AsiaBSDCon 2007
Proceedings

March 8-11, 2007
Tokyo, Japan

INDEX

A NetBSD-based IPv6/NEMO Mobile Router

Jean Lorchat, Koshiro Mitsuya
Keio University
Graduate School of Media and Governance
Fujisawa, Kanagawa 252-8520, Japan
Email: lorchat,mitsuya@sfc.wide.ad.jp

Romain Kuntz
The University of Tokyo
Gr. School of Information Science and Technology
Information and Communication Engineering
Email: kuntz@sfc.wide.ad.jp

Abstract— **This paper defines the problem statement of vehicle-embedded networking in order to communicate with the infrastructure (the Internet) as well as with other cars. Based on this problem statement, we explain the steps that allowed us to build a mobile router addressing this problem by using state of the art software. This software includes the NetBSD-current kernel and networking code developed by the Japan-based WIDE project working groups: the KAME IPv6 stack with SHISA extensions for Mobile IPv6 (MIPv6) and Network Mobility (NEMO) support, and the Zebra-based OLSR daemon with IPv6 extensions allowing for a Mobile Ad Hoc Networks (MANET) and NEMO cooperation, formerly known as MANEMO.**

I. Introduction

Current research on Intelligent Transportation System (ITS) focuses on vehicle-to-vehicle communication and information exchange between vehicles and the infrastructure network (Internet). In the near future, vehicles will embed various computers and sensor nodes, making a network, whose data will be exchanged with nodes in the Internet for various purposes such as monitoring, safety, or entertainment. For that purpose, the CALM[1] (Communications, Air Interface, Long and Medium Range) architecture specified at ISO (TC204, WG16) recommends the usage of IPv6 and IPv6 mobility protocols to ensure permanent and uninterrupted communication while moving in the Internet topology.

NEMO Basic Support [1], specified at the IETF in the NEMO Working Group [2], was standardized to solve the IPv6 network mobility problem. NEMO Basic Support allows a group of nodes to connect to the Internet via a gateway: the mobile router. It can change its point of attachment to the IPv6 Internet infrastructure while maintaining all the current connections transparently for all the nodes within the mobile network and their correspondent nodes. The only node in the mobile network that is managing the mobility is the mobile router itself. It updates its current location in the Internet to a special router known as the home agent, which is located in the home network. The home agent maintains a table of relationships between mobile routers permanent addresses, temporary addresses, and mobile network prefixes. All the traffic to or from the mobile network is exchanged between the mobile router and the home agent through an IPv6-over-IPv6 tunnel. This protocol can then be used to bring global access to any car component by adding a mobile router to the car environment. This is one of our goals in the InternetCAR project [3], and we show our communication model on Fig. 1.

NEMO Basic Support is thus a very likely architecture to ensure permanent connectivity in mobile environments. Although the main use case would be to connect any kind of vehicles to the Internet (such as cars, trains, buses, etc.), the underlying architecture would be the same (IPv6 and NEMO Basic Support) but customized at the application layer according to the usages: monitoring, safety, entertainment, etc.

The Internet Connected Automobile Research (InternetCAR) Working Group was established within the WIDE Project [4] since 1998 to con-

[1]http://www.calm.hu/

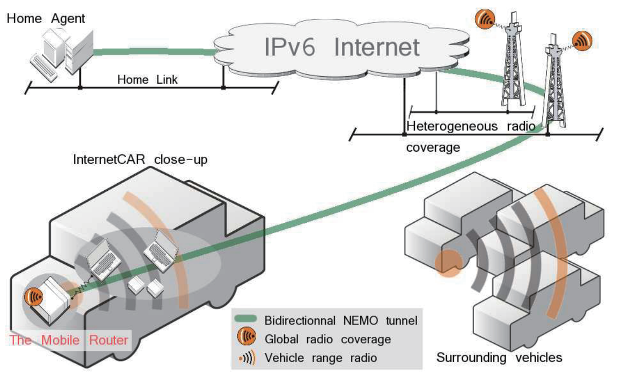


Fig. 1. InternetCAR communication model

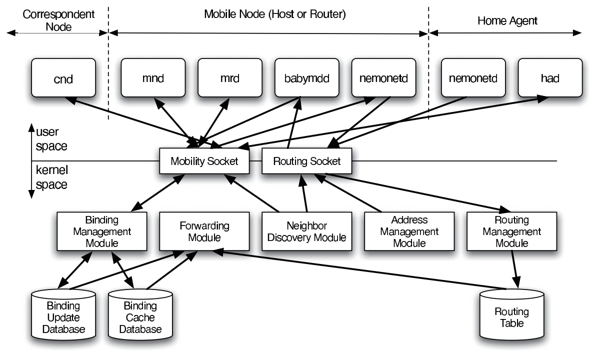


Fig. 2. SHISA daemons architecture

nect vehicles to the Internet [5], [6], [7] by developing the necessary tools and demonstrate their applicability in a real-life testbed. We aim to implement all the missing protocols to build an in-vehicle Mobile Router. It is designed to support IPv6 mobility and multihoming (simultaneous usage of several interfaces to share and load-balance the traffic, or for fault-tolerance purposes).

InternetCAR is also developing a monitoring application for demonstration purposes. Most of the outputs are freely available implementations for BSD operating systems, such as contribution to the KAME IPv6 stack [8], and the SHISA mobility stack [9].

The previous in-vehicle mobile router was based on the NetBSD 1.6.2 Operating System. The KAME IPv6 stack and SHISA stack [10] were used to manage the network mobility. Basic multihoming features allowed vertical handover between Wireless LAN and cellular interfaces. However, no vehicle-to-vehicle communication is possible, and no monitoring software on the mobile router makes the evaluation of the system difficult.

The purpose of this paper is twofold: first we make out and explain the constraints of such in-vehicle network architecture for both hardware and software sides. We then explain the work done on the implementation side to build a new version of the InternetCAR's in-vehicle mobile router, based on the NetBSD operating system with respect to the previously defined constraints. We then show an evaluation of this work.

II. A Constrained Environment

As explained in the previous section, one goal of the InternetCAR project is to build an embedded mobile router that is suitable for car operation. And while the car environment is not as constraining as some other environments from the embedded computing domain, it still has some features that must be accounted for when choosing hardware and software solutions to specific problems. We will explain these environment peculiarities by distinguishing between hardware and software considerations.

A. *Hardware related considerations*

Since the car is moving, we have to pay extra attention to the toughness of the hardware. The road conditions can make the embedded computer bump, so that moving parts should be avoided as much as possible. And of course, as long as the car is moving, it makes wireless communications (and especially all kinds of radio communications) mandatory.

As an embedded car equipment, the size of the mobile router must be kept to a minimum in order to save space and weight inside the car. This means avoiding traditional personal computer design and looking for single board computer alternatives.

Although AC current from the mains is not directly available inside the car, we have sufficient access to power resources through the car power source, either the alternator or the car battery.

Eventually, cars are equipped with lots of cable, yet there might be no networking cables available to interconnect equipments within the car, especially for current vehicles. Which means that we might have to resort to wireless communications again to provide in-car networking.

B. Networking related considerations

We have already introduced the ISO specified recommendation about IPv6 and NEMO protocols. These protocols allow to hide mobility in a very convenient way, especially by relieving car equipments from any mobility-related overhead, handled by the mobile router.

However, in addition to these next generation mobility protocols, we need to take the vehicle to vehicle (V2V) scheme into account. Using the previously mentioned mobility protocols, V2V communication can become very painful because the traffic has to go all the way back to both home agents. In this case, protocols for mobile ad hoc networks can be used to discover neighboring mobile routers and to route traffic between them.

This means that several daemons are going to be responsible for route injection (i.e. the mobility daemons and the ad hoc network daemons) and it is mandatory that they can collaborate in an efficient way. Above all, since mobility daemons are going to define the default route through the NEMO tunnel, this must be done in the regular way that allows longer prefixes to retain priority during the routing process.

C. Problem Statement

From the previous observations, we define the following problem statement: "*How to provide optimized connectivity to a moving car environment and all its attached equipment and users ?*". Keeping in mind that the recommended architecture from ISO must be followed to maintain compatibility with future solutions, but slightly amended to optimize V2V communication. And while these software goals must be achieved, we must also obey the guidelines that come from hardware constraints of the car environment.

III. InternetCAR team implementation

A. Hardware Platform

By taking all hardware-related constraints into account, we decided to use a Soekris[2] single-board computer as the key element for our mobile router. While it features an integrated processor, memory and two network controllers on the same board, it also provides room fox expansion with two Cardbus and one MiniPCI slots. This will allow us to use many wireless network interfaces.

The mass storage is fitted through a Compact Flash slot, which can accommodate either a flash card or a micro hard-disk. We chose to use flash cards to achieve a zero-spin architecture, which means that our mobile router has no moving part.

B. Operating System Choice

We chose the NetBSD operating system for several reasons. The first being the portability, which guarantees that switching from the current hardware platform to another will be easier. And since we wanted to use MiniPCI wireless network interfaces, we decided to use recent NetBSD kernels, as compared to the previous version of the mobile router.

NetBSD-Current (as of mid-October) was installed on the compact flash card using a laptop. Then, we proceeded to the customization of the root filesystem. This included removing system services that are not useful for our embedded operation, while adding specific services related to networking stack and monitoring (see III-C to III-F).

Then we modified the mountpoints so that the root filesystem is mounted read-only (to protect the flash storage) while a memory filesystem is mounted in frequently written locations like `/var/log`, `/var/run` and `/tmp`.

[2]http://www.soekris.com

C. Mobility protocols

As Mobile IPv6 and NEMO Basic Support implementation, SHISA is freely available for BSD variants. However, SHISA is an extension of KAME IPv6 implementation and it is not available for any BSD main branch yet. We thus have ported SHISA to NetBSD-Current with the help of the SHISA developers team.

The main design principle of SHISA is the separation of signaling and forwarding functions. The operation of Mobile IPv6 and NEMO Basic Support is basically IP packet routing (forwarding or tunneling). In order to obtain better performance, the packet routing should be done in the kernel space. The signal processing should be done in the user space, since the process is complex and it is easier to modify/update user space programs than kernel. This separation provides both good performance and efficiency in developing the stack.

In SHISA, a mobile node (host or router) consists of small kernel extensions to process mobility headers and to forward packets, and several user land daemons (MND, MRD, BABYMDD, MDD, NEMONETD, HAD and CND). MND/MRD are daemons which manage bindings on a mobile node. BABYMDD/MDD are daemons which detect the changes of temporary addresses and notifies them to MND or MRD. NEMONETD is a daemon which manages bi-directional tunnels. HAD is a daemon which manages bindings on a home agent. CND is a daemon which manages bindings on a correspondent node. Depending on the node type, one or several SHISA daemons run on a node. For instance, MRD, NEMONETD and MDD are running on our mobile router. The relationship between user land daemons and kernel is detailed in Fig. 2.

D. Mobile Ad Hoc Network protocol

To address the efficiency problem of V2V communication, we explained in section II-B that it would be possible to route traffic using a Mobile Ad Hoc Network (MANET) protocol. We decided to base our work on a proactive routing protocol because the neighboring nodes discovery process is more tightly integrated by distributing topology information. The protocol we chose is OLSR [11].

Although the RFC does include enhancements to announce host/network associations, it is not definite about IPv6 networks nor IPv6 hosts. We use a routing daemon developed in Keio University and functioning as a plug-in to the Zebra framework. It is already going beyond the scope of the RFC by supporting IPv6 hosts.

We modified this daemon to support IPv6 mobile routers announces (i.e. which prefix is reachable through which node) and added support for sub-second precision in Zebra timers. It is then possible to discover mobile routers that are nodes of the MANET at the current time, and use OLSR routing services to route V2V traffic. This is further described in [12] where we show that it is possible to discover new nodes and broken links in less than 100 milliseconds.

E. Interface management

Since an automobile moves around, the mobile router in the automobile needs to perform handovers between wireless access points. For this operation, it is required to carry out discovery of base stations, to connect to one of these stations, and to detach from the previous base station.

Casanova is an automatic Wireless LAN SSID/Wepkey switcher for FreeBSD and NetBSD. Casanova handles ESSID and wepkey and configures the wireless network interface while handing over from one access point to another. Casanova also acts as an IPv6 address manager. It requests to configure a new IPv6 address when the mobile router is connected to a new link, and to remove an old IPv6 address when disconnected.

F. Monitoring software

The embedded nature of the mobile router requires a flawless operation. However, there are some cases where mobile router operation is not possible, especially when no connection is available. This can be reported by our dedicated monitoring software. There are two small daemons sharing the same codebase. One is showing statistics locally using the front LED of the Soekris box (to diagnose connectivity problems). The other one is the SONAR[3] client, which records and prepares the data for transmission to a remote repository. It can be stored in the filesystem too, and sent later so as not to disturb ongoing experiments.

The monitoring architecture is very modular and allows for fast development and integration of new features to monitor. It is written in C and supports several platforms: NetBSD, FreeBSD and Linux. Statistics are polled on a regular basis using a user-defined interval, between a few milliseconds and several hours. This requires interaction with routing daemons (SHISA and OLSR) and the kernel.

A report is built for each polling interval using an XML tree structure. These reports can be sent regularly (another user-defined interval) or even kept in a local storage area and collected later. For an in-depth presentation of the monitoring architecture, please refer to [13].

IV. Evaluation of implementation

The mobile router that we described in this paper was implemented during the October month in year 2006. It can be seen on Fig. 3.

The picture on Fig. 3(a) shows the single-board computer (Soekris net4521) that we use to implement our mobile router architecture. Both pictures on Fig. 3(b) and Fig. 3(c) show experiments that we made using the mobile router. The former is an electric car manufactured by Toyota which has all approvals required to be driven on open road. The mobile router can be seen just behind the driver's seat. The latter illustrates the experiment led at the Open Research Forum in Tokyo (October 2006), where we equipped an electric bus with the mobile router, SNMP sensors and an IPv6 camera.

Before the implementation process, we were facing some questions about the behavior of networking component with respect to the each others. However, the interaction between the SHISA/KAME stack and the Zebra daemons went very smoothly. In fact, the main problem we faced during the implementation was a routing issue when using the KAME stack : at that time, it was impossible to send messages from SHISA daemons through a gif tunnel because the endpoints were not recorded in the neighbor cache.

With this achievement, we could perform some measurements on the workbench before the mobile router is put into vehicles. We especially investigated the booting time for the whole system and the amount of time required before the mobile router is successfully registered to its Home Agent, or before the network interface becomes available. These results are shown in table I.

Following this implementation, several software releases will happen. The casanova program used for interface management has already been released[4]. The SONAR client used for statistics monitoring is due to be released by the end of the year. Although we used a NetBSD-current port of the SHISA stack for NetBSD-current, we can not tell for sure when it is going to be released to the public.

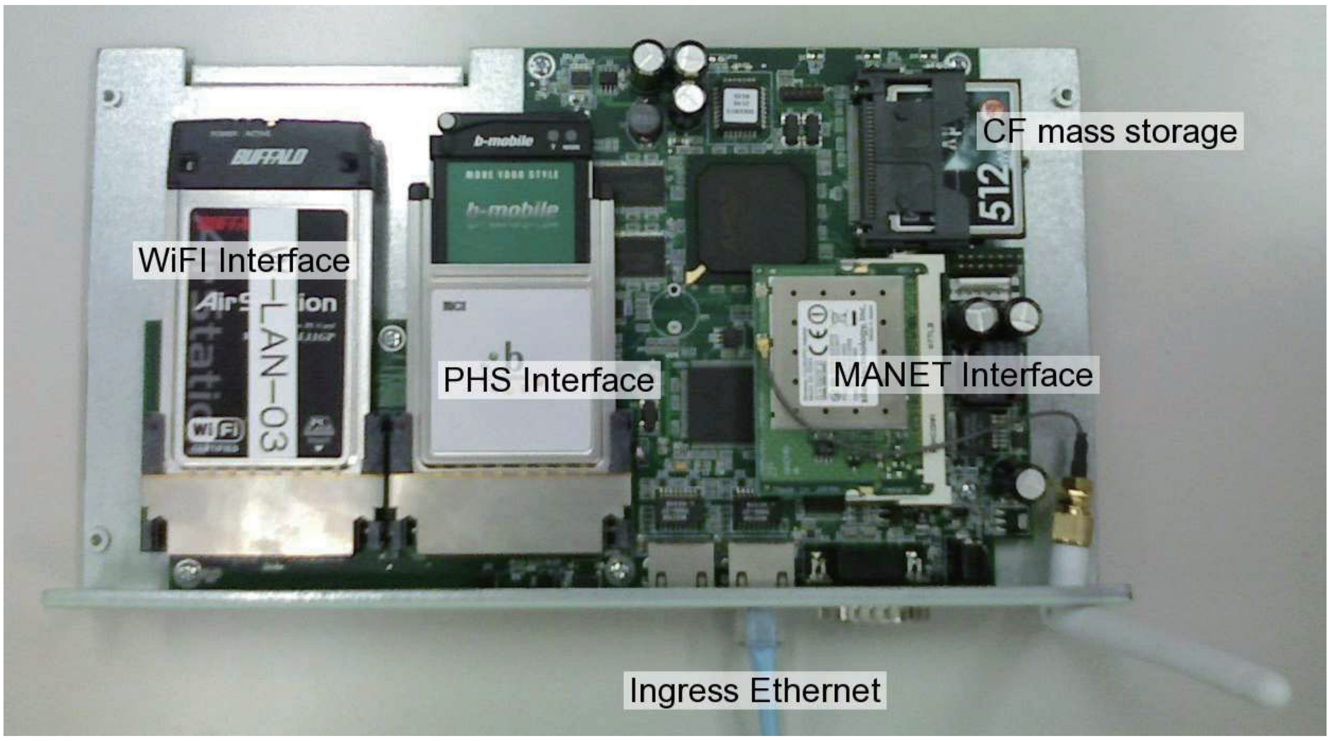
V. Conclusion

After a thorough description of the target environment and its constraints, we could define a specific problem statement that led us to the current implementation of a in-vehicle mobile router. It is based on the NetBSD-current (October 2006) operating system and uses several advanced networking code from the WIDE project.

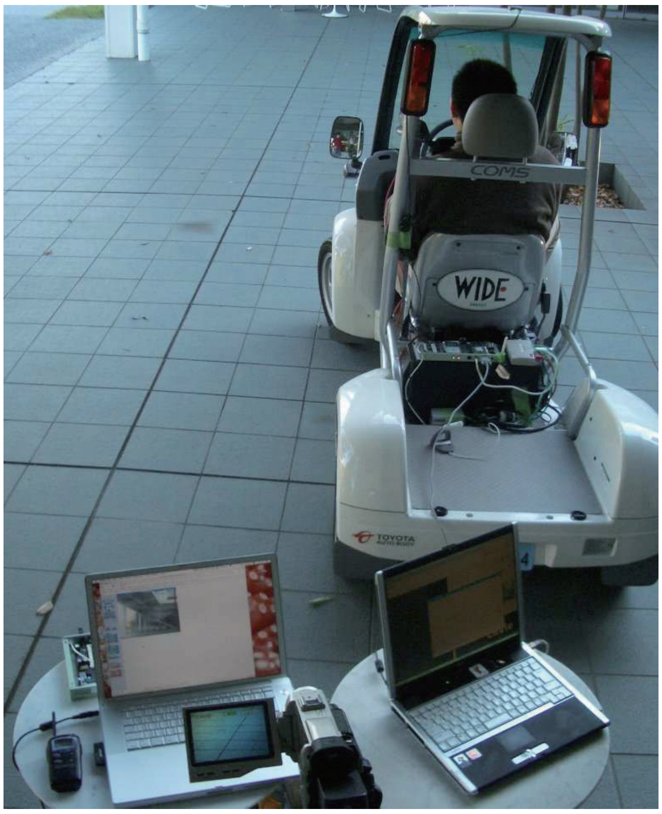
The core IPv6 and NEMO functionnality is supported by the KAME/SHISA networking stack, while MANET routing is handled by the OLSR routing protocol, using a modified implementation built as a plugin to the Zebra framework. These routing daemons were made to cooperate and routing decisions are based on the availability of mobile routers in the MANET vicinity. On a lower level, we developped software to control the network interfaces used by the SHISA stack. Eventually, monitoring is made by

[3]http://sonar.nautilus6.org

[4]http://software.nautilus6.org



(a) The Soekris SBC


(b) A test vehicle : Toyota COMS

(c) The ORF experiment : Marunouchi Shuttle

Fig. 3. The mobile router implementation

	Total boot time	NetBSD boot	Interface startup	MIPv6 / NEMO registration
PHS	79 s	60 s	11 s	8 s
WIFI	77 s	58 s	11 s	8 s

TABLE I
BOOTING TIME PROFILING

specific software that polls every component that needs to be reported.

Using this implementation, we were able to show preliminary results about booting time that confirm the fact that the choice of the NetBSD OS is suitable for in-car operation (with one minute and a half booting time).

As a next step, we are going to use this mobile router on a daily basis in personal cars, and at specific exhibitions like the ORF 2006 in Tokyo [5]. However, to be operable at public scale, we need to further investigate security issues at the network and datalink layers. The former will be achieved using IPSec transport between the mobile router and the home agent while we might resort to hardware encryption (WEP or WPA) for the latter.

ACKNOWLEDGMENTS

We would like to thank the people from SHISA team for helping us with NetBSD-current stack, and all the people who got involved in this project at any given time, especially Keiichi Shima, Keisuke Uehara and Manabu Tsukada.

REFERENCES

[1] V. Devarapalli, R. Wakikawa, A. Petrescu, and P. Thubert, "Network Mobility (NEMO) Basic Support Protocol," IETF, Request For Comments 3963, January 2005.

[2] "IETF Network Mobility (NEMO) Working Group Charter," URL, As of September 2004, http://www.ietf.org/html.charters/nemo-charter.html.

[3] "InternetCAR Working Group, WIDE Project," Web page, http://www.icar.wide.ad.jp.

[4] "WIDE Project (Widely Integrated Distributed Environment)," Web page, http://www.wide.ad.jp.

[5] T. Ernst and K. Uehara, "Connecting Automobiles to the Internet," in *ITST: 3rd International Workshop on ITS Telecommunications*, Seoul, South Korea, November 2002.

[6] K. Mitsuya, K. Uehara, and J. Murai, "The In-vehicle Router System to support Network Mobility," *LNCS*, vol. 2662, pp. 633–642, 2003.

[7] T. Ernst, K. Mitsuya, and K. Uehara, "Network mobility from the internetcar perspective," *Journal of Interconnection Networks(JOIN)*, vol. 4, no. 3, pp. 329–344, 2003.

[8] "KAME Working Group, WIDE Project," Web page, http://www.kame.net.

[9] "SHISA: an implementation of Mobile IPv6 within the KAME IPv6 stack," http://www.mobileip.jp/, http://www.kame.net/.

[10] K. Shima, R. Wakikawa, K. Mitsuya, T. Momose, and K. Uehara, "SHISA: The IPv6 Mobility Framework for BSD Operating Systems," in *IPv6 Today - Technology and Deployment (IPv6TD'06). International Academy Research and Industry Association*, Aug 2006.

[11] E. Clausen and E. Jacquet, "Optimized link state routing protocol (OLSR)," Internet Engineering Task Force, RFC 3626, Oct. 2003. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3626.txt

[12] J. Lorchat and K. Uehara, "Optimized Inter-Vehicle Communications Using NEMO and MANET," in *V2VCOM2006 : 2nd Vehicle-to-Vehicle Communications Workshop*, San Jose, California, USA, July 2006.

[13] K. Mitsuya, J. Lorchat, T. Noel, and K. Uehara, "SONAR : A Statistics Repository of Mobility Platforms," in *The First International Workshop on Network Mobility (WONEMO)*, Sendai, Japan, Jan 2006.

[5]http://orf.sfc.keio.ac.jp

Reflections on Building a High-performance Computing Cluster Using FreeBSD

Brooks Davis, Michael AuYeung, J. Matt Clark, Craig Lee, James Palko, Mark Thomas
The Aerospace Corporation
El Segundo, CA
{brooks,mauyeung,mclark,lee,jpalko,mathomas}@aero.org

Abstract

Since late 2000 we have developed and maintained a general purpose technical and scientific computing cluster running the FreeBSD operating system. In that time we have grown from a cluster of 8 dual Intel Pentium III systems to our current mix of 64 dual Intel Xeon and 289 dual AMD Opteron systems. This paper looks back on the system architecture as documented in our BSDCon 2003 paper "Building a High-performance Computing Cluster Using FreeBSD" and our changes since that time. After a brief overview of the current cluster we revisit the architectural decisions in that paper and reflect on their long term success. We then discuss lessons learned in the process. Finally, we conclude with thoughts on future cluster expansion and designs.

1 Introduction

From the early 1990's on, the primary thrust of high performance computing (HPC) development has been in the direction of commodity clusters, commonly referred to as Beowulf clusters [Becker]. These clusters combine commercial off-the-shelf hardware to create systems which rival or exceed the performance of traditional supercomputers in many applications while costing as much as a factor of ten less. Not all applications are suitable for clusters, but a signification portion of interesting scientific applications can be successfully adapted to them.

In 2001, driven by a number of separate users with supercomputing needs, The Aerospace Corporation (a California nonprofit corporation that operates a Federally Funded Research and Development Center) decided to build a corporate computing cluster (eventually named Fellowship for The Fellowship of the Ring [Tolkien]) as an alternative to continuing to buy small clusters and SMP systems on an ad-hoc basis. This decision was motivated by a desire to use computing resources more efficiently as well as reducing administrative costs. The diverse set of user requirements in our environment led us to a design which differs significantly from most clusters we have seen elsewhere. This is especially true in the areas of operating system choice (FreeBSD) and configuration management (fully network booted nodes).

At BSDCon 2003 we presented a paper titled "Building a High-performance Computing Cluster Using FreeBSD" [Davis] detailing these design decisions. This paper looks back on the system architecture as documented in that paper and our changes since that time. After a brief overview of the current cluster we revisit the architectural decisions in that paper and reflect on their long term success. We then discuss lessons learned in the process. Finally, we conclude with thoughts on future cluster expansion and designs.

2 Fellowship Overview

The basic logical and physical layout of Fellowship is similar to many clusters. There are six core systems, 352 dual-processor nodes, a network switch, and assorted remote management hardware. All nodes and servers run FreeBSD, currently 6.2-RELEASE. The core systems and remote management hardware sit on the Aerospace corporate network. The nodes and core systems share a private, non-routed network (10.5/16). The majority of this equipment is mounted in two rows of seven-foot tall, two-post racks residing in the underground data center at Aerospace headquarters in El Segundo, California. Figure 1 shows Fellowship in Fall 2006. The layout of a recent node racks is shown in Figure 2. The individual racks vary due to design changes over time.

When users connect to Fellowship, they do so via a

Figure 1: Fellowship Circa February 2007

core server named `fellowship` that is equipped to provide shell access. There they edit and compile their programs and submit jobs for execution on a node or set of nodes. The scheduler is run on the core server `arwen` that also provides network boot services to the nodes to centralize node management. Other core servers include: `frodo` which provides directory service for user accounts and hosts the license servers for commercial software including the Intel FORTRAN compiler and Grid Mathematica; `gamgee` which provides backups using the Bacula software and a 21 tape LTO2 changer; `elrond` and `legolas` which host shared temporary file storage that is fast and large respectively; and `moria`, our Network Appliance file server.

The nodes are currently a mix of older Intel Xeons and single and dual-core AMD Opterons. Table 1 gives a breakdown of general CPU types in Fellowship today. Figure 4 and Figure 5 show the composition of Fellowship over time by node and core count. Each node as an internal ATA or SATA disk that is either 80GB or 250GB and between 1 and 4 gigabytes of RAM. The nodes are connected via Gigabit Ethernet through a Cisco Catalyst 6509 switch[1] The

CPU Type	*Nodes*	*CPUs*	*Cores*
Xeon	64	128	128
Opteron single-core	136	272	272
Opteron dual-core	152	304	608
Total	352	704	1008

Table 1: CPUs in Fellowship nodes.

Opterons are mounted in custom 1U chassis approximately 18 inches deep with IO ports and disks facing the front of the rack. Figure 3 shows the front of first generation Opteron nodes.

Although the nodes have disks, we network boot them using PXE support on their network interfaces with `frodo` providing DHCP, TFTP, NFS root disk, and NIS user accounts. On boot, the disks are automatically checked to verify that they are properly partitioned for our environment. If they are not, they are automatically repartitioned. This means minimal configuration of nodes is required beyond determining their MAC address and location. Most of that configuration is accomplished by scripts.

Local and remote control of core machines is made

[1]This was originally a Catalyst 6513, but most slots in the 6513 have reduced available bandwidth so we upgraded to the smaller 6509.

Unit	*Contents*
45	Patch panel
44	(Connection to patch panel rack)
43	*empty*
42	*empty*
41	*empty*
40	node (r01n32: 10.5.1.32)
39	node (r01n31: 10.5.1.31)
38	node (r01n30: 10.5.1.30)
37	node (r01n29: 10.5.1.29)
36	node (r01n28: 10.5.1.28)
35	node (r01n27: 10.5.1.27)
34	node (r01n26: 10.5.1.26)
33	node (r01n25: 10.5.1.25)
32	Cyclades 10-port power controller
31	Cyclades 10-port power controller
30	node (r01n24: 10.5.1.24)
29	node (r01n23: 10.5.1.23)
28	node (r01n22: 10.5.1.22)
27	node (r01n21: 10.5.1.21)
26	node (r01n20: 10.5.1.20)
25	node (r01n19: 10.5.1.19)
24	node (r01n18: 10.5.1.18)
23	node (r01n17: 10.5.1.17)
22	rackmount KVM unit
21	node (r01n16: 10.5.1.16)
20	node (r01n15: 10.5.1.15)
19	node (r01n14: 10.5.1.14)
18	node (r01n13: 10.5.1.13)
17	node (r01n12: 10.5.1.12)
16	node (r01n11: 10.5.1.11)
15	node (r01n10: 10.5.1.10)
14	node (r01n09: 10.5.1.9)
13	Cyclades PM 10-port power controller
12	Cyclades PM 10-port power controller
11	node (r01n08: 10.5.1.8)
10	node (r01n07: 10.5.1.7)
9	node (r01n06: 10.5.1.6)
8	node (r01n05: 10.5.1.5)
7	node (r01n04: 10.5.1.4)
6	node (r01n03: 10.5.1.3)
5	node (r01n02: 10.5.1.2)
4	node (r01n01: 10.5.1.1)
3	
2	4 120V 30A & 1 120 V 20A Circuits
1	

Figure 2: Layout of Node Rack 1

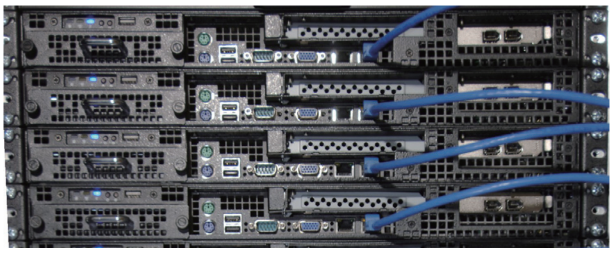


Figure 3: Front details of Opteron nodes

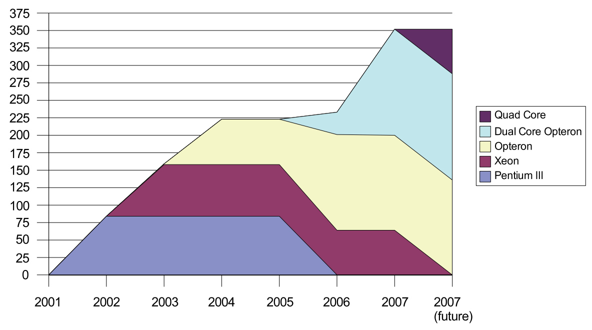


Figure 4: Fellowship node count and type over time

possible through a Lantronix networked KVM-switch connected to a 1U rackmount keyboard, and track pad and an 18-inch rackmount monitor which doubles as a local status display. In the newer racks, 1U rack mount keyboard, monitor, mouse (KVM) units are installed to allow administrators to easily access local consoles during maintenance. In addition to console access, everything except the terminal servers and the switch are connected to serial remote power controllers. The older racks use a variety of 8-port BayTech power controllers and the new ones use 10-port Cyclades AlterPath PM series controllers. All

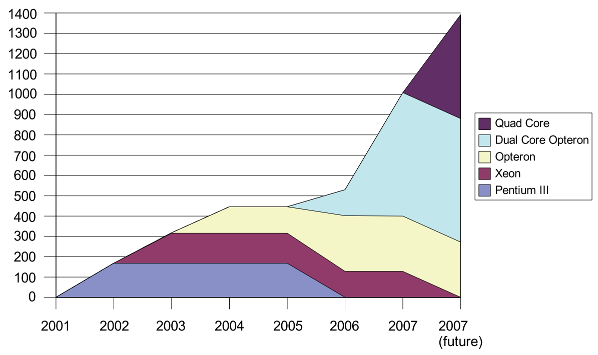


Figure 5: Fellowship core count and type over time

of these controllers are capable of supplying a total of 30 Amps of power at 120 Volts. This allows us to remotely reboot virtually any part of the system by connecting to the power controller via the appropriate terminal server.

On top of this infrastructure, access to nodes is controlled by Sun Grid Engine (SGE), a scheduler implementing a superset of the POSIX Batch Environment Services specification. SGE allows users to submit both interactive and batch job scripts to be run on one or more processors. Users are free to use the processors they are allocated in any reasonable manner. They can run multiple unrelated processes or massively parallel jobs.

To facilitate use of Fellowship, we provide a basic Unix programming environment, plus the parallel programming toolkits, and commercial parallel applications. For parallel programming toolkits we provide MPICH, MPICH2, and OpenMPI implementations of the Message Passing Interface [MPI] (MPI) as well as the Parallel Virtual Machine (PVM). We also provide Grid Mathematica and MATLAB under Linux emulation.

3 Design Issues

One of the biggest challenges in building Fellowship was our diverse user base. Among the users at the initial meetings to discuss cluster architecture, we had users with loosely coupled and tightly coupled applications, data intensive and non-data intensive applications, and users doing work ranging from daily production runs to high performance computing research. This diversity of users and applications led to the compromise that was our initial design. Many aspects of this design remain the same, but some have changed based on our experiences. In this section we highlight the major design decisions we made while building Fellowship and discuss how those decisions have fared in the face of reality.

3.1 Operating System

The first major design decision any cluster designer faces is usually the choice of operating system. By far, the most popular choice is a Linux distribution of some sort. Indeed, Linux occupies much the same position in the HPC community as Windows does in the desktop market to the point most people assume that, if it is a cluster, it runs Linux. In reality a cluster can run almost any operating system. Clusters exist running Solaris [SciClone], MacOS X, FreeBSD, and Windows[WindowsCCS] among others. NASA's Columbia [Columba] super computer is actually a cluster of 24 SGI Altix systems 21 of which are 512-CPU system.

For an organization with no operating system bias and straight-forward computing requirements, running Linux is the path of least resistance due to free clustering tool kits such as Rocks [ROCKS] or OSCAR [OSCAR]. In other situations, operating system choice is more complicated. Important factors to consider include chosen hardware platform, existence of experienced local system administration staff, availability of needed applications, ease of maintenance, system performance, and the importance of the ability to modify the operating system.

For a variety of reasons, we chose FreeBSD for Fellowship. The most pragmatic reason for doing so is the excellent out of the box support for diskless systems which was easily modifiable to support our nodes network booting model. This has worked out very well.

Additionally, the chief Fellowship architect uses FreeBSD almost exclusively and is a FreeBSD committer. This meant we had more FreeBSD experience than Linux experience and that we could push some of our more general changes back into FreeBSD to simplify operating system upgrades. We have been able to feed back a number of small improvements, particularly in the diskless boot process which has benefited us and other FreeBSD users. For changes which are too Aerospace- or Fellowship-specific to contribute back, we have maintained an internal Perforce repository with a customized version of FreeBSD.

The ports collection was also a major advantage of using FreeBSD. It has allowed us to install and maintain user-requested software quickly and easily. For most applications the ports collection has worked well. The major exception is anything related to MPI. MPI implementations are generally API compatible, but use different symbols, libraries, and header files. As a result MPI implementations traditionally provide a set of wrapper scripts for the compiler along the lines of `mpicc`, `mpic++`, `mpif77`, etc which take care of the details of linking. Unfortunately a different compilation of MPI is needed for each compiler and it is useful to have multiple MPI versions. The ports framework is not equipped to deal with such combinatorial explosions.

The availability of Linux emulation meant we did not give up much in the way of application compatibility. We were the launch customer for Grid Mathematica using the Linux version. We have also run other third-party Linux programs including MPI applications.

Initially the disadvantages of FreeBSD for our purposes were immature SMP and threading support, and an widely held view within the high performance computing community that if it isn't a commercial supercomputer, it must be a Linux system. SMP support was not a major issue for our users because most of our jobs are compute-bound so the poor SMP performance under heavy IO was a moot problem. With the 5.x and 6.x series of FreeBSD releases, this issue has largely been resolved. Threading was more of an issue. We had users who wanted to use threads to support SMP scaling in certain applications and with the old user space threading provided by `libc_r` they could not do that. With our migration to the FreeBSD 6.x series, this problem has been resolved.

The Linux focus of the HPC community has caused us some problems. In particular, many pieces of software either lack a FreeBSD port, or only have a poorly tested one which does not actually work out of the box. In general we have been able to complete ports for FreeBSD without issue. One significant exception was the LAM implementation of MPI. In 4.x it worked fine, but in 5.x and 6.x it builds but crashes instead of running. We have been unable to find and fix the problems. Initially there was a shortage of compiler support for modern versions of FORTRAN. This has been changed by two things: the gfortran project's FORTRAN 90 and 95 compiler and the wrapping of the Intel Linux FORTRAN compiler to build FreeBSD software. A FreeBSD FORTRAN compiler is also available from NAG. One other issue is the lack of a parallel debugger such as TotalView.

We are happy with the results of running FreeBSD on the cluster. It has worked well in for us and the occasional lack of software had been more than made up for by our existing experience with FreeBSD.

3.2 Hardware Architecture

The choice of hardware architecture is generally made in conjunction with the operating system as the two interact with each other. Today, most clusters are based on Intel or AMD x86 CPUs, but other choices are available. When developing Fellowship, SPARC and Alpha clusters were fairly command as were Apple XServe clusters based on the PowerPC platform. Today, the Alpha is essentially gone and Apple has migrated from PowerPC to Intel CPUs leaving x86 with the vast majority of the HPC cluster market. The major issues to consider are price, performance, power consumption, and operating system compatibility. For instance, Intel's Itanium2 has excellent floating point performance, but is expensive and power hungry. Early on it also suffered form immature operating system support. In general, x86 based systems are the path of least resistance given the lack of a conflicting operating system requirement.

When we were selecting a hardware architecture in 2001, the major contenders were Alpha and Intel or AMD based x86 systems. We quickly discarded Alpha from consideration because of previous experiences with overheating problems on a small Aerospace Alpha cluster. Alphas also no longer had the kind of performance lead they enjoyed in the late 1990's. We looked at both Pentium III and Athlon-based systems, but decided that while the performance characteristics and prices did not vary significantly, power consumption was too problematic on the Athlon systems.

Over the life of Fellowship, we have investigated other types of nodes including Athlon based systems, the Xeon systems we purchased for the 2003 expansion, AMD Opteron systems, and Intel Woodcrest systems. Athlons failed to match the power/performance ratios of Intel Pentium III systems, but with the Xeon and Opteron processors the balance shifted resulting in purchases of Opterons in 2004 through 2006. We are now evaluating both AMD and Intel based solutions for future purchase. One interesting thing we've found is that while the Intel CPUs themselves consume less power, the RAM and chipsets they use are substantially more power hungry. This illustrates the need to look a all aspects of the system not just the CPUs.

3.3 Node Architecture

Most of the decisions about node hardware will derive from the selection of hardware architecture, cluster form factor, and network interface. The biggest of the remaining choices is single or multi-processor systems. Single processor systems have better CPU utilization due to a lack of contention for RAM, disk, and network access. Multi-processor systems can allow hybrid applications to share data directly, decreasing their communication overhead. Additionally, multi-processor systems tend to have higher performance interfaces and internal buses then single processor systems. This was significant consideration with Fellowship's initial design, but the current direction of processor development suggest that only multiple core CPUs will exist in the near future.

Other choices are processor speed, RAM, and disk space. We have found that aiming for the knee of the price curve has served us well, since no single user dominates our decisions. In other environments, top of the line processors, large disks, or large amounts of RAM may be justified despite the exponential increase

CPU	2 x Pentium III 1GHz
Network Interface	3Com 3C996B-T
RAM	1GB
Disk	40GB 7200RPM IDE

Table 2: Configuration of first Fellowship nodes.

CPU	2 x Opteron 275 2x2.2GHz
Network Interface	On board gigabit
RAM	4GB
Disk	250GB 7200RPM SATA

Table 3: Configuration of latest Fellowship nodes.

in cost.

For Fellowship, we chose dual CPU systems. We were motivated by a desire to do research on code that takes advantage of SMP systems in a cluster, higher density than single processor systems, and the fact that the 64-bit PCI slots we needed for Gigabit Ethernet were not available on single CPU systems. As a result of our focus on the knee of the price curve, we have bought slightly below the performance peak on processor speed, with 2-4 sticks of smaller-than-maximum RAM, and disks in the same size range as mid-range desktops. This resulted in the initial configuration shown in Table 2. The most recent node configuration is shown in Table 3.

3.4 Network Interconnects

Like hardware architecture, the selection of network interfaces is a matter of choosing the appropriate point in the trade space between price and performance. Performance is generally characterized by bandwidth and latency. The right interface for a given cluster depends significantly on the jobs it will run. For loosely-coupled jobs with small input and output data sets, little bandwidth is required and 100Mbps Ethernet is the obvious choice. For other, tightly-coupled jobs, InfiniBand or 10 Gbps Myrinet which have low latency and high bandwidth are good options For some applications 1 Gbps or 10 Gbps Ethernet will be the right choice.

