explainatory: a filesystem subtree path, an hostname and an ip address are associated to each jail and by default they cannot be changed neither from inside nor from outside the jail during jail's life.The `version` is set to 0 when the jail is created and it is just an identifier for the API version.

Once jail(8) has filled the `jail` struct, it calls the jail(2) syscall as follows:

```
struct jail j;
```

[...]

```
i = jail(&j);
```

To complete the creation of the jail environment, jail(8) forks and his child process executes the command specified by the user on the command line:

```
if (execv(argv[3], argv + 3) != 0)
 err(1, "execv: %s", argv[3]);
exit(0);
```

jail(8) has now completed its tasks and can exit without errors (returning `0`).

2.2. Are jail and prison the same thing?

The answer is: *"No, they aren't"* but let's proceed with order.

In the previous section we analized how jail(8) works but, as the reader probably understood, the dirty job is accomplished by the jail(2) system call, so its implementation is, without doubts more interesting.

The file `/usr/src/sys/kern/kern_jail.c` contains the majority of jail-related kernel code: the jail(2) syscall is implemented in this file, together with `security.jail` sysctls and other routines strictly connected with jails' usage.

Although sysctls' code comes earlier in the file, we start our description from the jail(2) syscall's implementation:

```
int
jail(struct thread *td, struct jail_args *uap)
```

As you can see, the declaration for the syscall requires that two arguments should be passed, while, when called from the userland, just one is needed. This hidden first argument is present in every FreeBSD's syscall's code and represents the caller thread. The second argument too is different from the one expected: it is not a pointer to a `jail` struct, but to a different structure `jail_args`. This structure, defined in `<sys/sysproto.h>`, is just a fake structure that incapsulates the `struct jail` pointer passed at call time.

After having examined how the arguments list changed, we can concentrate on the code. The first operation accomplished by jail(2) is copying the `jail` struct from user- to kernel-space:

```
error = copyin(uap->jail, &j, sizeof(j));
```

Before going on with the next step, a new structure should be introduced: `prison`. As the name suggests, this structure is strongly related to the `jail` one: it's defined in the following way in `<sys/jail.h>`:

```
/*
 * Lock key:
 *   (a) allprison_mtx
 *   (p) locked by pr_mtx
 *   (c) set only during creation before the structure is shared, no mutex
```

```
 *        required to read
 *    (d) set only during destruction of jail, no mutex needed
 */
#if defined(_KERNEL) || defined(_WANT_PRISON)
struct prison {
        LIST_ENTRY(prison) pr_list;                     /* (a) all prisons */
        int              pr_id;                         /* (c) prison id */
        int              pr_ref;                        /* (p) refcount */
        char             pr_path[MAXPATHLEN];           /* (c) chroot path */
        struct vnode    *pr_root;                       /* (c) vnode to rdir */
        char             pr_host[MAXHOSTNAMELEN];       /* (p) jail hostname */
        u_int32_t        pr_ip;                         /* (c) ip addr host */
        void            *pr_linux;                      /* (p) linux abi */
        int              pr_securelevel;                /* (p) securelevel */
        struct task      pr_task;                       /* (d) destroy task */
        struct mtx       pr_mtx;
};
```

The fields are clearly explained by the comments and some of them are just the correspondants of other `struct jail`'s fields but others need at least a quick description. The kernel maintains a double linked list of existing jails, or better, of existing `prison` structs: the first field of the struct is, in practice, a pointer to this list. The `pr_id` identifies the jail in a unique way, since this field is set at jail's creation time and never changed thereafter. `pr_ref` is a counter for jail's process and is fundamental during the jail's lifetime: this counter is set to 1 at creation time and is incremented each time a process is created inside the jail and decremented once it dies. When it reaches a value of zero, the hooks in the process creation and destruction code free the entire prison structure. More on this later. A jail can have the Linux Binary Compatability Layer enabled and, if this is the case, the `pr_linux` field will point to a `linux_prison` struct (defined in `/usr/src/sys/compat/linux/linux_mib.c`) containing infos about the compatability layer (e.g. version). Kernel `securelevels` are one of the strongest forms of security, since they can affect he ability to apport crucial modifications to the system. A jail may have a `securelevel` different from the one set in the host system; obviously, for processes inside the jail, the higher is used. Jail's `securelevel` is set in the `pr_securlevel` field which, at creation time, is equal to that of the host system. In the end, `pr_task` is a pointer to the `task` structure that will be initialized and used when the jail needs to be destructed and its components freed. The `pr_mtx` is obviously the mutex protecting the `prison` structure and some of its fields.

