DETTE EKS TILHØRER
DIKUUG.

EuroBSDCon 2004

29. - 31. October 2004
Karlsruhe/Germany

Proceedings

Conference proceedings
EuroBSDCon 2004

Karlsruhe, Germany
29.-31. Oct. 2004

Proceedings of the 3rd European BSD Conference

Programme Committee:
Jan-Hinrich 'Oskar' Fessel
Brian Somers
Sam Smith
Wolfgang Zenker

Published by punkt.de GmbH
Editor: Jürgen Egeling

Some papers did not meet the deadline for inclusion in the printed proceedings. These and the other papers will be published for the foreseeable future on the conference website at http://2004.eurobsdcon.org after the conference.

punkt.de GmbH
Vorholzstr. 25
76137 Karlsruhe
Germany

Tel: +49 721 9109 0
Fax: +49 721 9109 100
Web: http://punkt.de

Contents

Track A Saturday **5**

Using Application-Driven. . . – Antti Kantee 7
NetBSD and handheld platorms – Alistair Crooks, Valeriy Ushakov 17
A portable Packaging System – Alistair Crooks 27
Cross–building packages – Krister Walfridsson 39
Fighting the Lemmings – Martin Husemann 45
But I'm not a developer. . . – Dru Lavigne 55

Track B Saturday **63**

Mac OS X binary compatibilty on NetBSD. . . – Emanuel Dreyfus 65
The flaf filesystem – Soren Jorvang 81
Integrating ALTQ QoS into FreeBsd – Adrian Penisoara 83
Lightwight FreeBSD package cluster in a jail – Dirk Meyer 85
DHCP: Unexplored Capacities – Yannick Cadin 93
A secure BGP implementation – Henniing Brauer 95

Track A Sunday **108**

An Introduction to Sysadmin Training. . . – Hubert Feyrer 109
NetBSD/Desktop: Scalable Workstation Solutions – Jan Schaumann 141
The Challendes of Dynamic Network Interfaces – Brooks Davis 161
NetBSD Status Report – Ignatios Souvatzis 173
A Machine-Independent Port of the SR Language. . . – Ignatios Souvatzis 181
The A–tree — a Simpler, More Efficient B–tree – Alistair Crooks 185

Track B Sunday **202**

Integrating Monitoring Data – Christoph Sold 203
FreeSBIE – Stucchi Massimiliano, Matteo Riondato 213
Using Aspect-Oriented Programming. . . – Michael Engel, Bernd Freisleben 219
Handling FreeBSD's latest firewall semantics. . . – Adrian Penisoara 221

Track A Saturday

Notes:

Using Application-Driven Checkpointing for Hot Spare High Availability

Antti Kantee
<pooka@cubical.fi>

Cubical Solutions Ltd.
http://www.cubical.fi/

ABSTRACT

For critical services downtime is not an option. The downtime of the service can be addressed by replicating the units that provide the service. However, if the session state is important, it is not enough to simply replicate units: sharing the continuously updated internal state of the units must also be made possible. If execution can be continued on another unit after the point-of-failure without any significant loss of state, the unit is said to have a Hot Spare.

Saving the state of a unit so that it can be restored at a later point in time and space is known as checkpointing. For the checkpointing approach to be a viable option in interactive services, it must not disrupt the normal program operation in any way noticeable to the user.

The goal of this work is to present a checkpointing facility which can be used in applications where checkpointing should and can not disrupt normal program operation. To accomplish this, the responsibility of taking a checkpoint is left up the application. The implications are twofold: checkpointing will done at exactly the right time and for exactly the right set of data, but each application must be individually modified to support checkpointing. A framework is provided for the application programmer so that it is possible to concentrate on the important issues when adding Hot Spare capabilities: what to checkpoint and when to checkpoint. Checkpointing efficiency is then further increased by introducing kernel functionality to support incremental checkpoints.

1. Introduction

Hot Spare High Availability support for an application means that if (when) the primary unit fails due to a fault in either software or hardware, a reserve unit will automatically take over the responsibilities of the primary unit. Execution will continue in the reserve unit with no or insignificant loss of internal application state. In a networking context this means that for Hot Spare support to be accomplished, the relevant pieces of the internal application state must be succesfully delivered to the spare units over the network at key points during execution. In addition to delivering the state to a spare unit, the system must have some cluster control mechanism that will take the necessary steps to transfer control to a reserve unit when the current primary unit fails. Once the problems involving saving state and restoring state are solved, the rest is mostly an issue which software professionals tend to call a

SMOP[1]. Therefore, the bulk of this work will concentrate on discussing the ideas involving saving and restoring process state.