The choice of Gigabit Ethernet for Fellowship's interconnect represents a compromise between the cheaper 100 Mbps Ethernet our loosely coupled applications would prefer (allowing us to buy more nodes) and 2Gbps Myrinet. When we started building Fellowship, Gigabit Ethernet was about one-third of the cost of each node whereas Myrinet would have more than doubled our costs. Today Gigabit Ethernet is standard on the motherboard and with the large switches required by a cluster Fellowship's size, there is no price difference between 100 Mbps and 1 Gbps ether ports.

Gigabit Ethernet has worked well for most of our applications. Even our computational fluid dynamics (CFD) applications have run reasonably well on the system. A few applications such as the CFD codes and some radiation damage codes would benefit from higher bandwidth and lower latency, but Gigabit Ethernet appears to have been the right choice at the time.

3.5 Core Servers and Services

On Fellowship, we refer to all the equipment other then the nodes and the remote administration hardware as core servers. On many clusters, a single core server suffices to provide all necessary core services. In fact, some clusters simply pick a node to be the nominal head of the cluster. Some large clusters provide multiple front ends, with load balancing and fail over support to improve up time.

Core services are those services which need to be available for users to utilize the cluster. At a minimum, users need accounts and home directories. They also need a way to configure their jobs and get them to the nodes. The usual way to provide these services is to provide shared home and application directories, usually via NFS and use a directory service such as NIS to distribute account information. Other core services a cluster architect might choose to include are batch schedulers, databases for results storage, and access to archival storage resources. The number of ways to allocate core servers to core services is practically unlimited.

Fellowship started with three core servers: the data server, the user server, and the management server. All of these servers are were dual 1GHz Pentium III systems with SCSI RAID5 arrays. The data server, `gamgee`, served a 250GB shared scratch volume via NFS, and performed nightly backups to a 20 tape LTO library using AMANDA. The user server, `fellowship`, served NFS home directories and gave the users a place to log in to compile and run applications. The management server, `frodo`, ran the scheduler, NIS, and our shared application hierarchy mounted at `/usr/aero`. Additionally, the management server uses DHCP, TFTP, and NFS to netboot the nodes.

Today, Fellowship has eight core servers. The data server has been split into three machines: `elrond`, `gamgee`, and `legolas`. `elrond` is a 2.4GHz dual Xeon with SCSI raid that provides `/scratch`. `gamgee` it self has been replaced with a dual Opteron with 1TB of SATA disk. It runs backups using Bacula a network based backup program. `legolas` provides 2.8GB of shared NFS disk at `/bigdisk`. The user server, `fellowship`, is now a quad Opteron system and home directories are served by `moria`, a Network Appliance FAS250 filer. It is supplemented by a second quad Opteron, `fellowship-64` which runs FreeBSD amd64. The original management servers, `frodo`, still exists, but currently only runs NIS. It has mostly been replaced by `arwen` which has taken over running the Sun Grid Engine scheduler and netbooting the nodes.

These services were initially isolated from each other for performance reasons. The idea was that hitting the shared scratch space would not slow down ordinary compiles and compiling would not slow down scratch space access. We discovered that, separation of services does work, but it comes at the cost of increased fragility because the systems are interdependent, and when one fails, they all have problems. We have devised solutions to these problems, but this sort of division of services should be carefully planned and would generally benefit from redundancy when feasible. Our current set of servers is larger than optimal from an administrative perspective. This situation arose because new core servers were purchased opportunistically when end of year funds were available and thus older servers have not been gracefully retired.

3.6 Node Configuration Management

Since nodes outnumber everything else on the system, efficient configuration management is essential. Many systems install an operating system on each node and configure the node-specific portion of the installation manually. Other systems network boot the nodes using Etherboot, PXE or LinuxBIOS. The key is good use of centralization and automation. We have seen many clusters where the nodes are never updated without dire need because the architect made poor choices that made upgrading nodes impractical.

Node configuration management is probably the most unique part of Fellowship's architecture. We start with the basic FreeBSD diskless boot process [diskless(8)]. We then use the diskless remount support to mount `/etc` as `/conf/base/etc`. For many applications, this configuration would be sufficient. However, we have applications which require significant amounts of local scratch space. To deal with this each node contains a disk. The usual way of handling such disks would be to manually create appropriate directory structures on the disk when the system was first installed and then let the nodes mount and fsck the disks each time they were booted. We deemed this impractical because nodes are usually installed in large groups. Additionally, we wanted the ability to reconfigure the disk along with the operating system.

In our original FreeBSD 4.x based installation, we created a program (`diskmark`) which used an invalid entry in the MBR partition table to store a magic number and version representing the current partitioning scheme. At boot we used a script which executed before the body of `rc.diskless2` to examine this entry to see if the current layout of the disk was the required one. If it was not, the diskless scripts automatically use Warner Losh's `diskprep` script to initialize the disk according to our requirements. With FreeBSD 6.x we adopted a somewhat less invasive approach. We still use a version of `diskprep` to format the disk, but now we use `glabel` volume labels to identify the version of the disk layout. The script that performs the partitioning is installed in `/etc/rc.d` instead of requiring modification of `/etc/rc`. We install the script in the node image using a port.

With this configuration, adding nodes is very easy. The basic procedure is to bolt them into the rack, hook them up, and turn them on. We then obtain their MAC address from the switch's management console and add it to the DHCP configuration so each node is assigned a well-known IP address. After running a script to tell the scheduler and Nagios about the nodes and rebooting them, they are ready for use.

Under FreeBSD 4.x, maintenance of the netboot image is handed by chrooting to the root of the installation and following standard procedures to upgrade the operating system and ports as needed. With our move to FreeBSD 6.x we have also moved to a new model of netboot image updating. Instead of upgrading a copy, we create a whole new image from scratch using parts of the nanobsd [Gerzo] framework. The motivation for switching to this mode was that in the four years since we got the initial image working we had forgotten all the customizations that were required to make it fully functional. Since 6.x has an large number of differences in configuration from 4.x, this made it difficult to upgrade. Our theory with creating new images each time is that it will force us to document all the customizations either in the script or in a separate document that will be manually modified. Thus far, this has worked to some degree, but has not been perfect as creation of some of the images has been rushed resulting in a failure to document everything. In the long term, we expect it to be a better solution than in place upgrades. Software which is not available from the ports collection is installed in the separate `/usr/aero` hierarchy.

One problem we found with early systems was poor

Mountpoint	*Source*
`/`	`arwen:/export/roots/freebsd/fbsd62`
`/etc`	`/dev/md0`
`/tmp`	`/dev/ufs/tmp`
`/var`	`/dev/ufs/var`
`/home`	`moria:/home`
`/usr/aero`	`frodo:/nodedata/usr.aero`
`/usr/local/sge/fellowship`	`arwen:/export/sge/fellowship`
`/scratch`	`elrond:/scratch`
`/bigdisk`	`legolas:/bigdisk`

Table 4: Sample node (`r01n01` aka 10.5.1.1) mount structure

quality PXE implementations. We have found PXE to be somewhat unreliable on nearly all platforms, particularly on the Pentium III systems, occasionally failing to boot from the network for no apparent reason and then falling back to the disk which is not configured to boot. Some of these problems appear to be caused by interactions with network switches and the spanning tree algorithm. To work around this problem we have created a `diskprep` configuration that creates an extra partition containing FreeDOS and an `AUTOEXEC.BAT` that automatically reboots the machine if PXE fails rather than hanging. It would be better if server motherboard vendors added an option to the BIOS to keep trying in the event of a PXE failure.

3.7 Job Scheduling

Job scheduling is potentially one of the most complex and contentious issues faced by a cluster architect. The major scheduling options are running without any scheduling, manual scheduling, batch queuing, and domain specific scheduling.

In small environments with users who have compatible goals, not having a scheduler and just letting users run what they want when they want or communicating with each other out of band to reserve resources as necessary can be a good solution. It has very little administrative overhead, and in many cases, it just works.

With large clusters, some form of scheduling is usually required. Even if users do not have conflicting goals, it's difficult to try to figure out which nodes to run on when there are tens or hundreds available. Additionally, many clusters have multiple purposes that must be balanced. In many environments, a batch queuing system is the answer. A number exist, including OpenPBS, PBSPro, Sun Grid Engine (SGE), LSF, Torque, NQS, and DQS. Torque and SGE are freely available open source applications and are the most popular options for cluster scheduling. When we started building Fellowship OpenPBS was quite popular and SGE was not yet open source.

For some applications, batch queuing is not a good solution. This is usually either because the application requires too many jobs for most batch queuing systems to keep up, or because the run time of jobs is too variable to be useful. For instance, we have heard of one computational biology application which runs through tens of thousands of test cases a day where most take a few seconds, but some may take minutes, hours, or days to complete. In these situations, a domain specific scheduler is often necessary. A common solution is to store cases in a database and have applications on each node that query the database for a work unit, process it, store the result in the database, and repeat.

On Fellowship, we have a wide mix of applications ranging from trivially schedulable tasks to applications with unknown run times. Our current strategy is to implement batch queuing with a long-term goal of discovering a way to handle very long running applications. We initially intended to run the popular OpenPBS scheduler because it already had a port to FreeBSD and it is open source. Unfortunately, we found that OpenPBS had major stability problems under FreeBSD (and, by many accounts, most other operating systems)[2]. About the time we were ready to give up on OpenPBS, Sun released SGE as open source. FreeBSD was not supported initially, but we were able to successfully complete a port based on some patches posted to the mailing lists. That initial port allowed jobs to run. Since then we have added more functionality and the FreeBSD port is essentially at feature parity with the Linux port.

The major problem we had with scheduling is that initially, we allowed direct access to cluster nodes without the scheduler. While we had few users and not

[2]The Torque resource manager is a successful fork of OpenPBS to support the Maui scheduler

all systems were full at all times, this was not a big deal. Unfortunately, as the system filled up, it became a problem. We had assumed that users would see the benefits of using the scheduler such as unattended operation and allocation of uncontested resources, but most did not. Worse, those few who did often found themselves unable to access any resources because the scheduler saw that all the nodes were overloaded. Those users then gave up on the scheduler. We eventually imposed mandatory use of the scheduler along with a gradual transition to FreeBSD 6.x on the nodes. There was a fair bit of user rebellion when this happened, but we were able to force them to cooperate eventually. In retrospect failure to mandate use of the scheduler as soon as it was operational was a significant error.

3.8 Security Considerations

For most clusters, we feel that treating the cluster as a single system is the most practical approach to security. Thus for nodes which are not routed to the Internet like those on Fellowship, all exploits on nodes should be considered local. What this means to a given cluster's security policy is a local issue. For systems with routed nodes, management gets more complicated, since each node becomes a source of potential remote vulnerability. In this case it may be necessary to take action to protect successful attacks on nodes from being leveraged into full system access. In such situations, encouraging the use of encrypted protocols within the cluster may be desirable, but the performance impact should be kept firmly in mind.

The major exception to this situation is clusters where jobs have access to data that must not be mingled. We have begun an investigation into ways to isolate jobs from each other more effectively. We believe that doing so will yield both improved security and better performance predictability.

For the most part we have chosen to concentrate on protecting Fellowship from the network at large. This primarily consists of keeping the core systems up to date and requiring that all communications be via encrypted protocols such as SSH and HTTPS. Internally we discourage direct connections between nodes except by scheduler-provided mechanisms that could easily be encrypted. Inter-node communications are unencrypted for performance reasons.

3.9 System Monitoring

The smooth operation of a cluster can be aided by proper use of system monitoring tools. Most common monitoring tools such as Nagios and Big Sister are applicable to cluster use. The one kind of monitoring tool that does not work well with clusters is the sort that sends regular e-mail reports for each node. Even a few nodes will generate more reports then most admins have time to read. In addition to standard monitoring tools, there exist cluster specific tools such as the Ganglia Cluster Monitor. Most schedulers also contain monitoring functionality.

On Fellowship we are currently running the Ganglia Cluster Monitoring system, Nagios, and the standard FreeBSD periodic scripts on core systems. Ganglia was ported to FreeBSD previously, but we have created FreeBSD ports which make it easier to install and make its installation more BSD-like. We have also rewritten most of the FreeBSD specific code so that it is effectively at feature parity with Linux (and better in some cases). A major advantage of Ganglia is that no configuration is required to add nodes. They are automatically discovered via multicast. We have also deployed Nagios with a number of standard and custom scripts. With Nagios notification we typically see problems with nodes before our users do.

3.10 Physical System Management

At some point in time, every system administrator finds that they need to access the console of a machine or power cycle it. With just a few machines, installing monitors on each machine or installing a KVM switch for all machines and flipping power switches manually is a reasonable option. For a large cluster such as Fellowship, more sophisticated remote management systems are desirable.

In Fellowship's architecture, we place a strong emphasis on remote management. The cluster is housed in our controlled access data center, which makes physical access cumbersome. Additionally, the chief architect and administrator lives around 1500 miles (about 2400 kilometers) from the data center, making direct access even more difficult. As a result, we have installed remote power controllers on all nodes are core systems and remote KVM access to all core systems. Initially we had also configured all nodes to have serial consoles accessible through terminal servers. This worked well for FreeBSD, but we had problems with hangs in the BIOS redirection on the Pentium III systems which forced us to disable it. That combined with the fact that we rarely used the feature and the

$100 per port cost lead us to discontinue the purchase of per-rack terminal servers when we built the second row of racks. One recent change we have made is adding a per-rack 1U KVM unit in newer node racks. At around $650/rack they are quite cost effective and should save significant administrator time when diagnosing failures.

In the future we would like to look at using IPMI to provide remote console and reboot for nodes eliminating the need for dedicated hardware.

3.11 Form Factor

The choice of system form factor is generally a choice between desktop systems on shelves, rack mounted servers, and blade systems. Shelves of desktops are common for small clusters as they are usually cheaper and less likely to have cooling problems. Their disadvantages include the fact that they take up more space, the lack of cable management leading to more difficult maintenance, and generally poor aesthetics. Additionally, most such systems violate seismic safety regulations.

Rack mounted systems are typically slightly expensive due to components which are produced in lower volumes as well as higher margins in the server market. Additionally, racks or cabinets cost more then cheap metal shelves. In return for this added expense, rack mount systems deliver higher density, integrated cable management, and, usually, improved aesthetics.

Blade systems are the most expensive by far. They offer higher density, easier maintenance, and a neater look. The highest density options are often over twice as expensive with significantly lower peak performance due to the use of special low-power components.

A minor sub-issue related to rack mount systems is cabinets vs. open, telco style racks. Cabinets look more polished and can theoretically be moved around. Their disadvantages are increased cost, lack of space making them hard to work in, and being prone to overheating due to restricted airflow. Telco racks do not look as neat and are generally bolted to the floor, but they allow easy access to cables and unrestricted airflow. In our case, we use vertical cable management with doors which makes Fellowship look fairly neat without requiring cabinets.

The projected size of Fellowship drove us to a rack mount configuration immediately. We planned from the start to eventually have at least 300 CPUs, which is pushing reasonable bounds with shelves. We had a few problems with our initial rack confirmation. First,

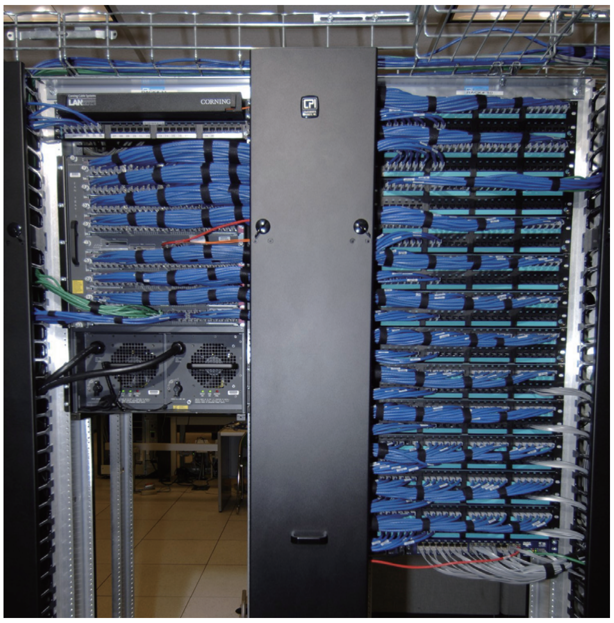


Figure 6: Fellowship's switch and patch panel racks

the use of six inch wide vertical cable management did not leave use with enough space to work easily. We used ten inch wide vertical cable management when we expanded to a second row of racks to address this problem. Second, the choice of making direct runs from nodes to the switch resulted in too much cable running to the switch. When we expanded to a second row of racks we added patch panels to them and the existing rack and moved the switch next to a new rack of patch panels. This substantially simplified our cabling. The patch switch and central patch panel can be seen in Figure 6. The third problem we encountered was that we were unable to mount some core systems in the racks we allocated for the core systems. We have mounted some of our core systems in a separate cabinet as a result and plan to add a dedicated cabinet in the future.

4 Lessons Learned

We have learned several lessons in the process of building and maintaining Fellowship. None took us completely by surprise, but they are worth covering as they can and should influence design decisions.

The first and foremost lesson we have learned is that with a cluster, relatively uncommon events can become common. For example, during initial testing with the Pentium III nodes we infrequently encountered two BIOS related problems: if BIOS serial port redirection was enabled, they system would occasionally hang and PXE booting would sometimes fail.

With the console redirection, we thought we had fixed the problem by reducing the serial ports speed to 9600 bps, but in fact we had just made it occur during approximately one boot in 30. This meant that every time we rebooted, we had to wait until it appeared everything had booted and then power cycle the nodes that didn't boot. In the end we were forced to connect a keyboard and monitor and disable this feature. Similarly, PXE problems did not appear serious and appeared resolved with one node, but with 40 nodes, they became a significant headache. In the end we implemented the reboot hack described in the Node Configuration Management section. In addition to these BIOS failures, we initially experienced hardware failures, most power supplies, at nearly every power cycle. This seemed high at first, but the problems mostly settled out over time. With a single machine this wouldn't have been noticeable, but with many machine it became readily apparent that the power supplies were poorly manufactured. Later on this was reinforced as at around three years of operation the nodes stared failing to POST. We eventually concluded the problem was with due to incremental degradation in the power supplies because the boards worked with a new supply. After the power supplies, the next most common component to fail has been the hard drives. In the Pentium III nodes they were the notorious IBM Deathstar disks which lead to a high failure rate. In other system the failure rate has been lower, but still significant. When we created specifications for the the custom cases used for the Opterons, we specified removable disks. This has simplified maintenance significantly.

A lesson derived from those hardware failures was that neatness counts quite a bit in racking nodes. To save money in the initial deployment, we ran cables directly from the switch to the nodes. This means we have a lot of slack cable in the cable management, which makes removing and reinstalling nodes difficult. We ended up adding patch panels in each rack to address this problem. Based on this experience we have considered blades more seriously for future clusters, particularly those where on site support will be limited. The ability to remove a blade and install a spare quickly would help quite a bit. Thus far the increased initial cost has out weighed these benefits, but it is something we're keeping in mind.

A final hardware related lesson is that near the end of their lives, nodes may start to fail in quantity as particular components degrade systemically. The main thing here is to keep an eye out for this happening. When a trend becomes apparent, it may be time for wholesale replacement rather than expending further effort on piecemeal repairs.

```
#!/bin/sh
FPING=/usr/local/sbin/fping
NODELIST=/usr/aero/etc/nodes-all

${FPING} -a < ${NODELIST} | \
    xargs -n1 -J host ssh -l root host $*
```

Figure 7: `oneallnodes` script

```
#!/bin/sh
restart_key=/home/root/.ssh/sge_restart.key
if [ -r ${restart_key} ]; then
        keyarg="-i ${restart_key}"
fi
export
QSTAT=/usr/local/sge/bin/fbsd-i386/qstat
FPING=/usr/local/sbin/fping
export SGE_ROOT=/usr/local/sge
export SGE_CELL=fellowship

${QSTAT} -f | grep -- -NA- | \
    cut -d@ -f2 | cut -d' ' -f1 | \
    ${FPING} -a | \
    xargs -I node ssh ${keyarg} root@node
/etc/rc.d/sge restart
```

Figure 8: `kickexecds` script

We have also learned that while most HPC software works fine on FreeBSD, the high performance computing community strongly believes the world is a Linux box. It is often difficult to determine if a problem is due to inadequate testing of the code under FreeBSD or something else. We have found that FreeBSD is usually the cause of application problems even when Linux emulation in involved. We have had good luck porting applications that already support multiple platforms to FreeBSD. There are some occasional mismatches between concepts such as the idea of "free" memory, but the work to add features such as resource monitoring is generally not difficult and simply requires reading manpages and writing simple code. We hope that more FreeBSD users will consider clustering with FreeBSD.

System automation is even more important than we first assumed. For example, shutting down the system for a power outage can be done remotely due to our power controllers, but until we wrote a script to allow automated connections to multiple controllers, it required manual connections to dozens of controllers making the process time consuming and painful. Other examples include the need to perform operations on all nodes, for example restarting a daemon to accept an updated configuration file. To simplify this we have created a simple script which handles most cases and nicely demonstrates the power of the Unix tool model. The script, `onallnodes` is shown in Figure 7. In general we find that many problems can be solved by appropriate application of xargs and appropriate Unix tools. For example the script in Figure 8 restarts dead SGE execution daemons on nodes. By running this out of `cron` we were able to work around the problem while working to find a solution.

In addition to the technical lessons above, we have learned a pair of related lessons about our users. Both are apparently obvious, but keep coming up. First, our users (and, we suspect, most HPC users) tend to find something that works and keep using it. They are strongly disinclined to change their method of operation and are unhappy when forced to do so. For this reason, we recommend that as much standard procedure as possible be developed and working before users are introduced to the system. It also suggests that voluntary adoption of practices will only work if a large benefit is involved and will never completely work. We have found this to be particularly true in regards to the scheduler. Second, because our users and domain experts[3] and not computer scientists, they often maintain mental models of the cluster's operation that are not accurate. For example, many believe that jobs that start immediately would inevitably complete before jobs that start later. While this seems logical, the only way jobs could start immediately would be for the system to be heavily over subscribed leading to substantial resource contention and thus large amounts of unnecessary swapping and excessive context switches which in turn can result in much longer job completion times. Another example is that while many users are interested in optimization and often micro-optimization, they often have a mental model of hardware the assumes no memory caches and thus discount cache effects.

5 Thoughts on Future Clusters

To meet our users ever expanding desire for more computing cycles and to improve our ability to survive disasters, we have been investigating the creation of a second cluster. In the process we have been in considering ways the system should be different from the current Fellowship architecture. The main areas we have considered are node form factor, storage, and network interconnect.

The locations we have been looking at have been engineered for cabinets so we are looking at standard 1U nodes and blades instead of our current custom, front-port solutions. In many regards the density and maintainability of blades would be ideal, but cost considerations are currently driving us toward 1U systems. The new systems will probably have at least 8 cores in a 1U form factor though.

[3]Including real rocket scientists.

Due to the fact that disks are the largest source of failures in our newer machines and that most users don't use them, we are considering completely eliminating disks in favor of high performance networked storage. The aggregate bandwidth from clustered storage products such as those from Isilon, NetApp, and Panasas easily exceeds that of local disk without all the nuisance of getting the data off the local storage at the end. There are two issues that concern us about this option. First, we can easily swap to network storage. Second, clustered storage is fairly expensive. We would be eliminating around $100 per node in disk costs, but that will not be enough to buy a large quantity of clustered storage. We think both of these issues are not too serious, but they are potential problems.

The use of Gigabit Ethernet as the Fellowship interconnect is working, but we do have some applications like computational fluid dynamics and radiation damage modeling where a higher bandwidth, lower latency link would be more appropriate. Additionally, our goal of moving away from having any storage on the nodes is leading us toward an architecture which places heavier demands on the networks. As a result we are considering both InfiniBand and 10Gb Myrinet interconnects.

For the most part, the other decisions in Fellowship's design have worked out and we think maintaining the basic architecture would be a good idea.

6 Future Directions & Conclusions

Currently Fellowship is working well and being used to perform important computations on a daily basis. We have more work to do in the area of scheduling, particularly on improving response time for short jobs, but things are working fairly well overall. Another area for improvement is better documentation to allow users to find what they need quickly and use the system correctly. We have made some progress recently with the migration of most of our documentation to a MediaWiki based Wiki system. We hope the ease of editing will help us write more documentation.

We are currently working with a team of students at Harvey Mudd College to add a web based interface to Fellowship. The interface is being built to allow users to submit inputs for specific jobs, but is being built on top of tools which allow generic access to the cluster. We hope this will allow us to attract new classes of users.

Additionally, we have ongoing research work in the area of job isolation to improve both the security of jobs and the predictability of their run time. We are looking at ways to extend the light weight virtualization facilities of the FreeBSD jail [jail(8)] framework to add support for stronger enforcement of job boundaries.

We feel that FreeBSD has served us well in providing a solid foundation for our work and is generally well-supported for HPC. We encourage others to consider FreeBSD as the basis for their HPC clusters.

References

[Becker] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer, *Beowulf: A Parallel Workstation for Scientific Computation* Proceedings, International Conference on Parallel Processing, 1995.

[Columba] *NAS Project: Columbia.* `http://www.nas.nasa.gov/About/Projects/Columbia/columbia.html`

[Davis] Brooks Davis, Michael AuYeung, Gary Green, Craig Lee. *Building a High-performance Computing Cluster Using FreeBSD.* Proceedings of BSDCon 2003, p. 35-46, September 2003.

[diskless(8)] *FreeBSD System Manager's Manual.* diskless(8).

[jail(8)] *FreeBSD System Manager's Manual.* jail(8).

[Gerzo] Daniel Gerzo. *Introduction to NanoBSD.* `http://www.freebsd.org/doc/en_US.ISO8859-1/articles/nanobsd/`

[MPI] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* `http://www.mpi-forum.org/docs/mpi-11.ps`

[OSCAR] *Open Source Cluster Application Resources.* `http://oscar.openclustergroup.org/`

[ROCKS] *Rocks Cluster Deployment System.* `http://www.rocksclusters.org/`

[SciClone] The College of William and Mary. *SciClone Cluster Project.* `http://www.compsci.wm.edu/SciClone/`

[Tolkien] J.R.R. Tolkien. *The Lord of the Rings* 1955.

[WindowsCCS] *Windows Compute Cluster Server 2003.* `http://www.microsoft.com/windowsserver2003/ccs/`

Support for Radio Clocks in OpenBSD

Marc Balmer <mbalmer@openbsd.org>
The OpenBSD Project, Micro Systems Marc Balmer

Abstract

Every computer is equipped with at least a clock chip or a general purpose device to provide a timer function. While these timers are certainly precise enough for measuring relatively short periods of time, they are not well suited for keeping the correct time and date over a longer period, since almost every chip drifts by a few seconds per day. Even so called real-time clocks only approximately meet the real time.

External time sources can be used to synchronize the local clock with a much preciser time information. Time signals are disseminated over various systems, the best known are the US american GPS (Global Positioning System) and Time Signal Stations. Time signal stations are available in many countries; while the coding schemes vary from time signal station to time signal station, the decoding principles are similar.

This paper outlines the general problems of setting a computers time at runtime; it will then give an overview about time signal stations and the GPS system. In the last sections the OpenBSD implementation is detailed.

1 Introduction

1.1 Adjusting the System Time

While receiving and decoding the time information is comparatively simple, introducing the current time into the computer system may be a complex task. Generally, it is recommended to set the system time during the boot procedure, for example as part of the startup procedure. In this case, everything is simple and no problems will arise. If it is, however, intended to update the system time while the computer is running and possibly executing programs that rely on absolute time or on time intervals, serious problems may occur.

There are two generally different concepts to change the system time at runtime. The first concept gives maximum priority to the continuity of the time, i.e. the time may be compressed or streched, but under no circumstances may a discrete time value get lost. The second concept regards time as a sequence of time units with fixed length which can neither be stretched nor compressed, but is allowed to miss or insert a time unit.

The distinction between the two different methods is necessary as in every environment the time must not be changed without prior consideration of the software that is running. Imagine a daemon program that has to start other programs at a given time: If the continuity of the time is broken up, a particular program may never be started. Such software would only run properly if time adjustment is done by stretching or compressing the time axis.

Other software may not rely on the absolute time but on the accuracy of the system clock (tick) rate. If, in this case, the time is adjusted by speeding-up or slowing-down the tick rate (i.e. stretching or compressing the time axis), this software will fail. Such software would only run properly if time adjustment is done just by changing the time settings.

If both types of software simultaneously run on the same system, the time cannot be adjusted without producing unpredictable results. In this case, the system time should better not be adjusted at runtime.

2 Time Signal Stations

In the following sections the focus is on time signal stations that emit official time using longwave transmitters.

2.1 Germany: DCF77

An ultra-precise time mark transmitter in Germany, called DCF77, emits a 77.5 kilohertz signal modulated by the encoded current time. This time signal can be used to adjust a computer's real-time clock and to ensure accurate calendar day and time of the system. An easy-to-build receiver and decoder can be directly attached to a free port; a special driver is needed to decode the incoming time information and to update the system clock whenever needed and desired.

Principally, there are two different possibilities to synchronize the system clock with the DCF77 signal. First, the system clock can be set every time a valid time information is received from the time-mark transmitter; secondly, the update can be done in predefined time periods, for example every 5 minutes. Since the accuracy of the real-time clock device is normally good enough to ensure precise system time and date over a short time period, the second possibility may not only suffice but also minimize system overhead.

2.1.1 The DCF77 Timecode

The DCF77 signal not only provides a very stable frequency, but is also continuously modulated with the current date and time information. The bit codes to provide date and time information are transmitted during the 20th and 58th second of a minute, each bit using a 1-second window. The transmitter signal is reduced to 30the beginning of each second. This reduction lasts for 100 or 200 milliseconds to encode a bit value of 0 or 1, respectively. There is no power reduction in the 59th second; this encodes the beginning of a new minute, so that the time information transmitted during the last minute may be regarded as valid. In consequence, the encoded time information has a maximum precision of one minute. The current second can only be determined by counting the bits since the last minute gap. The following additional information is included in the DCF77 code: daylight saving time, announcement of a leap second at midnight for time adjustments, spare antenna usage and others.

The DCF77 time signal station uses the following encoding scheme to transmit time information:

Bit 15	Call bit
Bit 16	Announcement of DST change
Bit 17-18	Indication of DST
Bit 19	Announcement of a leap second
Bit 20	Start of encoded time information
Bits 21-27	Minute
Bit 28	Parity
Bits 29-34	Hour
Bit 35	Parity
Bits 36-41	Day of month
Bits 42-44	Day of week
Bits 45-49	Month
Bits 50-57	Year
Bit 58	Parity

The time information is in German legal time, that is UTC+1 (or UTC+2 during daylight saving time).

2.2 Switzerland: HBG

The Swiss HBG time signal stations emits the official Swiss time on a frequency of 75 kHz from a transmitter located in Prangins near Geneva. Since 2001 it uses an encoding scheme that is compatible with the German DCF77 station. The only difference to the DCF77 code occurs during the first second of a minute: While there is only one reduction of the emission power in the DCF77 signal, there are two reductions of 100 ms with a gap of 100 ms in the HBG signal. This additional reduction of power can be used to differentiate between the two stations. During the first second of a new hour, we see three such power reductions, at noon and midnight, even four reductions are used.

2.3 Japan: JJY

The official japanese time is disseminated using two longwave time signal station transmitter, one is located on Mount Otakadoy near Fukushima and the second on Mount Hagane on Kyushu Island. The code is different from the Swiss and German codes and due to the lack of a suited receiver and for obvious geographical constraints, no driver support has yet been writte for JJY.

2.4 Connecting the Receiver

Various techniques are applicable to connect a DCF77 receiver to a computer system: The most obvious way is to convert the two states of the amplitude to a square wave signal, i.e. to a stream of zeroes and ones. In this case, the length of the reduction is determined to define the logical bit state.

A more sophisticated way that is used by most commercially available receivers is to provide a data line with the decoded time information and a separate clock line. These lines can be used to load a shift register or to enter the data on an input port bit using the clock line as an interrupt generating strobe. A device driver is required to read the data line and to combine the bits to the complete time information. In this case, the time of the amplitude reduction is measured by the receiver.

Most DCF77 receivers, however, not only provide the data and clock line but also the undecoded square wave signal on the clock line as additional information. This allows for a rather unconventional, but easy, method to both connect the receiver to the computer hardware and decode the time information. In essence, this method is based on interpreting the undecoded square wave signal as if it were an RS-232 bit stream signal so that a standard serial interface and the standard read command can be used.

2.4.1 Using the Serial Interface

Before the bit stream signal can be used as an RS-232 input signal, the required communication parameters must be determined and the controller be programmed accordingly: The longest duration of the low state of the square wave signal (200 ms) is taken as ten consecutive serial bits (one start bit, eight data bits, one parity bit) each of 20ms, so that a total of 50 bits would be transfered per second. Consequently, if the serial interface is set to 50 Baud, 8 data bits and even-parity, a 20ms section of the square wave signal represents one bit in the controllers input byte; the first 20ms section, however, is not considered since it is interpreted as a start bit.

```
_    __________________________
 |_|_|_|_|_|_|_|_|_|_|_|_|_|
Serial data format
 S 0 1 2 3 4 5 6 7 8 P S

_          ______________
 |_________| Transmitting a 0 bit (100ms)

_                    _____
 |___________________|
Transmitting a 1 bit (200ms)
```

A logical 0 is encoded from the time signal station as a low-level pulse of 100 ms duration and therefore causes the serial controller's input byte to contain binary 11110000 (hexadecimal F0). A logical 1 encoded as a 200 ms low-level pulse simply causes an input byte of 0.

The only hardware requirement to connect the square wave signal from the DCF77 receiver to a serial RS-232 port of a computer is a TTL-to-V.24 level shifter. Only the receive data (RxD) and ground (GND) pins are used, all other pins remain unused.

2.5 Decoding the Time

The very low level functions do nothing more than collecting the time bits and storing them in an appropriate structure. As the low-level interface may need to synchronize with the DCF77 transmitter, it cannot be expected to return very quickly. Under worst-case condition, i.e. if the function is entered just after the first bit has been transmitted, it may take up to two minutes until the function has completely collected the time information.

2.5.1 Collecting the Bits

There is, however, still a problem when using the serial interface to decode the time information. This problem is due to the evaluation of the parity bit. Both input byte values, hexadecimal F0 and 0, would need the parity bit to have the even (high level) state. This is alright in the first case; but in the second case (200 ms low-level state), the input signal has still low-level when the parity bit is expected so that a parity error is generated. The decoding routine has, therefore, to consider this condition.

2.5.2 The Decoding Algorithm

At any time the decoder is started, it must synchronize with the DCF77 time signal station. To do so, it waits for a delay between two bits that is significantly longer than one second, e.g. 1.5 seconds.

After this prolonged gap, the next bit to receive is bit zero of the subsequent time information packet. The algorithm then picks up the bits until all 59 data bits are collected. In case the algorithm falls out of synchronization and misses some bits, perhaps due to bad reception of the radio signal, it detects the data loss by the time elapsed between two bits which must not be more than one second. In this case, the algorithm synchronizes with the DCF77 signal again and restarts.

Once the complete time information is received, it is immediately decoded and marked valid as soon as bit 0 of the following minute starts. The time obtained can then be used to adjust the system time.

If the time decoding function is re-entered within less than one second interval, it does not need to resynchronize. In this case, the function waits for the next data bit and stores it. In consequence, such a function needs at least two minutes only for the first call; if, however, called at regular and short intervals, the function returns after about one second so that, for example, a display can continuously be updated.

2.6 We Are Always Late

The method described above to decode the DCF77 data bits has one disadvantage: We are always late. When the read function returns, indicating that a new second just started, we are already late for 100 or 200 milliseconds which is the time that is needed to transmit the data bit. This delay must be taken into consideration when the exact time of the starting of a new second is needed.

3 GPS

The American Global Positioning System, or GPS for short, works completely different than time signal stations and its primary purpose is not the dissemination of time, but accurate three-dimensional position information all over the world. To determine the exact position with the help of satellites, very high presion timing is used. This makes the GPS system a very interesting option to receive time information: It is available globally and it inherently carries very precise time information.

GPS receivers can be very cheap and are available as USB connected devices, serially attached receivers, even PC-Card or CF attached devices exist.

Professional GPS receivers are available as PCI cards, e.g. from the German manufacturor Meinberg Funkuhren.

OpenBSD currently has support for GPS in the nmea(4) and mbg(4) codes which are described in the following section.

4 OpenBSD Implementation

All time signal stations use their own code. All have some properties of their own, like the announcement of leap seconds or the announcement of a daylight saving time change in the DCF77 code. Time is encoded in a local timezone for most stations. In consequence, all drivers that decode a particular time signal station code should follow a common, yet minimal, protocol: Report the time in coordinated universal time (UTC), not the local time.

4.1 The Sensor Framework

OpenBSD has a sensor framework with sensors being read-only values that report various environmental values like CPU temparature, fan speeds or even acceleration values if the computer is equipped with accelerometers. Sensor values can be read using the sysctl(8) interface from the commandline or in userland programs.

To support radio clocks, a new type of sensor has been introduced, the timedelta sensor that reports the error (or offset) of the local clock in nanoseconds. A radio clock device driver provides a timedelta sensor by comparing the local time with the time information received.

A userland daemon like ntpd(8) can then pick up this time information and adjust the local clock accordingly.

Timedelta sensors not only report the error of the local clock, but they also have a timestamp value indicating when exactly the last valid time information has been decoded and a status flags indicating the quality of the time information. Initially, this status is set to UNKNOW, it will then change to OK once proper time information has been received. Some radio clock drivers, e.g. udcf(4), will degrade to the WARNING state if not valid time information has been recived for more than 5 minutes. If not time is received for a longer period, the state will eventually degrade to ERROR.

4.2 udcf(4)

During the development the actual driver implementation, I have used various Expert mouseCLOCK devices manufactured in Germany by Gude Analog und Digital Systeme.

The Expert mouseCLOCK USB and the Expert mouseCLOCK are inexpensive devices available in several variations. They can be interfaced serially or using USB and decode either the German DCF77 station, the Swiss HBG station, or the British MSF time signal station.

Although the serial devices use standard V.24 signal levels, they are not serial devices in the usual sense. They use the signal pins to provide a serial stream of bits that can be decoded with the method outlined above.

The USB attached devices interestingly contain an ISDN controller with USB interface that controls the receiving part using auxillary ports of the ISDN controller.

The implemented driver, udcf, attaches to a uhub device. When the device is plugged in to a USB hub, the driver programs the device to provide power to the receiver part. It then sets up a timer in 2000 ms to let the receiver stabilize. When this timer expires, the driver starts its normal operation by polling the device over the USB bus for the falling edge of the signal in a tight loop. Once the falling edge is detected, this fast polling stops and a set of four timers is used to decode the signal.

When the device is removed from the USB hub, all timers are stopped.

On the falling edge of the signal, i.e. at the beginning of a second, fast polling is stopped and some timers are started using timeout(9). Any pending timers are first reset using timeout del(9).

The first timer expires in 150 ms. Its purpose is to detect the bit being transmitted. The current signal level is measured — if the low power emission was 100 ms long, we have a high level again;, if it is a 200 ms emission, we still have a low level. The bit detected is stored.

The second timer expires in 250 ms and is used to detect if we decode the German DCF77 or the Swiss HBG signal.

The third timer expires in 800 ms. It is used to restart fast polling over the USB bus to detect the falling edge of the signal at the start of the next second. Note that there might not be a falling edge for more than one second during the minute gap after the 58th second. This situation is detected using a third timer.

The fourth timer expires in 1500 ms after the falling edge. When it expires, we have detected the 59th second. Note that this timer will not expire during seconds 0-58 as all timers are reset when the falling edge is detected using the fast polling loop.

In the 59th second we decode and validate the complete time information just received and at the beginning of the next second we stamp the time information with microtime(9) and mark it as valid. A userland program can get at this information and knowing the received time information and the exact system time when it was valid, the userland program can calculate the exact time.

The fifth timer expires in 3000 ms. If it ever expires, we have lost reception. We set an error bit and stop the whole decoding process for some time.

The four timers need not be very precise (10% tolerance is very acceptable) - the precision of the time decoding is solely determined by the detection of the falling edge at the start of a second. All means should be taken to make this detection as precise as possible.

When the algorithm is started we do not know in which second we are, so we first must synchronize to the DCF77 signal. This is done by setting the state to synchronizing, in which we don't store the bits collected, but rather wait for the minute gap. At the minute gap, the state is changed from synchronizing to receiving. Whenever we lose reception or time information gets invalid for other reasons, we fall back to the synchronizing state.

4.2.1 Using Interrupts

The driver described above is very easy to use. But it has limitations as polling over the USB bus has to be done to detect the falling edge at the beginning of a second. It is basically this polling loop that limits the precision of the time information. Higher precision can be obtained when the falling edge of the signal causes an interrupt. No polling is needed then and the decoding driver needs only some slight adjustments.

When the falling edge is detected, further interrupts from the device are disabled and the second timer, used to restart fast polling in the udcf driver, is used to reenable interrupts from the time signal receiver, thus debouncing the signal.

The serial versions of the time signal receivers can be rather easily used to generate these interrupts. Instead of using the standard wiring, the data line that provides the signal level is attached to an interrupt generating pin of the serial port.

The default serial driver must of course be disabled and the time signal station driver must program the UART accordingly.

4.3 nmea(4)

To use GPS receivers as time source, nmea(4) has been added to OpenBSD. Unlike the other implementations presented in this paper, nmea(4) is not a device driver, but a tty line discipline. A tty line discipline consists of a set of functions that are called by the tty driver on events like a character has been received, a character is to be sent etc. Thus a line discipline can look at (and manipulate) a serial data stream on a tty device.