Let's go back to the way jail(2) works: after having copied the `jail` structure in kernel space, it initializes a `prison` struct and fill it with the right values, either taken from the `jail` struct or set with a default value. At this time, the jail infrastructure is ready, but it must be conncected to the system; this is done by calling the jail_attach(2) syscall from the jail(2) syscall:

```
error = jail_attach(td, &jaa);
```

The `td` argument is a pointer to the current thread, while the second argument is a pointer to a `struct jail_attach_args` containing only one field, `jid`, to identify the jail. The action of jail_attach(2) consists of putting the process which the `td` thread belongs to in the new root directory as specified in the `prison` structure. It then modifies some fields of the `proc` structure so that the process gets the correct user credentials. jail_attach(2) returns to the caller jail(2) which in turns, if everything went right, returns to the calling process. The jail is now ready and the jail(8) binary can execv(3) the user specified command to start it.

2.3. Other jail guards

`kern_jail.c` contains the code for routines others than jail(2) and jail_attach(2), such as `prison_find` and `prison_free`, but their implementation is straight-forward so it will not be disscused here. At the beginning of the file, many `sysctls` are defined in the following way:

```
SYSCTL_DECL(_security);
SYSCTL_NODE(_security, OID_AUTO, jail, CTLFLAG_RW, 0,
 "Jail rules");

int      jail_set_hostname_allowed = 1;
SYSCTL_INT(_security_jail, OID_AUTO, set_hostname_allowed, CTLFLAG_RW,
 &jail_set_hostname_allowed, 0,
 "Processes in jail can set their hostnames");
```

Only one of them is presented here, since the code for the others is pretty similar. Actually, the first part of the code. creates the `security sysctls`' class, while the call to `SYSCTL_NODE` creates the `security.jail sysctls` subtree, or better, creates the root for it. The first argument is taken from `SYSCTL_NODE` is the parent `sysctl` tree and it can be seen as the mount-point for the `sysctl` subtree's root. The second argument, `OID_AUTO` is a magical number and its use will not be explained here; please note that nearly every FreeBSD `sysctl`'s code has this number as second argument. `jail`, the third argument, is the name for the subtree that is going to be created. `CTLFLAG_RW`, the fourth argument, means that the values of the `sysctls` can be changed by the user. The fifth argument should be an handler, that is, a pointer to a routine that should be called when more `sysctls` are added behind it. Here it is set to `0` because it is not needed. The last argument is a short description for the `sysctl`.

Now that both `security` and `security.jail` exist, children `sysctl` can be created with calls to `SYSCTL_INT`. This routine takes seven arguments: the *mount point* for the newly created `sysctl`, the magic `OID_AUTO` number, the new `sysctl`'s name, the flag that specifies wether the value can be changed, a pointer to the variable which to store the value in, the default value and a short description of the `sysctl`.

The `sysctls` created in `kern_jail.c` are:

- `security.jail.`*sysctl_name: default_value*
- `security.jail.set_hostname_allowed: 1`
- `security.jail.socket_unixiproute_only: 1`
- `security.jail.sysvipc_allowed: 0`
- `security.jail.enforce_statfs: 2`
- `security.jail.allow_raw_sockets: 0`
- `security.jail.chflags_allowed: 0`
- `security.jail.jailed: 0`

The use of each `sysctl` will be explained in Section 3.3a later section

Apart from `kern_jail.c`, many files in `/usr/src/sys/kern/` contain jail-related code. Many of them control the creation and the distruption of processes and, to accomplish their tasks, they must control wether the new process' parent is jailed or the dying process was in jail. One way to do this is calling the `jailed(struct ucred *cred)` routine defined in `kern_jail.c`: it returns `1` if the examined thread is in a jail, `0` otherwise.

Other checks for jailed processes are present throughout the code in many files. grep(1) can help finding them.

2.4. Future Developements

`"There is always room for improvement" (Anonymous)`. Although the jail subsystem seems fairly complete and its implementation straight forward, it can be improved in many ways. First of all, a jail can have only one IPv4 address and this can be restrictive. Pawel Jakub Davidek has patches for this in his `pjd_jail` branch on the FreeBSD Perforce Repository (http://cvs.freebsd.org/). Another feature that should be implemented is a clean way to shut down a jail: jails' start sequence is clearly documented and implemented, but a jail cannot be thought as a complete system when it comes to shutdown.