There are already plenty of good examples on fault tolerance in the world of computing. A popular example from the world of hardware is disk RAID. Certain RAID levels provide protection against data (state) loss in the case of unit failure. This is what we are looking for. However, in the world of software it is difficult to keep state across unit crashes. Therefore, most entry-level solutions for fault tolerance only include support for replacing the broken processing unit, and give the problem of keeping state less attention or outright ignore it. If the unit state at failure-time is ignored, it is not possible to provide a seamless user experience across points of failure.

The act of capturing a process state for later restoration is known as checkpointing. Traditionally checkpointing has played big role in scientific computation, where the requirements for checkpointing efficiency and application interruption have not been high on the list. If checkpointing is to be used in environments where pausing the application for abitrary periods is not acceptable, new techniques must be developed.

One of the reasons for the low efficiency of the methods mentioned above is that they are implemented below the application level and transparent to the application. While this means that the application does not need to worry about checkpointing, it also means that checkpointing cannot reach maximum efficiency, since the checkpointing facility does not know about the semantic behaviour of the application. This problem is magnified if the application is not "self-contained", i.e. it communicates with the outside world. The solution is to provide a checkpointing framework for the application, and then modify each individual application to use that framework. This way checkpointing effiency and accuracy can be increased to an acceptable level.

In Chapter 2 of this work I will concentrate on defining Application-Driven Checkpointing, and giving a general overview of the architecture. Chapter 3 discusses adapting a simple open-source application to the framework. Efficiency of the framework is discussed in Chapter 4, and the story closes with conclusions in Chapter 5. This paper will present the architecture very briefly. For a more through discussion on the subject, the interested reader is invited to look at my Master's Thesis [1].

2. Application-Driven Checkpointing

First of all, it should be noted that there are several components in a process checkpoint, and they can be divided in different ways [2]. However, I wish to define a simple division and only separate process *data* and *metadata*. Data involves memory used by application. This memory is reserved from the heap, memory reserved from the stack does not count as data in this definition (neither does it count as metadata). Metadata is all the other state related to the process, such as structures describing open files and existing threads. Usually most information involving metadata is hidden from the application e.g. in the kernel.

To record the processor physical state[2], the usual approach is to simply save the register contents for example by taking the core dump. However, we can observe that most programs are structured so that they have specific worker loops. An example of such a worker loop is a function (perhaps in its own thread) reading input from a network socket and processing it. In the usual case it suffices to record the informa-

[1] Simple Matter Of Programming

[2] By this I mean the register contents, and am less interested in the actual electrons running around.

Checkpointing Kernel Interface

```
struct cpt_range {
        void   *addr;
        size_t len;
};

pid_t   cptfork(void);
ssize_t cptctl(struct cpt_range *ranges,
               size_t nranges, int op);
```

tion that the program had a worker loop, and do a normal function call into the worker loop (with the correct input data, of course) during restoration.

The above strategy also takes care of the difficulties in checkpointing threaded programs [3], where great care must be taken to avoid other threads entering a bad state. Other threads could, for example, make a syscall between the timeframe the checkpointing thread decides to checkpoint and actually makes the checkpoint. A straightforward restoration from this checkpoint would cause invalid return values from the kernel[3]. Solutions such as suspending all threads for checkpointing have a fair performance hit, especially if checkpointing is to be attempted often for increased checkpoint granularity.

2.1. Architectural details

The implementation was carried out on two different levels. Most of the work is done by a userspace library (Hot Spare Library), but of course the work done by the kernel components is implemented inside the kernel. Ultimately the application does not need the Hot Spare Library at all, and could make the respective calls itself, but the idea was to make the application programmer be able to concentrate on the critical issues and get as much support from the system as possible.

The user library deals with issues related to capturing and restoring the application metadata, reserving checkpoint-safe memory, and also providing a grand unifying interface, `hs_cpt()`, for taking a checkpoint.

The kernel side takes care of providing cheap, atomic and asynchronous (from the point-of-view of the calling thread and application) snapshots of the memory area.