The purpose of the nmea(4) line discipline is to decode a serial NMEA 0183 data stream originating from an attached GPS device. NMEA is rather simple, ASCII based protocols where a NMEA speaking device emits so called sentences that always start with a $ character and extend to a CR-LF pair. No sentence is longer than 82 characters and there is an optional checksum. To decode the time information, it is sufficied to decode the GPRMC sentence, the "Recommended Minimum Soecific GPS/TRANSI Data".

nmea(4) supports all GPS devices that emit NMEA sentences and that attach to a serial port of some sort (RS-232, USB, or PC-Card).

There is a problem, however, with simply decoding the NMEA sentence. We have no indication **when** exactly the time information just received was actually valid. The nmea(4) line discipline takes a local timestamp when it receives the initial $ character and uses this timestamp as the base for the calculation of the local clock offset. This automatically leads to jitter but nevertheless this method gives us accurate date and time information.

4.3.1 TTY Timestamping

To address this problem, tty timestamping has been added to the OpenBSD tty driver.

Some GPS receivers provide a highly accurate pulse-per-second, PPS, signal. A PPS signal typically has microsecond accuracy and with PPS enabled, the GPRMC sentence indicates the time (and position) of the last PPS pulse. So if we can measure the exact local time when the pulse occurs, we can later, when we received the GPRMC sentence, calculate the local offset with very high precision.

This is done in the tty driver when tty timestamping is enabled. Once enabled, the tty driver will take a local timestamp at the very moment the PPS signal occurs (which must be wired to the serial ports RTS or DCD line). The nmea(4) line discipline will then use this timestamp as the base for its calculations once the GPRMC sentence is received.

To attach the nmea(4) line discipline to a tty device, the utility program nmeaattach(8) can be used which can also enable tty timestamping.

Userland programs that want to use the NMEA data as well can do so as nmea(4) does not consume the data, it only looks at it. So with the proper setup, a general GPS software like gpsd can be used to do whatever you want with e.g. the position data while the running kernel just uses the time information to keep the clock adjusted.

4.4 mbg(4)

The mbg(4) driver for radio-clocks supports the professional radio-clocks manufactured by Meinberg Funkuhren in Bad Pyrmont, Germany. Meinberg produces a range of industrial grade receivers for the German DCF77 time signal station and the global GPS system. All cards have a local real-time clock that can be free-running on a local oscillator, which on request is temperature compensated.

The mbg(4) currently supports the PCI32 and PCI511 DCF77 receiver cards and the GPS170 GPS receiver card. All cards provide the exact time information which is available to the driver at any time, plus status information.

Especially with the newer cards PCI511 and GPS170 a very high precision can be achieved, as these cards take the internal timestamp at the very moment the first command byte is written to the card over the PCI bus. The mbg(4) driver uses a very small critical section, protected by splhigh(9), to first take the local timestamp and then send the command to the card. The critical section is immediately left and the driver waits then for the card to return the time information.

Using a kernel timeout(9), the card is queried for time information every ten seconds.

As of the time of this writing, the mbg(4) driver is still under active development, so we expect to achieve higher precision with this driver in the future.

5 Conclusion

With the advent of timedelta sensors, tty timestamping and the drivers presented in this paper, OpenBSD now has complete support for precise time acquisition, keeping and distribution.

The elegant concept of timedelta sensors, an idea by Theo de Raadt, provides a very thin layer of abstraction that allows to provide time information in a uniform way to the sytem from devices as different as a time signal station receiver that is polled over the USB bus to a PCI based GPS receiver card.

The OpenNTPD daemon ntpd(8) can then be used to distribute the time information in the network.

All this makes OpenBSD an ideal platform for time servers.

6 Acknowledgments

Meinberg Funkuhren donated several PCI based GPS and time signal station receiver cards for the development of mbg(4).
Gude ADS donated several Expert Mouse CLOCK devices for the development of the udcf(4) driver.
The concept of timedelta sensors was an idea of Theo de Raadt who also did the implementation of the tty timestamping code.
Several OpenBSD users donated radio clocks of any kind to help with time related development, which was much appreciated.
Many OpenBSD developers helped in various ways, be it by testing the code or by pushing me in the right direction.

7 Availability

nmea(4) and udcf(4) are included in OpenBSD since the 4.0 release. The newer mbg(4) driver will be included in the upcoming 4.1 release.

`http://www.openbsd.org/`

About the Author

After working for Atari Corp. in Switzerland where he was responsible for Unix and Transputer systems, Marc Balmer founded his company micro systems in 1990 which first specialised in real-time operating systems and later Unix. During his studies at the University of Basel, he worked as a part time Unix system administrator.

He led the IT-research department of a large Swiss insurance company and he was a lecturor and member of the board of Hyperwerk, an Institute of the Basel University of Applied Sciences.

Today he fully concentrates on micro systems, which provides custom programming and IT outsourcing services mostly in the Unix environment.

Marc Balmer is an active OpenBSD developer; he was chair of the 2005 EuroBSDCon conference that was held at the University of Basel.

In his spare time he likes to travel, to photograph and to take rides on his motorbike. He is a licensed radioamateur with the call sign HB9SSB.

puffs - Pass-to-Userspace Framework File System

Antti Kantee <pooka@cs.hut.fi>
Helsinki University of Technology

ABSTRACT

Fault tolerant and secure operating systems are a worthwhile goal. A known method for accomplishing fault tolerance and security is isolation. This means running separate operating system services in separate protection domains so that they cannot interfere with each other, and can communicate only via well-defined messaging interfaces. Isolation and message passing brings inherent overhead when compared to services doing communication by accessing each others memory directly. To address this, the ultimate goal would be to be able to run the kernel subsystems in separate domains during development and testing, but have a drop-in availability to make them run in kernel mode for performance critical application scenarios. Still today, most operating systems are written purely with C and some assembly using the monolithic kernel approach, where all operating system code runs within a single protection domain. A single error in any subsystem can bring the whole operating system down.

This work presents *puffs* - the Pass-to-Userspace Framework File System - shipped with the NetBSD Operating System. It is a framework for implementing file systems outside of the kernel in a separate protection domain in a user process. The implementation is discussed in-depth for a kernel programmer audience. The benefits in implementation simplicity and increased security and fault tolerance are argued to outweigh the measured overhead when compared with a classic in-kernel file system. A concrete result of the work is a completely BSD-licensed sshfs implementation.

Keywords: userspace file systems, robust and secure operating systems, message-passing subsystems, BSD-licensed sshfs

1. Introduction

"*Microkernels have won*", is a famous quote from the Tanenbaum - Torvalds debate from the early 90's. Microkernel operating systems are associated with running the operating system services, such as file systems and networking protocols, in separate domains, and component communication via message passing through channels instead of direct memory references. This is known as isolation and provides an increase in system security and reliability in case of a misbehaving component [1]; at worst the component can corrupt only itself instead of the entire system. However, most contemporary operating systems still run all services inside a single protection domain with the popular argument being an advantage in performance. Even if we were to disregard research which states that the performance difference is irrelevant [2], we might be willing to make a tradeoff for a more robust system.

A separate argument is that we do not need to see issues only in black-and-white. An operating system's core can be monolithic with the associated tradeoffs, but offer the interfaces to implement some services in separate domains. An HTTP server or an NFS server can be implemented either as part of the monolithic kernel or as a separate user process, even though they both have their "correct" locations of implementation.

There is obviously room for both a microkernel and a monolithic kernel approach within the same operating system. Another relevant argument is the use of inline assembly in an operating system: almost everyone agrees that it is wrong, yet not using it makes the system less performant. Clearly, performance is not everything.

This work presents *puffs*, the Pass-To-Userspace Framework File System for NetBSD. *puffs* provides an interface similar to the kernel virtual file system interface, vfs [3], to a user process. *puffs* attaches itself to the kernel vfs layer. It passes requests it receives from the vfs interface in the kernel to userspace, waits for a result and provides the caller with the result. Applications and the rest of the kernel outside of the vfs module cannot distinguish a file system implemented on top of *puffs* from a file system implemented purely in the kernel. For the userspace implementation a library, libpuffs, is provided. libpuffs not only provides a programming interface to implement the file system on, but also includes convenience routines commonly required for implementing file systems.

puffs is envisioned to be a step in moving towards a more flexible NetBSD operating system. It clearly adds a microkernel touch with the associated implications for isolation and robustness, but also provides an environment in which programming a file system is much easier than compared to the same task done in the kernel. And instead of just creating a userspace file system framework, the lessons learned from doing so will be turned upside down and the whole system will also be improved to better facilitate creating functionality such as *puffs*. The latter part, however, is out of the scope of this paper.

Related Work

There are several other packages available for building file systems in userspace. When this project was begun in the summer of 2005, the only option available for BSD was nnpfs, which is supplied as part of the Arla [4] AFS implementation. Arla is a portable implementation of AFS. It relies on a small kernel module, nnpfs, which attaches to the host operating system's kernel and provides an interface for the actual userspace AFS implementation to talk to. A huge drawback was that at the time it only supported caching on a file level. Since, it has developed block level caching and some documentation on how to write file systems on top of it [5].

The best known userspace file system framework is FUSE, Filesystem in USErspace [6]. It supports already hundreds of file systems written against it. On a technical level, *puffs* is fairly similar to FUSE, since they both export similar virtual file system interfaces to userspace. However, the are differences already currently in, for example, pathname handling and concurrency control. The differences are expected to grow as the *puffs* project reaches future goals. Even so, providing a source compatible interface with FUSE is an important goal to leverage all the existing file systems (see Chapter 5). In the summer of 2005 FUSE was available only for Linux, but has since been ported to FreeBSD in the Fuse4BSD [7] project. A derivate project of the FreeBSD porting effort, MacFUSE [8], recently added support for Mac OS X. A downside from the BSD point-of-view is that userspace library for FUSE is available only under LGPL and that file systems written on top of it have a tendency of being GPL-licensed.

Apart from frameworks merely exporting the Unix-style vfs/vnode interface to userspace for file system implementation, there are systems which completely redesign the whole concept. Plan 9 is Bell Labs' operating system where the adage "everything is a file" really holds: there are no special system calls for services like there are on Unix-style operating systems, where, for example, opening a network connection requires a special type of system call. Plan 9 was also designed to be a distributed operating system, so all the file operations are encoded in such a way that a remote machine can decode them. As a roughly equivalent counterpart to the Unix virtual file system, Plan 9 provides the 9P [9] transport protocol, which is used by clients to communicate with file servers. 9P has been adapted to for example Linux [10], but the greater problem with 9P is that it is relatively different from the (Net)BSD vfs interface and it makes some assumptions about file systems in general not valid on Unix [10]. Therefore, it was not directly considered for the userspace library interface.

DragonFly BSD has started putting forth effort in creating a VFS transport protocol, which, like 9P, would be suitable for distributed environments in which the server can exist on a different network node than the client [11]. It is also usable for implementing a file system in userspace, but is a huge undertaking and restructures much of the kernel file system code.

The main reason for writing a framework from scratch is that the ultimate goal of the work is not to develop a userspace file system framework, but rather to improve the flexibility and robustness of the operating system itself. While taking a more flexible route such as that of 9P may eventually prove to be the right thing to do, it is easier to take n small steps in reaching a goal and keep the system functional all the time. Currently, especially the kernel side of *puffs* is very lightweight and tries to be a good kernel citizen in not modifying the rest of the kernel. The ultimate goal is to gradually change this in creating a more secure and reliable operating system.

Paper Contents

Chapter 2 discusses the architecture and implementation of *puffs* on an in-depth technical level. Chapter 3 presents a few file systems built on top of *puffs*. It discusses experiences in developing them. Chapter 4 presents performance measurements and analyses the measured results. Chapter 5 contains work being done currently and outlines some future visions for development. Finally, Chapter 6 provides conclusions.

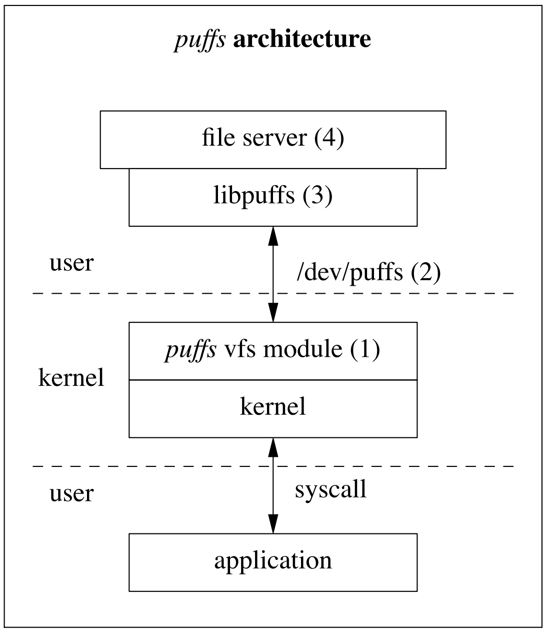
2. *puffs* Architecture

puffs is made up of four separate components (see figure):

1. VFS attachment, including virtual memory subsystem and page cache integration. This part interfaces with the kernel and makes sure that the kernel correctness is enforced. (Chapter 2.1.)
2. Messaging interface, which transports requests to and from the file system server. (Chapter 2.2.)
3. A user level adaption library, libpuffs, which handles the details of the kernel communication and provides supporting routines. (Chapter 2.3.)
4. The file system implementations themselves. (Chapter 3)

2.1. Virtual File System Attachment

Creating a new file system in the kernel is done by attaching it to the kernel's virtual file system (vfs) [3] interface. As long as the file system abides by the vfs layer's call protocols, it is free to provide the kind of file hierarchy and data content it wishes.



The vfs layer is made up of two separate interfaces: the actual virtual file system interface and the vnode interface. The former deals with calls involving file system level operations, such as mount and unmount, while the latter always involves an operation on a file; the vnode or virtual node is an abstract, i.e. virtual, representation of a file.

Vnodes are treated as reference counted objects by the kernel. Once the reference count for a vnode drops to zero, it is moved to the freelist and said to enter an *inactive* state. However, the file system in-memory data structures may still hold weak pointers to the vnode at this point and some vnode operations may prompt the file system to attempt to rescue the vnode from the freelist. Once a vnode is irreversibly freed and recycled for other use, it is said to be *reclaimed*. At this point a file system must invalidate all pointers to the vnode and in-memory file system specific data structures relating to the vnode are also freed [12].

A very central routine for every file system is the *lookup* routine in the vnode interface. This routine takes in a pathname component and produces a vnode. It must return the same vnode for the duration of the vnode's lifetime, or else the kernel could access the same file through multiple different interfaces destroying consistency guarantees. *puffs* uses cookie values to map node information between the kernel and the file server. The file server selects a cookie value and communicates it to the kernel upon node creation[1]. The kernel checks that it was not handed a duplicate, creates a new vnode and stores the cookie value in the private portion of the newly created vnode. This cookie value is passed to the file server for all subsequent operations on the kernel vnode. A *cookie* $\rightarrow$ *vnode* mapping is also stored in a hash list so that *lookup* can later determine if it should create a new vnode or if it should return the an existing one.

The cookie shared by the file server and kernel is of type `void *`. While this is not enough to cover all file system nodes on a 32bit architecture, it should be recalled that the cookie value is used only to locate an in-memory file system data structure and is valid only from node creation to the reclaim operation and that this cycle is controlled by the kernel. Most file servers will simply use the address of the in-memory data structure as the cookie value and do mapping from the cookie to the file server node structure with a simple pointer typecast. Even further, this address will be that of a generic libpuffs node, `struct puffs_node`, and the file system's private data structure can be found from the private data pointer in `struct puffs_node`. This is not required, but as we will later see when discussing the user library, the generic node provides some additional convenience features.

For interfacing between the kernel and the file server, the vfs layer acts as a translator between the in-kernel representation for vfs parameters and a serialized representation for the file server. This part is discussed further in Chapter 2.2. Additionally, the vnode portion of the vfs attachment implements the file system side of the vnode locking protocol.

The vfs layer also acts as a semantic police between the kernel and the user fs server. It makes sure that the file server does not return anything which the rest of the kernel cannot handle and would result in incorrect operation, data corruption or a crash.

Short circuiting Non-implemented Operations

All user file system servers do not implement all of the possible operations; open and close are examples of operations commonly not implemented at all on the vnode level. Therefore, unless mounted with the debug flag `PUFFS_KFLAG_ALLOPS`, operations unsupported by the file server will be short circuited in the kernel. To avoid littering operations with a check for a supported operation, the default vnode operations vector, *puffs_vnodeop_p*, defines some operations to be implemented by *puffs_checkop*(). This performs a table lookup to check if the operation is supported. If the operation is supported, the routine makes a `VOCALL()` for the operation from the vector *puffs_msgop_p* to communicate with the file server. Otherwise it returns immediately. To make this approach feasible, the script generating the vnode interface was modified to produce symbolic names for the operations, e.g. `VOP_READDIR_DESCOFFSET`, where they were previously generated only as numeric values. It should be noted that all operations cannot be directed to *puffs_checkop*(), since e.g. the reclaim operation must do in-kernel bookkeeping regardless of if the file server supports the operation in question. These operations use the macro `EXISTSOP()` to check if they need to contact the file server or is in-kernel maintenance enough.

puffs vnode op vector

```
{&vop_lookup_desc, puffs_lookup },
{&vop_create_desc, puffs_checkop },
{&vop_mknod_desc, puffs_checkop },
{&vop_open_desc, puffs_checkop },
 ...
{&vop_reclaim_desc, puffs_reclaim },
{&vop_lock_desc, puffs_lock },
{&vop_unlock_desc, puffs_unlock },
```

Kernel Caching

Caching relatively frequently required information in the kernel helps reduce roundtrips to the fs server, since operations can be short circuited already inside the kernel and cached data provided to the caller. Caching is normal behavior even for in-kernel file systems, as disk I/O is very slow compared to memory access.

The file system cache is divided into three separate caches: the page cache, the buffer cache and the name cache. The page cache [13] is a feature of the virtual memory subsystem and caches file contents. This avoids reading the contents of frequently used files from the backing storage. The buffer cache in turn [12,14] operates on disk blocks and is meant for file system metadata. The

[1] A node can be created by the following operations: *lookup*, *create*, *mknod*, *mkdir* and *symlink*. The first one just creates the node, while the final four create the backing file and the node.

name cache [12,15] is used to cache the results of the lookup from pathname to file system node to avoid the slow path of the frequent *VOP_LOOKUP*() operation.

To avoid doing expensive reads from the file server each time data is accessed, *puffs* utilizes the page cache like any other file system would. Additionally, it provides the file server with an interface to either flush or invalidate the page cache contents for a certain file for a given page range. These facilities can be used by file servers which use backends with distributed access. Since *puffs* does not operate on a block device in the kernel, it does not use the buffer cache at all. However, caching metadata is advantageous [16] even if it is not backed up by a block device. Support for caching metadata in the kernel is planned in the near future. Finally, *puffs* uses the name cache as any other file system would, but additionally provides the file server with a method to invalidate the name cache either on a per-file basis, per-directory basis or for the entire file system.

2.2. User-Kernel Messaging Interface

Messaging between the kernel and file server is done through a character device. Each file server opens `/dev/puffs` at mount time and the communication between the file server and kernel is done through the device. The only exception is mounting the file system, for which the initial stage is done by the file server by calling the *mount*() system call. Immediately when the device descriptor is closed the file system is forcibly unmounted in the kernel, as the file server is considered dead. This is an easy way to unmount a misbehaving file system, although normally `umount` should be preferred to make sure that all caches are flushed.

VFS and Vnode Operations

All vfs and vnode operations are initiated in the kernel, usually as the result of a process doing a system call involving a file in the file system. Most operations follow a query-response format. This means that when a kernel interface is called, the operation is serialized and queued for transport to the file server. The calling kernel context is then put to sleep until a response arrives (or the file system is forcibly unmounted). However, some operations do not require a response from the file server. Examples of such operations are the vnode reclaim operation and fsync not called with the flag `FSYNC_WAIT`. These operations are enqueued on the transport queue after which the caller of the operation continues executing. *puffs* calls these non-blocking type operations Fire-And-Forget (FAF) operations.

Before messages can be enqueued, they must be transformed to a format suitable for transport to userspace. The current solution is to represent parameters of the operation as structure members. Some members can be assigned directly, but others such as `struct componentname` must be translated because of pointers and other members the userland does not have direct access to. Currently all this modifying is done manually for each operation, but it is hoped that this could be avoided in the future with an operation description language.

Transport

As mentioned above, the format of messages exchanged between the kernel and file server is defined by structures. Every request structure is subclassed from `struct puffs_req`, which in C means that every structure describing a message contains the aforementioned structure as its first member. This member describes the operation enough so that it can be transported and decoded.

puffs_req members

```
struct puffs_req {
    uint64_t preq_id;

    union u {
        struct {
            uint8_t opclass;
            uint8_t optype;
            void   *cookie;
        } out;
        struct {
            int     rv;
            void   *buf;
        } in;
    } u;
    size_t  preq_buflen;
    uint8_t preq_buf[0]
      __aligned(ALIGNBYTES+1);
};
```

The messaging is designed so that each request can be handled by in-place modification of the buffer. For most operations the request structures contain fields which should be filled, but the operations *read* and *readdir* may return much more data so it is not sensible to include this space in the structure. Conversely, *write* does not need to return all the data passed to userspace.

puffs_vnreq_read/_write

```
struct puffs_vnreq_readwrite {
    struct puffs_req  pvn_pr;

    struct puffs_cred pvnr_cred;
    off_t             pvnr_offset;
    size_t            pvnr_resid;
    int               pvnr_ioflag;

    uint8_t           pvnr_data[0];
};
```

When querying for requests from the kernel, the file server provides a pointer to a flat buffer along with the size of the buffer. The kernel places requests in this buffer either until the next operation would not fit in the buffer or the queue of waiting operations is empty. To facilitate in-place modification for operations which require more space in the response than in the query (read, readdir), the kernel leaves a gap which can fit the maximal response.

This solution, however, is suboptimal. It was designed before the continuation framework (see Chapter 2.3) and does not take into account that the whole flat buffer is not available every time a query is made. The currently implemented workaround is to *memcpy*() the requests from the buffer into storage allocated separately for the processing of each operation. To fix this, the query operation will eventually be modified to use a set of buffers instead of one big buffer.

Responses from the user to the kernel use a scatter-gather type buffering scheme. This facilitates both operations which return less or more data than what was passed to them by the kernel and also operations which do not require a response at all. To minimize cross-boundary copy setup costs, the ioctl argument structure contains the address information of the first response. The `puffs_req` in the first response buffer contains the information for the second response buffer and so forth. This way only one copyin is needed per buffer instead of one for the header describing how much to copy from where and one for the buffer itself.

Snapshots

puffs supports building a snapshotting file system. What this means is that it supports the necessary functionality to suspend the file system temporarily into a state in which the file system server code can take a snapshot of the file system's state. Denying all access to the file system for the duration of taking the snapshot is easy: the file system server needs only to stop processing requests from the kernel. This is because, unlike in the kernel, all requests come through a single interface: the request queue. However, the problem is flushing all cached data from the kernel so that the file system is in a consistent state and disallowing new requests from entering the request queue while the kernel is flushing the information.

NetBSD provides file system suspension routines [17] for implementing suspending and snapshotting a file system within the kernel. These helper routines are designed to block any callers trying to modify the file system after suspension has begun and before all the cached information has been flushed. Once all caches have been flushed, the file system enters a suspended state where all writes are blocked. After a snapshot has been taken, normal operation is resumed and blocked writers are allowed to continue. Note that using these synchronization routines is left up to the file system, since generic routines cannot know where the file system will do writes to backing storage and where not.

puffs utilizes these routines much in the same fashion as an in-kernel file systems would. A file server can issue a suspend request to the kernel module. This causes the kernel vfs module to block all new access to the file system and flush all cached data. The kernel uses four different operations to notify the file server about the progress in suspending the file system. First, `PUFFS_SUSPEND_START` is inserted at the end of the operations queue to signal that only flushing operations will be coming from this point on. Second, when all the caches have been flushed, `PUFFS_SUSPEND_SUSPENDED` is issued to signal that the kernel is now quiescent. Note that at this point the file system server must still take care that it has completed all operations blocked with the continuation functionality or running in other threads and can only then proceed to take a clean snapshot. Finally, the kernel issues an explicit `PUFFS_SUSPEND_RESUME`, even though it always follows the suspend notification. In case of an error while attempting to suspend, the kernel issues PUFFS_SUSPEND_ERROR. This also signals that the file system continues normal operation from the next request onwards.

2.3. User Level Library

The main purpose of the user library, libpuffs, is to take care of all details irrelevant for the file system implementation such as memory management for kernel operation fetch buffers and decoding the fetched operations.

The library offers essentially two modes of operation. The file server can either give total control to the library by calling *puffs_mainloop*(), or invoke the library only during points it chooses to with the *puffs_req* family of functions. The former is suited for file systems which handle all operations without blocking while the latter is meant for file systems which need to listen multiple sources of input for asynchronous I/O purposes. Currently, the library does not support a programming model where the library issues a separate worker thread to handle each request.

Interface

The current *puffs* library interface closely resembles the in-kernel virtual file system interface. The file server registers callbacks to the library for operations and these callbacks get executed when a request related to the callback arrives from the kernel.

For file system operations, only three operations from vfsops are exported: sync, statvfs and unmount. The sync callback is meant to signal the file server to synchronize its state to backing storage, statvfs is meant to return statistics about the file system, and unmount tells the file server that the kernel has requested to unmount the file system. The user server can still fail an unmount request which was not issued with MNT_FORCE. The kernel will respect this.

The operations dealing with file system nodes are greater in number, but some operations are missing when compared to the kernel vnode interface. For example, the kernel uses *VOP_GETPAGES*() and *VOP_PUTPAGES*() for integration with the virtual memory subsystem[2] and as a backend for *VOP_READ*() and *VOP_WRITE*() on most file systems. However, since *puffs* userspace file servers do not integrate into the kernel virtual memory subsystem, they do not need *VOP_GETPAGES*() and *VOP_PUTPAGES*() and can simply make do with read and write.

The parameters for the node operations follow in-kernel vnode operations fairly closely. Operations are given an opaque library call context pointer, pcc, and the operation cookie, opc, which the file server can use to find its internal data structure. The meaning of the operation cookie depends on each operation, but it is either the directory which the operation affects or the node itself if the operation is not a directory operation. For example, in the signature of rmdir, the operation cookie is the cookie of the directory from which the file is supposed to be removed from, targ is the cookie of the node to be removed and pcn describes the directory entry to remove from the directory.

puffs_node_rmdir

```
int
node_rmdir(struct puffs_cc *pcc,
    void *opc, void *targ,
    const struct puffs_cn *pcn);
```

Full descriptions of each operation and involved parameters can be found from the *puffs* manual pages [18].

Filenames and Paths

The kernel vnode layer has only minimal involvement with file names. Most importantly, the vnode does not contain a pathname. This has several benefits. First, it avoids confusion with hardlinks where there are several pathnames referring to a single file. Second, it makes directory rename a cheap operation, since the pathnames of all nodes under the given directory do not need to be modified. Only operations which require a pathname component are passed one. Examples are lookup, create and rmdir. The latter two require the pathname component to know what is the name of the directory entry they should modify.

However, most file system backends operate on paths and filenames. Examples include the sftp backend used by psshfs and the puffs null layer (discussed further in Chapter 3.1). To

[2] In NetBSD, file system read and write are commonly implemented as *uiomove*() on a kernel memory window. getpages is used to bring file data into memory while putpages is used to flush it to storage. This is how the file data is cached into the page cache and written from it.

facilitate easier implementation of these file systems, *puffs* provides the mount flag `PUFFS_FLAG_BUILDPATH` to include full pathnames[3] in componentnames passed to interface functions as well as store the full path in `struct puffs_node` for use by the file server. In addition to providing automatic support for building pathnames, *puffs* also provides hooks for file systems to register their own routines for pathname building in case a file system happens to support an alternative pathname scheme. An example of this is sysctlfs (Chapter 3.1), which uses sysctl MIB names as the pathnames stored in `struct puffs_nodes`. This alternate scheme helps keep pathnames in the same place as other file systems, but it requires some extra effort from the file system: the file system must itself complete the path in routines such as lookup after it figures out its internal representation for the pathname component; file systems based on "regular" pathnames do not require this extra burden.

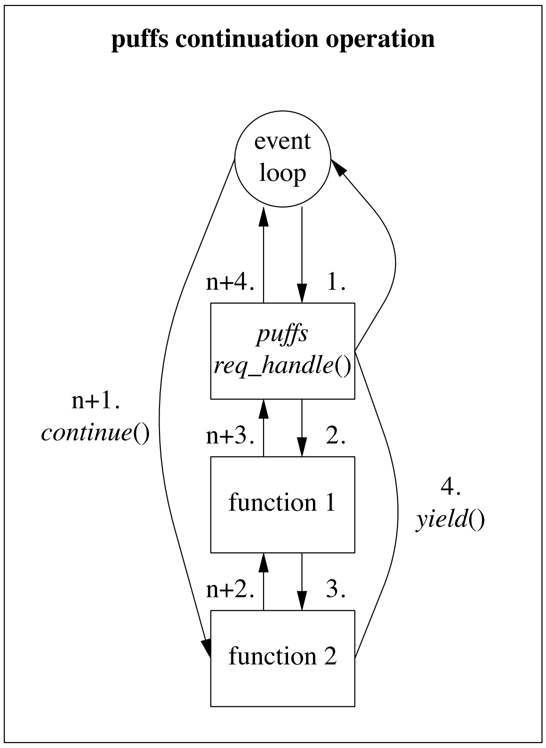
The advantage of having pathnames as an optional feature provided by the framework is that file servers implemented more in the style of classical file system do not need to concern themselves unnecessarily with the hassle of dealing with pathnames, and yet backends which require pathnames have then readily available. The framework also handles directory renames and modifies the pathnames of all child nodes of a renamed directory.

Continuations

libpuffs operates purely as a single threaded program. The question between the preference for an event loop or multiple threads is mostly an open question and the conscious decision was to in no way bias the implementation in such a fashion that threading with all its uncertainties [19] would be required to create a working file system which does not block while waiting for operations to complete.

The *puffs* solution is to provide a continuation framework in the library. Multitasking with continuations is like multitasking with cooperative threads: the program must explicitly indicate scheduling points. In a file system these scheduling points are usually very clear and similar to the kernel: a yield happens when the file system has issued an I/O operation and starts waiting for the result. Conversely, a continue is issued once the result has been produced. This also bears resemblance to how the in-kernel file systems operate (*ltsleep*()/*wakeup*() and the buffer cache operations *biowait*()/*biodone*()) and should provide a much better standing point for running unmodified kernel file systems under *puffs* than relying on thread scheduling.

[3] "full" as in "starting from the mount point"



The programming interface is extremely simple. The library provides an opaque cookie, `struct puffs_cc *pcc`, with each interface operation. The file system can put itself to sleep by calling *puffs_cc_yield*() with the cookie as the argument and resume execution from the yield point with *puffs_cc_continue*(). Before yielding, the file system must of course store the `pcc` in its internal data structures so that it knows where to continue from once the correct outside event arrives. This is further demonstrated in the above figure and also Chapter 3.1, where the *puffs* ssh file system is discussed.

However, since the worker thread model is useful for example in situations where the file system must call third party code and does not have a chance to influence scheduling points, support for it will likely be added at some stage. Also, a file system can be argued to be an "embarrassingly parallel" application, where most operations, depending slightly on the backend, can run completely independently of each other.

3. Results and Experiences

puffs has been imported to the NetBSD source tree. It will be featured in the upcoming NetBSD 4.0 release as an unsupported experimental subsystem. Example file systems are shipped in source form to make it clear no binary compatibility is going to be provided for NetBSD 4.0. Full support is planned for NetBSD 5.0.

3.1. Example File Systems

psshfs - puffs sshfs

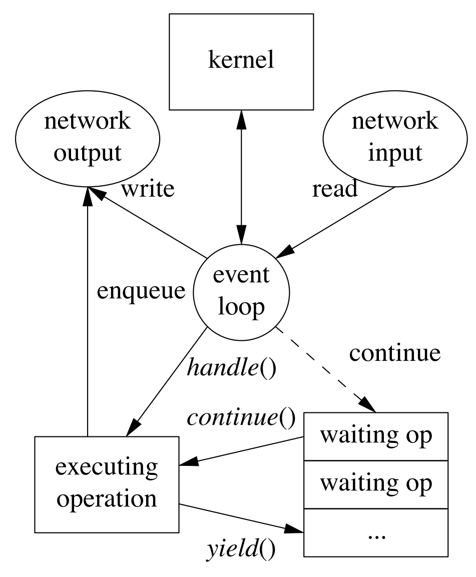
One desired feature commonly associated with userspace file systems is sshfs. It gives the ability to mount a remote file system through the sftp ssh subprotocol [20]. The most widely known sshfs implementation is FUSE sshfs. It was originally available only for Linux, but is currently available also for FreeBSD and Mac OS X. However, since all the other projects use (L)GPL licensed original FUSE code, with *puffs* NetBSD is only operating system to provide a completely BSD-licensed sshfs solution out-of-the-box.

While psshfs will be supported fully by the eventual release of NetBSD 5.0, NetBSD 4.0 ships with an experimental source-only simple sshfs, ssshfs, found under `share/examples/puffs/ssshfs` in the source tree. The difference between ssshfs and psshfs is that ssshfs was written as simple glue to OpenSSH code and cannot utilize puffs continuations. psshfs was written completely from scratch with multiple outstanding operations in mind.

The operational logic of psshfs is based on an event loop and puffs continuations. The loop is the following:

1. read and process all requests from the kernel. some of these may enqueue outgoing network traffic and *yield*().
2. read input from the network, locate continuations waiting for input, issue *continue*() for them. if a request blocks or finishes, continue from the next protocol unit received from the network. do this until all outstanding network traffic has been processed.
3. send traffic from the outgoing queue until all traffic has been sent or the socket buffer is full.
4. issue responses to the kernel for all operations which were completed during this cycle.

psshfs operational diagram



dtfs

dtfs was used for the final development of *puffs* before it was integrated into NetBSD. It is a fully functional file system, meaning that it can do all that e.g. ffs can. The author has run it on at least */tmp*, */usr/bin* and */dev* of his desktop system. For ease of development dtfs uses memory as the storage backend. However, it is possible to extend the file system for permanent storage by using a permanent storage backed memory allocator, such as one built on top of *mmap*() with `MAP_FILE`.

Development of dtfs was straightforward, as it does what the exported kernel virtual file system layer assumes a file system will do and it very closely resembles the operational logic of in-kernel file systems.

puffs nullfs

A nullfs [12] layer, also known in some contexts as a loopback file system [21], is provided by libpuffs. A null or loopback layer maps a directory hierarchy from one location to another. The puffs nullfs is conceptually similar to the in-kernel nullfs in that it acts as a simple passthrough mechanism and always relays unmodified calls the file system below it. However, since it is implemented in the user library instead of the kernel, it cannot simply push the request to the next layer. Instead, it uses pathnames and system calls to issue requests to the new location.

The null layer in itself is not useful, especially since NetBSD already provides a fully functional alternative in the kernel. However, it can be used to implement various file systems which modify filenames or file content with very little effort for the backend. An example of a user of the null layer is rot13fs, which is less than 200 lines of code and even of those almost half are involved with setting up the file system and parsing command line options. rot13fs translates pathnames and file content to rot13 for any given directory hierarchy in the file system.

sysctlfs

sysctlfs was an experiment in writing a file system which provides the storage backend through other means than a traditional file system block device -like solution. It maps the sysctl namespace as a file system and supports querying (with e.g. *cat*) and changing the values of integer and string type sysctl nodes. Nodes of type "struct" are currently not supported. Traversing the sysctl namespace is possible with standard tools such as *find*(1) or *fts*(3). sysctlfs does not currently support dynamically adding or removing sysctl nodes. While support for the latter would be possible, the former is problematic, since the current file system interface exported to processes in the form of system calls does not provide any obvious way to specify all the information, such as node type, required to create a sysctl node. Non-obvious kludges such as abusing mknod are possible, though.

Development was mostly done during a single day. One of the features introduced to *puffs* because of sysctlfs was the ability to instruct the kernel vfs attachment to bypass cache for all operations. This is useful here because re-querying the information each time from sysctl(3) is not expensive and we want changes in both directions to show up as quickly as possible in the other namespace.

3.2. Experiences

The above clearly demonstrates that adapting a name hierarchy and associated data under the file system interface is possible with relative ease and in a very short time. It can be argued that the development time was cut down greatly due to the author's intimate familiarity with the system. But it must also be pointed out that some time included in the development time was spent tracking down generic kernel bugs triggered by the corner-case vfs uses of userspace file systems and that some effort was used on framework development. Currently, the development of simple file systems should take only hours or days for someone with a reasonable familiarity in the problem scope.

3.3. Stability

One of the obvious goals is to "bulletproof" the kernel from mistakes or malice in other protection domains. The author has long since developed file systems purely on his desktop machine instead of inside an emulator or test environment. This has resulted in a few crashes in cases where the userspace file server has been acting erroneously. There are no known cases of *puffs* leading to a system crash when the file system is operating properly and many people in fact already run psshfs on their systems. Incidents where a misbehaving file server manages to crash the system are being fixed as they are discovered and discoveries are further and further apart.

It is, however, still very easy to figure out a way to maliciously crash the system, such as introduce a loop. This is more of a convenience problem than a security problem, though, since mounting a file system still requires special privileges not available to regular users.

Simply using the system long enough and developing new file systems will iron out all fairly easy-to-detect bugs. However, to meet the final goal and accomplish complete certainty over the stability and security of the system, formal methods more developed than cursory analysis and careful C coding principles are required.

4. Performance

These performance measurements are meant to give a rough estimate of the amount of overhead that is caused by *puffs*. Naturally a userspace file system will always be slower than a kernel file system, but the question is if the difference is acceptable. Nevertheless, it is important to keep in mind that the implementation has not yet reached a performance tuning stage and what has been measured is code which was written to work instead of be optimal.

The measurements were done on 2GHz Pentium 4 laptop running NetBSD 4.99.9. Note that the slowness of disk I/O is exacerbated on a laptop.

The first measurement used was extracting a tarball which contains the author's kernel compilation directory hierarchy from memory to the target file system. The extracted size for this is 127MB and contains 2332 files. It will therefore reasonably exercise both the data and name hierarchy sides of a file system.

The files were extracted in two different fashions: a single extract and two extractions running concurrently. For non-random access media the latter will stress disk I/O even more.

Four different setups were measured in two pairs: ffs and ffs through puffs nullfs; dtfs and tmpfs[4]. Technically this grouping gives a rough estimate about the overhead induced by *puffs*. It should be noted that the double test for the dtfs case is not entirely fair, as the machine used for testing only has 512MB of memory. The tree and the associated page cache does not fit into main memory twice. The tmpfs test does not have this problem, as it does not store the tree in memory and in the page cache.

tar extraction test

	tmpfs (s)	dtfs (s)	diff (%)
single	3.203	11.398	256%
double	5.536	22.350	303%
	ffs (s)	ffs+null (s)	diff (%)
single	47.677	53.826	12.9%
double	109.894	113.836	3.6%

Another type of test performed was the reading of a large file. It was done both directly off of ffs and through a *puffs* null layer backed by ffs and it was done both for an uncached file (uc) and a file in the page cache (c). Additionally, the null layer test was done so that the file was in the page cache of the backing ffs mount but not the cache of the null mount itself (bc). This means that the read travelled from the kernel to the user server, was mapped as a system call to ffs, and the data was found from the ffs file system's page cache, so no disk I/O was necessary.

4.1. Analysis of Results

The results for extraction show that *puffs* is clearly slower than an in-kernel file system. This is expected. But what is surprising is how little overhead is added. tmpfs is a high optimized in-kernel memory efficient file system. dtfs is a

[4] tmpfs is NetBSD's modern memory file system

read large file

	system (s)	wall (s)	cpu (%)
ffs (uc)	0.2	11.05	1.8
null (uc)	0.6	11.01	5.9
ffs (c)	0.2	0.21	100.0
null (c)	0.2	0.44	61.6
null (bc)	0.6	1.99	31.7

userspace file system written for testing purposes and not optimized at all. It uses *malloc*() as a storage backend and as a extreme detail it does not do block level allocation; rather it *realloc*()s the entire storage for a file when it grows.

tmpfs contains 4828 lines of code while dtfs is 1157 lines. The difference in code size is over four times as many lines of code for tmpfs. The difference in development effort probably was probably even greater than this, although of course there is no measurable evidence to back it up. Development cycles for fatal errors for a kernel file system are also considerably slower: even though loadable modules can be used to reduce the test cycle time to not require a complete reboot, this will not help if the file system under test crashes the kernel.

Even though tmpfs and dtfs are compared here, it is important to keep in mind that they in no way attempt to compete with each other.

A regular system call for a file operation requires the user-kernel privilege boundary to be crossed twice, while the puffs null scheme requires it to be crossed at least six times: system calls do not map 1:1 to vnode operations, but rather they usually require several vnode operations per system call. However, as the results show, the wall time penalty is very much hidden under the I/O time imposed by the media.

The large file read test mostly measures cache performance. The interaction of *puffs* with the page cache is less efficient than ffs. The reasons will be examined in the future. Also an interesting result is the direct read from disk, which was always slower than the read from disk via nullfs. This result cannot yet be fully explained. One possible explanation is that the utility *cat* used for testing issues *read*() system calls using the file system blocksize as the buffer size and this creates suboptimal interaction with ffs. When reading the file through the null layer the read-ahead code requests 64k (`MAXPHYS`) chunks and these are converted back to system calls at the null layer and ffs is accessed in 64k chunks providing better interaction. This is, however, just a hypothesis.

The "backend cached" test (bc) gives yet another idea of overhead introduced by *puffs*. It shows that reading a file in backend cache is ten times as expensive in terms of wall time as reading it directly from an in-kernel file system's cache is. It shows a lot of time was spent waiting instead of keeping the CPU busy. This will be analyzed in-depth later.

5. Current and Future Work

Even though *puffs* is fully functional and included in the NetBSD source tree, work is far from complete. This chapter outlines the current and future work for reaching the ultimate goals of the project.