Together with the implementation of new features, code auditing is a must and should be performed on the totality of the FreeBSD code, since information leaks can exist everywhere.

3. The perfect Daemon County Jail Director (or "How to build, run and administer a jail under FreeBSD")

In the first part of this paper, the implementation of the jail subsystem was described in depth. This section is addressed to system administrator and users who want to understand how to make jails work.

3.1. Building the jail

As the jail(8) man page clearly states, building a jail requires the following steps:

```
D=/here/is/the/jail
mkdir -p $D ❶
cd /usr/src
make world DESTDIR=$D ❷
make distribution DESTDIR=$D ❸
mount_devfs devfs $D/dev ❹
```

Then the jail can be started. The meaning of the above steps should be understood to anyone having even little experience with FreeBSD, anyway, they are quickly explained here.

❶ Obviously, the first thing to do is to decide where the jail will physically reside in the host system. A good choice can be `/usr/jailN`, where *N* is a number identifying the jail. `/usr/` usually has enough room for the jail filesystem, which is, in practice, a replication of every file present in a default FreeBSD installation.

❷ This command will populate the directory chosen for the jail filesystem subtree with the necessary binaries, libraries, man pages and so on. Everything is done in the typical FreeBSD's style: first build/compile, then install. This command alone can also be used to upgrade a previously created jail.

❸ The `distribution` for **make** installs every needed configuration file, so in poor words, it copies `/usr/src/etc/` to `$D/etc/`.

❹ To work, a jail does not need `devfs` to be mounted in it. However, any, or quite any, application requires access to at least one device. It is really important to control access to devices from inside a jail, as improper settings could bypass the jail sandboxing.Refer to the devfs(8) man page for more info.

Final hint about building a jail: there are two kinds of jails, the *complete* and the *minimal*. The difference between the two is that a complete jail is really an FreeBSD system inside a real system, while a minimal jail just has the binaries and the libraries needed to run a particular program, usually a daemon. The easiest way to build a minimal jail is to start from a complete jail and then remove the unnecessary parts of it. Be conservative in what you remove.

3.2. Starting the jail

As said before, the jail(8) command is used to start a jail. It takes a variety of arguments and only some of them will be examined here, so please refer to the man page for additional info.

jail(8) is usually invoked with four parameters as follows:

```
#jail path hostname IPaddress command
```

The meaning of each argument should be clear. A little note about the `IPaddress` argument: it should be an alias to network interface configured in the system. To set this up, proceed as follows:

```
# ifconfig ifX inet alias aliasIPaddress
```

The `command` argument too may need some clarifications. If you are running a minimal jail, you should pass the path of the deamon you want to be run inside the jail. On the contrary, if the jail is a complete one, the best choice is to have the `/etc/rc` script run, as it will replicate the starting of a real FreeBSD system, setting up services and applying configuration settings.

Usually jails are started a boot time and the FreeBSD `rc` mechanism provides an easy way to do that. First of all, a general section about jails should be added to `/etc/rc.conf`. It contains instructions for enabling jails' starting.

```
jail_enable="YES"        # Set to NO to disable starting of any jails
jail_list="www"          # Space separated list of names of jails
```

Together with this, an entry similar to the following should be added to `/etc/rc.conf` for each jail to be started and the `/etc/rc.d/jail` script, called at boot-time, will start it.

```
jail_www_rootdir="/usr/jail/www"    # Jail's root directory
jail_www_hostname="www.domain.com"    # Jail's hostname
jail_www_ip_="192.168.0.10"       # Jail's IP number
jail_www_exec_start="/bin/sh /etc/rc"     # command to execute in jail for starting
jail_www_exec_stop="/bin/sh /etc/rc.shutdown"  # command to execute in jail for stopping
jail_www_devfs_enable="YES"       # mount devfs in the jail
jail_www_fdescfs_enable="NO"       # mount fdescfs in the jail
jail_www_procfs_enable="NO"       # mount procfs in jail
jail_www_mount_enable="NO"        # mount/umount jail's fs
jail_www_devfs_ruleset="www_ruleset"      # devfs ruleset to apply to jail
jail_www_fstab=""          # fstab(5) for mount/umount
```

The `/etc/rc.d/jail` script can be used to start or stop a jail by hand, if an entry for it is in `rc.conf`. The following statement shows an example of this feature:

```
# /etc/rc.d/jail
   start
```

3.3. Managing the jail from inside and outside

As they reside entirely in a real FreeBSD system, jails can be administer from outside, by the super user in the host system, and from inside, by the jail's `root`. It was already mentioned that they have different levels of powers and this is reflected in the administration of the jail. In the following subsection jail's administration is covered both from the outside and from the inside.