2.2. Checkpointing data

On UNIX®systems, the *fork*() system call creates a process, which is almost an exact duplicate of the calling process the only main difference being the process ID number. Historically, the *fork*() call really did copy the entire address space of a process when executed. However, this was mostly wasteful, since *fork*() is frequently used in conjugation with the *exec*() system call, which replaces the entire address space with a binary image from the disk. Therefore a technique called copy-on-write, or COW for short, was employed in AT&T System V UNIX. The copy-on-write property of *fork*() is close to what we are looking for: it will give us both asynchronous checkpointing ability and an atomic snapshot of the checkpoint-range.

Incremental checkpointing means saving only the portions changed from the previous checkpoint, and provides a cheap performance boost in nearly every imaginable case. Therefore it is desireable to

[3] No, there are no actual returned values in this case. That's why they are funny.

implement support for incremental checkpoints. Most userspace solutions employ *mprotect*() to deny writing to critical areas and track modification information with the help of a SIGSEGV handler [4]. However, since we are allowed to play in kernel land, it is possible to avoid jumping between the kernel and userspace to track modification information. Modification information can be tracked for example in the copy-on-write fault handler or my asking the MMU. Tracking modifications in the fault handler would have its advantages, but since the latter was easier to implement[4], it was done for this work.

Kernel interface for checkpointing data

We must modify the kernel and VM [5] to support three different operations for incremental checkpointing to be possible:

- Add and remove memory areas which contains checkpoint data.
- Take the checkpoint itself.
- Ask the kernel which pages of memory in checkpoint areas have been modified since the previous checkpoint.

The call sequency for the application (or actually a programming library) which wishes to use the interface is approximately the following:

1. Decide which memory areas contain data critical enough to be worth checkpointing. Add those memory areas.
2. Decide it is time to checkpoint. Make the checkpointing syscall.

2½. The parent process from *cptfork*() continues execution as normal, and makes all the changes it wants to the checkpoint memory areas. They are "protected" by copy-on-write.

3. The child process queries the kernel for modified areas.
4. The child process writes the changed memory areas (along with other checkpoint data, we'll get to that soon) to back storage using its method of choice, e.g. write to file or TCP socket.
5. The child process exits.

2.3. Checkpointing metadata

For checkpointing metadata a slightly different approach was taken. Most of the information related to process metadata is hidden in the kernel away from the application. While we could simply add a kernel interface to extract the in-kernel information for the Hot Spare Library to trasmit over the network, it would not be a good idea. The kernel structures, such as vnodes [6], are very integrally linked to each other. Attempting to extract and restore them as opaque data is not possible without creating a huge mess. To grasp the concept of checkpointing metadata, thinking of Java Serializable [7] or Python Pickle [8] may help.

I will go over one example of capturing and restoring process metadata. The rest of the descriptions are available in my thesis [1].

Checkpointing file descriptors

There are several different type of file descriptors: normal files, pipes, sockets, crypto descriptors, and so forth. Not all of them are supported. The serialization information depends entirely on the type of file descriptor we wish to serialize. For example, for a file the important facts are the filename used to open the descriptor, the mode it was opened in, and the current seek offset into the file. None of that information applies to a networking socket and we must provide other routines for it.

The information related to file descriptors is not static: for example file offset will constantly change if the file is accessed. Therefore the library provides an

[4] At least for platforms which have an MMU that keeps this information. The SPARCv9 MMU for example does not.

option to "refresh" the information related to a file descriptor during each checkpoint by asking the kernel. For a normal file this would consitute of calling *lseek*(), while for networking sockets it would most likely be a matter of *getpeername*() and *getsockname*(). As there is a minor cost-penalty for doing this, it is not done for all file descriptors, but rather the choice of which descriptors are critical in this respect is left up to the application programmer.

Of course there is one huge downside to querying the information at checkpoint-time: since the entity doing the checkpoint and the application itself are not (necessarily) synchronized, the state that gets written into the checkpoint does not necessarily reflect the state present in the memory dump. The application programmer is encouraged to very carefully think how important the exact file descriptor state is, and possibly even take steps to record the state in the lock-protected checkpoint-area, where it will be guaranteed to be correct. However, doing so will probably open a whole other can-of-worms™, and currently there is no easy solution to the problem.

3. Adapting the framework

Adapting the checkpointing framework to the all-important game Tetris is presented next[5] [6]. While the loss of a Tetris score may not be the most tragic episode that has hit human history, migrating the Tetris game to a nearby system when the original gets (literally) axed makes for a powerful visual effect.