File System Layering

File system layering or stacking [12,22] is a technique which enables file system features to be stacked on top of each other. All layers in the stack have the ability to modify requests and the results. A common example of such a file system is the union file system [23], which layers the top layer in front of the bottom layer in such a fashion that all modifications are done on the top layer and shadow the file system in the bottom layer.

While rot13fs is a clear example of a layering file system implemented on top of the puffs null layer, libpuffs does not yet support any kind of layering. Making layering support an integral, easy-to-use, non-intrusive part of libpuffs a future goal.

Improving Caching

As mentioned in Chapter 2, kernel caching is already at a fairly good stage, although it could still use minor improvements. However, library support for generalized caching is missing. The goal is to implement caching support on such a level in libpuffs that most file systems could benefit from the caching logic by just supplying information about their backend's modification activity.

This type of library caching is useful for distributed file system where the file system backend can be modified through other routes than the kernel alone. In cases where the file system is accessed only through the local kernel, the file server does not need to take care about caches: the kernel will flush its caches correctly whenever it is required, for example when a file is removed.

Another use is more aggressive read-ahead than what the kernel issues. To give an example, when reading a file in bulk over psshfs, the kernel read-ahead code eventually starts issuing reads in large blocks. However, an aggressive caching subsystem could issue a read-ahead already for the next large block to avoid latency at a block boundary. It could also measure the backend latency and bandwidth figures and optimize its performance based on those.

Messaging Interface Description

Currently the message passing interface between the kernel and libpuffs is described with struct definitions in `puffs_msgif.h`. All request encoding and decoding is handled manually in code both in the kernel and libpuffs. This is both error-prone and requires manual labour in a number of places. First of all, multiple locations must be modified both in the kernel and in the library in case of an interface change. Second, since all semantic information is lost when the messages are written as C structures, it is difficult to facilitate a tool for automatically creating a skeleton file system based on the properties of the file system about to be written.

By representing the message passing interface by a higher level description with, for example XML, much of the code written manually can be autogenerated. Also, this would lend to skeleton file system creation and to building limited userspace file system testers based on the properties of the created file system skeletons.

Abolishing Vnode Locking

Currently the system holds vnode locks while doing a call to the file server. The intent is to release vnode locks and introduce locking to the userspace file system framework. This will open up several opportunities and will enable the file system itself to decide what kind of locking it requires; it knows its own requirements better than the kernel.

Self-Healing and Self-Recovery

In case a file server hangs due to a programming error, processes accessing the file system will hang until the file server either starts responding again or is killed. While the problem can always be solved by killing the file server, it requires the intervention from someone with the correct credentials. Detecting malfunctioning servers and automatically unmounting them would introduce recovery and self-healing properties into the system. Remounting the file system automatically afterwards would minimize a break in service.

Compatibility

To leverage the huge number of userspace file systems already written and available, it makes sense to be interface compatible with some projects. The most important of these is FUSE, and a source code level compatibility layer to *puffs* for FUSE file systems, dubbed *refuse*, is being developed as a third party effort. As of writing this, the compatibility layer is able run simple FUSE file systems such as hellofs. Progress here has been fast.

Another interesting compatibility project is 9P support. Even though, as stated earlier, supporting it in the kernel would require a huge undertaking, emulating it on top of the *puffs* library interface may prove to be a manageable task. Currently though, the author knows of no such effort.

Longer Term Goals

A large theme is improving the vfs layer by identifying some of its properties through formal techniques [24] and using these to show that the *puffs* kernel side correctly shields the kernel from malicious and/or accidentally misbehaving user file system servers. It also allows for the development of the vfs subsystem into a more flexible and less fragile direction.

6. Conclusions

The Pass-to-Userspace Framework File System (*puffs*), a standard component of the NetBSD operating system, was presented in depth, including the kernel and user level architecture. *puffs* was shown to be capable of supporting multiple different kinds of file systems:

- psshfs - the puffs sshfs file system capable of mounting a remote location through the ssh sftp protocol
- dtfs - an in-memory general-purpose file system
- sysctlfs - a file system mapping the sysctl tree to a file system hierarchy
- nullfs - a file system providing any directory hierarchy in the system in another location

The ease of development of these file systems was observed to be good. Similarly, the development test cycle time and time for error recovery from crashes was observed to be very close to nil. The comparison is the typical times measured in minutes for kernel file systems. Additionally, *puffs* does not require any special tools or setup to develop, as is typical for kernel development. Rather, standard issue user program debuggers such as *gdb* can be attached to the file system and the live file system can be debugged on the same host as it is being developed on.

Performance of file systems built on top of *puffs* was shown to be acceptable. In cases where the storage backend has any significant I/O cost, i.e. practically anything but in-memory file systems, the wall time cost for *puffs* overhead was shown to be shadowed by the I/O cost. As expected, *puffs* was measured to introduce some additional CPU cost.

Finally, since *puffs* is entirely BSD licensed code, it provides a significant advantage to some parties over (L)GPL licensed competitors.

Acknowledgements

The *puffs* project was initially started under Google Summer of Code 2005 mentored by Bill Studenmund. Some later funding was provided by the Ulla Tuominen Foundation.

Professor Heikki Saikkonen helped review and finalize this paper.

References

1. Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos, "Can We Make Operating Systems Reliable and Secure?," *IEEE Computer,* vol. 39, no. 5, pp. 44-51.
2. Brian N. Bershad, *The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems,* pp. 205-211, Workshop on Micro-Kernels and Other Kernel Architectures (1992).
3. S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX,* pp. 238-247, Summer Usenix Conference, Atlanta, GA (1986).
4. Assar Westerlund and Johan Danielsson, *Arla---a free AFS client,* pp. 149-152, Usenix Freenix Track (1998).

5. Kristaps Dzonsons, *nnpfs File-systems: an Introduction,* Proceedings of the 5th European BSD Conference (November 2006).

6. Miklos Szeredi, *Filesystem in USErspace, http://fuse.sourceforge.net/* (referenced February 1st 2007).

7. Csaba Henk, *Fuse for FreeBSD, http://fuse4bsd.creo.hu/* (referenced February 1st 2007).

8. Amit Singh, *A FUSE-Compliant File System Implementation Mechanism for Mac OS X, http://code.google.com/p/macfuse/* (referenced February 4th, 2007).

9. Bell Labs, "Plan 9 File Protocol, 9P," *Plan 9 Manual.*

10. Eric Van Hensbergen and Ron Minnich, *Grave Robbers from Outer Space: Using 9P2000 Under Linux,* pp. 83--94, USENIX 2005 Annual Technical Conference, FREENIX Track (2005).

11. *"The Clustering and Userland VFS transport protocol - summary"* (May 2006). DragonFly BSD Kernel mailing list thread title.

12. Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley (1996).

13. Chuck Silvers, *UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD,* pp. 285-290, Usenix Freenix Track (2000).

14. *buffercache(9) -- buffer cache interfaces* (October 2006). NetBSD Kernel Developer's Manual.

15. Marshall Kirk McKusick, Samuel J. Leffler, and Michael J. Karels, "Name Cacheing," *Measuring and Improving the Performance of Berkeley UNIX* (April 1991).

16. David C. Steere, James J. Kistler, and M. Satyanarayanan, *Efficient User-Level File Cache Management on the Sun Vnode Interface,* pp. 325-332, Summer Usenix Conference (1990).

17. Juergen Hannken-Illjes, *fstrans(9) -- file system suspension helper subsystem* (January 2007). NetBSD Kernel Developer's Manual.

18. *puffs -- Pass-to-Userspace Framework File System development interface* (February 2007). NetBSD Library Functions Manual.

19. Edward A. Lee, "The Problem with Threads," UCB/EECS-2006-1, EECS Department, University of California, Berkeley (2006).

20. J. Galbraith, T. Ylönen, and S. Lehtinen, *SSH File Transfer Protocol draft 03,* Internet-Draft (October 16, 2002).

21. Sun Microsystems, "lofs - loopback virtual file system," *SunOS Manual Pages, Chapter 7FS* (April 1996).

22. David S. H. Rosenthal, *Evolving the Vnode Interface,* pp. 107-118, Summer Usenix Conference (1990).

23. J. Pendry and M. McKusick, *Union Mounts in 4.4BSD-Lite,* pp. 25-33, New Orleans Usenix Conference (January 1995).

24. Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi, *Using Model Checking to Find Serious File System Errors,* pp. 273-288, OSDI (2004).

An ISP Perspective, jail(8) Virtual Private Servers

Isaac (.ike) Levy, <ike@lesmuug.org>

The denial of complexity is the beginning of failure.

- Swiss historian, Jacob Burkhardt

..with proper design, the features come cheaply. This approach is arduous, but continues to succeed.

- UNIX co-creator, Dennis Ritchie

...As in all Utopias, the right to have plans of any significance belonged only to the planners in charge.

- Jane Jacobs, *"The Death and Life of Great American Cities"* [0]

One of the first significant elements of UNIX [1], was process time-sharing [2]. It's easy to forget these early times, as we now commonly touch relatively inexpensive multi-cpu hardware, eclipsing the power of a PDP-11; with smp and multi-threading kernels. Computers therefore manage simultaneous processes scaled to levels only the most adventurous could dare imagine back when UNIX first appeared. Active and persistent memory have of course scaled with raw CPU power. And it continues to get faster. We all know this.

We all know about machines, and have come to repeat the design intentions of time-sharing in many forms, including the FreeBSD jail(8) facility- a *virtual* machine.

The jail(8) subsystem in FreeBSD is well known to be an incredibly secure and durable system for partitioning processes, memory, network, and disk i/o. Building on the simplest of core UNIX subsystems, jail is an elegant base for creating Virtual Private Servers (`# man 8 jail`) To bastardize this rich and elegant system on FreeBSD:

chroot(2), bound to an **IP address**, minus some relevant system calls **= jail**

(Simply add a BSD userland, and a full virtual system is born, with a confined root!)

This material assumes the reader is familiar with the jail(8) utility, and generally familiar with the mechanisms of the underlying jail(2) system call. Further reading on the use and implimentation of jail(8) can be found in the paper written by jail's original author, '*Jails: Confining the omnipotent root.*', (PHK/Watson, FreeBSD Core) [3].

This material aims to share real-world experiences running massively jailed systems, from a ISP perspective. Diverse goals and agendas can be liberated by applying modular, self-contained, and disposable technologies- (in short, traditional UNIX principles).

Audience for these materials:

- UNIX System Administrators with demanding users, *and limited hardware resources*

- Internet Service Providers who wish to provide robust shared hardware services

- Internet Service Providers with rigorous high-availability requirements, where mutually untrusted users and processes pose a threat to service reliability (uptime)

- Institutions with fast-paced development, learning, or short-lived server requirements

The iMeme Experience, my time at a small jailing ISP- (the first of it's kind?)

Around 2000 I became a customer at a small web hosting company called iMeme. The iMeme specialty, root-access virtual servers (using FreeBSD jail(8)). My need, was to run and further develop the behemoth web application server, Zope. I needed basics- root, a compiler, cron, logfile analysis and reporting tools- (a full server). My budget was under $70/mo usd, and back then a dedicated server was unrealistic at that rate- I needed virtual-hosting scaled prices.

By 2002, iMeme hit some stiff 'problems' when a partner left, I was then asked to join the company- and we gave it quite a go. During my time at the company we hit a mark of 1000 domains hosted, in around 470 jailed systems. The ISP was unique in that once you paid for your jailed system online, it was 'booted', and you had access to your new server- no Administrator action was necessary. iMeme, as a company, later died based on external business problems.

Mutually Untrusted Users, (and processes).

2007, it can be estimated there are 785 million people using the ipv4 internet [4], arguably a critical mass. Most of these users have personal computers, yet a great deal of computing today, again, happens on servers, offering services in various contexts.

As the needs of users become more sophisticated and varied, the applications become a uniquely fragmented environment. From a birds eye view, an astounding amount of computing machinery makes all these network applications run. From a micro view, it doesn't take much computing machinery to run a single Gmail account- (from the CPU clock perspective).

With that, the proliferation of network software which looks suspiciously like 'websites', (and perhaps mislabeled as such), are starting to to take various business applications off the PC, and onto the webserver, en masse'. Everything from content and asset management systems, to financial accounting and transaction systems, to the core of the internet- information exchange through blogs, online communities, and on, and on. Through a sort of promiscuity of form [5], http applications are evolving to manifest timeless forms of 'traditional' software.

Users of any given ISP always include developers, hackers [6] , us. The mass of internet users who do not hack, have the same sophisticated and diverse demands. For example, thank MySpace for escalating user expectations in mass-market accessibility in http server applications. With that, iMeme aimed to provide an inexpensive base platform for new internet applications like this to grow.

The real world of iMeme users: A hacker: "I want to compile LISP", An undergraduate sociology student: "I want to install 'Foo' blog software, it's PHP and the instructions say I need to run Cron", A web designer: "I want to run an http server on port 8080". A business owner: "I want to run Foo web application for my business." A community leader: "I want to run Mailman List Manager", A 13 year old hacker: "I want to run both an IRC and jabber server for my friends". Most iMeme users simply, just wanted to hack Python/Zope.

Fairly simple requirements, yet so hard for commodity web hosting to accommodate!

Each of these users demands, and deserves, root.

The real world of iMeme users was extremely diverse. From a business perspective, the 'markets' served were all considered niche- hosting companies thought we were crazy. However, we felt the internet is merely niches stitched together to make a whole, and jail enabled a unique opportunity to build our ISP in the model of a metropolitan city [7].

Timeless Methodology in Computing (UNIX, the undead in computing)

Ancient UNIX computing models revolved around a model which the PC era did away with: server applications, feeding thin clients (server + many UNIX terminals). PC's evolved, and network computing became largely a peer-to-peer affair. The internet, has now brought a swing in the pendulum back to thin clients, as the Web Browser, as software, takes on the same role a terminal did years ago- and UNIX is right there, ready and waiting to handle the applications- with an astounding wealth of time-tested (and some ancient) tools well suited for managing multi-user multi-process servers.

With that, simple, modular, disposable utilities are vital to meeting the diverse needs of the iMeme user, in providing a full Virtual Private Server environment.

When jail(8) was first introduced to FreeBSD, it was (and still is) a simple utility, written in the spirit of old UNIX. As a simple utility, jail(8) provided iMeme the opportunity to build on the work of others and avoid reinvention and incompatibilities, (classic UNIX methodology).

jail(8) therefore proved itself well suited to to taking on the complexities of our user needs, which were essentially limitless. Other Virtualized system designs come close, but insomuch as most Virtual OS systems take on the monolithic responsibility of providing all system interfaces, (virtualized memory, networking, filesystem), they all critically failed to meet the iMeme needs in one area or another- as their respective histories were to meet a particular computing problem, or use case.

The history of computing is littered with the corpses of Virtual OS systems, all of which end up withering under the sheer weight of the computational responsibilities they take on. However, like UNIX time sharing, simple and modular components of computational virtualization seem to be the only elements which persist. Subsystems like UNIX users and ACL's, actually the entire concept of UNIX privilege separation, follows in the footsteps of the simple mechanism of time-sharing. Enter, jail(8), 1998.

As a small and complete utility, jail(8) is much like the invention of of the Otis Elevator and it's affect on the design of skyscrapers,

"In the era of the staircase all floors above the second were considered unfit for commercial purposes, and all those above the fifth, uninhabitable." [8]

The jail(8) utility, enabled the same sort of liberation of space, and with the same overtones of 'safety'- if one compares security features to elevator safety concerns, (falling).

(Running the risk of sounding silly, I am directly comparing an internet hosting ISP to a skyscraper, and skyscrapers are different from other types of buildings.)

The iMeme Experience (System Specifics)

The iMeme systems were quite simple for UNIX administrators to understand.

We ran high-density 2u (and then 1u) servers, which we aimed to have approximately 50 jails running on at any given time. In 2001, a base account was provisioned 4gb of disk space, and 100mb of what we called 'process space', the amalgamation of memory and cpu usage. Bandwidth was rarely an issue worth metering back then, so very basic QOS oriented throttling was performed to ensure every user had a fair slice of available network traffic.

For disk space, we ran scripts from the host server which simply used du, and shoved the output into MySQL databases- where we then automated the process of implementing policies of charging for extra disk usage. We choose to give 1 month of 'grace time', insomuch as sometimes logfiles would explode, or users would accidentally consume undue disk space- and we felt this was a simple buffer our customers appreciated.

Hard limits for disk space were always a consideration. Disk slices were far too rigid to meet user demands, (creating extreme overhead in managing upgrading disk space), though we did experiment with them. A persistent risk was that a user, by choice, accident, or compromise, could consume all the available disk space for a jailing system. With that, again, simple unix strategies came back into place to contain the problem. The strategy we ended up liking best was to absolutely a partition for jails, (the majority of available disk), and then perhaps break it into a few chunks to isolate various jailed disk space from each other. After time, 80gb slices worked nicely, and fitting 4x 300gb drives into 1u, this afforded a sort of 'neighborhood' partitioning. Extreme cases of disk consumption were further restricted on a per-case basis, using file-backed memory disks (disk images); **but,** especially in recent FreeBSD releases, this incurs an additional i/o penalty, which users do not appreciate- (and it soaks RAM on the host system as well). Disk images are not necessarily a practical solution for every jailed system, however flexible they are in providing hard limits to disk space.

Memory and CPU usage was polled on a regular basis for each jail. Shell scripts were originally setup to run as cron jobs *inside each jail*, which took cumulative memory consumption and cpu usage by parsing ps(1) output inside a given jail. While iMeme originally ran thes scripts inside of each jailed system, outputting totals to text files in /jail/dir/var/log/, however this always carried the risk that a user could (trivially) bypass this system to avoid increased billing or otherwise. In their jail, remember, the user has root. That stated, eventually iMeme moved this system out to the host system with new jailing features in FreeBSD 5.x- insomuch as one can list/kill processes based on the jail id, information availble to ps, and processes listed in the /proc filesystem.

FreeBSD 4.x jailing relied heavily on a jailed hostname for host-level process identification (and subsequent management)- which created problems. If a user changed their hostname, accidentally or maliciously, havoc would follow for management systems in the host system. FreeBSD 5.x solved this problem by pinning a 'jail id' to each process on the system, and providing a sysctl to lock down the ability to change hostnames within a jail.

Jailed process restrictions were then handled neatly using renice(8). Processes which hogged undue CPU were simply renice'd by the host server, releasing the process renice level after 5 minutes to see if the process was again behaving. If not, it was reniced again. This crude strategy was wildly successful in maintaining fair-share cpu and memory usage for processes. Problem processes, (things with memory leaks, for example), were then in the hands of the jailed user to deal with- without negatively impacting the other jailed users.

Fork bombs were still a threat, but from FreeBSD 5.x onward, each jail could be set to start with an escalated securelevel, and maxprocs could be locked for a jail, chflags(2) disabled in jails via host sysctl settings, and viola- fork bombs as a threat are mitigated, with relatively minimal management and resource consumption.

Network resource management is far outside the scope of this material, however, it is worth mentioning one thing: at iMeme, each jailing hardware server was conceptually treated like a network border or gateway, with routing and filtering tasks carried out inside the machine. This paradigm shift in management greatly simplified the physical network requirements, (making routers, firewalls, non-existent). With that, we ran NAT for our external IP blocks, and mapped addresses to our jails- which all ran using a private netblock, (192.168.x.x). This NAT strategy had pros and cons and is hardly worth discussion- except to state it all was run from the host servers, with negligible impact on jailed systems. Also, back then, ipfw(8) and dummynet(4) were used for very minimal network management- dummynet(4) configured to provide eqal-share bandwidth (ad-hock QOS), and IPFW was crudely used to put out fires. Today, in my Diversaform jail cluster, pf(4) nicely replaces these tools- and is becoming the de-facto packet filter- and in 5 more years, there may be something else, but it will still be running from the jailing host hardware.

Large Scale Management Techniques

(System Specifics)

At iMeme, we maintained Master Record Server (obviously a redundant system). This system primarily kept the MySQL database which recorded everything from resource usage, to billing and contact information. This strategy worked well, provided any modifications/additions to this system were thoroughly tested. This was easy, insomuch as we could replicate this system in one of our jails at any time, and then dispose of the jail. There was no reason in particular for the MySQL database, it was just used in the beginning and stuck with us reliably.

The website, where users bought jailed systems, and managed their account and billing, was all written in Zope, and had PHP elements added over time. This could have been any web technology.

As each iMeme jailed system had some custom tweaks, we maintained a pre-compiled FreeBSD useraland, preconfigured with any small tweaks to our enviornment (like the cpu/memory polling cron job mentioned before). These jailed systems were built, and put into cvs(1) repositories for long-term management, however tar(1) became the deployment tool of choice. Scripts to add new systems would effectively untar the current jailing userland, and then run scripts to add an initial user, add the root password, and start the jail.

Upgrading jails was a trivial technical process. System upgrades were handled similarly, un-tarring updated userland sources to jailed userland directories. Following the hier(7) man page, users additional applications ended up in /usr/local, and only in extreme edge cases did a customer application have problems with minor dot upgrades, (4.5 to 4.6, for example).

In FreeBSD 5.x, it became clear that running installworld, and tossing it an additional flag for the jailed directories, was even simpler than the tarballs, with the additional benefit of dispensing with keeping userland (binaries!) in CVS.

When monitoring the systems, based on the rapid scaling possibilities with the ease of adding jails, keep monitoring simple- and quiet. When problems occur on jailed systems, it's *always* possible that all jails on a particular host are affected, so if they all trip alarms, administrators can get lost in white noise. An experiment, was logger(1)/syslog(3). iMeme tried pushing all jailed logs out to master syslogd(8) server, with nearly worthless results. The valuable information was covered by the white noise of everything users were doing and running in their systems, and it also provided outright surprising breaches of privacy- so iMeme abandoned this idea immeadiately. While there are ways to sanely utilize syslogd(8) schemes, they are far outside of the scope of this material.

Jailing Redundancy (failure is life)

Jails present a uniquely simplistic mechanism for backup and fail-over. At iMeme, each jailing host kept jails in /usr/local/jails. As time and internal methodology evolved, (disk slice strategies, etc...)
/usr/local/jails/hostname.jailing.host became collected mount ponts and soft links, but the userland interface was always the same to find a given jail:
/usr/local/jails/hostname.jailing.host/JAIL_DIR

Then, each jaiing host both exported, and mounted, all other jail directories as an NFS mount. This carried extreme management benefits, worth the hassle and cursing associated with heavy NFS use. Operations could be carried out on each jailed userland *from any jailing host in the cluster*! With that stated, backups and restore became simple operations. Backing up became an operation of tarballing each jail to a backup server, (independently redundant), and restores consisted of untarring the jailed userland in the NFS mount of a jailed host. If a jailing host server died, all of it's jailed systems could then be rapidly re-distributed and re-started across the whole cluster. This process required Administrator intervention.

Post-iMeme, Diversaform jailed systems are run slightly differently- without NFS. Each jailing host has an identical hardware machine, which jailed systems are regularly synchronized to. If a jailed application requires time-based backups, it is synchronized to another jailing server (itself having a hardware twin). Diversaform systems have also been experimenting with a combination of carp(4) and ggated(8) (GEOM Gate), providing network interface virtualization and network block-level disk mirroring, but due to discovered inconsistencies of the FreeBSD carp(4) mechanism, and the relatively low adoption (and documentation) of ggated(8), this setup is still considered experimental. However, as these tools mature, they promise to help bring real-time failover of jailed systems- *without* Administrator intervention.

One last strategy for jailing failover has been called 'The Golden Jailing Formula': NFS mass storage backing for jails' userland, running on thin servers in a cluster. This is an excellent strategy, excepting it's restrictiveness for scaling. Many jailing administrators have attempted this formula, yet it doesn't scale as modularly as the iMeme strategy- which uses many jailing hosting servers, (or one jailing host). Total storage, i/o throughput, and then redundancy of this system make scaling jails difficult- and the reality of jailing, is that in many contexts, jailed systems grow far beyond initial expectations. So while this is a technologically viable plan, social, political, economic, and human factors limit it's success in most manifestations of massively jailed environments.

User Segregation (a bad idea)

Back to the various mutually untrusted users, a component of any massively jailed systems environment is to segregate users according to their threat level to the whole. This quickly takes jailing into philosophical approaches to social, political, economic, and human factors.

The wily hacker is your friend. iMeme founders' roots literally grew up at the annual Defcon security conference, in the USA. With that, many iMeme customers were politely put, a bit insane- and very demanding. Should these users be identified and placed on their own hardware, so some hackhing hyjinks don't get out of control and affect the 'small business' or 'nice' customers? This is a common question iMeme wrestled with.

However, at this real-world massively jailed ISP, the very opposite scenario manifest. The 'small business' user often followed less than adequate security practices, as well as running less stable software. With that, it was more often the 'wiley hacker' complaining about their small-business or blogging neighbor.

Regardless, it became clear that there was no viable metric for how or when to segregate users to given hardware servers , and in the end it became irrelevant in mitigating the risks inherent in any shared system. So what did iMeme do? This problem is a constant, use this environment to advantage for all.

Blindly distributing types of users across all hardware, had the distinct advantage of leveraging everyone's shared needs- keeping all systems online. The use of a customer/community mailing list created an enviornment where business owners and wiley hackers alike, could share experiences and discuss problems- all with the common aim of solving the problems. Additionally, this community took a great deal of impossible administrative overhead out of iMeme Administrator hands. It helped set the expectation that we just ran the servers, but had no expertise in using FOO PHP blog software, or BAR irc server, etc...

That stated, hackers who monitor uptime were a guard for the 'business owner' who did not, and the various social and cultural diversity of the user base ensured *somebody* was online 24/7. While this made for excellent catch-all systems monitoring, it also made it difficult to schedule upgrades- a trivial point in the context of the benefits.

Developers vs. Production Users is likewise a poor segregation line, insomuch as 'developers' are often hammering systems that 'production users' may rarely touch- and can help spot system problems before they become critical (arbitrary inode corruption, for an anecdotal example).

In the end, user diversity decreased overall failure risks in iMeme systems.

Conclusion

Through a combination of building on thirty years of UNIX, attention to social concerns, and respect for undeniable complexity, jail(8) was leveraged to great success at the ISP iMeme.

The most valuable elements in successfully running massively jailed systems were not cutting-edge technologies, but the application of ancient practices in computing, urban design, and to a great extent social and political sciences.

Now that iMeme is gone, who's next? What ISP, in private, commercial, or other contexts will step foreword to provide virtual systems?

Doesn't everyone deserve root?

Awknowledgements

Jon Ringuette (wintermute),
founding partner of iMeme, to it's end.
<bobskr@gmail.com>

Ethan and Jake, iMeme founding partners

Dave Lawson, (reality), iMeme core friend

Elise, the P1rate, whit537, beren1hand, and all the iMeme community who LIVED on irc

Footnotes

Note: parens in the text, ( ) refer to a corresponding UNIX man page, notatin of brackets [ ] refer to the footnotes below.

[0] Jane Jacobs , '*The Death and Life of Great American Cities*' Copyright 1961 Jane Jacobs, Renewed 1989.

[1] D. M. Ritchie and K. Thompson , '*The UNIX Time-Sharing System*', First presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

[2] L. P. Deutsch and B. W. Lampson, `SDS 930 time-sharing system preliminary reference manual,' Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley, April 1965.

[3] Poul-Henning Kamp <phk@FreeBSD.org> , Robert N. M. Watson <rwatson@FreeBSD.org>, '*Jails: Confining the omnipotent root.*' The FreeBSD Project

[4] World population is 6.45 billion today according to World Gazetteer <http://www.gazetteer.de/home.htm> ;it was 6.3 billion in 2002 according to the UN, <http://www.un.org/esa/population/publications/wpp2002/WPP2002-HIGHLIGHTSrev1.PDF>. Internet World Stats says 785M (March 2004) <http://www.internetworldstats.com/stats.htm>. NUA said 606M (Sept. 2002) <http://www.nua.ie/surveys/how_many_online/>. A claim that "roughly" or "about" 10% of the world's people have Internet access seems safe.

[5] Jonathan Crary, *J. G. Ballard and the Promiscuity of Forms*, Zone 1/2 (1986), pp. 159–65. 38- also reprinted in various MIT press publications.

[6] Hacker is mostly used in the Eric S. Raymond (ESR) sense of the word- "The term 'hacker' also tends to connote membership in the global community defined by the net" <http://www.catb.org/~esr/jargon/html/H/hacker.html>

[7] Christopher Alexander, '*A City is Not a Tree* , (1968), Originally published in: Architectural Forum, Vol 122, No 1, April 1965, pp 58-62 (Part I), Vol 122, No 2, May 1965, pp 58-62 (Part II). Available online in C. Alexander's Archives, <www.patternlanguage.com>

[8] Rem Koolhaas, '*Delirious New York: A Retroactive Manifesto for Manhattan*', (1997), Monacelli Press

Without the amazing support and existence of the following institutions, none of this material would have been possible :

NYCBUG
NEW YORK CITY *BSD USER GROUP

diversaf■rm inc.

Nsswitch development: nss-modules and libc separation and caching daemon extensions

Michael Bushkov
bushman@freebsd.org

Southern Federal University,
Rostov-on-Don, Russia

Abstract

This paper describes the extensions to the FreeBSD nsswitch subsystem, that should be committed to the source tree in the nearest future and the issues that had to be solved to make them. These changes are:

1) The separation of the libc and nsswitch modules, which makes libc code much lighter and nsswitch subsystem more dynamic. It also allows proper use of the nsdispatch(3) calls from the userland.
2) New features, that were added to the caching daemon (full "perform-actual-lookups" option support, "precache" and "check-files" option). They make it much more usable and similar in functionality to Linux/Solaris nscd, while having its own unique features.

Preface

The work, described below, was made during and after the Google Summer Of Code 2006, which I was lucky to participate in, working for the FreeBSD community. It is not yet committed to the -CURRENT, but I hope it to be finally reviewed and committed in the nearest future.

Nss-modules and libc separation

The idea of nss-modules and libc separation is quite straight-forward: we should make several dynamic libraries (nss_files, nss_dns, nss_compat, nss_nis) and move appropriate code from libc to them. Appropriate code is the functions which were specified as the sources during nsdispatch(3) calls.

Several issues had to be solved to separate nss-modules from the libc.

Issue 1. Common functionality

Common functionality was the almost ubiquitous problem of all nss-modules. As all nsswitch sources for the particular database usually reside in 1 file (getpwent.c, for example), their functions usually use some common routines (pw_scan, for example). To move such modules from the libc with minimal changes, common functions were moved to the internal libnssutil static library. This library is compiled with ${PICFLAG} to allow linking with shared libraries - i.e. nsswitch modules. It contains quite general routines (like copy_htent() and copy_netent()) and is used from nss_files, nss_nis, nss_dns, and nss_compat. It can also be useful if some new nss-module is introduced.

Getipnodeby**(3) functions had a lot of common functionality issues. It turned out that the simplest way to solve them is to implement getipnodeby**(3) functions not through nsdispatch(3) calls but through gethostby**(3) calls. Such modifications were made and tested for compatibility with current implementations (nsswitch regression tests, that are described below were used to ensure that the behavior of these functions didn't change).

Issue 2. Threading and private libc includes issues

All nss-modules use thread specific storage (thread local storage) by using either NSS_TLS_HANDLING or NETDB_THREAD_ALLOC macros from nss_tls.h and netdb_private.h respectively. Both of these files are libc-internal. And they both require all pthread-related calls to be hidden with namespace.h/un-namespace.h includes. To allow nss-modules to be moved out of the libc with minimal changes, <pthread.h> and <pthread_np.h> includes are enclosed with "namespace.h" and "un-namespace.h" in their source code. Path to libc/include is added to the standard include path for each module to allow "nss_tls.h", "netdb_private.h" and other libc-private files inclusion.

Such an approach allows to move out the modules from the libc to separate libraries with minimal changes to their sources, which is very useful, until this work is finally committed. The drawback of such decision is the dependency of the nss-modules code on the libc code. This dependency can surely be broken after the modules are separated in –CURRENT. For example, if all modules use only NSS_TLS_HANDLING macro to handle thread local storage data, then it will make netdb_private.h unneeded. The nss_tls.h can be modified not to use hidden versions of pthread calls and placed in the libnssutil folder (it would have to be left in libc also - as it is used not only from nss-modules but also from the libc itself). Other libc-private includes can also be easily eliminated from the modules' sources.

Issue 3. Statically linked binaries

Statically linked binaries can't call dlopen(3). But when all nss-modules are moved out from the libc, dlopen(3) is the only way to use them. To solve this issue, not only the dynamic versions of the nss-modules, but also their static versions, should be built. Libc's Makefile was modified to link statically built nss-modules in (please see Appendix A for details).

Nsdispatch.c has the nss_load_builtin_modules() function, which loads the statically linked modules into the libc at program startup. In the shared libc.so each modules' entry function is now replaced with an empty stub. In static libc.a each modules' real entry functions are used. nsdispatch.c was slightly modified to correctly distinguish real module entry functions from a stub.

The approach, that was used to link-in nss-modules into the static libc.a is quite flexible - new module can be added to the list of linked-in modules without any problems as long as it can be built as a static library (plus some 1-line changes would need to be made to the libc). The possible extension of this approach is to:

1. Make the list of the linked-in modules extendable via macro definitions, that can be defined during the buildworld.
2. Add an option to the nss-modules ports to build statically linked libraries along with shared ones.

With these changes made, the user will be able to link-in any prebuilt nss-module into the libc during the buildworld process. This would allow him to use this module's functionality with any of the statically linked binaries (/rescue is the most important example, probably) without any restrictions

Benefits of separating nss-modules from the libc

1. The code of both libc and nsswitch modules became much cleaner. The common functionality was placed into the libnssutil library and the number of interdependencies between libc and nsswitch modules sources was reduced to minimum. The code of the particular nsswitch module is not spread over several libc files, but is located in one library.

2. The described above ability to add the particular module support to the libc without any pain is now present.

3. There is now an ability to actually use nsdispatch(3) routine not only from the libc. The use of nsdispatch(3) was limited because of the number of the opaque pointers (in the dtab structure, that describes the list of nss-modules and their entry-points), that were needed to be passed in order for nsdispatch(3) to use libc built-in modules. When all nss-modules are standalone, the need in these pointers became obsolete, so nsdispatch(3) can be used and will properly work not only in the libc, but also in any other place. That gives an ability to properly support "perform-actual-lookups" option for all nsswitch databases in the caching daemon (please see the details below).

Nsswitch Regression Tests

The basic idea of the regression tests is to check that the expected functions behavior doesn't change after their sources modification. The idea of the regression testing for nsswitch is that the nsswitch query results should be generally the same after the system or nss-modules upgrade (if we don't change the databases, of course). The test procedure itself is very simple: we make a set of nsswitch queries (get**ent(3), get**byname(3) and get**byid(3) calls) and store their results in a file. When the test is done next time, it does the same queries in the same order and checks that their results are equal to the stored ones.

So, the testing is done in 2 stages.

First stage is the snapshot creation stage. We run the test and it builds a snapshot file of the nsswitch queries results. It also checks these results for correctness – numerical values must be in the correct range, (char *) strings that should not be NULL must not be NULL. For the resolver functions we can check that the ip address length corresponds to ip address type and, if the address was mapped from ipv4 to ipv6, that it was mapped correctly.

During stage 2 we use the already created snapshot to perform the same set of queries and then compare their results to the ones in the snapshot. We also check all results for correctness on this stage.

Such kind of testing can be used to test any existent nsswitch module. For example, we can take FreeBSD6-STABLE, run the first stage of the test, then upgrade to CURRENT and run the second stage of the test. The test will show all the compatibility issues between versions of nsswitch-dependent functions.

All nsswitch regression tests are C programs, that use the same testutil.h file, which carries most of the common logic (mostly in the form of macro definitions). The command line arguments are the same for almost all tests:
  -d - enables debug output, which helps to debug the test itself and to get more information in case of test failure
  -n - runs test for the get**byname(3) function
  -e - runs the test for the get**ent(3) functions
  -g, -u, -p – run the test for getgrgid(3), getpwuid(3) or getservbyport(3) functions accordingly
  -s <file> - causes the snapshot file to be created or, if it already exists, to be used to check the equality of the nsswitch queries results

The described regression tests were used while work on libc and nsswitch modules separation was being done. Their output was used to ensure that the behavior of the system with all modules built into the libc is equal to its behavior with all modules separated. They've especially helped during the getipnodeby**(3) functions reimplementation through the gethostby**(3) calls.

Regression tests can also be used to ensure that the caching daemon works correctly. To do that, we make a snapshot, when the caching for the particular nsswitch database is turned off, then we turn it on, run the stage 1 again (without rewriting the snapshot file), so that all necessary data are cached and then run stage 2 test with the snapshot file. If any error occurs during the caching process or caching daemon's marshalling/demarshalling process, it will most probably be mentioned in the test output.

Cached performance analysis

Cached gives tremendous and easily explainable performance boost for network-related nsswitch queries – LDAP is the best example, probably. That's why comparing the performance of the, for example, "passwd" nsswitch database queries to LDAP with and without caching is not of much interest. Much more interestingly is to compare caching daemon speed with the speed of the fastest nsswitch source: "files".

To do the comparison, we've used the "passwd" and "services" databases, which are quite different in their current implementation: "passwd" relies on BDB and "services" – on plain files.

We modified the sources of the getent utility so that it began to write getrusage(2) information to the stdout after each nsswitch query. Then, for each test we ran getent multiple times, forcing it to do 2 queries at 1 run. Only the speed of second query from each run was taken into account, because the first query always involves much overhead for reading nsswitch.conf file, loading nsswitch modules, caching the results, when caching was enabled and so on. The results were collected in the files and then processed by python script. For each type of testing (we used getpwnam(3) for "passwd" testing and getservbyname(3) for "services" testing), total of 10000 requests were made, 5000 of them were taken into account. For "services" database, half of requests were made for the data in the top part of the /etc/services file and half – for the data in the bottom of this file, because the time of the getservbyname(3) call is proportional to position of the needed data in /etc/services.

Here are the numbers (in microseconds), evaluated in different caching conditions:

Caching turned off

“passwd” nsswitch database	
Total time:	44880.00
Average time:	44.88
Median time:	47.00
Standard deviation:	13.39
Minimal time:	27.00
Maximum time:	157.00
“services” nsswitch database	
Total time:	5529766.00
Average time:	552.98
Median time:	1069.00
Standard deviation:	492.91
Minimal time:	30.00
Maximum time:	1209.000

Caching turned on (caching daemon is in single threaded mode):

“passwd” nsswitch database	
Total time:	102717.00
Average time:	102.72
Median time:	100.00
Standard deviation:	21.58
Minimal time:	71.00
Maximum time:	197.00
“services” nsswitch database	
Total time:	1010379.00
Average time:	101.04
Median time:	169.00
Standard deviation:	22.31
Minimal time:	71.00
Maximum time:	214.00

Caching turned on (caching daemon is in multithreaded mode – 8 threads):

“passwd” nsswitch database	
Total time:	124147.00
Average time:	124.15
Median time:	150.00
Standard deviation:	27.58
Minimal time:	78.00
Maximum time:	232.000
“services” nsswitch database	
Total time:	1213242.00
Average time:	121.32
Median time:	137.00
Standard deviation:	28.25
Minimal time:	80.00
Maximum time:	257.00

While showing good results (about 5,5 times faster) with caching enabled for "services" database, this test shows the ugly truth - it's nearly impossible to beat BDB query time with caching daemon’s query time. This fact makes using cached for local sources very questionable (not impossible, though). BDB is obviously the fastest solution, but caching daemon caches all plain files information in the uniform way, it can perform checks on local files to update cache if they are changed (with all precautions of not flushing the old data if something is wrong with the updated file) and do precaching on startup (please see below), it is more lightweight solution, that does not require BDB in tree. But, once again, if the speed is the main and only concern, then BDB is the choice.

Actually there are 3 areas, where cached’s speed can be improved:

1) Socket I/O
2) Multithreading
3) Lack of performance-improvement features

Socket IO optimizations appeared very hard to be done without major changes of the cached's architecture. And, most of the socket I/O-related calls have normal execution time, which however is much longer than BDB-related calls time. Because of these 2 reasons, no significant changes were made to the socket I/O part.

Multithreading issues doesn’t seem (according to the numbers above) to affect the caching daemon’s speed much.

Because of the described reasons, Item 3 was considered to be the most perspective way to improve cached's performance in certain cases., so the precaching feature was added to the caching daemon (please see below).

Cached extensions

"perform-actual-lookups" option full support

The nss-modules and libc separation allowed adding full support for the "perform-actual-lookups" option to the FreeBSD caching daemon. With this option turned on, cached acts exactly like Linux/Solaris nscd daemon for the particular nsswitch database - i.e. it makes requests by itself and not only caches the results, supplied by the user.

"precache" option support

"precache [cachename] [yes|no]" option support was added to the caching daemon. With this option turned on, the caching daemon precaches the specified database at startup (and, possibly, recaches it in case of local file change – please see below).

Precaching can be very useful for such databases as "services" when "perform-actual-lookups" method is turned on. If we precache data on startup, all queries to the cached would be read_request-search-read_response queries (without any write operations). And this type of queries is the fastest one in the caching daemon. It has no overhead of writing to cache, or of performing the nsdispatch(3) lookup.

This option proper support was also made possible only by the libc and nsswitch modules separation.

"check-files" option support

"check-files [cachename] [yes|no]" option is now also supported by the caching daemon. With this option turned on, cached flushes the cache for the particular nsswitch database automatically when its corresponding local file is changed. For example, cache for groups is flushed in case of /etc/group file change.

The lack of this option made caching daemon sometimes unusable during several ports installation process and required system administrator to flush the cache manually after any local database update.

FreeBSD caching daemon and nscd

The libc and nss-modules separation and cached extensions, that were made possible because of it, are directed to make nsswitch subsystem more powerful and flexible.

With all its current features FreeBSD caching daemon became similar in many terms to the nscd daemon, used in other OSes. It has its unique feature, though - the ability to rely all the nsswitch requests on the user side, and only cache their results by itself. However, because of the similar functionality and compatible configuration files, caching daemon will be probably renamed to nscd, when the work, described in this paper is committed.