3.3.1. Jail's Administration from outside

The `root` user of the host system can deeply influence the powers of the jail's superuser. These can be accomplished thanks to many `sysctls` and others tools.

Jail-related `sysctl` were presented in a previous section and their use is accurately described in the jail(8) man page, so please refer to it for additional information.

Some tools useful for jail administration are included in the base system, while others, less useful, are in the ports collection. jls(8) and jexec(8) are the ones in the base system: the first prints a list of active jails and their correspondant jail identifier (JID), IP address, hostname and path. More useful than jls(8) is surely jexec(8), which allows the `root` in the real system to run programs inside the jail environment, as if he was the superuser of the jail. This is expecially useful when the `root` want to cleanly shut down a jail.

Among the many suites of tools for jail administration that can be found in the ports collection, the probably most complete and useful is `sysutils/jalutils` (http://memberwebs.com/nielsen/freebsd/jails/jailutils/), a set of small applications that help the administrator in jail management. Refer to the web page for more info.

3.3.2. Jail's Administration from inside

Administering a jail from inside is not too different from administering a real FreeBSD system. In this section, we will cover more what the jail's `root` can not do than what he can.

Obviously the main limitations to the jail's superuser are kernel-related ones and those that come from `sysctls`' setting done by the external `root`. The superuser inside the jail is not allowed to mount filesystems nor to unmount them. He can not change devfs(8) ruleset as the in-jail mounted device filesystem was mounted from the outside. He can not set firewall rules and is not allowed to do many other administration tasks that requires modifications to in-kernel data.

Limitations imposed by the kernel `root` using `sysctls` were covered in the previous section.

3.4. Shutting down the jail

As said before, a clean way to shut down a jail is not available at the moment as commands normally used to accomplish the shutdown cannot be used inside a jail. The least ugly way to halt a jail is to run

```
# sh /etc/rc.shutdown
```

from inside it or using jexec(8). More info on this can be found in the jail(8) man page.

4. A Proof of Concept for Jail Usefulness

This part will be outlined during the talk.

5. Conclusions

Jails are a powerful and flexible tool that every system administrator should know and use. Their widespread usage in environments with high security requirements demonstrates how much they can be trusted. Their usefulness can be applied in many different environments, from UNIX courses for students to SSH account servers, to hosting. The future for them is clear and shiny, with the promise of new features' addition that would make them even more powerful.

Bibliography

The Design and Implementation of the FreeBSD Operating System, pp. 123-129, Marshall Kirk McKusick and George V. Neville-Neil, Addison-Wesley, 2004.

Jails: Confining the omnipotent `root`. *(http://docs.freebsd.org/44doc/papers/jail)*, Poul-Henning Kamp and Robert Watson, The FreeBSD Project, 2000.

Using Jails in FreeBSD for Fun and Profit (http://www.usenix.org/publications/login/2002-06/pdfs/hope.pdf), Paco Hope, USENIX Association, June 2002.

Inside Jail (http://www.daemonnews.org/200109/jailint.html), Evan Sarmiento, Daemon News, September 2001.

jail(2) man page, The FreeBSD Documentation Project, The FreeBSD Project.

jail(8) man page, The FreeBSD Documentation Project, The FreeBSD Project.

Conference Sponsors

Conference Venue:

Universität Basel
Petersplatz 1, CH-4003 Basel
http://www.unibas.ch/

Silver Sponsor:

The FreeBSD Foundation
7321 Brockway Dr., Boulder, CO 80303, USA
http://www.freebsd.org/

Bronze Sponsor:

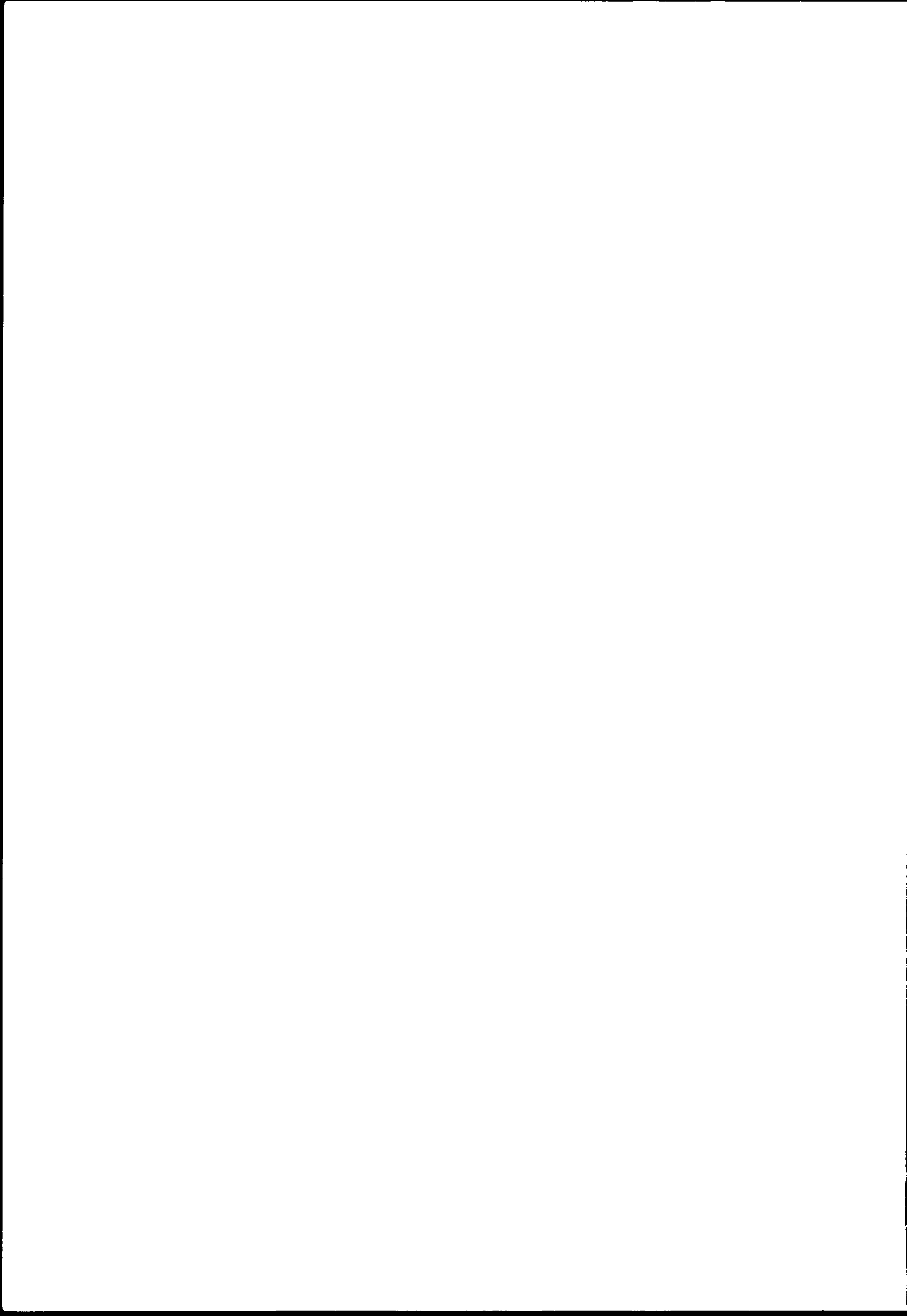
HOSTPOINT AG
Zürcherstrasse 2, CH-8640 Rapperswil
http://www.hostpoint.ch/

GENOTEC Internet Consulting AG
Hegenheimermattweg 119a, CH-4123 Allschwil
http://www.genotec.ch/

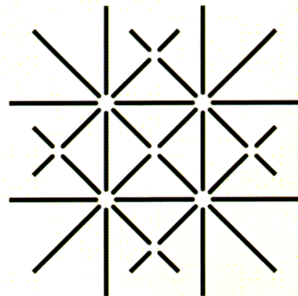
Sponsor:

Improware AG
Zurlindenstrasse 29, CH-4133 Pratteln
http://www.imp.ch/

BSD Consult
Rued Langgaardsvej 7, 5D-19, DK-2300 Copenhagen S
http://www.bsdconsult.dk/

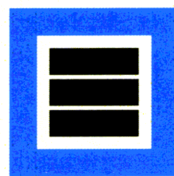


We thank our Sponsors:



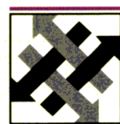
UNI
BASEL

The FreeBSD Foundation



HOSTPOINT

THE DATA RESIDENCE



ImproWare AG

www.imp.ch

Offsetdruck
Digitaldruck
Siebdruck

Druckerei Dietrich AG
CH-4019 Base, Pfarrgasse 11
Tel. 061 639 90 39