Tetris from the BSDgames package is a fairly small program. The version against which this discussion is written can be found from the NetBSD CVS Repository in *src/games/tetris* with the tag `netbsd-1-6-PATCH002`. It constitutes of less than 2000 lines of code.

The state of the Tetris game can be broken into the following elements:

- score
- current piece
- next piece
- state (of pieces already placed) on the board

There are two good choices for checkpointing places: at the beginning of each cycle when a new piece appears at the top, or each time a piece moves. The latter option introduces much overhead into the game, and the former would be a natural choice. But since it can be argued that the latter is "better" (better granularity), and it does not kill performance, it was chosen.

The worker loop

The main loop of Tetris does practically everything from user input monitoring to moving the piece to checking if the piece fits to bumping the score. Therefore it makes a very good candidate to be registered as the worker function. The only thing we need to do is take the loop out of *main*(), and place it into its own function. This is done because we need to call the main loop directly if we wish to do a restore from a checkpointed situation. If the program would go through *main*() also when restoring from a checkpoint, it would initialize its runtime state to zero, and defeat any purpose of Hot Spare checkpointing.

In addition to moving the main loop into the worker function, we also move some screen-related initialization there. This is done because we need to set up the screen also on the spare if the program

[5] Creating a clustered Tetris solution was suggested by Marcin Dobrucki, obviously as a joke, but he should be more careful around humor impaired people.

[6] NetHack was of course considered, but since it, as most games, always comes with its own application-driven checkpointing mechanism (savegames), the redundancy did not seem worth the effort.

execution is handed over. Normally the Hot Spare Library provides routines for all necessary state-saving functionality, but since it was written with daemons, not interactive applications, in mind, it does not provide routines to save screen state. Nonetheless, this serves as an example of the fact that when the Hot Spare Library does not provide the necessary routines, it is possible for the application to define them in its own domain.

Finally, the code that takes care of returning screen setup to a sane state needs to be moved into the worker function after the main loop. The spare program has no knowledge it should fall back to *main*(), since the worker function was called directly from the Hot Spare Library, and will exit after returning from the worker function.

Saving state

Since this version of Tetris was written in the early 90's, it was written like most programs of old: state is kept in the data segment as global variables. This is unacceptable for us, since we need to store critical data in areas which will be included in the checkpoint.

The task of moving the information from the data segment to checkpoint-safe memory is a fairly simple one: we simply "collect" the state from global scope in the source module `tetris.c`, and create `struct tetstate`, in which all the variables essential to the state are placed. This structure is added to the checkpoint memory area when Tetris is initially started. All the references to the state variables must be fixed to point inside the checkpoint-safe structure. It can be accomplished either by using cheap tricks with the preprocessor (`#define`) or by a simple search-and-replace operation with a text editor or shell utility. Most of the time taking the effort to do an actual search-and-replace pays off and avoids unwanted and weird side-effects, although the bulk of the differences may then amount to changes in variable referencing.

Normally multithreaded programs avoid using global state and pass the context of the call as a parameter. In this case the program state will most likely already be readily contained, and no modifications such as with Tetris and other older non-threaded programs should be required.