Conclusion

Most notable features, that the work, described here, gives to developers are: cleaner libc and nsswitch-modules code, the easy process of adding a particular module to the list of libc's built-in modules and ability to use nsdispatch(3) not only in the libc. The latter was used to add several useful options to the caching daemon and can be possibly used to build specific nsswitch tools (like the mentioned caching daemon or getent command, for example). The described regression tests can be used in future nsswitch development to ensure the invariance of the nsswitch-related libc functions behavior.

Appendix A

```
# Include nss-modules's sources so that statically linked apps can work
# normally
NSS_STATIC+= ${.OBJDIR}/../nss_files/libnss_files.a
NSS_STATIC+= ${.OBJDIR}/../nss_dns/libnss_dns.a
NSS_STATIC+= ${.OBJDIR}/../nss_compat/libnss_compat.a
.if ${MK_NIS} != "no"
NSS_STATIC+= ${.OBJDIR}/../nss_nis/libnss_nis.a
.endif
NSS_STATIC+= ${.OBJDIR}/../libnssutil/libnssutil.a

# NSS-modules should be linked into the libc.a
nss_static_modules.o:
        ${LD} -o ${.TARGET} -r --whole-archive ${NSS_STATIC}

# libc.so should have stubs instead of module-load
# functions
nss_stubs.So:
        ${CC} ${PICFLAG} -DPIC ${CFLAGS}\
        -c ${.CURDIR}/net/nss_stubs.c -o ${.TARGET}


.if ${MK_PROFILE} != "no"
nss_static_modules.po:
        ${LD} -o ${.TARGET} -r --whole-archive ${NSS_STATIC}
.endif

DPSRC=  nss_static_modules.c nss_stubs.c
STATICOBJS+= nss_static_modules.o
SOBJS+= nss_stubs.So
CLEANFILES+= nss_static_modules.o nss_stubs.So
```

Appendix B

The details of the described work along with the patches can be found on the FreeBSD wiki:
http://wikitest.freebsd.org/LdapCachedDetailedDescription
http://wikitest.freebsd.org/MichaelBushkov

The code is located in the perforce branch:
http://perforce.freebsd.org/depotTreeBrowser.cgi?FSPC=//depot/projects/soc2006/nss%5fldap%5fcached/src&HIDEDEL=YES

How the FreeBSD Project Works

Robert N. M. Watson
rwatson@FreeBSD.org

FreeBSD Project

Computer Laboratory
University of Cambridge

1 Introduction

FreeBSD is a widely deployed open source operating system. [3] Found throughout the industry, FreeBSD is the operating system of choice for many appliance products, embedded devices, as a foundation OS for several mainstream commercial operating systems, and as a basis for academic research. This is distinct, however, from the FreeBSD Project, which is a community of open source developers and users. This paper discusses the structure of the FreeBSD Project as an organization that produces, maintains, supports, and uses the FreeBSD Operating System. As this community is extremely large, I approach this from the perspective of a FreeBSD developer. This necessarily captures the project from my perspective, but having had the opportunity to discuss the FreeBSD Project extensively with many people inside and outside the community, I hope it is also more generally applicable.

2 Introduction to FreeBSD

FreeBSD is an open source BSD UNIX operating system, consisting of a kernel, user space environment, extensive documentation, and a large number of bundled third party applications. It is widely used as an ISP server platform, including at well-known providers such as Yahoo!, Verio, New York Internet, ISC, Demon, and Pair. It is also widely used in part or in whole for appliances and embedded devices, including Juniper's JunOS, Nokia's IPSO, and for commercial operating system products, such as VXWorks and Mac OS X. The product of one of the most successful open source projects in the world, FreeBSD development work has focused on the areas of storage, networking, security, scalability, hardware support, and application portability.

The highly active FreeBSD development community centers on services offered via FreeBSD.org, which include four CVS repositories and a Perforce repository. These represent the life-blood of the development and documentation work of the Project. There are over 300 active developers working in CVS, which hosts the official development trees for the base source code, Ports Collection, projects tree, and documentation project. Significant project work also takes place in Perforce, which supports a heavily branched concurrent development model as well as guest accounts and external projects.

Another defining feature of the FreeBSD Project is its use of the liberal Berkeley open source license. Among features of the license are is remarkable simplicity (the license can be fully displayed in an 80x24 terminal window) and its ability to support derived works that are closed source, key to commercial and research adoption of FreeBSD.

3 What do you get with FreeBSD?

FreeBSD is a complete, integrated UNIX system. The core of FreeBSD is a portable multi-processing, multi-threaded kernel able to run on a variety of hardware platforms including Intel/AMD 32-bit and 64-bit processors, Intel's Itanium platform, and Sun's UltraSparc platform. FreeBSD is also able to run on several embedded platforms based on i386, ARM, and PowerPC; a MIPS port is also underway.

FreeBSD implements a variety of application programming interfaces (APIs) including the POSIX and Berkeley Sockets APIs, as well as providing a full UNIX command line and scripting environment. The FreeBSD network stack supports IPv4, IPv6, IPX/SPX, EtherTalk, IPSEC, ATM, Bluetooth, 802.11, with forthcoming support for SCTP. Security features include access control lists (ACLs), mandatory access control (MAC), security event auditing, pluggable authentication modules (PAM), and a variety of cryptographic services. FreeBSD ships with both workstation/server and embedded development targets, and comes with extensive user and programmer documentation.

FreeBSD also ships with ports of over 16,000 third party open- and closed-source software packages, providing programming and user interfaces such as X11, KDE, Gnome, OpenOffice, and server software such as Java, MySQL, PostgreSQL, and Apache.

4 The FreeBSD Project

The FreeBSD Project's success can be measured by the extremely wide deployment of FreeBSD-based systems. From root name servers to major web hosts, search engines, and routing infrastructure, FreeBSD may be found at most major service providers. FreeBSD is also the foundation for a number of commercial operating systems. The FreeBSD Project is more than just software, or even software development: it includes a global community of developers, port maintainers, advocates, and an extensive user community. Central to this community are the FreeBSD.org web site, FTP site, CVS repository, and mailing lists.

Several papers and studies have been written on the topic of the FreeBSD Project and its development process, including a papers by Richards [7], Jorgensen [4], and Dinh-Trong [1].

5 The FreeBSD Foundation

The FreeBSD Foundation is a non-profit organization based in Boulder, CO. By design, the Foundation is separate from the FreeBSD Project. When the Foundation was created, it was not clear that a non-profit supporting open source development was a viable concept. As such, it was important to the founders that the Foundation be a separate legal entity that would support the Project, but that the Project not be dependent on the long-term viability of a Foundation. It was also important to the founders of the Foundation that there be a differentiation between the people managing the monetary, legal, and administrative matters and those administering the software development work in the project. In practice, the Foundation has proved financially and administratively successful, and plays an important role in supporting the daily operation and long term success of the Project.

The FreeBSD Foundation is responsible for a broad range of activities including contract development (especially relating to Java), managing of intellectual property, acting as a legal entity for contractual agreements (including non-disclosure agreements, software licensing, etc), providing legal support for licensing and intellectual property issues, fund-raising, event sponsorship (including BSDCan, EuroBSDCon, AsiaBSDCon, and several FreeBSD developer summits a year), providing travel support for FreeBSD developers and advocates, negotiating collaborative R&D agreements, and more.

The FreeBSD Foundation is currently managed by a board of directors, and has one part-time employee who is responsible for day-to-day operation of the Foundation as well as sitting on the board. The board also consists of four volunteer members drawn from the FreeBSD developer community. The FreeBSD Foundation Board is in regular communication with other administrative bodies in the FreeBSD Project, including the FreeBSD Core Team.

The FreeBSD Foundation is entirely supported by donations, and needs your help to continue its work!

6 What We Produce and Consume

The FreeBSD Project produces a great deal of code: the FreeBSD kernel, user space, and the Ports Collection. But the FreeBSD Project does not produce "just source code". FreeBSD is a complete software product, consisting of software, distribution, documentation, and support:

- FreeBSD kernel, user space
- Ports collection, binary package builds
- FreeBSD releases
- FreeBSD manual pages, handbook, web pages, marketing material
- Architecture and engineering designs, papers, reports, etc
- Technical support, including answering questions and debugging problems
- Involvement in and organization of a variety of FreeBSD user events

This would not be possible without support of a larger community of users and consumers, who provide certain necessary commodities:

- Beer, wine, soda, chocolate, tea, and other food/beverage-related vices in significant quantity.
- Donated and sponsored hardware, especially in racks at co-location centers, with hands to help manage it.
- Bandwidth in vast and untold quantities.
- Travel grants, developer salaries, contracts, development grants, conference sponsorship, organization membership fees, etc.
- Thanks, user testimonials and appreciation, good press.
- Yet more bandwidth.

None of these has a trivial cost–by far the most important resource for the project is developer time, both volunteered and sponsored.

7 Who are the Developers?

FreeBSD developers are a diverse team, made up of members from 34 countries on six continents. They vary in age between 17 and 58, with a mean age of 32 and median age of 30; the standard deviation is 7.2 years. FreeBSD developers include professional systems programmers, university professors, contractors and consultants, students, hobbyists, and more. Some work on FreeBSD in a few spare hours in the evening once a week–others work on FreeBSD full time, both in and out of the office. FreeBSD developers are united by common goals of thoroughness and quality of work. Unlike many open source projects, FreeBSD can legitimately claim to have developers who have worked on the source base for over thirty years, a remarkable longevity that would be the envy of many software companies. This diversity of experience contributes to the success of FreeBSD, combining the pragmatic "real world problem" focus of consumers building products with the expertise of researchers working on the cutting edges of computer science research.

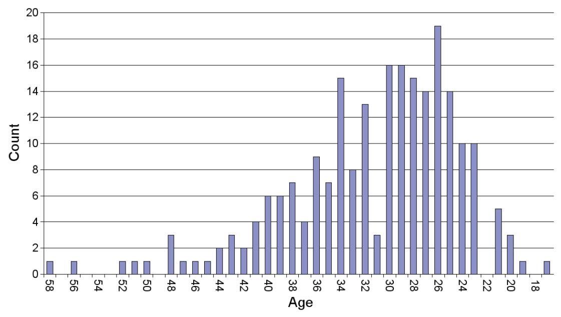


Figure 1: Age Distribution of FreeBSD Developers (2005)

8 FreeBSD Processes

The FreeBSD Project is successful in significant part because it encapsulates not just many experienced and highly competent individuals, but also because it has a set of well-defined development processes and practices that are universally accepted and self-sustaining.

- *Committer life cycle and commit bits* - The process by which new developers are inducted into the community and mentored as new members of the community is well-defined and successful.
- *Core Team* - Project leadership is selected and renewed via regular elections from the developer team as a whole, insuring both continuity, continued engagement, and fresh voices lead the project over time.
- *Mailing lists* - Through extensive and courteous use of mailing lists for almost all project communications over many years, consensus is almost universal in project decision making, and there is relatively little "stepping on toes" for a project that spans dozens of countries and time zones.
- *Web pages and documentation* - A well-designed and extremely complete set of web pages and documentation provide access to both the current condition and history of the project, from tutorial content for new users to detailed architectural information on the design of the kernel.
- *Groups/projects* - A hallmark of FreeBSD's success is the scalable community model, which combines the best of centralized software development with project-oriented development, allowing long-term spin-off projects to flourish while maintaining close ties and involvement in the central project.
- *Events* - The FreeBSD Project exists primarily through electronic communication and collaboration, but also through in-person developer and user events occurring continuously throughout the year. These include developer summits and involvement in both BSD-specific and general purpose conferences.
- *Honed development and release cycle* - With over ten years of online development and release engineering experience, the FreeBSD Project has pioneered many online development practices, combining professional software engineering approaches with pragmatic approaches to volunteer-driven open source development. One of the key elements of this approach is effective and highly integrated use of software development tools and revision control, including the use of multiple revision control systems, CVS and Perforce.
- *Centralized computing resources* - Also key to the success of the project has been the use of several globally distributed but centrally managed computing clusters, organized and maintained by project donors and a highly experienced system administration team. The FreeBSD.org infrastructure "just works", providing flawless support for the daily activities of the project.
- *Conflict resolution* - In any development project, but especially in widely distributed organizations, effective management of technical disagreements and conflicts is critical; the FreeBSD Project's history is full of examples of successful conflict resolution leading to both good technical and social outcomes.

8.1 FreeBSD Committers

A FreeBSD committer is, in the most literal sense, someone who has access to commit directly to the FreeBSD CVS repository. Committers are selected based on four characteristics: their technical expertise, their history of contribution to the FreeBSD Project, their clear ability to work well in the FreeBSD community, and their having made the previous three extremely obvious. Key to the induction of new committers is the notion of a mentor: this is an existing committer who has worked with the candidate over an extended period and is willing to both sponsor their candidacy and also act in a formal role in introducing them to the project. The mentor proposes the candidate to one of the Core Team, Port Manager, or Doceng, who respectively approve commit rights for the src tree, the ports tree, or the documentation tree. A typical proposal includes a personal introduction of the candidate, a history of their background and contribution, and volunteers to mentor them.

Once approved, typically by a vote, the new committer is given access to the FreeBSD.org cluster and authorized access to CVS. Mentorship does not end with the proposal: the mentor and new committer will have a formal ongoing relationship for several months, in which the mentor works with the new committer to review and approve all commits they will make, helps them circumnavigate the technical and social structure of the project. This relationship often continues informally in the long term, beyond the point where the mentor has "released" the new committer from mentorship. Typically, there is significant technical interest overlap between the proposing mentor and the new committer, as this will be the foundation on which familiarity with their work, as well as competence to review their work, will have been formed.

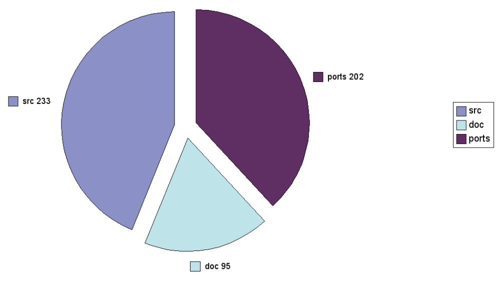


Figure 2: Number of FreeBSD committers by commit bit type (2005)

Committers often begin working in one of the various trees, and gradually spread to working in others. For example, it is not uncommon for documentation committers to expand the scope of their work to include source development, or for src developers to also maintain a set of application ports. Some of FreeBSD's most prolific and influential kernel developers have begun life writing man pages; "upgrading" a commit bit to allow access to new portions of the tree is a formal but lightweight process, in which a further proposal by a potential mentor is sent to the appropriate team for approval. As with an entirely new committer, a formal mentorship will take place, in which the new mentor takes responsibility for reviewing their commits during their earlier work with their new commit bit.

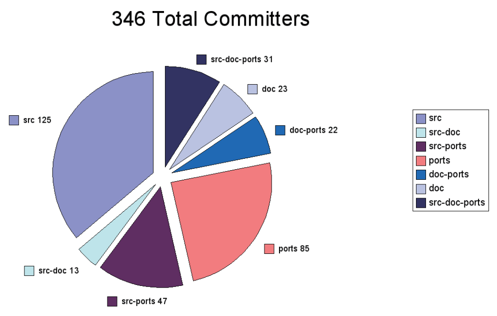


Figure 3: There is significant overlap, with many committers working in more than one area of the source tree. (2005)

8.2 FreeBSD Core Team

The FreeBSD Core Team is the nine-member elected management body of the FreeBSD Project, and is responsible for a variety of administrative activities. Historically, the Core Team consisted of a self-selected set of the leading developers working on FreeBSD; however, in 2000, the model was changed to an elected model in order to adopt a more sustainable model. Every two years, nominees from the FreeBSD committer team volunteer to be placed on the role, and a one month online election is held. The FreeBSD Core Team then appeals for and selects a volunteer to act as Core Secretary.

While the process of selecting the Core Team is well-defined, the precise responsibilities of the Core Team are not, and have evolved over time. Some activities are administrative in nature: organizing successive elections, assisting in writing and approving charters for specific teams, and approving new FreeBSD committers. Other activities are more strategic in nature: helping to coordinate developer activity, making sure that key areas are being worked in by cajoling or otherwise convincing developers they are important, and assigning authority to make significant (possibly contentious) architectural decisions. Finally, the FreeBSD Core Team is responsible for maintaining and enforcing project rules, as well conflict resolution in the event that there is a serious disagreement among developers.

8.3 Ports Committers, Maintainers

The FreeBSD Ports Collection is one of the most active areas of FreeBSD work. At its heart, the ports tree is a framework for the systematic adaptation of third party applications to FreeBSD, as well as a vast collection of ported applications. In 2005, there were 158 ports committers working on 16,000 application ports. In addition to ports committers, the notion of a ports maintainer is also important: while ports committers are often involved in maintaining dozens or even hundreds of ports themselves, they also work to funnel third party porting work by over 1,500 ports maintainers into the ports tree. Particularly prolific maintainers often make good candidates for ports commit bits. With an average of 100 ports per committer and 11 ports per maintainer, the ports work is critical to the success of FreeBSD.

The Port Manager (portmgr) team is responsible for administration of the ports tree, including approving new ports committers as well as administering the ports infrastructure itself. This involves regression testing and maintaining the ports infrastructure, release engineering and building of binary packages across half a dozen hardware platforms for inclusion in FreeBSD releases, as well as significant development work on the ports infrastructure itself. Regression testing is a significant task, involving large clusters of build systems operating in parallel; even minor infrastructure changes require the rebuilding of tens of thousands of software packages.

8.4 Groups and Sub-Projects

The FreeBSD Project is a heavily structured and sizable organization with many special interest groups working in particular areas. These groups focus on specific technical areas, support, advocacy, deployment and support of FreeBSD in various languages and in different countries. Some sub-groups are formally defined by the project, and in some cases, have approved charters and membership. Others exist more informally, or entirely independent of the central FreeBSD.org infrastructure, shipping derived software products.

8.5 A FreeBSD Project Org Chart

While the concept of an organizational chart applies somewhat less well to a loose-knit volunteer organization than a traditional company, it can still be instructive.

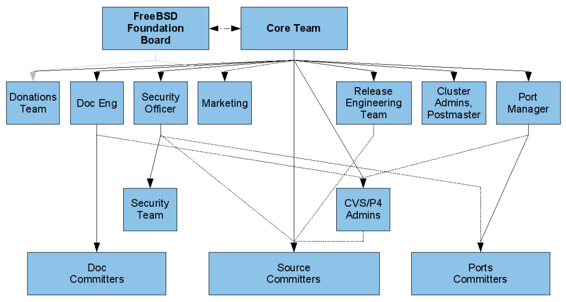


Figure 4: Lines in this FreeBSD Project Org chart represent more than just downward delegation of authority found in commercial organizations.

In a traditional organization chart, arrows would represent delegation of responsibility. In the FreeBSD Project organization chart, this is only partially true: typically arrows represent delegation of authority: i.e., the FreeBSD Core Team, the elected management body of the project has assigned authority, by means of voting to approve a written chart, for security advisory and other Security Officer activities to the Security Officer and Security Officer team. As the organization is volunteer-driven, delegation of of responsibility occurs up as much as down: the larger body of FreeBSD committers select a Core Team to take responsibility for a variety of administrative activities.

8.6 Derived Open Source Projects

FreeBSD provides components, and in some cases the foundation, of a large number of derived open source software projects.

- FreeSBIE, a FreeBSD-based live CD image
- m0n0wall, an embedded FreeBSD-based firewall package
- pfSense, an extensible firewall package based on m0n0wall
- PC-BSD, a workstation operating system based on FreeBSD
- Darwin, the open source foundation of the Mac OS X operating system, which includes both portions of the FreeBSD kernel and user space
- DesktopBSD, a workstation operating system based on FreeBSD
- DragonflyBSD, a FreeBSD-derived research operating system project

- FreeNAS, a FreeBSD-based network storage appliance project

In addition, FreeBSD code may be found in an even greater number of projects that software components developed in FreeBSD; this includes open source projects such as OpenBSD, NetBSD, and Linux systems.

8.7 Mailing Lists

Mailing lists are the life-blood of the project, and the forum in which almost all project business takes place. This provides a long term archive of project activities. There are over 40 public mailing lists hosted at FreeBSD.org, as well as a number of private mailing lists associated with various teams, such as the Core Team, Release Engineering team, and Port Manager team. Mailing lists serve both the developer and user communities. A great many other mailing lists relating to FreeBSD are hosted by other organizations and individuals, including regional user groups, and external or derived projects.

8.8 FreeBSD Web Pages

Web sites are a primary mechanism by which the FreeBSD Project communicates both internally and with the world at large. The main FreeBSD.org web site acts as a distribution point for both FreeBSD as software and documentation, but also as a central point for advocacy materials. Associated web sites for the mailing lists and mailing list archives, bug report system, CVSweb, Perforce, and many other supporting services are also hosted as part of the FreeBSD.org web site.

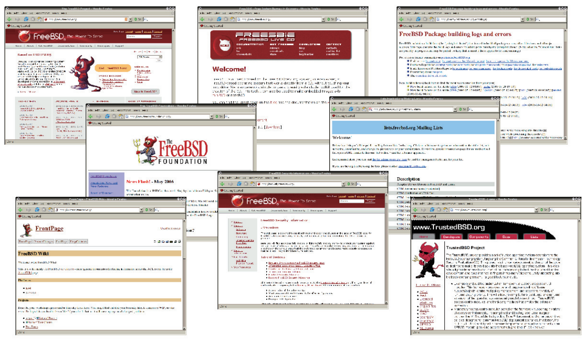


Figure 5: Web sites play an integral role in how the FreeBSD Project communicates with both users and contributors.

In addition, there are a number of project-specific web sites for FreeSBIE, TrustedBSD, PC-BSD, DesktopBSD, and others, which are linked from the main FreeBSD.org web site, but are separately authored and hosted.

8.9 Events

While electronic communications are used as the primary method of communication for most on-going work, there is no substitute for meeting people you are working with in-person. The FreeBSD Project has a presence at a great many technical workshops and conferences, such as USENIX and LinuxWorld, not to mention a highly successful series of BSD-related conferences, such as BSDCan, EuroBSDCon, AsiaBSDCon, NYCBSDCon, MeetBSD, and a constant stream of local user group and developer events.

As these conferences bring together a great many FreeBSD developers, there are often Developer Summits occurring concurrently, in which FreeBSD developers meet to present, discuss, hack, and socialize. Summits typically consist of a formal session containing both presentations and moderated discussion, and information activities, such as hacking and gathering at a bar or pub.

8.10 FreeBSD Development Cycle

FreeBSD is created using a heavily branched development model; in revision control parlance, this means that there is a high level of concurrent work occurring independently. The central FreeBSD src CVS repository contains a large number of branches; the main of these is the HEAD or CURRENT branch, where new features are aggressively developed.

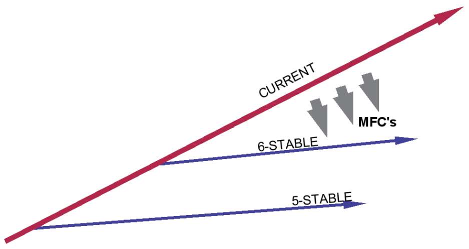


Figure 6: Branching is a key element of the FreeBSD development model: simultaneous work on several complete versions of FreeBSD at once allows changes to be merged from one branch to another as they gain stability, exposing them to successively wider testing and use.

A series of STABLE branches contains more conservative development, one per major release series, with changes being trickled from the CURRENT branch to other branches as they stabilize; this process is referred to as "Merged From Current", or MFC. Minor releases are cut from STABLE branches at regular intervals, typically three to six months. Major releases are cut around every 18 months, although sometimes less frequently, and involve the creation of a new STABLE branch; this allows extremely large features, inappropriate for merge to a STABLE release series, to be released as part of new major (.0) releases.

In addition to the CURRENT and STABLE branches, RELEASE branches are used for release cycles as well as for security and errata patches following release.

7-current	cutting edge development
6-stable	active development with releases
5-stable	legacy branch with releases
4-stable	legacy branch

Branched development is also used extensively during early feature development. Due to limitations in CVS, discussed later, this work typically occurs in branches in the FreeBSD Perforce server.

8.11 FreeBSD Releases

Release engineering is one of the most tricky aspects of running any large software project, let alone a large-scale, volunteer-run open source project. The release team (RE) is responsible for the coordinating the combination of technical and technical engineering necessary to bring a FreeBSD release to fruition. With membership approved by the Core Team, RE is given significant leeway to steer the FreeBSD development process, including placing administrative limits on development in the tree (code slushes, freezes), performing CVS branching and tagging operations, not to mentioning begging and cajoling developers into doing that which is necessary to make a release possible.

As FreeBSD development is centered on revision control, the revision control operations involved in a release are important to understanding how releases occur. Releases occur in release branches, which are typically branched from a -STABLE development branch. In preparation for a release, development on the -STABLE branch is slowed to a more conservative set of changes in order that existing new work can stabilize. First a "code slush" occurs, in which new features are eschewed, but bug fixing and refinement occurs largely unhindered; any significant changes for the release require approval by the Release Engineering team during this period. After a period of slush, a "code freeze" is started, after which point commits to the tree may only occur with the specific approval of the release am. This change in process increases the level of review taking place for changes, as well as allowing the Release Engineering team to manage risk for the release as a whole.

A series of beta test releases will be made during the code freeze, in which major and minor problems are incrementally identified and corrected. Once the Release Engineering team is satisfied with the quality of the tree, branching of the release branch may occur, which can allow more active development on the -STABLE branch to resume. A series of release candidates is used to continue to refine the release, with successively more broad testing, especially of the install procedure, which sees less exposure during normal development. Once a final release candidate is created, the release itself may occur, and the release is tagged.

Coordinated with this process for the base tree is both a release process for the ports and documentation trees. Final third party package builds occur prior to the release candidate series, ensuring testing and compatibility after significant changes have been completed in the base source tree. The Port Manager team also places a slush and freeze on the ports tree, allowing testing of the packages together rather than in isolation. The documentation tree is likewise tagged as part of the release process; an important aspect of the release is preparation of the release documentation, including the release notes identifying changes in FreeBSD, finalization of translated versions, and updates to the web site and documentation to reflect the release.

The release branches continue to serve an important role after the tagging and release of a FreeBSD version. Once the Release Engineering team believes that there is no risk of a re-roll of the release due to a last minute issue, it will transfer ownership of the branch to the Security Officer team, which will then maintain security patches against the release in that branch. The Release Engineering team may also coordinate the addition of errata patches to the branch for major stability or functional problems identified after the release. Freezes requiring approval of the Release Engineering or Security Officer teams are not released on release branches.

The FreeBSD 6.1 release process is fairly representative, in that it contained the typical snags and delays, but produced a very technically successful and widely deployed release:

25 Jan 2006	Schedule finalized
31 Jan 2006	Code freeze begins
5 Feb 2006	Ports schedule, announced
5 Feb 2006	6.1-BETA1
19 Feb 2006	6.1-BETA2
23 Feb 2006	Ports tree frozen
3 Mar 2006	6.1-BETA3
6 Mar 2006	Doc tree slush
14 Mar 2006	6.1-BETA4; ports tagged
5 Apr 2006	RELENG_6_1 branch
10 Apr 2006	6.1-RC1
17 Apr 2006	Doc tree tagged, unfrozen
2 May 2006	6.1-RC2
7 May 2006	Release tagged
7 May 2006	Build release
8 May 2006	6.1-RELEASE released

Major (.0) releases occur in a similar manner to minor releases, with the added complexity of creating a new -STABLE branch as well as a new release branch. As this occurs quite infrequently, often as much as several years apart, the process is more variable and subject to the specific circumstances of the release. Typically, the new -STABLE branch is created after a long period of code slush and stabilization in the -CURRENT branch, and occurs well in advance of the formal release process for the .0 release. Critical issues in this process include the finalization of application binary interfaces (ABIs) and APIs for the new branch, as many ABIs may not be changed in a particular release line. This includes library version updates, kernel ABI stabilization for device drivers, and more.

Incremental releases of FreeBSD, such as the 6.1 and 6.2 releases, largely require appropriately conservative strategies for merging changes from the CURRENT branch, along with some amount of persuasion of developers to address critical but less technically interesting issues. Typical examples of such issues are device driver compatibility issues, which tend to rear their heads during the release process as a result of more broad testing, and a few individuals bravely step in to fix these problems.

Larger releases, such as 3.0, 4.0, 5.0, and 6.0, require much more care, as they typically culminate several years of feature development. These have been handled with varying degrees of success, with the most frequent source of problems the tendency to overreach. While the FreeBSD 4.0 and 6.0 releases were largely refinements and optimizations of existing architecture, the FreeBSD 3.0 and 5.0 releases both incorporated significant and destabilizing architectural changes. Both resulted in a series of incremental releases on a STABLE branch that did not meet the expectations of FreeBSD developers; while these problems were later ironed out, they often resulted from a "piling on" of new features during an aggressive CURRENT development phase.

The success of the FreeBSD 6.x release series has been in large part a result of a more moderated development and merge approach, facilitated by the heavy use of Perforce, which allows experimental features to be maintained and collaborated on without merging them to the CVS HEAD before they are ready. Prior to the use of Perforce, experimental features were necessarily merged earlier, as there were not tools to maintain them independently, which would result in extended periods of instability as the base tree ceased to be a stable platform for development. The more mature development model leaves the CVS HEAD in a much more stable state by allowing a better managed introduction of new features, and actually accelerates the pace of development by allowing avoiding slowdowns in concurrent development due to an unstable base.

8.12 Revision Control

Most major technical activities in the project are centered on revision control. This includes the development of the FreeBSD source code itself, maintenance of the tends of thousands of ports makefiles and metadata files, the FreeBSD web site and documentation trees (including the FreeBSD Handbook), as well as dozens of large-scale on-going projects. Historically, FreeBSD has depended heavily on CVS, but has both extended it (via cvsup), and made extensive use of Perforce as the project has grown. The FreeBSD Project is now actively exploring future revision control options.

8.12.1 Revision Control: CVS

CVS, or the Concurrent Versions System, is the primary revision control system used by the FreeBSD Project, and holds the authoritative FreeBSD source trees, releases, etc. [2] This repository has over twelve years of repository history. The FreeBSD CVS repository server, repoman.FreeBSD.org, actually holds four separate CVS repositories:

/home/ncvs	FreeBSD src
/home/pcvs	FreeBSD ports
/home/dcvs	FreeBSD documentation
/home/projcvs	FreeBSD project

The FreeBSD Project supplements CVS in a variety of ways; the most important is cvsup, which allows high-speed mirroring and synchronization of both the CVS repository itself, as well as allowing CVS checkouts without use of the heavier weight CVS remote access protocol. This permits the widespread distribution of FreeBSD, as well as avoiding concurrent access to the base repository, which with CVS can result in a high server load. Most developers work against local CVS repository mirrors, only using the central repository for check-in operations.

Over time, the technical limitations of CVS have become more apparent; cvsup significantly enhances the scalability of CVS, but other limits, such as the lack of efficient branching, tagging, and merging operations have become more of an issue over time.

8.12.2 Revision Control: Perforce

While CVS has served the project extremely well, its age is showing. CVS fails to offer many key features of a distributed version control system, nor the necessary scalability with respect to highly parallel development. To address these problems, the FreeBSD Project has deployed a Perforce server, which hosts a broad range of on-going "projects" derived from the base source tree. [6] The most important feature that Perforce brings to the FreeBSD Project is support for highly branched development: it makes creating and maintaining large-scale works in progress possible through lightweight branching and excellent history-based merging of changes from parent branches to children.

Currently, most major new kernel development work is taking place in Perforce, allowing these projects to be merged to the base tree as they become more mature, avoiding high levels of instability in the CURRENT branch. Perforce also makes collaboration between developers much easier, allowing developers to monitor each other's works in progress, check them out, test them, and modify them. Projects that have been or are being developed in Perforce include SMPng, KSE, TrustedBSD Audit, TrustedBSD MAC, SEBSD, superpages, uart, ARM, summer of code, dtrace, Xen, sun4v, GEOM modules, CAM locking, netperf, USB, ZFS, gjournal, and many others. CVS remains the primary and authoritative revision control system of the FreeBSD Project, with Perforce being reserved for works in progress, but it plays a vital role in the growth of the project, so cannot be ignored in any serious consideration of how the project operates.

8.12.3 Revision Control: The Future

The FreeBSD Project is in the throes of evaluating potential future distributed version control systems as a potential successor to CVS and Perforce, with the goal of subsuming all activity from both into a single repository. The Project's requirements are complicated, both in terms of basic technical requirements, as well as being able to support our development processes and practices. Primary of these requirements is that the entire current CVS repository and history be imported into the new repository system, a task of non-trivial complexity, and that it support the new branched development model used heavily in Perforce. Another important consideration is continued support for the cvsup infrastructure for the foreseeable future.

8.13 Clusters

The FreeBSD Project makes use of several clusters scattered around the world, typically located at colocation centers. These clusters are possible due to the generous donations of companies using FreeBSD. One of the most important aspects of these donations is that they are not just significant donations of servers or rack space, but donations of administrative staff time and expertise, including hands to rearrange and handle new and failing hardware, reinstall and update systems, and help troubleshoot network and system problems at bizarre hours of the day and night.

8.13.1 FreeBSD.org cluster

While there are several FreeBSD Project clusters, The FreeBSD.org Cluster is hosted in Santa Clara by Yahoo!, and is home of many of the most critical systems making up the FreeBSD.org domain.

Mail servers	hub, mx1, mx2
Distribution	ftp-master, www
Shell access	freefall, builder
Revision control	repoman, spit, ncvsup
Ports cluster	pointyhat, gohans, blades
Reference systems	sledge, pluto, panther, beast
Name server	ns0
NetApp filer	dumpster

All of these systems have been made available through the generous donations of companies supporting FreeBSD, such as Yahoo!, NetApp, and HP. The systems are supported by remote power, serial consoles, and network switches.

8.13.2 Other Clusters

The FreeBSD.org cluster hosted at Yahoo! is not the only concentration of FreeBSD Project servers. Three other major clusters of systems are used by the FreeBSD Project:

- The *Korean ports cluster* hosted by Yahoo! in Korea provides a test-bed for ports work.
- *allbsd.org* in Japan provides access to many-processor Sun hardware for stress and performance testing.
- The *Sentex cluster* hosts both the FreeBSD Security Officer build systems, as well as the Netperf cluster, a network performance testing cluster consisting of a dozen network booted systems with gigabit networking. This cluster has also been used to test dtrace, hwpmc, and ZFS.
- The *ISC cluster* hosts half of FreeBSD.org, as well as a large number of ports building systems, the FreeBSD.org Coverity server, test systems, and more.

8.14 Conflict Resolution

Conflict resolution is a challenging issue for all organizations, but it is especially tricky for volunteer organizations. FreeBSD developers are generally characterized by independence, a good sense of cooperation, and common sense. This is no accident, as the community is self-selecting, and primary criteria in evaluating candidates to join the developer team are not just technical skills and technical contribution, but also the candidate's ability to work successful as part of a larger global development team. Conflict is successfully avoided by a number of means, not least avoiding unnecessary overlap in work areas and extensive communication during projects that touch common code.

Despite this, conflicts can and do arise: some consist purely of technical disagreements, but others result from a combination of the independence of spirit of FreeBSD developers and the difficulty of using solely online communications to build consensus. Most conflicts are informal and self-resolving; on the rare occasion where this is not the case, the FreeBSD Core Team is generally responsible for mediating the conflict. For purely technical disagreements, reaching a decision by careful consideration (and fiat) is often successful, relying on the elected authority of the Core Team to make a final decision. As technical disagreements are often only the trigger in more serious conflicts, the Core Team typically selects a mediator (usually a Core Team member) to help work to improve communications between the disagreeing parties, not just pick a "right" technical solution.

8.15 Bike sheds

"Bike sheds" are a very special kind of conflict found, most frequently, in technical communities. First described by Parkinson in a book on management, the heart of the issue of the bike shed lies in the observation that, for any major engineering task, such as the designing of a nuclear power plant, the level of expertise and investment necessary to become involved is so significant that most contributions are productive; however, the building of a bike shed is something that anyone (and everyone) can, and will, express an opinion on. [5] Strong opinions prove easiest to have on the most trivial details of the most unimportant topics; recognizing this problem is key to addressing it. Bike sheds, while not unique to FreeBSD, are an art-form honed to perfection by the project. Since they have become better understood, they have become much easier to ignore (or dismiss once they happen). This terminology has now been widely adopted by many other open source projects, including Perl and Subversion.

9 Conclusion

The FreeBSD Project is one of the largest, oldest, and most successful open source projects. Key to the idea of FreeBSD is not just software, but a vibrant and active online community of developers, advocates, and users who cooperate to build and support the system. Several hundred committers and thousands of contributors create and maintain literally millions of lines of code in use on tens of millions of computer systems. None of this would be possible without the highly successful community model that allows the FreeBSD Project to grow over time, as well as permitting other projects to build on FreeBSD as a foundation.

References

[1] DINH-TRONG, T. T., AND BIEMAN, J. M. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions on Software Engineering 31*, 6 (2005).

[2] FREE SOFTWARE FOUNDATION. cvs - Concurrent Versions System.
http://www.nongnu.org/cvs/.

[3] FREEBSD PROJECT. FreeBSD Project home page.
http://www.FreeBSD.org/.

[4] JORGENSEN, N. Putting it all in the trunk: incremental software development in the FreeBSD open source project. *Information Systems Journal 11*, 4 (2001), 321–336.

[5] PARKINSON, C. N. *Parkinson's Law; or, the Pursuit of progress*. John Murray.

[6] PERFORCE SOFTWARE. Perforce, the Fast Software Configuration Management System.
http://www.perforce.com/.

[7] RICHARDS, P. eXtreme Programming: FreeBSD a case study. In *UKUUG Spring Conference and Tutorials: Conference Proceedings* (2006), UKUUG.

SHISA: The Mobile IPv6/NEMO BS Stack Implementation Current Status

Keiichi Shima, Internet Initiative Japan Inc., Japan, `keiichi@iijlab.net`
Koshiro Mitsuya, Keio University, Japan, `mitsuya@sfc.wide.ad.jp`
Ryuji Wakikawa, Keio University, Japan, `ryuji@sfc.wide.ad.jp`
Tsuyoshi Momose, NEC Corporation, Japan, `momose@az.jp.nec.com`
Keisuke Uehara, Keio University, Japan, `kei@wide.ad.jp`

Abstract

Mobile IPv6 and Network Mobility Basic Support (NEMO BS) are the IETF standard mobility protocols for IPv6. We implemented these protocols and we call the implementation SHISA. SHISA supports most of the features in these mobility protocol specifications and has high level interoperability with other stacks compliant to the specifications. We are now working towards adapting the SHISA code to fit the latest BSD source tree. In this paper we explain the detailed implementation design of the stack, current status of the porting work and the future plans of our project.

1 Introduction

The rapid growth of the IPv4 Internet raised a concern of the IPv4 address exhaustion. IPv6 was designed as the essential solution of the problem. We are now on the transition period from the IPv4 Internet to the IPv6 Internet. As a result of the transition, a vast number of IPv6 devices connected to the Internet using various communication technologies will appear in the future. The devices will not only be computers and PDAs but also cars, mobile phones, sensor devices and so on. Since many devices will potentially move around changing its point of attachment to the Internet, mobility support for IPv6 is considered necessary. The IETF has discussed the protocol specification and finally standardized two IPv6 mobility protocols, Mobile IPv6 [1] for host mobility and Network Mobility Basic Support (NEMO BS) [2] for network mobility.

When we deploy a protocol, it is one of the efficient ways to provide the protocol stack as open source software. The developers of the protocol stack can get many feedback from worldwide users and can enhance their implementation. We implemented the mobility protocol stack, called *SHISA*[1] [3, 4], that supports both Mobile IPv6 and NEMO BS to provide a full featured mobility stack on top of BSD operating systems as a part of the KAME project activity [5], and released the stack as open source software. After the KAME project concluded in March 2006, we started to adapt the stack to fit the latest BSD tree aiming to merge the mobility code.

This paper presents the current status of our work on IPv6 mobility and future plans. We will provide the basic knowledge of Mobile IPv6 and NEMO BS in Section 2 and discuss the design principle and implementation detail in Section 3 and 4. Section 5 discusses the remaining stuffs to be designed and implemented to give advanced mobility features and also discusses the future plans of our project. Section 6 concludes this paper.

2 Mobile IPv6 and NEMO BS Overview

Mobile IPv6 is a protocol which adds a mobility function to IPv6. Figure 1 illustrates the operation of Mobile IPv6. In Mobile IPv6, a moving node (*Mobile Node*, *MN*) has a permanent fixed address which is called a *Home Address* (*HoA*). HoAs are assigned to the MN from the network to which the MN is originally attached. The network is called a *Home Network*. When the MN moves to other networks than the home network, the MN sends a message to bind its HoA and the address assigned at the foreign network. The message is called a *Binding Update* (*BU*) message. The address at the foreign network is called a *Care-of Address* (*CoA*) and the networks other than the home network are called *Foreign Networks*. The message is sent to a special node, called a *Home Agent* (*HA*) located in the home network. The HA replies to the MN with a *Binding Acknowledgement* (*BA*) message to confirm the request. A bi-directional tunnel between the HA and the CoA of the MN is established after the binding information has been successfully exchanged. All packets sent to the HoA of the MN are routed to the home network by the Internet routing mechanism. The HA intercepts the packets and for-

[1]SHISA was named after a traditional roof ornament in Okinawa Japan, where we had the first design meeting.

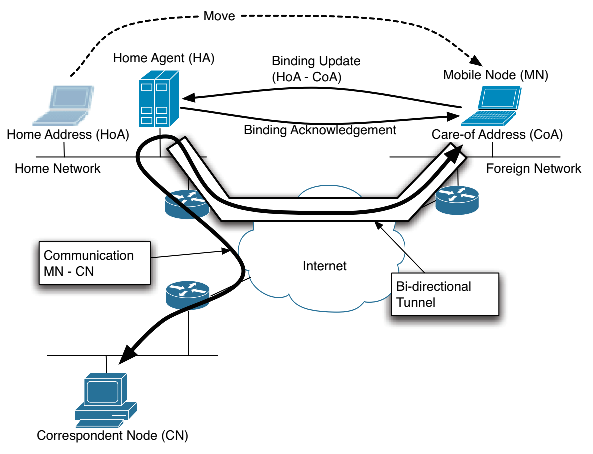


Figure 1: Basic Operation of Mobile IPv6.

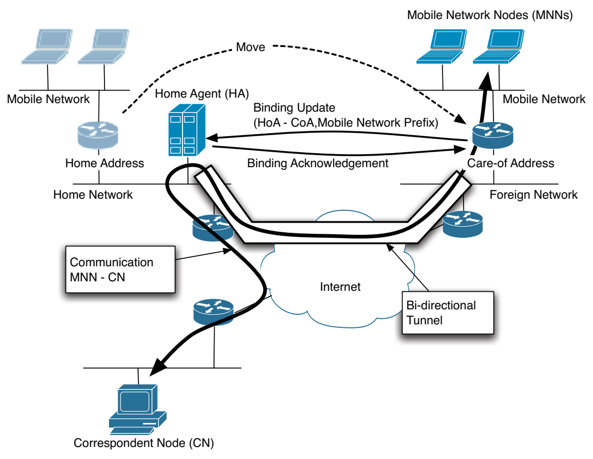


Figure 2: Basic Operation of NEMO BS.

wards them to the MN using the tunnel. Also, the MN sends packets using the tunnel when communicating with other nodes. The communicating nodes (called as *Correspondent Nodes*, *CN*) do not need to care about the location of the MN, since they see the MN as if it is attached to the home network.

In Figure 1, the communication path between the MN and its peer node is redundant since all traffic is forwarded through the HA. Mobile IPv6 allows an MN to optimise the path to an IPv6 node which is aware of the Mobile IPv6 protocol by sending a BU message. When an MN send a BU message to a CN, the MN must perform a simple address ownership verification procedure called *Return Routability* (*RR*). The MN sends two messages (*Home Test Init* (*HoTI*) and *Care-of Test Init* (*CoTI*) messages) to the CN, one from its HoA and the other from its CoA. The CN responds these two messages with *Home Test* (*HoT*) and *Care-of Test* (*CoT*) messages with cookie values. The MN then generates secret information using these two cookies and sends a BU message cryptographically protected with the secret information. Once the CN accepts the BU message, the MN can directly send a packet to the CN from its CoA. To provide the HoA information to the CN, the MN stores its HoA in a Destination Options Header as the *Home Address option* (*HAO*). The option is newly defined in the Mobile IPv6 specification. The CN can also directly send a packet to the MN using the *Routing Header Type 2* (*RTHDR2*), which is a new type of a routing header. This direct path is called a *Route Optimized* (*RO*) path.

NEMO BS is an extension of Mobile IPv6. The basic operation of a moving router (*Mobile Router*, *MR*) is same as that of an MN except the MR has a network (*Mobile Network*) behind it. The network prefix is called a Mobile Network Prefix (*MNP*). A node in the mobile network, which is called a *Mobile Network Node* (*MNN*), can communicate with other nodes as if they are attached to the home network, thanks to the tunneling between the HA and the MR. NEMO BS does not provide the RO feature. Figure 2 depicts the operation of NEMO BS.

3 SHISA Design

Mobile IPv6 and NEMO BS are layered between the Network Layer and Transport Layer. The first version of our mobility stack, known as the KAME Mobile IPv6 stack, was implemented as a part of the kernel as other Network Layer and Transport Layer protocols.

When the Mobile IPv6 specification was published as an RFC, we were considering to extend the features of the mobility stack. We thought it would not be a good idea to keep all mobility functions in the kernel, considering its extensibility and maintainability[2]. We redesigned the entire stack and moved most of the protocol functions to user space. In the process of redesign, we also referred the basic design of another Mobile IPv6 stack (SFCMIP [7]) for BSD that was being developed at Keio University. The remaining functions in the kernel was packet forwarding processing. All the mobility signal processing and binding information management processing were moved to user space. The design gives us the following benefits.

- Easy development and maintenance: Since the signaling processing of Mobile IPv6 and NEMO

[2]There was a separate project that provided a NEMO BS implementation [6] based on the KAME Mobile IPv6 stack, which was also implemented in the kernel.

BS is complicated, it is better to implement it in user space. We can develop and debug the complex part of the protocol easier than doing it in the kernel, without reducing packet forwarding performance.

- Extensibility for additional features: Developing user space programs is easier than the kernel programming in most cases and for most users. Moving the core mobility implementation from the kernel to user space will encourage third party developers to add new features.
- Minimum modification of the kernel code: When considering to merge the developed code into BSD trees, the smaller amount of kernel modification is the better. Moving signaling part to the user space reduces the amount of kernel modification.

In the user space, we also divided the entire stack into 6 pieces as follows.

- MN functions
- MR functions
- HA functions
- RO responder functions
- Movement detection functions
- NEMO BS tunnel setup functions

The design allows users to chose only necessary components when they build mobility aware nodes. For example, if one wants to build an MN that does not act as an RO responder, he can disable it. The design also allows to replace components with their own implementation. Especially, the ability to replace the movement detection mechanism is useful when deploying mobility services in a specific network infrastructure that supports a good movement detection mechanism, such as the Layer 2 movement notification scheme. In that case, the system integrator can create a special movement detection program, keeping other signaling processing code untouched.

The components, including the kernel, communicate each other through a newly designed socket domain dedicated to mobility information exchange. When a user program put some information (e.g. binding information) to the kernel, this socket domain is used. The socket domain is used to exchange such information between user space programs too. It can also be used as a notification mechanism from the kernel to user space programs. When the kernel has to notify information that can only be retrieved inside the kernel, such as extension header processing errors or tunneled packet input events, the kernel writes the event information to the socket domain so that all the listening programs of the socket in user space can receive the event data.

4 Implementation

SHISA was originally developed on top of the KAME IPv6 stack [9] for NetBSD 2.0 and FreeBSD 5.4. We ported SHISA to the NetBSD-current tree as the first step of porting effort. There are two reasons why we chose NetBSD as the first platform for the porting work. The first reason is that it supports various kinds of architectures. The mobility functions are useful especially when it is integrated to a moving entities such as PDAs and cars or trains, and so on. They usually use an architecture that runs with limited resources. NetBSD supports many such architectures that is suitable for embedded use, and we wanted to realize such small devices using our code. The other reason is the difference between the KAME tree (that was based on NetBSD 2.0) and the latest NetBSD is relatively small compared to other BSD variants that KAME supported. This makes it easier to port the SHISA code from KAME to NetBSD-current.

Figure 3 shows the relationship of the SHISA modules. The objects with solid lines are newly implemented modules. The dotted line objects exist in the original BSD system and the shaded ones of them have been modified for the SHISA system.

There are 6 user space programs; **mnd**, **babymdd**, **cnd**, **mrd**, **nemonetd** and **had**. Each program handles, the MN signaling messages, the movement detection procedure, the RO responder signaling messages, the MR signaling messages, the tunnel setup procedure for NEMO BS, and the HA signaling messages respectively. The binding database that corresponds the HoA and CoA of an MN is maintained by **mnd** and **mrd** on the MN/MR side, and by **cnd** and **had** on the CN/HA side. The subset information of the databases that is necessary for the packet input and output processes in the kernel is injected by these programs using the Mobility socket discussed in Section 4.1.

The communication interface used between the kernel and the user space programs, and between the user space programs is provided by the newly implemented Mobility socket domain (AF_MOBILITY). The mechanism and message formats used in the domain are similar to the Routing socket [10]. Unlike the Routing socket, we use this socket to exchange

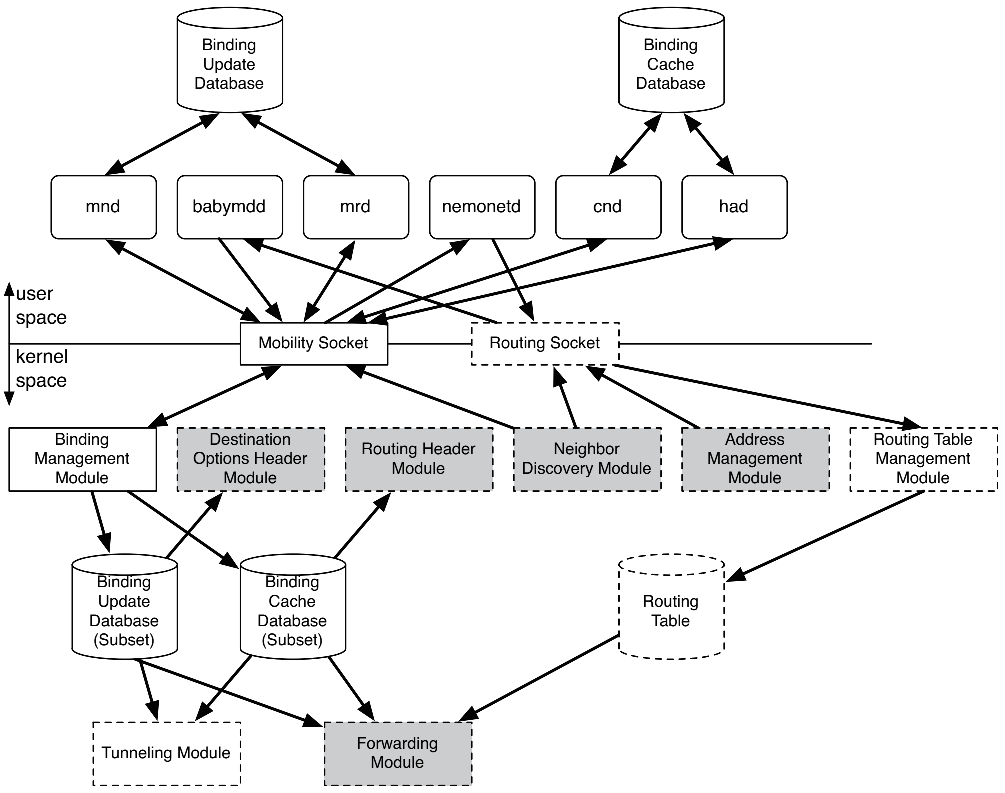

Binding Update Database
Binding Cache Database
mnd
babymdd
mrd
nemonetd
cnd
had
user space
kernel space
Mobility Socket
Routing Socket
Binding Management Module
Destination Options Header Module
Routing Header Module
Neighbor Discovery Module
Address Management Module
Routing Table Management Module
Binding Update Database (Subset)
Binding Cache Database (Subset)
Routing Table
Tunneling Module
Forwarding Module

Figure 3: The relationship of the SHISA modules.

mobility related information even between user space programs. For example, the movement detection program (**babymdd**) uses this socket to notify other programs of movement events through the socket when it considers the node is attached to a new network. The socket is also used as a broadcasting channel from the kernel to user space programs. For example, when the kernel of a mobile host receives a tunneled packet from a correspondent node, it notifies the **mnd** program of the fact so that it can start the route optimization procedure. Such information is not usually available from the user space.

We created two new pseudo interfaces, `mip` and `mtun`. The `mip` interface represents the home network of an MN/MR and keeps HoAs of the node. If an MN/MR has more than one home network, the node will have multiple `mip` interfaces. The `mtun` interface is used as a tunnel interface between an MR and its HA. The interface is basically a copy of the `gif` interface with some extension to keep the next-hop information of the interface. The `mtun` interfaces are controlled by the **nemonetd** program based on the signaling messages exchanged between an MR and an HA by monitoring the Mobility socket. The `mip` and `mtun` interfaces are discussed in Section 4.2 and 4.3 respectively.

Mobile IPv6 and NEMO BS extended IPv6 extension headers. These protocols use a new destination option (HAO) and a new routing header type (RTHDR2). The processing code is implemented by extending the existing extension header processing code in the kernel, because these headers cannot be handled in user space. The normal packets, that are not mobility signal messages, are automatically processed based on the binding information stored in the kernel by the extended processing code. The signaling packets are sent and received by the user space programs using the socket API specified in RFC4584 [11].

4.1 Mobility Socket: AF_MOBILITY

The Mobility socket [8] is implemented as a variant of the raw sockets. The usage of this socket is similar to that of the Routing socket. The mobility socket can be opened as follows.

```
s = socket(AF_MOBILITY, SOCK_RAW, 0);
```

At the this moment, there are 12 message types as shown in Table 1.

Type	Description
NODETYPE_INFO	Set or reset the operation mode (MN, MR, HA or CN).
BC_ADD	Add a binding cache entry.
BC_REMOVE	Remove a binding cache entry.
BC_FLUSH	Remove all binding cache entries.
BUL_ADD	Add a binding update list entry.
BUL_REMOVE	Remove a binding update list entry.
BUL_FLUSH	Remove all binding update list entries.
MD_INFO	A hint message that indicates the movement of an MN.
HOME_HINT	A hint message from the kernel that notifies returning home of an MN from the kernel.
RR_HINT	A hint message from the kernel that indicates receiving or sending a bi-directional packet.
BE_HINT	A hint message from the kernel that an error message has to be sent due to protocol processing error in the kernel.
DAD	Request the kernel to perform the DAD (Duplicate Address Detection) procedure for a specific address.

Table 1: The Mobility socket message types.

The NODETYPE_INFO message enables (or disables) mobility functions in the kernel. The user space programs issue this message to enable (or disable) specific mobility processing code in the kernel, for example, the **mnd** program issues this message to enable mobile node functions in the kernel such as the binding update list management and the extension header processing. The BC_* messages are used by the **had** and **cnd** programs to add or remove binding cache entries in the kernel. The BUL_* messages are used for binding update list entries by the **mnd** and **mrd** programs similarly. The MD_INFO message is issued by the **babymdd** program to notify the node movement of the **mnd** or **mrd** program. As discussed earlier, any system integrator can prepare their specific movement detection program that issues the MD_INFO message for better or optimized performance of the node movement. The *_HINT messages are issued by the kernel to notify the events that cannot be obtained in user space of user space programs. The HOME_HINT message is issued when an MN/MR returns home by comparing received prefix information in a Router Advertisement message and the configured home network prefix. When receiving this message, the MN/MR stops mobility functions.

Flag	Description
IN6_IFF_HOME	The address is an HoA.
IN6_IFF_DEREGISTERING	The address is being de-registered.

Table 2: The address flags used by an HoA.

The address information in these messages are stored in the form of the sockaddr structure so that any kind of address family can utilize this socket mechanism. IPv6 is the only supported address family at this moment.

4.2 The mip Interface and Home Address

The mip interface represents the home network of an MN/MR. This interface is used to keep the HoAs of an MN/MR when the node is in foreign networks. The HoAs are assigned to the physical network interface attached to the home network of the MN/MR while it is at home. However the physical interface is used to attach to a foreign network when the node leaves from the home network. In this case, the HoAs are moved from the physical interface to the mip interface.

The address assigned as an HoA has special flags as shown in Table 2. All HoAs have the IFF_HOME flag. The IFF_HOME flag is used by the source address selection procedure to prefer an HoA as a source address. The IFF_DEREGISTERING flag is used in the returning home procedure. The IFF_DEREGISTERING flag is added while an MN/MR is performing de-registration procedure of its HoA when it returns to home. Until the procedure has successfully completed, the HoA is not valid and is not used for communication.

4.3 The mtun Interface

The mtun interface is used when the NEMO BS function is used. This interface is used by an MR and an HA to create an unnumbered tunnel between them. The physical endpoint address of the tunnel is the CoA of the MR and the HA's address. On the HA, the traffic addressed to the mobile network of the MR is sent to the mtun interface established between it and the MR that manages the mobile network. On the MR, the mtun interface is set as the default route of the outgoing packets. All packets generated by the MR or the nodes in the mobile network of the MR will be tunneled to the HA.

Since the mtun interface is used as the default route on the MR, the loop condition occurs if we do not specify the next hop router when sending tunneled packets. Figure 4 shows the situation. When the MR

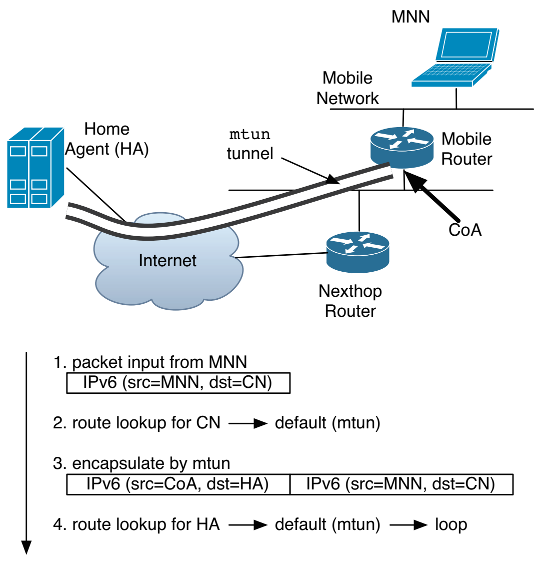


Figure 4: The loop of a tunneled packet.

sends a packet to the default route that is the `mtun` tunnel, the MR creates an encapsulated packet whose outer source is its CoA and the outer destination is the HA. The output function of the encapsulated packet will try to send it based on the routing table and it will try to send it to the default route again.

To avoid this problem, the `mtun` interface keeps the next hop router's address in its interface structure. The information is retrieved from the default router list managed by the kernel. The **nemonetd** program checks the default router list and picks up one of them that are attached to the same network as the CoA of the MR. The next hop information is stored by the **nemonetd** program using I/O control message of the `mtun` interface. When sending an encapsulated packet, the output function of the `mtun` interface will add the next hop information as an IPv6 packet option.

As we have mentioned already, the `mtun` interface is originally copied from the `gif` interface. The only difference is the next hop information storing mechanism and output mechanism using the information.

4.4 Sending and Receiving Signaling Messages

All the signaling messages are processed by the user space programs. The signaling messages are carried by the Mobility Header which is introduced by the Mobile IPv6 specification. Although the header is defined as one of the IPv6 extension headers, it is treated as a final header at this moment. Therefore, there is no following upper layer or other extension headers after a Mobility Header. To support this header, we implemented a simple input validation routine in the kernel and used the raw IPv6 packet delivery mechanism. The Mobility Header processing function is added using the protocol switch mechanism. When a packet whose last header is a Mobility Header (protocol number 135) is input, then the `mip6_input()` function is called.

As shown in the following code fragment, the `mip6_input()` function performs the validation check of the input packet and calls the raw IPv6 input function (`rip6_input()`) to deliver the packet to applications. The application with the Mobility socket will receive all Mobility Header messages.

```
int
mip6_input(mp, offp, proto)
    struct mbuf **mp;
    int *offp, proto;
{

    validation of the input packet.

    /* deliver the packet using Raw IPv6
       interface. */
    return (rip6_input(mp, offp ,proto));
}
```

When sending a Mobility Header packet, the same output function as that of the raw IPv6 socket (`rip6_output()`) is used.

4.5 Extension Header Processing

The Mobile IPv6 specification defines a new destination option, the HAO option, to carry the HoA of an MN to an HA or a CN, and the RTHDR2 to deliver packets to an MN directly from an HA or a CN. We simply extended the existing code to support these new messages, since both Destination Options Header and Routing Header processing code had been already implemented as a basic IPv6 feature in NetBSD.

The input processing code of the HAO option is implemented in the `dest6_input()` function. The function checks an HAO option and related binding cache entry of the HoA included in the HAO option. If the cache entry exists, the source address of the input packet and the HoA are swapped. The transport layer and above layer will process the HoA as the source of the packet. Note that this operation is a kind of source spoofing operation and we need to verify the operation is safe. The existence of the binding cache entry is used for the validation.

The exception of the swapping is a BU message. A BU message has an HAO option to request a peer node to create a binding cache entry that binds the

HoA in the HAO option and the CoA stored in the source address field of the IPv6 header. When a node receives a BU message first time, there is no binding cache entry, and we cannot rely on the cache existence to validate the message. In the Mobile IPv6 specification, it is specified that a BU message is protected by some cryptographic mechanisms. When an MN sends a BU message to its HA, the message is protected by the IPsec mechanism. When an MN sends a BU message to a CN, the message is protected by the secret created through the RR procedure. If a bogus MN tries to send a BU message to a victim HA, then the message will be dropped during the IPsec header processing. If a bogus MN tries to send a BU message to a CN, then the message will be delivered to the **cnd** program because it does not have any IPsec headers. The **cnd** program checks if the message is protected by the secret created by the RR procedure, and drop it if it is not protected. Once the message is accepted, the **had** or **cnd** program creates a new binding cache entry for the message. The following packets with an HAO option will be accepted.

The input processing of the RTHDR2 is implemented in the `route6_input()` function. The function calls the `rthdr2_input()` function when the type number of the input routing header is 2. The RTHDR2 includes the HoA of an MN. The basic procedure is same as that of the Type 1 Routing Header (RTHDR1). The destination address of the input IPv6 header and the address in the Routing Header are swapped. Unlike the RTHDR1, the RTHDR2 only include one address and the address must be an HoA. The `rthdr2_input()` validates the RTHDR2 and swaps the addresses if it is valid. Since the original IPv6 destination address (which is the CoA of an MN) and the address in the RTHDR2 (the HoA of the MN) both belong to the same MN, a peer node can send a packet directly to the MN without using the tunnel established between the MN and its HA.

The output processing of these headers is handled by the `ip6_output()` function. Since the mobility functions are transparent to all the applications, the packet passed to the `ip6_output()` function does not have any mobility related data, except signaling packets that are handled in the user space programs and have extension headers specified by the user space programs. The `ip6_output()` function checks binding update list entries and binding cache entries at the beginning of the packet processing, and inserts a HAO and/or a RTHDR2 if there is a binding entry related to the addresses of the outgoing IPv6 packet. For example, if the packet's source address is the HoA of an MN and the MN has a valid binding update list entry of the HoA, then a HAO option, that includes the CoA of the MN stored in the binding update list entry, is created and inserted to the outgoing packet. Similarly, a RTHDR2 is also inserted if there is a valid binding cache entry that is related to the destination address of the outgoing packet.

4.6 Tunneling

When an MN/MR sends packets to CNs, or when an MR forwards packets from its mobile network to the nodes outside, the nodes encapsulate packets to its HA using the tunnel established between them. The same operation is performed in the reverse direction. In the SHISA stack, we use two different encapsulating mechanisms for tunneling. One is the mechanism for packets sent/delivered to a moving node itself, the other is for packets sent/delivered to the nodes in a mobile network.

In fact, these two tunnels have the same function. The reason why we have two different tunnels is that the stack has been build step-by-step based on the previous KAME Mobile IPv6 design. In KAME Mobile IPv6 that did not support NEMO BS, the tunneling was implemented as a part of packet processing in the kernel. SHISA re-used the design as a Mobile IPv6 tunneling mechanism. When we started implementing NEMO BS in SHISA, we chose to use a specific tunnel interface (the `mtun` interface) as a tunneling mechanism for mobile network nodes, so that the `nemonetd` programs can easily control the tunnel endpoints. We do not think the current design is the best and keep discussing to revise the design. Section 5 mentions this topic further.

Figure 5 shows the output flow of tunneling packets on an MN/MR. When an MN sends a tunneled packet, the `mip6_tunnel_output()` function is used. For the forwarding packets from the mobile network of an MR, the `mtun_output()` which is the output function of the `mtun` interface is used instead.

Figure 6 shows the input flow of tunneling packets on an MN/MR. Similar to the output case, the tunneled packets sent to the moving node itself is processed by the special input function `mip6_tunnel_input()`. For the forwarding packets to the mobile network nodes, the `mtun_input()` function handles tunneled packets.

4.7 Intercepting Packets

Thanks to the backward compatibility of Mobile IPv6, all IPv6 nodes can communicate with an MN/MR or nodes inside the mobile network of the MR. In this case, all packets sent to the moving entities are routed to the home network of them. The HA of these mov-

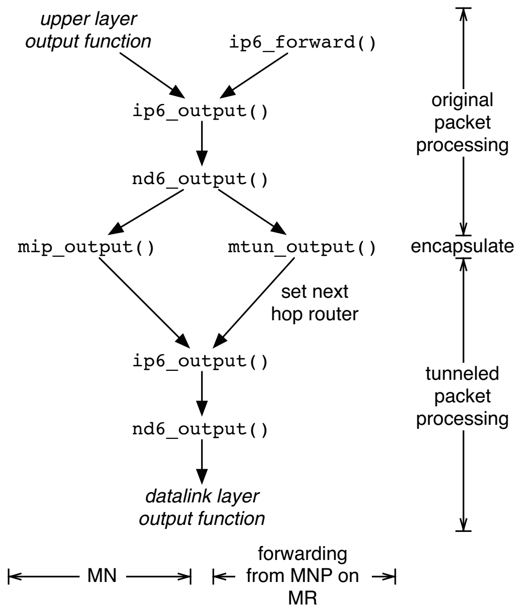

upper layer output function
ip6_forward()
ip6_output()
nd6_output()
mip_output()
mtun_output()
set next hop router
ip6_output()
nd6_output()
datalink layer output function
original packet processing
encapsulate
tunneled packet processing
MN
forwarding from MNP on MR

Figure 5: The output flow of a tunneled packet on an MN/MR.

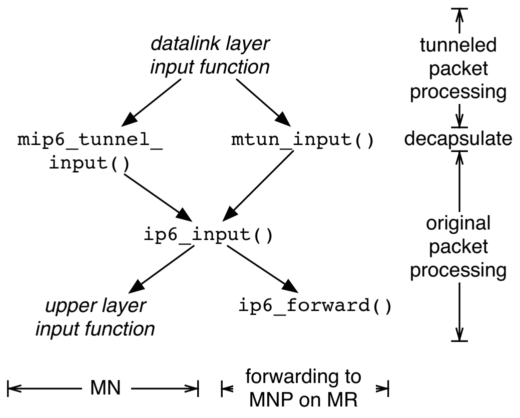

datalink layer input function
mip6_tunnel_ input()
mtun_input()
ip6_input()
upper layer input function
ip6_forward()
tunneled packet processing
decapsulate
original packet processing
MN
forwarding to MNP on MR

Figure 6: The input flow of a tunneled packet on an MN/MR.

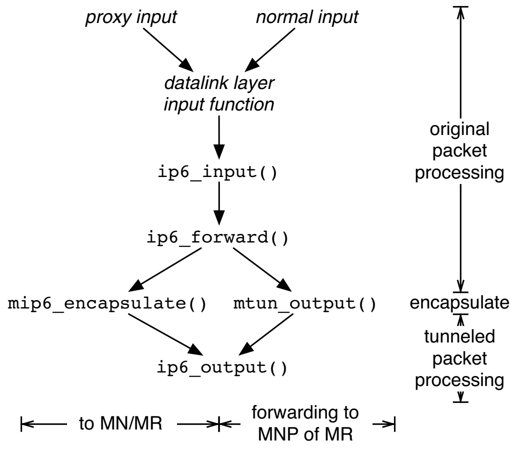


Figure 7: The output flow of a tunneled packet on an HA.

ing entities have to intercept the packets and forward them properly.

When an HA intercepts packets sent to the HoA of an MN or MR, the HA uses proxy Neighbor Discovery mechanism. The proxy is started after the HA receives a valid BU message for registration from the MN/MR, and is stopped when it receives a de-registration BU message. The intercepted packets are forwarded using the tunneling mechanism. In contrast to the packets sent to HoAs, the packets sent to the mobile network of the MR are processed by the normal forwarding mechanism. The HA has a routing entry for the MNP whose outgoing interface is set to the tunnel interface. Figure 7 shows the flow.

The input processing of the tunnel packets at an HA is a simple forwarding processing. The only difference is that the packets originated by an MN/MR itself are input by the special input function `mip6_tunnel_input()`. Figure 8 shows the flow. The `mip6_tunnel_input()` function is defined as a part of the protocol switch structure for Mobile IPv6 as shown in Figure 9. The protocol switch structure is used internally in the kernel when the Mobile IPv6 function is enabled.

4.8 Movement Detection

Movement detection is also performed in user space in the SHISA stack. At this moment, we are providing a simple detection program `babymdd` as a sample code for developers of more enhanced detection program. The `babymdd` programs detects the node movement based on the validity of the CoA currently assigned. In the BSD Operating Systems, all IPv6 addresses have a special flag called *DETACHED* that is proposed in [12]. The flag means that the address is valid but the router that advertised the prefix of the address is unreachable. This implies that an MN/MR once received prefix information and formed an address from the prefix, but left the network.

The `babymdd` program sends a Router Solicitation message when the status of the network interfaces used to connect the node to the Internet is changed from ‘down’ to ‘up’. If the node leaves and attaches to a new network, then the old routers will become unreachable by the Neighbor Unreachability Detection

```
struct ip6protosw mip6_tunnel_protosw =
{ SOCK_RAW,     &inet6domain,    IPPROTO_IPV6,    PR_ATOMIC|PR_ADDR,
  mip6_tunnel_input, rip6_output, 0,              rip6_ctloutput,
  rip6_usrreq,
  0,            0,               0,               0,
};
```

Figure 9: The tunneled packet protocol switch entry.

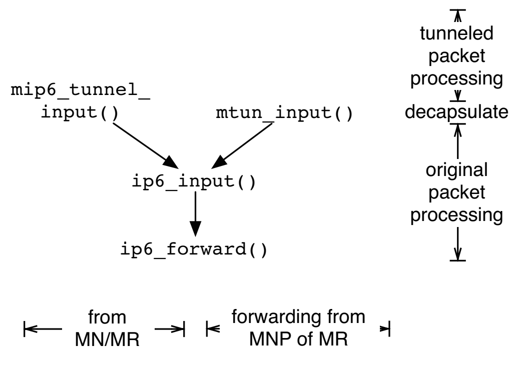


Figure 8: The input flow of a tunneled packet on an HA.

(NUD) mechanism. As a result the corresponding addresses formed from the prefix advertised by these old routers will become detached. If the CoA is one of these detached addresses, the babymdd program will search other appropriate address and inform the mnd or mrd program of the new CoA by the MD_INFO Mobility socket message.

5 Discussion

SHISA provides full functional Mobile IPv6 and NEMO BS implementation. We have conformed its interoperability with other implementations through a couple of interoperability test events. However we still need to develop the SHISA stack in order to support extensions of Mobile IPv6/NEMO BS, which are currently discussed in the IETF, and to support more operating platforms. In this section, we explain some of these remaining stuffs.

5.1 Multiple Tunnel Mechanisms

As discussed in Section 4.6, the SHISA stack is currently providing two different tunneling mechanisms related to mobility function for the same purpose, one for Mobile IPv6 and the other for NEMO BS. When a node is acting as an MR, it can also work as an MN. However the packet tunneled to its HA goes to

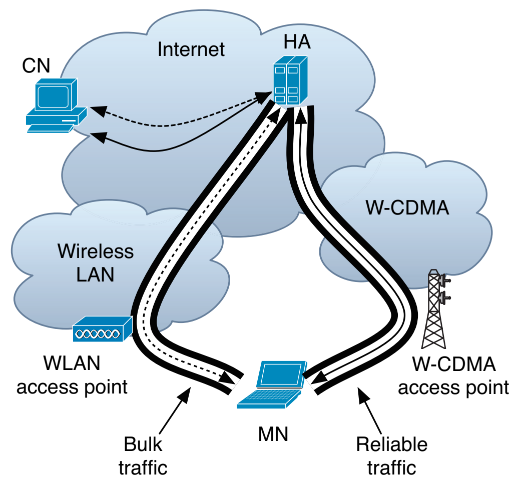


Figure 10: The usage scenario of multiple network interfaces simultaneously.

the mtun interface when it is an MR, and goes to the internal in-kernel tunnel function if it is an MN. This causes not only the code duplication problem but also cases functional restrictions.

The IETF MONAMI6 WG is standardizing the mechanism to utilize multiple network interfaces at the same time on Mobile IPv6 and NEMO BS. For example, if a mobile device has a wireless LAN interface and a W-CDMA interface, it might want to utilize both of them according to the local traffic policy. The mobile device can use the wireless LAN interface as a cheap bulk data transfer interface and use the W-CDMA as a reliable interface (Figure 10). Recently, as many mobile devices often have multiple communication interfaces, utilizing them simultaneously is urgent matter for Mobile IP and NEMO. The Multiple Care-of Addresses Registration (MCoA) mechanism [13] proposed at the MONAMI6 WG provides a method to register more than one CoA at the same time.

The current SHISA implementation supports MCoA for NEMO BS as described below. The tunnel mechanism of NEMO BS is implemented as the mtun interface as discussed in Section 4.3. The current design of the MCoA mechanism in SHISA is to define the same number of `mtun` interfaces as the number of physical network interfaces, and to bind each physical interface to a `mtun` interface. The default route of an MR is set to one of the `mtun` interfaces (for example, `mtun0`) and packet flow is distributed using a packet filter mechanism, such as IP Filter [14] or PF [15]. If a wireless LAN interface `wi0` is bound to the `mtun0` interface, and a W-CDMA interface `ppp0` is bound to the `mtun1` interface, then we may use the rules described in Figure 11 to distribute traffic. With these rules, all traffic except the SSH traffic is sent to the `mtun0` interface which is bound to the wireless LAN interface. This mechanism cannot be used with the Mobile IPv6 case of the SHISA implementation, because the `mtun` interface is not used in Mobile IPv6.

We once tried to solve this problem using the PF mechanism for Mobile IPv6 too. In the trial, we stopped using the special in-kernel tunnel mechanism and passed all the Mobile IPv6 traffic to the `mtun` interface. It worked with one network interface, however we noticed that we would have a problem when we use multiple network interfaces and the RO communication.

The stack has multiple binding update list or cache entries when the node registers multiple CoAs to its HA. When the RO is used, the source address of the packet (which is one of the CoAs of the MN) must be decided based on the local flow distribution policy. That means, we need two policy judgement points for the essentially same traffic, one in the CoA selection part, and the other in the packet filtering part.

We are now designing a new tunnel mechanism for mobility functions. In the idea, the moving node always outputs packets to a special tunnel interface bound to the home network of the node (similar to the `mip` interface). In the output function, the local traffic distribution policies are applied to the packets and they are redirected to the HA with an encapsulating header with a proper CoA of the node based on the policy. With this procedure, we can put both the CoA selection task and policy application task in the same place that will solve the problem described above. We will verify if this is feasible to implement.

5.2 IPsec Policy Management

As specified in RFC, some of the signaling messages between an MN and an HA must be protected by the IPsec mechanism. In these messages, the HoT and HoTI messages cause IPsec configuration problem when a node returns to home. These messages must be protected by the ESP tunnel mechanism while the node is in a foreign network. In the current implementation, the node has static IPsec tunnel policy entries for these messages. Figure 12 is a sample policy definition for these packets. 135 is the protocol number of the Mobility Header and 1 and 3 represents the HoTI and HoT message types respectively.

```
spdadd HoA ::/0 135 1,0 -P out ipsec
  esp/tunnel/HoA-HA/;
spdadd ::/0 HoA 135 3,0 -P in ipsec
  esp/tunnel/HA-HoA/;
```

Figure 12: IPsec policy entries to protect the HoTI and HoT messages

These tunnels are used only the node is in a foreign network and must not be used at home. This restriction cases a problem. An MN has to de-register its binding information registered in CNs when the MN returns to home. To de-register binding information, the MN needs to perform the RR procedure that requires HoTI/HoT message exchange. The HoTI message sent from the node will match the IPsec policy statically installed on the MN and may be dropped at the tunnel end point (the HA). The HoT messages will come from CNs directly to the MN, because the MN has already returned to home and the HA is not proxying its address anymore. The IPsec policy will discards the incoming HoT message because it is not protected by the IPsec mechanism as required in the policy entry.

To solve this problem, the mobility stack must inactivate all the policy related to the HoTI/HoT messages installed in the kernel. Currently, there is no standard way to inactivate the policy entries, other than removing them. Removing policy entries may work if the node uses IPsec only for Mobile IPv6. However if other communication frameworks are also using IPsec policy database, then removing and adding policy entries may influence the policy matching order, that may result in unexpected IPsec processing. We are considering a new policy management message to activate/inactivate a specific policy entry and planning to implement and test the mechanism.

5.3 Standard Mobility Interface

We have moved all the signal processing code to user space. That means, if the kernel supports packet forwarding mechanisms for Mobile IPv6 and NEMO BS, then we can use the same signal management program on different kernels. We defined a generic mobility information exchange mechanism as the Mobility Socket for this purpose. The messages used in the socket is basically platform independent. Thus, if we can cleanly separate kernel functions and user space

```
pass out route-to mtun0 inet6 from MNP::/64 to any
pass out route-to mtun1 inet6 from MNP::/64 to any port 22
```

Figure 11: The filter rules to distribute mobile network traffic to multiple NEMO BS tunnels

functions, we can develop the kernel and the signal processing program independently. SHISA now runs only on BSD operating systems that support the Mobility Socket and in-kernel mobility functions, however it can run on other operating systems if they provide the Mobility Socket interface and equivalent functions in their kernel.

One obvious missing feature of the current Mobility Socket implementation is a message filtering mechanism. Currently, all the messages sent by mobility entities are delivered to all the listening sockets regardless of its necessity. However, some Mobility Socket messages are meaningless to some of the mobility entities. For example, the HA module may not want to receive any messages related to MN/MR functions. Suppressing unnecessary messages will alleviate the exhaustion of the socket buffer when there are many messages.

We once submitted the basic specification (not including the filtering mechanism) of the Mobility Socket at the IETF, but more than one year has passed since the draft expired. We may need to resume the standardization work when we have gotten clear understanding of the roles in the kernel and user space though the development.

5.4 IKE Interaction

At this moment, the SHISA stack works with IPsec security associations (SAs) manually configured. The manual operation works only with a small number of nodes and it is not scalable. The essential solution is the Internet Key Exchange (IKE) protocol that provides a dynamic SA generation mechanism. IKE creates a pair of SA between two nodes, however the constructed SA is based on the addresses used by the IKE procedure. This causes a problem in mobility environment. In the Mobile IPv6 (and NEMO BS) case, the communication is originated from the HoA of the MN/MR. Because the HoA cannot be used for communication, the MN/MR cannot start the IKE procedure using their HoA.

The solution is to use CoAs for IKE communication and creates SAs for HoA during the IKE negotiation[3]. How to use IKE with Mobile IPv6 is further described in [16, 17]. Unfortunately, most of the current IKE programs are not aware of Mobile IPv6. To provide the dynamic keying feature to our stack and encourage mobility technology deployment, we are now working with the Racoon2 project [18] that is developing an open source IKE implementation. The interaction mechanism between a mobility stack and an IKE program is proposed in [19]. The proposal defines an optional data structure to provide CoA and HoA information to an IKE program of an MN, when it needs to start the SA negotiation process. We are planning to join the discussion of the standardization process of the proposal and to provide the implementation.

[3]The code contributed by Francis Dupont exists waiting to be merged to the SHISA code.

5.5 Porting to Other Platforms

The KAME version of the SHISA stack supported both NetBSD 2.0 and FreeBSD 5.4. Unfortunately we could not support OpenBSD mainly because lack of developers using OpenBSD in our team, but it could be supported potentially. As explained, we are now focusing on NetBSD-current. Once we have completed the porting work to NetBSD-current, we will work on FreeBSD-current. The work will be harder than the NetBSD work, since the difference of the kernel code between FreeBSD 5.4 and FreeBSD-current is bigger than that of NetBSD. In addition, recent FreeBSD introduced the fine-grained locking mechanism for better performance in a multi-processor environment. The IPv6 code and mobility related code in the kernel does not have support for the fine-grained locking mechanism. The adaptation will need some additional development and more test to stabilize. The OpenBSD port is planned after the FreeBSD port.

It is possible to port SHISA to other platforms than BSDs if they support kernel modifications as described in Section 5.3. In fact, there is a port to the Darwin operating system as announced in the Darwin IPv6 developers mailing list.

6 Conclusion

We developed the Mobile IPv6 and NEMO BS protocol stack on the KAME platform. We are now porting it to the latest BSD distributions. We have started to port it to NetBSD as the first step and will try to work on other platforms based on the progress of the current work. The stack provides most of the specified features. It is confirmed interoperable with many other independently developed Mobile IPv6 and NEMO BS stacks, and it works stably. We are now focusing to refine the code, especially the kernel code, to make the quality high enough to be merged to the NetBSD main tree.

Although we have completed the implementation of the basic mobility functions, we still have many things to do to support advanced mobility features under standardization in the IETF. We continue to work on supporting these advanced functions to provide more useful mobility stack to various BSD operating systems.

Acknowledgement

The authors would like to thank the WIDE Project for the support on our development activity.

References

[1] David B. Johnson, Charles E. Perkins, and Jari Arkko. Mobility Support in IPv6. Technical Report RFC3775, IETF, June 2004.

[2] Vijay Devarapalli, Ryuji Wakikawa, Alexandru Petrescu, and Pascal Thubert. Network Mobility (NEMO) Basic Support Protocol. Technical Report RFC3963, IETF, January 2005.

[3] WIDE project. SHISA, February 2007. http://www.mobileip.jp/.

[4] Keiichi Shima, Ryuji Wakikawa, Koshiro Mitsuya, Tsuyoshi Momose, and Keisuke Uehara. SHISA: The IPv6 Mobility Framework for BSD Operating Systems. In *IPv6 Today – Technology and Deployment (IPv6TD'06)*. International Academy Research and Industry Association, IEEE Computer Society, August 2006.

[5] WIDE project. KAME Working Group, March 2006. http://www.kame.net/.

[6] Koshiro Mitsuya. ATLANTIS: NEMO Basic Support Implementation, January 2005. http://www.nautilus6.org/implementation/atlantis.html.

[7] Ryuji Wakikawa, Susumu Koshiba, Keisuke Uehara, and Jun Murai. Multiple Network Interfaces Support by Policy-Based Routing on Mobile IPv6. In *2002 International Conference on Wireless Networks (ICWN'02)*, July 2002.