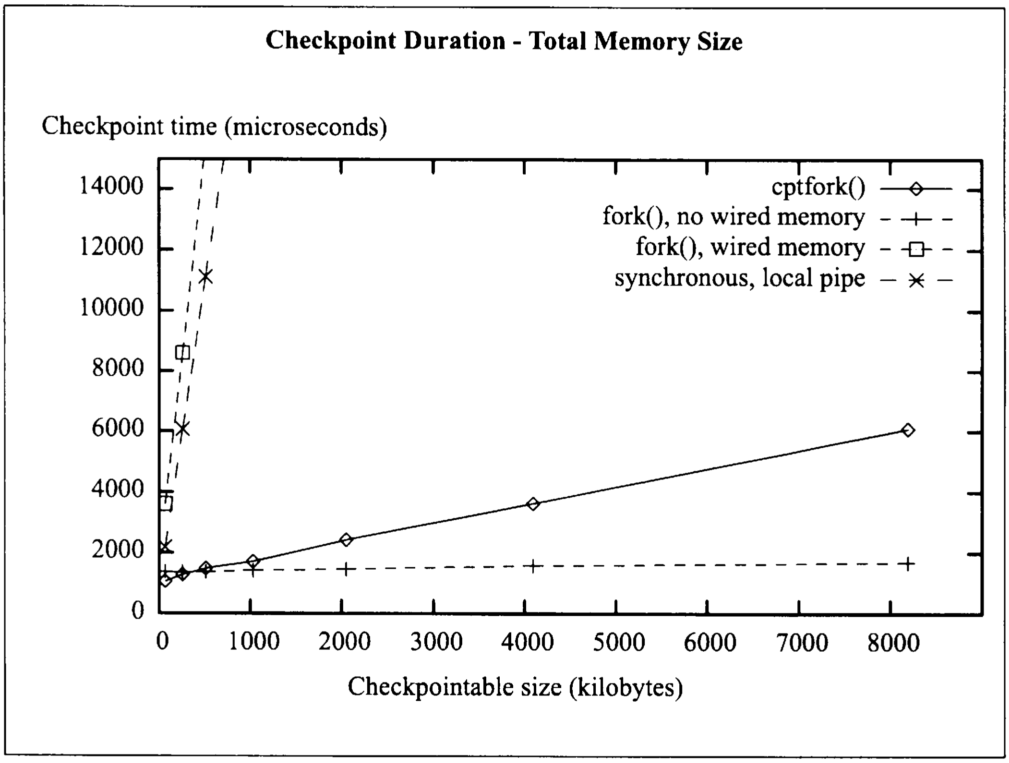
In addition to the memory and worker "thread" state, the game registers a few signal handlers. Although they could be registered via the *hs_sigreg*() facility, they are an integral part of the screen setup code. Since we run that code anyway, the signals get proper treatment even without explicitly including them in the checkpoint.

Conclusions

Adapting Tetris from the BSDgames package for application-driven checkpointing was a simple job. It was accomplished in just a few hours time after first looking at the source code. The factors that amounted to the ease of checkpointing adaption were the limited size and instantly clear intuition on what to checkpoint. The non-threaded programming approach and consequent lack of state grouping were the only difficulties encountered.

4. Performance

In this chapter I present some key benchmarks. Since we are interested in the performance of the checkpointing module and less interested in the operating system and network performance, the checkpointing process does not transmit the checkpoints anywhere for restoration. Checkpoint data is simply written into `/dev/null`. All the tests were run on a 300MHz AMD K6-2. It is not "current" technology, but this work is not targeted for any specific machine, so it is a safe choice.



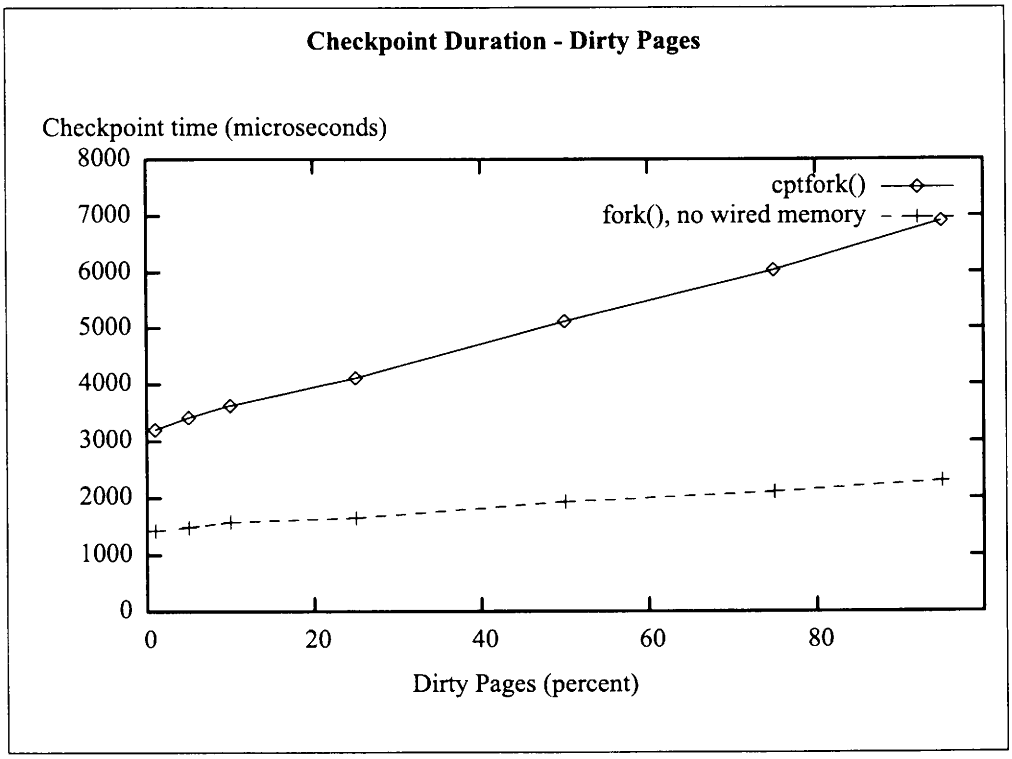
The first test examines how the checkpointing time from the application point-of-view is influenced by the amount of checkpoint-safe memory registered. This amounts to the time in between calling *hs_cpt*() and returning from the function. Between checkpoints the parent modifies 10% in sets of four contiguous pages and sleeps for one second.