[8] Tsuyoshi Momose, Keiichi Shima, and Anti Tuominen. The application interface to exchange mobility information with Mobility subsystem (Mobility Socket, AF_MOBILITY). Technical Report draft-momose-mip6-mipsock-00, IETF, June 2005.

[9] Tatuya Jinmei, Kazuhiko Yamamoto, Jun-ichiro Hagino, Shoichi Sakane, Hiroshi Esaki, and Jun Murai. The IPv6 Software Platform for BSD. *IEICE Transactions on Communications*, E86-B(2):464–471, February 2003.

[10] Keith Sklower. A Tree-based Packet Routing Table for Berkeley UNIX. In *Proceedings of the Winter 1991 USENIX Conference*, pages 93–103. USENIX Association, January 1991.

[11] Samita Chakrabarti and Erik Nordmark. Extension to Socket API for Mobile IPv6. Technical Report RFC4584, IETF, July 2006.

[12] Tatuya Jinmei, Jun-ichiro Ito, and Munechika Sumikawa. Efficient Use of IPv6 Auto-Configuration in a Mobile Environment. In *The 7th Research Reporting Session.* Information Processing Society of Japan, SIG Mobile Computing, December 1998.

[13] Ryuji Wakikawa, Thierry Ernst, and Kenichi Nagami. Multiple Care-of Addresses Registration. Technical Report draft-wakikawa-mobileip-multiplecoa-05, IETF, February 2006.

[14] Darren Reed. IP Filter. Web page, February 2007. http://coombs.anu.edu.au/~avalon/.

[15] Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *USENIX 2002 Annual Technical Conference*, pages 171–180, June 2002.

[16] Jari Arkko, Vijay Devarapalli, and Francis Dupont. Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. Technical Report RFC3776, IETF, June 2004.

[17] Vijay Devarapalli and Francis Dupont. Mobile IPv6 Operation with IKEv2 and the revised IPsec Architecture. Technical Report draft-ietf-mip6-ikev2-ipsec-07, IETF, October 2006.

[18] WIDE project. The Racoon2 Project, February 2007. http://www.racoon2.wide.ad.jp/.

[19] Shinta Sugimoto, Francis Dupont, and Masahide Nakamura. PF_KEY Extension as an Interface between Mobile IPv6 and IPsec/IKE. Technical Report draft-sugimoto-mip6-pfkey-migrate-03, IETF, September 2006.

Implementation and Evaluation of Dual Stack Mobile IPv6

Koshiro Mitsuya[1], Ryuji Wakikawa[1], and Jun Murai[1]

Keio University, Japan. {mitsuya,ryuji,jun}@sfc.wide.ad.jp

Abstract. Dual Stack Mobile IPv6 (DSMIPv6) is an extension of Mobile IPv6 to support IPv4 care-of address and to carry IPv4 traffic via bi-directional tunnels between mobile nodes and their home agents. Using DSMIPv6, mobile nodes only need the Mobile IPv6 protocol to manage mobility while moving within both the IPv4 and IPv6 Internet. This is an important feature for IPv6 mobility during its deployment phase because IPv6 access network is not widely deployed yet. This paper describes the DSMIPv6 implementation on BSD operating systems and presents results of the experiments using the implementation.

1 Introduction

Mobility support in IPv6 is important, as mobile computers are likely to account for a majority or at least a substantial fraction of the population of the Internet during the lifetime of IPv6. Hence, the IETF has standardized Mobile IPv6 [1] in 2004 and NEMO Basic Support [2] in 2005, to address host and network mobility.

Mobile IPv6 and NEMO allow mobile nodes (host and router) to move within the Internet while maintaining IP reachability and ongoing sessions, using a permanent IPv6 address for the host or a permanent IPv6 prefix inside the moving network. In these schemes, a mobile node keeps a home address and a mobile network prefix, which are permanent, and an IP address acquired from the network it is visiting, which is called care-of address. A special redirection server called home agent maintains the mappings between the home address and the care-of address, and between the home address and the mobile network prefix. The home agent intercepts packets on behalf of the mobile node and sends them to the mobile node's care-of address when the mobile node is away from its home network; thus the ongoing sessions can be kept alive.

Mobile IPv6 and NEMO are now in the deployment phase and there are two issues. First, it is acknowledged that mobile nodes will use IPv6 addresses only for their connections and will, for a long time, need IPv4 home addresses that can be used by upper layers, since IPv6 applications are not widely deployed. Second, as IPv6 wireless access networks are not widely deployed, it is also reasonable to assume that mobile nodes will move to a network that does not support IPv6 and therefore need the capability to support an IPv4 care-of address.

The Dual Stack Mobile IPv6 (DSMIPv6) specification [3] extends the Mobile IPv6 capabilities to allow mobile nodes to request their home agent, to forward IPv4/IPv6 packets addressed to their home addresses, to their IPv4/IPv6 care-of address(es). Using DSMIPv6, mobile nodes only need Mobile IPv6 or NEMO Basic Support to manage mobility while moving within the Internet; hence eliminating the need to run both IPv6 and IPv4 mobility management protocols simultaneously.

This paper describes the DSMIPv6 implementation on SHISA [4, 5], a Mobile IPv6 and NEMO implementation on BSD operating systems, and its evaluation. Considerations for the current specification are also discussed in this paper.

This paper is organized as follow. We give an overview of DSMIPv6 in Sec. 2 followed by an overview of SHISA in Sec. 3. We then present the design of our implementation in Sec. 4 and the implementation details in Sec. 5. We performed experiments by using our implementation and the results and considerations are reported in Sec. 6. This paper concludes in Sec. 7.

2 Dual Stack Mobile IPv6

This section presents an overview of the Dual Stack Mobile IPv6 (DSMIPv6).

2.1 Overview

A node supporting both IPv4 and IPv6 is called a dual stack node. It is important to develop dual stack nodes under IPv6 deployment phase because we cannot rapidly make the transition to IPv6. In fact, many applications are still using IPv4 and most of the access networks support only IPv4. The situation is the same for Mobile IPv6 (MIPv6) deployment. Mobile node will visit IPv4 only access networks and will use IPv4 only applications in the deployment phase of MIPv6.

In order to use MIPv6 by dual stack nodes, mobile nodes need to manage an IPv4 and IPv6 home or care-of address simultaneously and update their home agents' bindings accordingly. This concept is shown in Fig. 1. On the figure, MN is a mobile node, HA is a home agent, and CN is a correspondent node. MN1 is connected to an IPv6 network and MN2 is connected to an IPv4 network.

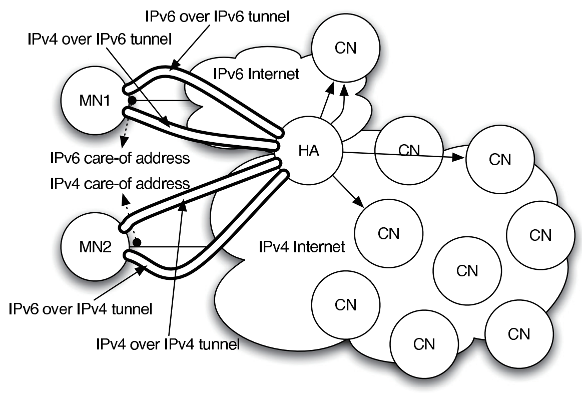


Fig. 1. The concept of DSMIPv6

A MN has both IPv4 and IPv6 home addresses. HA is a dual stack node connected to both IPv4 and IPv6 Internet. As MN1 is visiting IPv6 network, MN1 configures a global unique IPv6 address as its care-of address. MN1 registers the care-of address to HA, and both IPv4 and IPv6 home addresses bound to the address. IPv4 traffic goes through IPv4 over IPv6 tunnel between MN1 and HA, and IPv6 traffic goes through IPv6 over IPv6 tunnel. In a similar way, MN2 registers IPv4 care-of address. Traffic goes through IPv6 over IPv4 tunnel or IPv4 over IPv4 tunnel. By this way, mobile nodes need only MIPv6 to manage mobility while moving within both IPv4 and IPv6 Internet.

We give details of DSMIPv6 operation in the rest of this section. However, Home Agent Address Discovery feature and Network Address Translator (NAT) Traversal feature are not mentioned because they are not implemented yet.

2.2 Binding Management

A mobile node needs to update the bindings on its home agent with its current care-of address. If the mobile node is a dual stack node and has an IPv4 and IPv6 home address, it creates a binding cache entry for both addresses. The format of the IP packet carrying the binding update and acknowledgment messages varies depending on the visited network. IPv6 network is always preferred than IPv4 network, so there are three different scenarios to consider with respect to the visited network:

1. The mobile node configures a global unique IPv6 as its care-of address.
2. The mobile node only configures a global unique IPv4 address as its care-of address.
3. The mobile node only configure a private IPv4 address as its care-of address.

The operation for case 1 is explained in Sec. 2.3 and the operation for case 2 is explained in Sec. 2.4. Case 3 is not explained in this paper as mentioned in Sec. 2.1.

2.3 Visiting IPv6 Global Foreign Network

In this case, the mobile node configures a global unique IPv6 address as its care-of address (V6CoA). The mobile node sends a binding update message (BU) to the IPv6 address of its home agent (V6HA), as defined in [1]. The binding update message includes the IPv4 home address option, which is defined in [3]. The packet format is shown in Fig. 2. The packet format of the normal MIPv6 binding update message is also shown in the figure for comparison.

```
MIPv6 BU:
  IPv6 header (src=V6CoA, dst=V6HA)
    Destination option
      HoA (IPv6 home address)
    Mobility header
      BU

DSMIPv6 BU:
  IPv6 header (src=V6CoA, dst=V6HA)
    Destination option
      HoA (IPv6 home address)
    Mobility header
      BU [IPv4 home addres]
```

Fig. 2. Binding update message formats

```
MIPv6 BA:
  IPv6 header (src=V6HA, dst=V6CoA)
    Routing header type 2
      HoA (IPv6 home address)
    Mobility header
      BA

DSMIPv6 BA:
  IPv6 header (src=V6HA, dst=V6CoA)
    Routing header type 2
      HoA (IPv6 home address)
    Mobility header
      BA [IPv4 addr. ack.]
```

Fig. 3. Binding Acknowledgement message formats

After receiving the binding update message, the home agent creates two binding cache entries, one for the mobile node's IPv4 home address and one for the mobile node's IPv6 home address. Both entries will point to the mobile node's IPv6 care-of address. Hence, whenever a packet is addressed to the mobile node's IPv4 or IPv6 home addresses, it will be forwarded via the bi-directional tunnel to the mobile node's IPv6 care-of address. Effectively, the mobile node establishes two different tunnels, one for its IPv4 traffic (IPv4 over IPv6) and one for its IPv6 traffic (IPv6 over IPv6) with a single binding update message.

After binding cache entries are created, the home agent sends a binding acknowledgment message (BA) to the mobile node as defined in [1]. If the binding update message included an IPv4 home address option, the binding acknowledgment message includes the IPv4 address acknowledgment option. The packet format is shown in Fig. 3. This option informs the mobile node whether the binding was accepted for the IPv4 home address. The packet format of the normal MIPv6 binding acknowledgment message is also shown in the figure for comparison.

2.4 Visiting IPv4 Only Global Foreign Network

In this scenario, the mobile node needs to tunnel IPv6 packets containing a binding update message to the home agent's IPv4 address (V4HA). The mobile node uses the IPv4 care-of address (V4CoA) as a source address in the outer header.

The binding update message contains the mobile node's IPv6 home address in the home address option. However, since the care-of address in this scenario is the mobile node's IPv4 address, the mobile node must include its IPv4 care-of address in the IPv6 packet. The IPv4 address is represented in the IPv4-mapped IPv6 address format (V4MAPPED) and is included in the source address field of the IPv6 header. If the mobile node has an IPv4 home address, it also includes the IPv4 home address option. The packet format is as shown in Fig. 4.

```
DSMIPv6 BU (IPv4):
  IPv4 header (src=V4CoA, dst=V4HA)
    UDP header
      IPv6 header (src=V4MAPPED, dst=V6HA)
        Destination option
          HoA (IPv6 home address)
        Mobility header
          BU [IPv4 home address]
```

Fig. 4. Binding update message format in IPv4 network

After accepting the binding update message, the home agent will update the related binding cache entry or create a new binding cache entry if such entry does not exist. If an IPv4 home address option was included, the home agent will update the binding cache entry for the IPv4 address or create a new entry for the IPv4 address. Both binding cache entries point to the mobile node's IPv4 care-of address.

All packets addressed to the mobile node's home address(es) (IPv4 or IPv6) will be encapsulated in an IPv4 header that includes the home agent's IPv4 address in the source address field and the mobile node's IPv4 care-of address in the destination address field.

After creating the corresponding binding cache entries, the home agent sends a binding acknowledgment message. If the binding update message included an IPv4 home address option, the binding acknowledgment message includes the IPv4 address acknowledgment option as shown in Fig. 5. The binding update message is encapsulated in an IPv4 payload whose destination is the IPv4 care-of address (represented as an IPv4-mapped IPv6 address in the binding update message).

```
DSMIPv6 BA (IPv4):
  IPv4 header (src=V4HA, dst=V4CoA)
    [UDP header] (if NAT is detected)
      IPv6 header (src=V6HA, dst=V4MAPPED)
        Routing header type 2
          HoA
        Mobility header
          BA ([IPv4 addr. ack.], NAT DET)
```

Fig. 5. Binding acknowledgement message format in IPv4 network

3 SHISA: Mobile IPv6 and NEMO Implementation on BSD operating systems

As our DSMIPv6 implementation is an extension of SHISA, we give an overview of the SHISA stack in this section.

SHISA [4, 5] is an open source Mobile IPv6 and NEMO Basic Support implementation on BSD operating systems. It has been developed by the KAME project [6].

The design feature of SHISA is to separate packet forwarding functions and signal processing functions into the kernel and user land programs. The operation of Mobile IPv6 and NEMO Basic Support is mainly IP packet routing (forwarding or tunneling). In order to obtain better performance, the packet routing has been implemented in the kernel space. The signal processing has been implemented in the user land space because it is easier to modify or update user land programs then the kernel. This is important because the signaling processing of Mobile IPv6 and NEMO Basic Support is complicated. This separation provides both good performance and efficiency when developing the stack.

In SHISA, a mobile node (host or router) consists of small kernel extensions for packet routing and several user land daemons (MND, MRD, BABYMDD, NEMONETD, HAD and CND). MND/MRD are daemons which manage bindings on a mobile node. BABYMDD is a daemon which detects the changes of care-of addresses and notifies MND/MRD of the changes. NEMONETD is a daemon which manages bi-directional tunnels. HAD is a daemon which manages bindings on a home agent. CND is a daemon which manages bindings on a correspondent node. Depended on the node type, one or several SHISA daemons run on a node.

Fig. 6 shows the relation of the SHISA modules. The objects with solid line are modules implemented for SHISA. The dotted line objects are modules existing in the original BSD system and the shaded ones are modified for SHISA.

SHISA programs communicate with the kernel and other programs using the Mobility Socket [7]. The Mobility Socket is a special socket to exchange the mobility related information between the kernel and user land programs.

The Neighbor Discovery module and the Address Management module notify the movement detection daemon (BABYMDD) of the changes of IP address through the Routing Socket.

The signaling messages such as binding updates and acknowledgments are exchanged between the SHISA programs on two nodes, for example, MRD on a mobile router and HAD on its home agent. The binding information retrieved from the signal exchange is stored in the user land space as the Binding Update database for mobile nodes and the Binding Cache database for home agents or correspondent nodes. Only subsets of the databases that are necessary for packet forwarding are installed into the kernel.

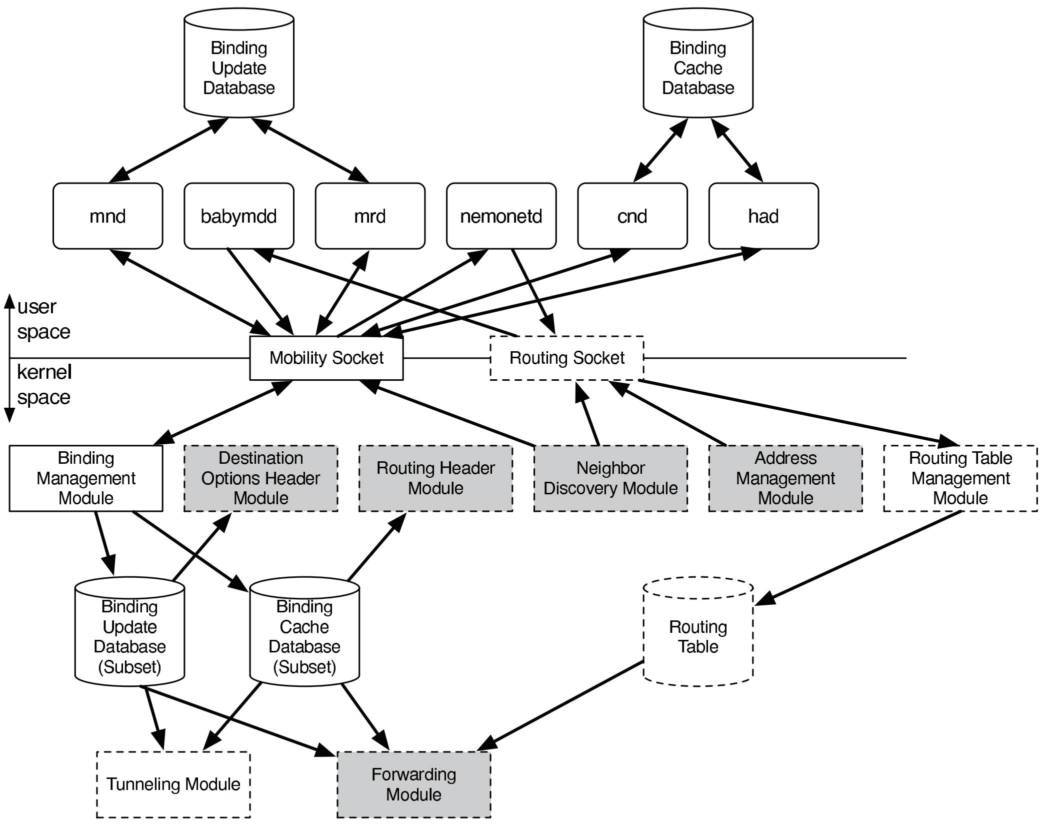

Binding Update Database
Binding Cache Database
mnd
babymdd
mrd
nemonetd
cnd
had
user space
kernel space
Mobility Socket
Routing Socket
Binding Management Module
Destination Options Header Module
Routing Header Module
Neighbor Discovery Module
Address Management Module
Routing Table Management Module
Binding Update Database (Subset)
Binding Cache Database (Subset)
Routing Table
Tunneling Module
Forwarding Module

Fig. 6. SHISA Overview

4 Design

4.1 Approaches

This section presents the design principle of our DSMIPv6 implementation. In accordance with the SHISA design principles, we defined the requirements of our DSMIPv6 implementation as follows:

1. Separation of signaling processing and packet routing:
   This is for good performance and efficiency in developing the stack, as explained in Sec. 3.
2. Minimum modification on the existing kernel:
   In order to integrate the implementation to the main branch of BSD operating systems, it is reasonable to minimize the modification on the kernel.
3. Minimum modification on the existing SHISA daemons:
   As the specification reuses the Mobile IPv6 functions for an IPv4 mobility management, it can be expected that many parts of the Mobile IPv6 implementation will be reused.

4.2 Functional Requirements

We defined functional requirements as listed below based on the approaches. The process flowchart of DSMIPv6 is shown in Fig. 7. The circles are functions in the user land space and the square boxes are functions in the kernel space. The requirements are corresponded to each function in the figure.

1. Binding management
   Support IPv4 care-of addresses and IPv4 home addresses. It should be done in the user land programs according to SHISA desgin.
2. Detecting IPv4 care-of addresses
   It should be done in the user land programs same as IPv6 care-of address detection.
3. Sending binding update messages
   Send a binding update message with the IPv4 home address option via IPv6 network, or a

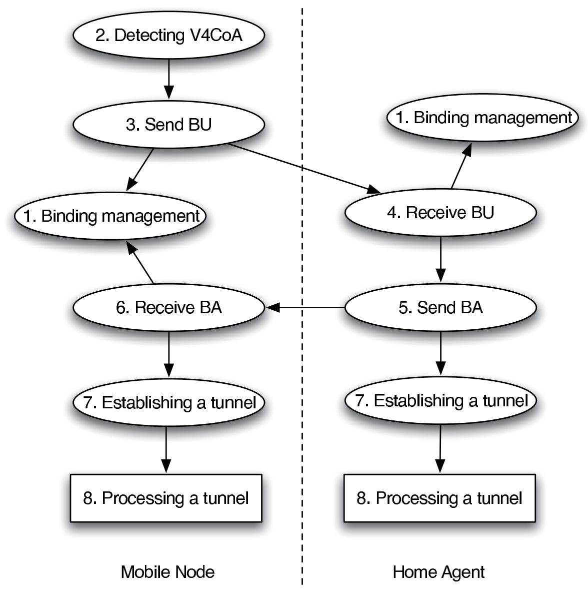


Fig. 7. DSMIPv6 process flowchart

binding update via IPv4 network. It should be done in the user land programs according to SHISA design.

4. Receiving binding update messages
It should be done in the user land programs according to SHISA design.
5. Receiving binding acknowledgment messages
Receive a binding acknowledgment message with the IPv4 home acknowledgment option via IPv6 network, or a binding acknowledgment message via IPv4 network. It should be done in the user land programs according to SHISA design.
6. Sending binding acknowledgment messages
It should be done in the user land programs in the same way as sending binding update messages.
7. Establishing a bi-directional tunnel
Establish a tunnel between an IPv4 care-of address and an IPv4 home agent address. The control should be done in the user land programs.
8. Processing bi-directional tunnel
the processing should be done in the kernel, according to SHISA design.

5 Implementation

The SHISA daemons are modified as explained in the following sections to support the DSMIPv6 signaling processing and tunnel setup. Note that the kernel is not modified for IPv4 tunnel processing because it is already existing in IPv6 stack. Since we do not introduce any new deamon for DSMIPv6, the overview of the DSMIPv6 implementation is the same as Fig. 6.

5.1 New Data Structures and Functions

The following data structures (Table 1) and functions (Table 2) are newly defined for the DSMIPv6 extensions.

Table 1. New Data Structures

Name	**Purpose**
IPv4 UDP socket	send/receive IPv4 UDP-encapsulated signaling
IPv4 Raw socket	send/receive IPv4 encapsulated signaling
IPv4 home address (struct in_addr)	added on the binding update list (struct binding_update_list)
IPv4 home address (struct in_addr)	added on the binding cache (struct binding_cache)
IPv4 home address mobility header option	(struct ip6_mh_opt_ipv4_hoa)
IPv4 home address mobility option	added on the Mobility Header option list (struct mip6_mobility_options)

5.2 Binding Management

The DSMIPv6 specification requires to create a binding cache entry and a binding update list entry for each IPv4 and IPv6 home addresses. However, we put an IPv4 home address entry on the binding update list entry and the binding cache entry of the related IPv6 home address, as shown in Table 1, to simplify the implementation. This is consistent with the specification because all parameters in the IPv4 home address binding expect the home address itself are the same as those of the IPv6 home address.

Table 2. New functions

Name	Purpose
nemo_tun4_set()	setup IPv4 tunnel
nemo_tun4_del()	delete IPv4 tunnel
upd4_input_common()	a common (used by both mobile nodes and home agents use) routine for incoming IPv4 UDP packet
udp4sock_open()	open an IPv4 UDP socket
raw4_input_common()	a common routine for incoming IPv4 Raw packet
raw4sock_open()	open an IPv4 Raw socket
v4_sendmessage()	send mobility signaling via IPv4
mnd_get_v4hoa_by _ifindex()	find IPv4 Home Address by an interface index on the MIP interface (where IPv6 home address is stored)
mip6_find_hal_v4()	find IPv4 Home Agent address

An IPv4 care-of address is stored in the binding update list entries or the binding cache entries using the IPv4-mapped IPv6 format. By this way, the same code are reused to maintain bindings. Whenever a care-of address is used, correspondent functions are called according to its address family.

5.3 Mobile Node Modifications

In order to meet requirements defined in Sec. 4.2, the following modifications are needed for mobile node.

Detecting IPv4 Care-of Addresses First, a mobile node have to detect when it configures IPv4 care-of addresses on its interface. One possible way is to modify a DHCP client to have the same function as BABYMDD (notify MND/MRD of a new care-of address via the Mobility Socket). The advantage is easy to control the address configuration function. The mobile node can actively perform adding or deleting a care-of address depending on the link status. However, this approach makes the comparison procedure between an IPv4 care-of address and an IPv6 care-of address difficult as following. When both IPv4 access and IPv6 access are available on a link, IPv6 care-of address must be chosen as the primary care-of address of the node. If the care-of address detections are separately performed for IPv4 and IPv6, MND/MRD will have the address change notification asynchronously. Therefore, it is reasonable to modify BABYMDD to deal with IPv4 care-of address too.

The overview of this idea is shown in Fig. 8. The circles in the figure represent functions related to the DSMIPv6 operation, and the bold squares represent a part modified especially for the DSMIPv6 implementation.

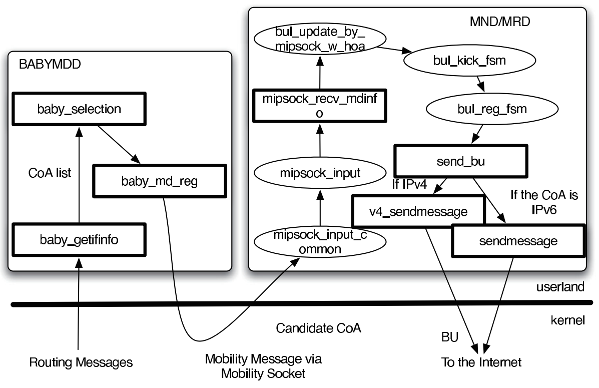


Fig. 8. Detecting IPv4 care-of address and Sending a binding update message

When an IPv4 care-of address is assigned or deleted, a routing message such as RTM_NEWADDR, RTM_DELADDR or RTM_ADDRINFO can be received by monitoring the routing socket. BABYMDD monitors the routing messages to make a list of candidate IPv6 care-of addresses in baby_getifinfo(). We use the same approach for IPv4. A list of candidate IPv4 care-of addresses is also built in baby_getifinfo().

In the BSD operating systems, the *sysctl* allows to retrieve kernel state. The state to be retrieved or set is described using a Management Information Base (MIB) name style. In order to obtain all addresses assigned on a mobile router, baby_getifinfo() uses a *sysctl* with the following MIB.

```
mib[0] = CTL_NET;
mib[1] = PF_ROUTE;
```

```
mib[2] = 0;
mib[3] = AF_UNSPEC;
mib[4] = NET_RT_IFLIST;
mib[5] = 0;
```

BABYMDD then selects a primary care-of address from the list. The function to determine the primary care-of address, baby_selection(), is modified to make an IPv4 care-of address as the primary care-of address. The interface which has the smallest interface index becomes the primary, and an IPv6 care-of address has priority over an IPv4 care-of address as described in [3].

BABYMDD notifies MND/MRD of the selected care-of address via the Mobility Socket. The function to notify MND/MRD of the selected care-of address via the Mobility Socket, baby_md_reg(), is modified to carry both IPv6 and IPv4 addresses. The data structure to store the care-of address was changed from *struct sockaddr_in6* to *struct sockaddr_storage*. Whenever used, the structure is casted with *struct sockaddr_in* or *sockaddr_in6* according to its address family.

Sending a Binding Update Message The notification is passed though generic mipsock processing routines such as mipsock_input_common(), mipsock_input() and mipsock_recv_mdinfo() as shown in Fig. 8. MND/MRD then updates its binding update list with the primary care-of address by bul_update_by_mipsock_w_hoa(). The binding update list is processed by SHISA binding state machine with bul_kick_fsm() and bul_reg_fsm(). A binding update message is sent by calling send_bu() if everything is successfully processed.

The function to receive Mobility Socket messages, mipsock_recv_mdinfo(), is modified to receive an IPv4 care-of address. If the care-of address is IPv4, it will be encoded into an IPv4-mapped IPv6 address. Thanks to this operation, it is not required to modify all other binding management functions.

The function to send a BU, send_bu(), is modified to include the IPv4 home address mobility option. If the care-of address is an IPv4 address, Alternate Care-of Address option is not added but IPv4 home address mobility option is added. If the care-of address is an IPv4 address, v4_sendmessage() is called.

According to [3], the BU is encapsulated in an IPv4 UDP packet. Thus, a new function to send an IPv4 message, v4_sendmessage(), is needed to send an IPv4 message.

IPv4 tunnel packets are decapsulated by the forwarding module implemented in the kernel (shown in Fig. 6), and a pair of a care-of address and a home agent address is to be referred by the module for the decapsulation. Therefore, a tricky hack to setup an IPv4 tunnel (store the pair into the kernel) at the same time as sending a BU is implemented in order to receive the correspondent BA.

Receiving a Binding Acknowledgment Message If the care-of address is IPv4, the binding acknowledgement message is encapsulated in whether IPv4 or IPv4 UDP as explained in Sec. 5.4. The mobile node' operation after receiving an binding acknowledgment message is shown in Fig. 9.

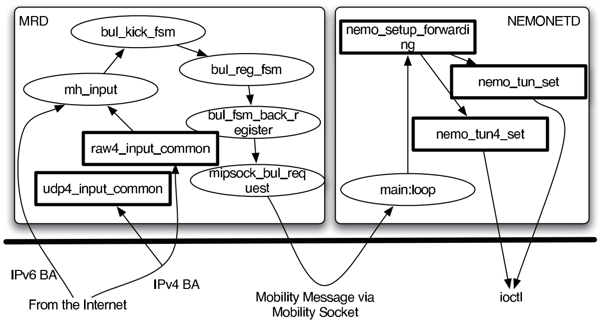


Fig. 9. Receiving a BA and Establishing a Bi-directional Tunnel

When MND/MRD receives an IPv4 or IPv4 UDP encapsulated binding acknowledgment message, the packet will be processed in raw4_input_common(), or in udp4_input_common() in the case of UDP.

The received IPv4/IPv4 UDP packet goes through sanity checks. The sanity checks consist of verifying whether the packet contains an IPv6 home address option, and whether the source address in the IPv4 header is the same as the source address in the IPv6 header (in a IPv6-mapped IPv4 address format). If they are not the same, it is assumed that a NAT is existed between the mobile node and the home agent. After the checks, the packet will be decapsulated and carried to mh_input().

The acknowledgment is then passed to the SHISA mobility header processing routine, mh_input(). The status of the binding is updated as [the binding is accepted] in bul_kick_fsm(), bul_reg_fsm(), and bul_fsm_back_register().

Establishing a Bi-directional Tunnel If a mobile node is a mobile router, a request to create a bi-directional tunnel between the home agent and the mobile router is sent via the Mobility Socket to NEMONETD by mipsock_bul_request() after the binding information has been registered.

NEMONETD receives the request at its main loop, and the bi-directional tunnel is established by calling nemo_setup_forwarding() and nemo_tun4_set().

The function to setup a bi-directional IPv6-in-IPv6 tunnel, nemo_tun_setup(), is modified to establish an IPv4 tunnel. Since the tunnel is already established before sending the BU, most parts of this function will be skipped if the care-of address is an IPv4 address.

The function to setup IPv4 tunnel, nemo_tun4_set(), is added. The function to delete IPv4 tunnel, nemo_tun4_del(), is also added. Those functions are called according to the address family of the primary care-of address.

5.4 Home Agent Modifications

As defined in Sec. 4.2, the following modifications are needed on a Home Agent: Receiving Binding Update Message, Setup a bi-directional tunnel and Sending Binding Acknowledgment Message. Fig. 10 shows the details.

Receiving a Binding Update Message When HAD receives an IPv4 UDP encapsulated binding update message, it will be processed in upd4_input_common(). The received IPv4 UDP packet goes through sanity checks defined in [3].

The binding update message is passed to the SHISA mobility header processing routine,

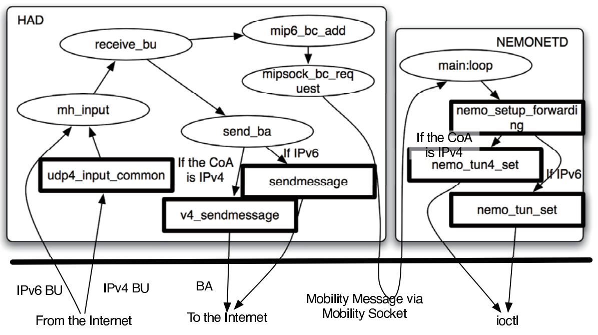


Fig. 10. Receiving a binding update message, Establishing a bi-directional tunnel, and Sending a binding acknowledgement message

mh_input(), and a function to process the binding update, receive_bu(), is called. If the binding update message is processed correctly, the binding cache entry is created or updated by mip6_bc_add() and a request to create a bi-directional tunnel between the mobile router and the home agent is sent via the Mobility Socket to NEMONETD by misock_bc_request().

The function to process a binding update message, receive_bu(), is modified to accommodate IPv4-mapped IPv6 address in the care-of address field. The function to process/expand mobility options, get_mobility_options(), is also modified to process the IPv4 home address mobility option.

Sending a Binding Acknowledgment Message The binding acknowledgment message is then sent by send_ba() as shown in Fig. 10. The function to send a binding acknowledgment message, send_ba(), is modified to add the IPv4 acknowledgment option. If the care-of address is an IPv4 address, the IPv4 address acknowledgment option is added. The binding acknowledgment message is sent via IPv4 network by v4_sendmessage(). The v4_sendmessage() is a function for IPv4 encapsulation or IPv4 UDP encapsulation.

Establishing a Bi-directional Tunnel If the home agent is capable of NEMO Basic Support, a request to create a bi-directional tunnel is sent via the Mobility Socket to NEMONETD by mip-sock_bul_request() after the binding registered as shown in Fig. 10.

NEMONETD receives the request at its main loop, and the bi-directional tunnel is established by calling nemo_setup_forwarding() as well as the mobile node's operation.

5.5 Initialization

The main() in HAD is modified to initialize new sockets. The main() in MND/MRD is also modified as follows. Users can specify an IPv4 home agent address as one of the MND/MRD's argument. This is because the Dynamic Home Agent Address Discovery is not defined in the specification [3]. New sockets are initialized here as well.

The function to build the home agent list from MND/MRD's arguments, add_hal_by_commandline_xxx(), is modified to add IPv4 home agent addresses too. As both IPv4 and IPv6 home agent addresses are stored into the same list, appropriate home agent address is chosen by confirming its address family (the address family must be the same as the current primary care-of address). A new function, mnd_get_v4hoa_by_ifindex() which gets an IPv4 home address from mip device is added.

The function to make a list of home addresses, hoainfo_insert(), is added to add an IPv4 home address on the list. There is an design assumption that an IPv4 home address is assigned on the same interface as where an IPv6 home address is assigned.

6 Evaluation

We performed experiments using our DSMIPv6 implementation in order to measure signalling processing costs newly introduced by DSMIPv6 and to confirm that our implementation works as expected.

6.1 Testbed Details

In order to evaluate our DSMIPv6 implementation, we have setup a testbed as shown in Fig. 11.

A mobile router (MR) is a laptop, IBM ThinkPad X31, with Intel Pentium M 1298.97MHz, 512MB memory, Intel Pro/1000 VE Network Controller, BUFFALO WLI2-CF-S11 (IEEE802.11b), and an EvDO 1x (2GHz) cellular card . SHISA on NetBSD 2.0 is running on this router.

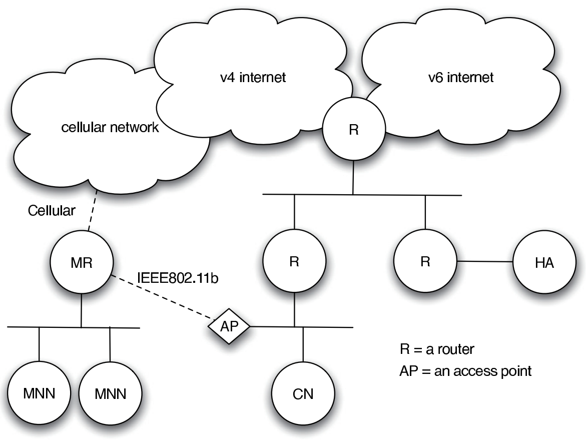


Fig. 11. Testbed topology

A home agent (HA) is a 1U server with Intel Pentium 2527.16MHz, 256MB memory, and Sundance ST-201 10/100 Ethernet. SHISA on NetBSD 2.0 is running on this server.

6.2 Signaling Processing Costs

The DSMIPv6 specification introduces the IPv4 care-of address detection and the additional IPv4 headers. The processing costs are listed as follows and measured by using the testbed.

1. Detecting a care-of address
   a time from when the mobile router attached to the link, to when baby_getifinfo() is called.
2. Sending a binding update
   a time from when baby_getifinfo() is called to when v4_sendmessage() is called
3. Receiving a binding update
   a time from when udp4_input_common() is called to when receive_bu() is called.
4. Sending a binding acknowledgment
   a time from when receive_bu() is called to when v4_sendmessage() is called.
5. Receiving a binding acknowledgment
   from when raw4_input_common() to when bul_kick_ fsm() is called.

The scenario is that an address configuration function (DHCP or NDP) is launched when the mobile node attaches to the Wireless LAN link and the above operations are then performed. This experiments are performed 200 times for the case that the mobile node attaches to IPv6 global foreign network (MIPv6 is used) and the case of IPv4 global foreign network (DSMIPv6 is used). The results are shown in Table 3.

Table 3. Signaling Costs (msec)

Item	**1**	**2**	**3**
`MIPv6`	819.077	1.612	0.232
`DSMIPv6`	1818.758	2.351	0.268

Item	**4**	**5**
`MIPv6`	1.101	0.234
`DSMIPv6`	1.140	0.316

The time to detect the care-of address are largely differ. This is because DHCP requires two round trips between the mobile node and the access router whereas NDP requires only one round trip.

As expected, additional costs due to the outer IPv4 header are observed. However, these costs are negligible because they are smaller than the round trip between the mobile node and the home agent, and these processing are not happened so often (for example, the default lifetime of bindings in SHISA is 40 seconds).

6.3 Forwarding Operation Checks and its Performances

The evaluation of the forwarding module is conducted for all cases (IPv4 or IPv6 home addresses v.s. IPv4 or IPv6 care-of addresses) by using the EvDO 1x 2GHz cellular with the experimental access point at YRP Research Center, KDDI R&D Laboratories. The experimental access point can be configured as an IPv6 global foreign network or an IPv4 global foreign network. This environment is less interference than the commercial access points because there is no user other than us.

The results are listed in Table 4. The first column represents the test case, whether the care-of address (CoA) is IPv6 or IPv4 and whether the correspondent node (CN) uses IPv6 or IPv4. The round trip time (RTT) is measured by using *ping* and the throughput is measured by using *iperf.*

Table 4. Performance in All Situations

Case	RTT (ping)	Throughput (bps)	
CoA-CN	(msec)	TCP (up/down)	UDP (up/down)
v6-v6	174.787	87K/238K	95.3K/332K
v6-v4	183.6	104.3K/701K	96.3K/344.4K
v4-v6	149.8	112K/1.05M	111K/324K
v4-v4	183.27	103.2K/1.08M	110K/308.6K

It is confirmed that the forwarding functions works in all situations. In general, the header cost reduction effect is expected when IPv4 is used because the IPv4 header (20 bytes) is smaller than the IPv6 header (40 bytes). However, it is not observed in this experiment. The performance thus might be greatly depended on other elements such as the round trip time, MTU, network congestion, and the packet loss.

6.4 Consideration

The DSMIPv6 protocol works in all situation without adding remarkable header processing costs. Since only 874 lines are added on the SHISA user land programs for the DSMIPv6 support , it can be managed to minimize the modification (SHISA has 20787 lines). However, the following issues are needed to be revised in the spec:

– UDP header in a binding acknowledgment
  In order to receive a IPv4 encapsulated binding acknowledgment, an IPv4 tunnel is setup before accepting the binding acknowledgment. This can become a security hole. To avoid this, we suggest to force a UDP header in the binding acknowledgment always. (This suggestion has been adopted in the IETF working group and included in the next version of the specification).
– Uses of the IPv4-mapped IPv6 address
  The mapped address in an IPv6 header is not preferable because of several reasons [8].

Thus, we suggest to put the IPv6 home address in the IPv6 header and a kind of IPv4 care-of address option in the mobility headers.

7 Conclusion

This paper describes the DSMIPv6 implementation on an open source Mobile IPv6 and NEMO implementation for BSD operating systems. The DSMIPv6 implementation was designed to separate the signaling function and the forwarding function, and to minimize modifications on the existing kernel and user land programs. The binding management code is shared with both IPv4 and IPv6 mobility functions by using IPv4-mapped IPv6 address format. When care-of or home addresses are used, new or modified correspondent functions are called according to its address family. An IPv4 care-of address detection feature is also added to the user land programs. By evaluating the implementation, it is confirmed that the DSMIPv6 implementation works in the all situations without adding remarkable header processing overhead. It is thus said that the specification is stable to forward IPv4/IPv6 packets address to their home addresses/mobile networks.

Acknowledgment

The authors would like to thank KDDI R&D Laboratories and KDDI for their helps.

References

1. D. Johnson et al. Mobility Support in IPv6. Request For Comments 3775, The Internet Engineering Task Force, June 2004.
2. V. Devarapalli et al. Network Mobility (NEMO) Basic Support Protocol. Request For Comments 3963, The Internet Engineering Task Force, Jan. 2005.
3. H. Soliman Ed. Mobile IPv6 support for dual stack Hosts and Routers (DSMIPv6). Internet Drafts draft-ietf-mip6-nemo-v4traversal-02, The Internet Engineering Task Force, June 2006.
4. SHISA Project, As of July. 2006. http://www.mobileip.jp/.
5. Keiichi Shima, Ryuji Wakikawa, Koshiro Mitsuya, Keisuke Uehara, and Tsuyoshi Momose. SHISA: The IPv6 Mobility Framework for BSD Operating Systems. In *IPv6 Today – Technology and Deployment Workshop (IPv6TD'06)*, August 2006.
6. KAME Project, As of July. 2006. http://www.kame.net/.
7. T. Momose et al. The application interface to exchange mobility information with Mobility subsystem. Internet Drafts draft-momose-mip6-mipsock-00, The Internet Engineering Task Force, July 2005.
8. Craig Metz et al. IPv4-Mapped Address API Considered Harmful. Internet Drafts draft-cmetz-v6ops-v4mapped-api-harmful-01, The Internet Engineering Task Force, October 2003.

Security measures in OpenSSH

Damien Miller
OpenSSH Project
djm@openssh.com

ABSTRACT

This paper examines several security measures that have been implemented in OpenSSH. OpenSSH's popularity, and the necessity for the server to wield root privileges, have made it a high-value target for attack. Despite initial and ongoing code audits, OpenSSH has suffered from a number of security vulnerabilities over its 7.5 year life. This has prompted the developers to implement several defensive measures, intended to reduce both the likelihood of exploitable errors and the consequences of exploitation should they occur.

This paper examines these defensive measures; each measure is described and assessed for implementation effort, attack surface reduction, effectiveness in preventing or mitigating attacks, applicability to other network software and possible improvements.

General Terms

Security

Keywords

SSH, OpenSSH, Security, Attack Surface, Network Applications

1 Introduction

OpenSSH [22] is a popular implementation of the SSH protocol [32]. It is a network application that supports remote login, command execution, file transfer and forwarding of TCP connections between a client and server. It is designed to be safely used over untrusted networks and includes cryptographic authentication, confidentiality and integrity protection.

Since its release in 1999, OpenSSH quickly gained popularity and rapidly became the most popular SSH implementation on the Internet [23]. Today it is installed by default on almost all modern Unix and Unix-like operating systems, as well as many network appliances and embedded devices.

OpenSSH is has been developed to run on Unix-like operating systems and must operate within the traditional Unix security model. Notably, the OpenSSH server, *sshd*, requires root privileges to authenticate users, access the host private key, allocate TTYs and write records of logins. OpenSSH is also based on a legacy code-base, that of ssh-1.2.16 [33]

OpenSSH's popularity, and the knowledge that a successful compromise gives an attacker a chance to gain super-user privileges on their victim's host has made it an attractive target for both research and attack. Since its initial release in 1999, a number of security bugs have been found in OpenSSH. Furthermore some of the libraries that OpenSSH depends on have suffered from bugs that were exposed through OpenSSH's use of them.

Some of these errors have been found despite OpenSSH being manually audited on several occasions. This, and the occurrence of vulnerabilities in dependant libraries, have caused the developers to implement a number of proactive measures to reduce the likelihood of exploitable errors, make the attacker's work more difficult and to limit the consequences of a successful exploit. These measures include replacement of unsafe APIs, avoidance of complex or error-prone code in dependant libraries, privilege separation of the server, protocol changes to eliminate pre-authentication complexity and a mechanism to maximise the benefit of OS-provided attack mitigation measures.

A key consideration in implementing these measures has been their effect on reducing OpenSSH's *attack surface*. Attack surface [17] is a qualitative measure of an application's "attackability" based on the amount of application code exposed to an attacker. This quantity is scaled by the ease with which an attacker can exercise the code – for example, code exposed to unauthenticated users would be weighted higher than that accessible only by authenticated users. A further weighting is given to code that holds privilege during its execution, as an attacker is likely to inherit this privilege in the event of a successful compromise. Attack surface may therefore be considered as a measure of how well developers have applied Saltzer and Schroeder's *Economy of Mechanism* and *Least Privilege* design principles [7].

This paper examines these security measures in OpenSSH's server daemon, *sshd*. Each measure is considered for implementation ease, applicability to other network applications, attack surface reduction, actual attacks prevented and possible improvements.

2 Critical vulnerabilities

Table 1 lists and characterises several critical vulnerabilities found in OpenSSH since 1999. We consider a vulnerability *critical* if it has a moderate to high likelihood of successful remote exploitation.

File	*Problem*	*Found*
session.c	sanitisation error	Friedl, 2000 [15]
deattack.c	integer overflow	Zalewski, 2001 [18]
radix.c	stack overflow	Fodor, 2002 [11]
channels.c	array overflow	Pol, 2002 [8]
auth2-chall.c	array overflow	Dowd, 2002 [12]
buffer.c	integer overflow	Solar Designer, 2003 [13]
auth-chall.c	logic error	OUSPG, 2003 [21]

Table 1: Critical vulnerabilities in OpenSSH

OpenSSH has also been susceptible to bugs in libraries it depends on. Over the same period, zlib [9] and OpenSSL [24] have suffered from a number of vulnerabilities that could be exploited through OpenSSH's use of them. These include heap corruption [14] and buffer overflows [27] [16] in zlib, and multiple overflows in the OpenSSL ASN.1 parser [20].

Note that many of these vulnerabilities stem from memory management errors. It follows that measures that reduce the likelihood of memory management problems occurring, or that make their exploitation more difficult are likely to yield a security benefit.

3 OpenSSH security measures

OpenSSH will naturally have a raised attack surface because of its need to accept connections from unauthenticated users, while retaining the *root* privileges it needs to record login and logout events, open TTY devices and authenticate users.

The approaches used to reduce this attack surface or otherwise frustrate attacks generally fall into the following categories: defensive programming, avoiding complexity in dependant libraries, privilege separation and better use of operating system attack mitigation measures.

3.1 Defensive programming

Defensive programming seeks to prevent errors through the insertion of additional checks [29]. An expansive interpretation of this approach should also include avoidance or replacement of APIs that are ambiguous or difficult to use correctly. In OpenSSH's case, this has included replacement of unsafe string manipulation functions with the safer `strlcpy` and `strlcat` [30] and the replacement of the traditional Unix `setuid` with the less ambiguous [2] `setresuid` family of calls.

A source of of potential errors may be traced to POSIX's tendency to overload return codes; using `-1` to indicate an error condition, but zero for success and positive values as a result indicator (a good example of this is the `read` system call). This practice leads to a natural mixing of unsigned and signed quantities, often when dealing with I/O. Integer wrapping and signed-vs-unsigned integer confusion have caused a number of OpenSSH security bugs, so this is of some concern. OpenSSH performs most I/O calls through a "retry on interrupt" function, `atomicio`. This function was modified to always return an unsigned quantity and to instead report its error via `errno`. Making this API change did not uncover any bugs, but reducing the need to use signed types it made it easier to enable the compiler's signed/unsigned comparison warnings and fix all of the issues that it reported.

Integer overflow errors are often found in dynamic array code. A common C language idiom is to allocate an array using `malloc` or `calloc`, but attacker-controlled arguments to these functions may wrap past the maximum expressible `size_t`, resulting in an exploitable condition [1]. `malloc` and array resizing using `realloc` are especially prone to this, as their argument is often a product, e.g. `array = malloc(n * sizeof(*array))`.

OpenSSH replaced all array allocations with an error-checking `calloc` variant (derived from the OpenBSD implementation) that takes as arguments a number of elements to allocate and a per-element size in bytes. These functions check that the product of these quantities does not overflow before performing an allocation. The `realloc` function, which has no calloc-like counterpart in the standard library (i.e. accepting arguments representing a number of elements and an element size), was replaced with a calloc-like error-checking array reallocator. Implementing this change added only 17 lines of code to OpenSSH, but has not yet uncovered any previously-exploitable overflows. A similar change was subsequently made to many other programs in the

OpenBSD source tree.

Once valid criticism of API replacements is that they make a program more difficult to read by an new developer, as they must frequently recurse into unfamiliar APIs. In OpenSSH's case, effort has been made to use standardised APIs as replacements wherever possible as well as using logical and consistent naming for non-standard replacements (e.g. `calloc` → `xcalloc`)

3.2 Avoiding complexity in dependant libraries

Significant complexity, and thus attack surface, can lurk behind simple library calls. If there is sufficient risk, it may be worthwhile to replace them with more simple, or limited versions. Replacing important API calls is not without risk or cost; it represents additional development and maintenance work and it provides the opportunities for new errors to be made in critical code-paths. If replacement is considered to risky, simply avoiding the call may still be an option – OpenSSH avoids the use of regular expression libraries for this reason.

An example of this approach is OpenSSH's replacement of RSA and DSA cryptographic signature verification code. Prior to late 2002, OpenSSH used the OpenSSL `RSA_verify` and `DSA_verify` functions to verify signatures for user- and host-based public-key authentication. The OpenSSL implementations use a general ASN.1 parser to unpack the decrypted signature object. This adds substantial complexity – in the case of `RSA_verify` at least 282 lines of code, not including calls to the raw OpenSSL cryptographic primitives, or its custom memory allocation, error handling and binary I/O functions.

These calls were replaced with minimal implementations that avoided generic ASN.1 parsing in favour of a simple comparison of the structure of the decrypted signature to an expected form. The replacement `openssh_RSA_verify` function was implemented in 63 lines of code, of much simpler structure (basically decrypt then compare) and with no calls to complex subroutines other than the necessary cryptographic operations.

The replacement functions clearly reduce the attack surface of public key authentication in OpenSSH and have avoided a number of critical bugs in the OpenSSL implementations, notably an overflow in the ASN.1 parsing [20] and a signature forgery bug [3], both of which were demonstrated to be remotely exploitable.

3.3 Protocol changes to reduce attack surface

The SSH protocol includes a compression facility that is intended to improve throughput over low-bandwidth links. Compression is negotiated during the initial key exchange phase of the protocol and activated, along with encryption and message authentication, as soon as the key exchange has finished. The next phase of the protocol is user authentication, but by this time compression is already enabled and any bugs in the underlying zlib code have been exposed to an unauthenticated attacker.

OpenSSH introduced a new compression method *zlib@openssh.com* [5] as a protocol extension (the SSH protocol has a nice extension mechanism that allows the use of arbitrary extension method names under the developer's domain name – unadorned names are reserved for standardised protocol methods). The zlib@openssh.com compression method uses exactly the same underlying compression algorithm (zlib's *deflate*), it merely delays its activation until successful completion of user authentication. This eliminates all zlib exposure to unauthenticated users.

An alternate solution to this problem that does not a require protocol change is to refuse compression in the initial key exchange proposal, but then offer it in a re-exchange immediately after user authentication has completed. This approach was rejected, as key exchange is a heavyweight operation in the SSH protocol; usually consisting of a Diffie-Hellman [4] key agreement with a large modulus. Performing a re-exchange to effectively flip a bit was considered too expensive.

The benefit of delayed compression is clear, despite there not having been any zlib vulnerabilities published since it was implemented. Network application developers considering making non-standard protocol changes to reduce attack surface should consider interoperability carefully, especially if the protocol they are implementing lacks a orthogonal extension mechanism like SSH's.

3.4 Privilege separation

[*Privilege separation in OpenSSH is described in detail in [25], this is a brief summary*].

The design principle of *Least Privilege* [7] requires that privilege be relinquished as soon as it is no longer required, but what should application developers do in cases where privilege is required sporadically through an applications life? sshd is such an application; it must retain root privileges after the user has authenticated and logged in as it needs to record login and logout records and allocate TTYs. Furthermore the SSH protocol allows multiple sessions over a single SSH transport and these sessions may be started any time after user authentication is complete.

OpenSSH 3.3 implemented *privilege separation* (a.k.a *privsep*), where the daemon is split into a privileged *monitor* and an unprivileged *slave* process. Before authentication (*pre-auth*), the slave runs as a unique, nonprivileged user. After authentication (*post-auth*) the slave runs with the privileges of the authenticated user. In all cases, the slave process is jailed (via the `chroot` system call) into an empty directory, typically `/var/empty`.

The slave is responsible for the SSH transport, including cryptography, packet parsing and managing open "channels" (login sessions, forwarded TCP ports, etc.). When the slave needs to perform an action that requires privilege, or any interaction with the wider system, it messages the monitor, which performs the requested action and returns the results.

The monitor is structured as a state-machine, enforcing constraints over which actions a slave may request at its stage in the protocol (e.g. opening login sessions before user authentication is complete is not permitted). The monitor is intended to be as small (codewise) as possible; the initial implementation removed privilege from just over two thirds of the OpenSSH application [25].

OpenSSH's privsep implementation is complicated somewhat by the need to offer compression before authentication. Once user authentication is complete, the pre-auth slave must serialise and export its connection state for use by the post-auth slave, including cryptographic keys, initialisation vectors (IVs), I/O buffers and compression state. Unfortunately the zlib library offers no functions to serialise compression state. However it does support allocation hooks that it will use instead of the standard `malloc` and `free` functions. OpenSSH's privsep includes a memory manager that is used by zlib. This manager uses anonymous memory mappings that are shared between the pre-auth slave and the monitor. The post-auth slave inherits this memory from the monitor when it is started. Since the monitor treats these allocations as completely opaque and never invokes zlib functions, there is no risk of monitor compromise through deliberately corrupted zlib state.

The OpenSSH privsep implementation builds the monitor and both the pre- and post-authentication slaves into the one executable. This may be contrasted with the Postfix MTA [31], which uses separate cooperating executables that run at various privilege levels. The OpenSSH implementation could probably be simplified if it the monitor were split into a dedicated executable that in turn executed separate slave executables. An additional benefit to this model would be the slaves would no longer automatically inherit the same address space layout as the monitor (further discussed in section 3.5), but it would carry some cost: it would no longer be possible to disable privsep, and it would be impossible to support the standard compression mode though the above shared memory allocator – zlib would have to be modified to allow state export, or pre-authentication compression would have to be abandoned.

Another criticism [19] of OpenSSH's privsep implementation is that it uses the same buffer API as the slave to marshal and unmarshal its messages. This renders the monitor susceptible to the same bugs as the slave if they occur in this buffer code (one such bug has already occurred [13]). However, the alternative of reimplementing the buffer API for the monitor is not very attractive either; maintaining two functionally identical buffer implementation raises the attack surface for questionable benefit. A better approach may be to automatically generate the marshalling code (discussed further in section 4).

Privilege separation in OpenSSH has been a great success; it has reduced the severity of all but one of the memory management bugs found in OpenSSH since its implementation, and the second layer of checking that the privsep monitor state machine offers has prevented at least one logic error in the slave from being exploited. It has suffered from only one known bug, that was not exploitable without first having compromised the slave process. OpenSSH is something of a worst-case in terms of the complexity required to implement privilege separation, other network applications seeking to implement it will likely find it substantially easier.

3.5 Assisting OS-level attack mitigation

Modern operating systems are beginning to implement attack mitigation measures [28] intended to reduce the probability that a given attack will succeed. These measures include stack protection as well as a suite of *runtime randomisations* that are applied to stack gaps, executable and library load addresses, as well as to the addresses returned by memory allocation functions. Collectively, these randomisations (often referred to as Address Space Layout Randomisation – ASLR) render useless any exploits that use fixed offsets or return addresses.

These randomisations are typically applied *per-execution*, as it would be very difficult to otherwise re-randomise an application's address space at runtime. Stack protectors such as SSP/Propolice [6] also employ random "canaries" that are initialised per-execution.

A typical Unix daemon that forks a subprocess to service each request will inherit the address space layout of its parent. In the absence of other mitigation measures, an attacker may therefore perform an *exhaustive search*, trying every possible offset or return address until they find one that works. They will be guaranteed success, as there is a finite and unchanging space of possible addresses (as little as 16-bits in some implementations [26]).

To improve this situation, OpenSSH implemented self-re-execution. sshd was modified to fork, then execute itself to service each connection rather than simply forking. Each daemon instance serving a connection is therefore re-randomised, approximately doubling the effort required to guess a correct offset and removing any absolute guarantee of success.

This change, while straightforward to implement, does incur some additional overhead for each connection, and has been criticised as offering little benefit on systems that do not support any ASLR-like measures.

4 Future directions

There are several opportunities to further improve OpenSSH's security and attack resistance. Perhaps the most simple is to disable or deprecate unsafe or seldom-used protocol elements. Removing support for pre-authentication compression (once a delayed compression method is standardised) would permanently remove complexity and significantly simplify the privilege separation implementation. Likewise, deactivating and ultimately removing support for the legacy SSH protocol version 1 would remove a lot of complexity from the code (and may hasten the demise of a protocol with known weaknesses).

Further attack resistance may be gained by including measures to frustrate *return-to-executable* attacks, where the attacker sets up a stack frame with controlled arguments and then returns to a useful point inside the currently executing process. In OpenSSH's case, they may select the `do_exec` function, that is responsible for spawning a subshell as part of session creation. These attacks may be made more difficult by pervasively inserting authentication checks into code that has the potential to be subverted. However these checks have the potential to bloat and obfuscate the code and their effectiveness at preventing this attack is not entirely clear.

Improved attack resistance could also be achieved by having the listener sshd process check the exit status of the privsep monitor and (indirectly) slave processes. If an abnormal exit status is detected, such as a forced termination for a segmentation violation, then the listener could take remedial action such as rate-limiting similar connections as defined by a (source address, username) tuple. This would work especially well on operating systems that support ASLR – exploits will be nondeterministic on these platforms, and an attacker will be forced to make many connections to find working offsets. Attacks may be rendered infeasible or unattractive by limiting the rate at which these attempts can be made. This concept could be extended to form the basis of a denial of service mitigation, where the listener could impose a rate-limit on connections from hosts that repeatedly fail to authenticate within the login grace period, or that experience frequent login failures.

Another potential approach to reducing errors is to generate mechanical parts of the source automatically. Packet parsers are an excellent candidate for automatic generation, and this technique is used by many RPC implementations and at least one SSH implementation already [10]. The channel and privilege separation state-machines could also be represented at a higher level, allowing easier verification.

5 Conclusion

Network software that accepts data from unauthenticated users while requiring privilege to operate presents a significant security challenge to the application developer. This paper has described the OpenSSH project's approach to this problem, has detailed a number of specific measures that have been implemented and explored areas of possible future work.

These measures focus on reducing the attack surface of the application and making better use of any attack mitigation facilities provided by the underlying operating system. They have been shown to be effective in stopping exploitable problems occurring or in reducing their impact when they do occur. Finally, these measures have been shown to be relatively easy to implement and widely applicable to other network software

References

[1] blexim. Basic Integer Overflows. *Phrack, Vol 11, Number 60, Build 3*, December 2002.

[2] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[3] Daniel Bleichenbacher. OpenSSL PKCS Padding RSA Signature Forgery Vulnerability. `www.securityfocus.com/bid/19849`, September 2005.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, 1976.

[5] Markus Friedl and Damien Miller. Delayed compression for the SSH Transport Layer Protocol, 2006. INTERNET-DRAFT draft-miller-secsh-compression-delayed-00.txt.

[6] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. `www.trl.ibm.com/projects/security/ssp/`, August 2005.

[7] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE 63, number 9*, pages 1278–1308, September 1975.

[8] Joost Pol. OpenSSH Channel Code Off-By-One Vulnerability. `online.securityfocus.com/bid/4241`, March 2002.

[9] Jean loup Gailly and Mark Adler. zlib. `www.zlib.net`.

[10] Anil Madhavapeddy and David Scott. On the Challenge of Delivering High-Performance, Dependable, Model-Checked Internet Servers. In *In the proceedings of the First Workshop on Hot Topics in System Dependability (HotDep)*, pages 37–42, June 2005.

[11] Marcell Fodor. OpenSSH Kerberos 4 TGT/AFS Token Buffer Overflow Vulnerability. `online.securityfocus.com/bid/4560`, April 2002.

[12] Mark Dowd. OpenSSH Challenge-Response Buffer Overflow Vulnerabilities. `online.securityfocus.com/bid/5093`, June 2002.

[13] Mark Dowd and Solar Designer and the OpenSSH team. OpenSSH Buffer Management Vulnerabilities. `online.securityfocus.com/bid/8628`, September 2003.

[14] Mark J. Cox and Matthias Clasen and Owen Taylor. ZLib Compression Library Heap Corruption Vulnerability. `www.securityfocus.com/bid/4267`, March 2002.

[15] Markus Friedl. OpenSSH UseLogin Vulnerability. `online.securityfocus.com/bid/1334`, June 2000.

[16] Markus Oberhumer. Zlib Compression Library Decompression Buffer Overflow Vulnerability. `online.securityfocus.com/bid/14340`, July 2005.

[17] Michael Howard. Fending Off Future Attacks by Reducing Attack Surface. `msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp`, February 2003.

[18] Michal Zalewski. SSH CRC-32 Compensation Attack Detector Vulnerability. `online.securityfocus.com/bid/2347`, February 2001.

[19] Sun Microsystems. Sun's Alternative "Privilege Separation" for OpenSSH. `src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/ssh/README.altprivsep`, 2006.

[20] NISCC and Stephen Henson. OpenSSL ASN.1 Parsing Vulnerabilities. `www.securityfocus.com/bid/8732`, September 2003.

[21] OUSPG. Logic error in PAM authentication code. Private email, September 2003.

[22] The OpenSSH Project. OpenSSH. `www.openssh.org`.

[23] The OpenSSH Project. SSH usage profiling. `www.openssh.org/usage`.

[24] The OpenSSL Project. OpenSSL. `www.openssl.org`.

[25] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, August 2003.

[26] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[27] Tavis Ormandy. Zlib Compression Library Buffer Overflow Vulnerability. `online.securityfocus.com/bid/14162`, July 2005.

[28] Theo de Raadt. Exploit Mitigation Techniques. `www.openbsd.org/papers/ven05-deraadt`, November 2005.

[29] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. Technical report, August 1976.

[30] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference. Monterey, CA*, pages 175–178, June 1999.

[31] Wieste Venema. Postfix MTA. `www.postfix.org`.

[32] T. Ylönen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[33] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, July 1996.

Porting the ZFS file system to the FreeBSD operating system

Pawel Jakub Dawidek
pjd@FreeBSD.org

1 Introduction

The ZFS file system makes a revolutionary (as opposed to evolutionary) step forward in file system design. ZFS authors claim that they throw away 20 years of obsolute assumptions and designed an integrated system from scratch.

The ZFS file system was developed by Sun Microsystems, Inc. and was first available in Solaris 10 operating system. Although we cover some of the key features of the ZFS file system, the primary focus of this paper is to cover how ZFS was ported to the FreeBSD operating system.

FreeBSD is an advanced, secure, stable and scalable UNIX-like operating system, which is widely deployed for various internet functions. Some argue that one of the largest challenges facing FreeBSD is the lack of a robust file system. Porting ZFS to FreeBSD attempts to address these short comings.

2 ZFS file system and some of its features

Calling ZFS a file system is not precise. ZFS is much more than only file system. It integrates advanced volume management, which can be utilized by the file system on top of it, but also to provide storage through block devices (ZVOLs). ZFS also has many interesting features not found in other file systems. In this section, we will describe some of the features we find most interesting.

2.1 Pooled storage model

File systems created by ZFS are not tied to a specified block device, volume, partition or disk. All file systems within the same ”pool”, share the whole storage assigned to the ”pool”. A pool is a collection of storage devices. It may be constructured from one partition only, as well as from hundreds of disks. If we need more storage we just add more disks. The new disks are added at run time and the space is automatically available to all file systems. Thus there is no need to manually grow or shrink the file systems when space allocation requirements change. There is also no need to create slices or partitions, one can simply forget about tools like fdisk(8), bsdlabel(8), newfs(8), tunefs(8) and fsck(8) when working with ZFS.

2.2 Copy-on-write design

To ensure the file system is functioning in a stable and reliable manner, it must be in a consistent state. Unfortunately it is not easy to guarantee consistency in the event of a power failure or a system crash, because most file system operations are not atomic. For example when a new hard link to a file is created, we need to create a new directory entry and increase link count in the inode, which means we need two writes. Atomicity around disk writes can only be guaranteed on a per sector basis. This means if a write operation spans more than a single sector, there can be no atomicity guarantees made by the disk device. The two most common methods to manage consistency of file system are:

- Checking and repairing file system with fsck [McKusick1994] utility on boot. This is very inefficient method, because checking large file systems can take serval hours. Starting from FreeBSD 5.0 it is possible to run fsck program in the background [McKusick2002], significantly reducing system downtime. To make it possible, UFS [McKusick1996] gained ability to create snapshots [McKusick1999]. Also file system has to use Soft updates [Ganger] guarantee that the only inconsistency the file system would experience is resource leaks steming from unreferenced blocks or inodes. Unfortunately, file system snapshots have few disadvantages. One of the stages of performing a snapshot blocks all write operations. This stage should not depend on file system size and should not take too long. The time of another stage, which is responsible for snapshot perparation grows linearly with the size of the file system and generates heavy I/O load. Once snapshot is taken, it should not slow the system down appreciably except when removing many small files (i.e., any file less than 96Kb whose last block is a fragment) that are claimed by a snapshot. In addition checking file system in the background slows operating system performance for many hours. Practice shows that it is also possible for background fsck to fail, which is a really hard situation, because operating system needs to be rebooted and file system repaired in foreground, but what is more important, it means that system was working with inconsistent file system, which implies undefined behaviour.

- Store all file system operations (or only metadata changes) first in a special "journal", once the whole operation is in the journal, it is moved to the destination area. In the event of a power failure or a system crash incomplete entires in the journal are removed and not fully copied entries are copied once again. File system journaling is currently the most popular way of managing file system consistency [Tweedie2000, Best2000, Sweeney1996].

The ZFS file system does not need fsck or jounrals to guarantee consistency, instead takes an alternate "Copy On Write" (COW) approach. This means it never overwrites valid data - it writes data always into free area and when is sure that data is safely stored, it just switches pointer in block's parent. In other words, block pointers never point at inconsistent blocks. This design is similar to the WAFL [Hitz] file system design.

2.3 End-to-end data integrity and self-healing

Another very important ZFS feature is end-to-end data integrity - all data and metadata undergoes checksum operations using one of several available algorithms (fletcher2 [fletcher], fletcher4 or SHA256). This allows to detect with very high probability silent data corruptions cased by any defect in disk, controller, cable, driver, or firmware. Note, that ZFS metadata are always checksummed using SHA256 algorithm. There are already many reports from the users experiencing silent data corruptions successfully detected by ZFS. If some level of redundancy is configured (RAID1 or RAID-Z) and data corruption is detected, ZFS not only reads data from another replicated copy, but also writes valid data back to the component where corruption was originally detected.

2.4 Snapshots and clones

A snapshot is a read-only file system view from a given point in time. Snapshots are fairly easy to implement for file system storing data in COW model - when new data is stored we just don't free the block with the old data. This is the reason why snapshots in ZFS are very cheap to create (unlike UFS2 snapshots). Clone is created on top of a snapshot and is writable. It is also possible to roll back a snapshot forgetting all modifications introduced after the snapshot creation.

2.5 Built-in compression and encryption

ZFS supports compression at the block level. Currently (at the time this paper is written) only one compression algorithm is supported - lzjb (this is a variant of Lempel-Ziv algorithm, jb stands for his creator - Jeff Bonwick). There is also implementation of gzip algorithm support, but it is not included in the base system yet. Data encryption is a work in progress [Moffat2006].

2.6 Portability

A very important ZFS characteristic is that the source code is written with portability in mind. This is not an easy task, especially for the kernel code. ZFS code is very portable, clean, well commented and almost self-contained. The source files rarely include system headers directly. Most of the times, they only include ZFS-specific header files and a special `zfs_context.h` header, where one should place system-specific includes. Big part of the kernel code can be also compiled in userland and used with ztest utility for regression and stress testing.

3 ZFS and FreeBSD

This section describes the work that has been done to port the ZFS file system over to the FreeBSD operating system.
The code is organized in the following parts of the source tree:

- `contrib/opensolaris/` - userland code taken from OpenSolaris, used by ZFS (ZFS control utilities, libraries, etc.),
- `compat/opensolaris/` - userland API compatibility layer (implementation of Solaris-specific functions in FreeBSD way),
- `cddl/` - Makefiles used to build userland utilities and libraries,
- `sys/contrib/opensolaris/` - kernel code taken from OpenSolaris, used by ZFS,
- `sys/compat/opensolaris/` - kernel API compatiblity layer,
- `sys/modules/zfs/` - Makefile for building ZFS kernel module.

The following milestones were defined to port the ZFS file system to FreeBSD:

- Created Solaris compatible API based on FreeBSD API.
- Port userland utilities and libraries.
- Define connection points in the ZFS top layers where FreeBSD will talk to us and those are:
  - ZPL (ZFS POSIX Layer) which has to be able to communicate with VFS,
  - ZVOL (ZFS Emulated Volume) which has to be able to communicate with GEOM,
  - `/dev/zfs` control device, which actually only talks to ZFS userland utilities and libraries.
- Define connection points in the ZFS buttom layers where ZFS needs to talk to FreeBSD and this is only VDEV (Virtual Device), which has to be able to communicate with GEOM.

3.1 Solaris compatibility layer

When a large project like ZFS is ported from another operating system one of the most important rules is to keep number of modifications of the original code as small as possible, because the fewer modifications, the easier porting new functionality and bug fixes is. The programmer that does the porting work is not the only one responsible for number of changes needed, it also depends on how portable the source code is.

To minize the number of changes, a Solaris API compatability layer was created. The main goal was to implement Solaris-specific functions, structures, etc. using FreeBSD KPI. In some cases, functions needed to be renamed, while in others, functionality needed to be fully implemented from scratch. This technique proved to be very effective (not forgetting about ZFS code portability). For example looking at files from the `uts/common/fs/zfs/` directory and taking only non-trivial changes into account, only 13 files out of 112 files were modified.

3.1.1 Atomic operations

There are a bunch of atomic operations implemented in FreeBSD (atomic(9)), but there are some that exist in Solaris and have no equivalents in FreeBSD. The missing operations in pseudo-code look like this:

```
<type>
atomic_add_<type>_nv(<type> *p, <type> v)
{
    return (*p += v);
}

<type>
atomic_cas_<type>(<type> *dst, <type> cmp, <type> new)
{
    <type> old;

    old = *dst;
    if (old == cmp)
        *dst = new;
    return (old);
}
```

Another missing piece is that FreeBSD implements 64bit atomic operations only on 64bit architectures and ZFS makes heavy use of such operations on all architectures.

Currently, atomic operations are implemented in assembly language located in the machine dependant portions of the kernel. As a temporary work around, the missing atomic operations were implemented in C, and global mutexes were used to guarantee atomicity. Looking forward, the missing atomic operations may be imported directly from Solaris.

3.1.2 Sleepable mutexes and condition variables

The most common technique of access synchronization to the given resources is locking. To guarantee exclusive access FreeBSD and Solaris use mutexes. Unfortunately we cannot use FreeBSD mutex(9) KPI to implement Solaris mutexes, because there are some important differences. The biggest problem is that sleeping with FreeBSD mutex held is prohibited, on Solaris on the other hand such behaviour is just fine. The way we took was to implement Solaris mutexes based on our shared/exclusive locks - sx(9), but only using exclusive locking. Because of using sx(9) locks for Solaris mutex implementation we also needed to implement condition variables (condvar(9)) to use Solaris mutexes.

3.2 FreeBSD modifications

There were only few FreeBSD modifications needed to port ZFS file system.

The sleepq_add(9) function was modified to take `struct lock_object *` as an argument instead of `struct mtx *`. This change allowed to implement Solaris condition variables on top of sx(9) locks.

The mountd(8) program gained ability to work with multiple exports files. With this change we can automatically manage private exports file stored in `/etc/zfs/exports` via zfs(1) command.

The VFS_VPTOFH() operation was turned into VOP_VPTOFH() operation. As confirmed by Kirk McKusick, vnode to file handle translation should be a VOP operation in the first place. This change allows to support multiple node types within one file system. For example in ZFS v_data field from the vnode structure can point at two different structures (znode_t or zfsctl_node_t). To be able to recognize which structure it is, we define different functions as vop_vptofh operation for those two different vnodes.

lseek(2) API was extended to support SEEK_DATA and SEEK_HOLE [Bonwick2005] operation types. Those operations are not ZFS-specific. They are useful mostly for backup software to skip "holes" in files. "Holes" like those created with truncate(2).

3.3 Userland utilities and libraries

Userland utilities and libraries communicate with the kernel part of the ZFS via `/dev/zfs` control device. We needed to port the following utilities and libraries:

- `zpool` - utility for storage pools configuration.
- `zfs` - utility for ZFS file systems and volumes configuration.
- `ztest` - program for stress testing most of the ZFS code.
- `zdb` - ZFS debugging tool.
- `libzfs` - the main ZFS userland library used by both zfs and zpool utilities.
- `libzpool` - test library containing most of the kernel code, used by ztest.

To make it work we also ported libraries (or implemented wrappers) they depend on: `libavl`, `libnvpair`, `libuutil` and `libumem`.

3.4 VDEV_GEOM

ZFS have to use storage provided by the operating system, so at the bottom layers it has to be connected to disks. In Solaris there are two "leaf" VDEVs (Virtual Devices) that allow to use storage from disks (VDEV_DISK) and from files (VDEV_FILE). We don't use those in FreeBSD. The interface to talk to disks in FreeBSD is totally incompatible with what Solaris has. That's why we decided to create a FreeBSD-specific leaf VDEV - VDEV_GEOM. VDEV_GEOM was implemented as consumer-only GEOM class, which allows to use **any** GEOM provider (disk, partition, slice, mirror, encrypted storage, etc.) as a storage pool component. We find this solution very flexible, even more flexible than what Solaris has. We also decided not to port VDEV_FILE, because files can always be accessed via md(4) devices.

3.5 ZVOL

ZFS can serve the storage in two ways - as a file system or as a raw storage device. ZVOL (ZFS Emulated Volume) is a ZFS layer responsible for managing raw storage devices (GEOM providers in FreeBSD) backed by space from a storage pool. It was implemented in FreeBSD as a provider-only GEOM class to fit best in FreeBSD current architecture (all storage devices in FreeBSD are GEOM providers). This way we can put a UFS file system or swap on top of a ZFS volume or we can use ZFS volumes as components in other GEOM tranformations. For example we can encrypt ZFS volume with GELI class.

3.6 ZPL

ZPL (ZFS POSIX Layer) is the layer that VFS interface communicates with. This was the hardest part of the entire ZFS port. The VFS interfaces are most of the time very system-specific and also very complex. We belive

that VFS is one of the most complex subsystem in the entire FreeBSD kernel.
There are many differences in VFS on Solaris and FreeBSD, but they are still quite similar. VFS on Solaris seems to be cleaner and a bit less complex than FreeBSD's.

3.7 Event notification

ZFS has the ability to send notifications on various events. Those events include information like storage pool imports as well as failure notifications (I/O errors, checksum mismatches, etc.). This functionality was ported to send notifications to the devd(8) daemon, which seems to be the most suitable communication channel for those kind of messages. We may consider creating dedicated userland daemon to manage messages from ZFS.

3.8 Kernel statistics

Various statistics (mostly about ZFS cache usage) are exported via kstat Solaris interface. We implemented the same functionality using FreeBSD sysctl interface. All statistics can be printed using the following command:

```
# sysctl kstat
```

3.9 Kernel I/O KPI

The configuration of a storage pool is kept on its components, but in addition configuration of all pools is cached in `/etc/zfs/zpool.cache` file. When the pools are added, removed or modified `/etc/zfs/zpool.cache` file is updated. It was not possible to access files from the kernel easly (without using VFS internals), so we created KPI that allows to perform simple operations on files from the kernel. We called the KPI "kernio". Below are the list of operations supported. All functions are equivalents of userland functions, the only difference is that they operate on vnode, not file descriptor.

struct vnode *kio_open(const char *file, int flags, int cmode)

- Opens or creates a file returning a pointer to a vnode related to the file. Returns NULL if file can't be opened or created.

void kio_close(struct vnode *vp)

- Close the file related to the given vnode.

ssize_t kio_pread(struct vnode *vp, void *buf, size_t size, off_t offset)

- Reads data at the given offset. Returns number of bytes read or -1 if the data cannot be read.

ssize_t kio_pwrite(struct vnode *vp, void *buf, size_t size, off_t offset)

- Writes data at the given offset. Returns number of bytes written or -1 if the data cannot be written.

int kio_fstat(struct vnode *vp, struct vattr *vap)

- Obtains informations about the given file. Return 0 on success or error number on failure.

int kio_fsync(struct vnode *vp)

- Causes all modified data and file attributes to be moved to a permanent storage device. Return 0 on success or error number on failure.

int kio_rename(const char *from, const char *to)

- Renames file **from** to a name **to**. Return 0 on success or error number on failure.

int kio_unlink(const char *name)

- Removes file **name**. Return 0 on success or error number on failure.

4 Testing file system correctness

It is very important and very hard to verify that file system works correctly. File system is a very complex beast and there are many corner cases that have to be checked. If testing is not done right, bugs in a file system can lead to applications misbehaviour, system crashes, data corruptions or even security holes. Unfortunately we didn't find freely available file system test suits, that verify POSIX conformance. Because of that, during the ZFS port project the author developed fstest test suite [fstest]. At the time this paper is written, the test suite contains 3438 tests in 184 files and tests the following file system operations: chflags, chmod, chown, link, mkdir, mkfifo, open, rename, rmdir, symlink, truncate, unlink.

5 File system performance

Below we present some performance numbers to compare current ZFS version for FreeBSD with various UFS configurations. All file systems were tested with the `atime` option turned off.
Untaring src.tar archive four times one by one:

UFS	256s
UFS+SU	207s
UFS+gjournal+async	127s
ZFS	237s

Removing four src directories one by one:

UFS	230s
UFS+SU	94s
UFS+gjournal+async	48s
ZFS	97s

Untaring src.tar by four processes in parallel:

UFS	345s
UFS+SU	333s
UFS+gjournal+async	158s
ZFS	199s

Removing four src directories by four processes in parallel:

UFS	364s
UFS+SU	185s
UFS+gjournal+async	111s
ZFS	220s

Executing `dd if=/dev/zero of=/fs/zero bs=1m count=5000`:

UFS	78s
UFS+SU	77s
UFS+gjournal+async	200s
ZFS	111s

6 Status and future directions

6.1 Port status

ZFS port is almost finished. 98% of the whole functionality is already ported. We still need to work on performance. Here are some missing functionalities:

- ACL support. Currently ACL support is not ported. This is more complex problem, because FreeBSD has only support for POSIX.1e ACLs. ZFS implements NFSv4-style ACLs. To be able to port it to FreeBSD, we must add required system calls, teach system utilities how to manage ACLs and prepare procedures on how to convert from one ACL-type to another on copy, etc. (if possible).

- ZFS allows to export file systems over NFS (which is already implemented) and ZVOLs over iSCSI. At this point there is no iSCSI target deamon in the FreeBSD base system, so there is nothing to integrate this functionality with.

- Clean up some parts of the code that were coded temporarily to allow to move forward.

6.2 Future directions

Of course there is a plan to import ZFS into FreeBSD base system, it may be ready for 7.0-RELEASE. There is no plan to merge ZFS to the RELENG_6 branch.

One of the interesting things to try is to add jails [Kamp2000] support to ZFS. On Solaris, ZFS has support for zones [Price] and will be nice to experiment with allowing for ZFS file system creation and administration from within a jail.

FreeBSD UFS file system supports system flags - chflags(2). There is no support for those in the ZFS file system. We consider adding support for system flags to ZFS.

There is no encryption support in the ZFS itself, but there is an ongoing project to implement it. It may be possible to cooperate with SUN developers to help finish this project and to protect portability of the code, so we can easly integrate encryption support with the opencrypto [Leffler2003] framework.

7 Acknowledgments

I'd like to thank ZFS Developers who created this great file system.

I'd like to thank the FreeBSD Foundation [fbsdf] for their support. I'm using machines from the FreeBSD Netperf Cluster [netperf] for my development work. This paper was also first presented at AsiaBSDCon conference, where the FreeBSD Foundation covered my transportation costs.

I'd like to thank Wheel LTD [wheel]. I was able to do the work during my day job.

I'd like to thank the FreeBSD community for their never-ending support and warm words.

References

[McKusick1994] M. McKusick and T. Kowalski, *Fsck - The UNIX File System Check Program*, 1994

[McKusick2002] M. McKusick, *Running 'Fsck' in the Background*, 2002

[McKusick1996] M. McKusick, K. Bostic, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, 1996

[McKusick1999] M. McKusick, G. Ganger, *Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem*, 1999

[Ganger] G. Ganger, M. McKusick, C. Soules, and Y. Patt, *Soft Updates: A Solution to the Metadata Update Problem in File Systems*

[Tweedie2000] S. Tweedie, *EXT3, Journaling Filesystem*, 2000

[Best2000] S. Best, *JFS overview*, 2000

[Sweeney1996] A. Sweeney, *Scalability in the XFS File System*, 1996

[Hitz] D. Hitz, J. Lau, and M. Malcolm, *File System Design for an NFS File Server Appliance*,

[SHA-1] SHA-1 hash function, `http://en.wikipedia.org/wiki/SHA-1`

[fletcher] Fletcher's checksum, `http://en.wikipedia.org/wiki/Fletcher's_checksum`

[Moffat2006] D. Moffat, *ZFS Encryption Project*, 2006

[Bonwick2005] J. Bonwick, *SEEK_HOLE and SEEK_DATA for sparse files*, 2005

[Kamp2000] P. Kamp and R. Watson, *Jails: Confining the omnipotent root*, 2000

[Leffler2003] S. Leffler, *Cryptographic Device Support for FreeBSD*, 2003

[Price] D. Price and A. Tucker, *Solaris Zones: Operating System Support for Consolidating Commercial Workloads*

[fstest] File system test suite, `http://people.freebsd.org/ pjd/fstest/`, 2007

[fbsdf] The FreeBSD Foundation, `http://www.FreeBSDFoundation.org/`

[netperf] FreeBSD Netperf Cluster, `http://www.freebsd.org/projects/netperf/cluster.html`

[wheel] Wheel LTD, `http://www.wheel.pl`