For taking a synchronous checkpoint in application context the system was modified somewhat. Writing the checkpoint to `/dev/null` also in the case of a synchronous checkpoint would be unfair, since transfer speed to the spare is the limiting factor. For application context synchronous checkpoints the preferred way is getting the checkpoint contents as quick as possible somewhere else, so that the application can continue with its normal tasks. For benchmarking purposes I opted to write to a local pipe, since it is faster than transmitting the data over the network. In this case the other end of the pipe just reads data to empty the pipe buffer, but in real life it would naturally also take care of making sure the data reaches the spare units.

The results are what was expected. When dealing with wired pages, plain *fork*() is hideously expensive. This is because it copies all wired memory to the child process. As you can see, the results go "off the scale" fairly early.

Without wiring pages *fork*() is the cheapest alternative from the application point-of-view. It starts out slightly heavier base cost than *cptfork*(), but quickly catches up and follows an almost constant trend after that. The difference in base cost can be accounted to the fact that *fork*() marks all regions copy-on-write, while *cptfork*() shares most of them. However, the price to pay for doing a full asynchronous checkpoint with *fork*() is of course the amount of data to be sent over to the spare.

The cost of doing *cptfork*() is very close to linear with an added base cost for



doing common tasks required when *fork*()ing. The linear cost can be explained by having to go through all pages marked checkpoint-safe and checking them for modification information before allowing the parent of the *cptfork*()ing process to return.

Taking a synchronous checkpoint by writing to a pipe starts out about as cheap as the non-wired *fork*()ing alternatives, but exhibits high costs when the checkpoint size is even slightly increased.

Varying Amount of Dirty Pages

To see how the amount of dirty pages affects checkpointing cost from the application perspective, a test which modifies a varying number of pages was run. The test reserved 4MB of memory and did modifications in sets of four contiguous pages. In the case where 95% of the pages were modified, 25 contiguous pages were used instead to make the test runnable. The main purpose of this test was to see if it becomes clearly cheaper to take a full checkpoint instead of using the *cptfork*() approach at some point.

The asynchronous approach exhibits a clearly linear trend in addition to the *cptfork*() base-cost. The same can be said about normal *fork*(), except that the linear coefficient is much smaller in the latter case. I am not totally sure where the linear coefficient comes from, but my educated guess is that accessing pages influeces various caches in the system. The *cptfork*() case takes more performance penalty from this, because it does more lookups than a normal *fork*(). If nearly all pages are modified, *cptfork*() is 5ms slower than plain *fork*(). This difference is significant, since the longer the checkpoint memory area is locked, the longer other threads can be blocked[7].

[7] Checkpoints are taken with important areas locked by the application to avoid them being in a "bad state" in the checkpoint.

Analysis of Results

Ultimately we wish to know which approach is the cheapest for each given situation. It is clear that application context checkpointing is not worthwhile unless there is extremely little data to checkpoint, perhaps only a page of memory or so[8]. Once the checkpoint size gets into the range of tens of kilobytes and beyond, asynchronous checkpoints stall the application for much less.

While doing full asynchoronous checkpoints employing *fork*() is a win from the point-of-view of the application, that is only half of the truth. The cost of transferring the checkpoint to spare units becomes a huge factor for applications which wish to register a myriad of memory, but only modify it seldom. This limits the granularity of full checkpoints. Available bandwidth will most likely be saturated by information which remains the same from one checkpoint to another.

Looking at all the graphs presented in this chapter, it is clear that *cptfork*() is the most performant alternative as long as there is enough memory in the checkpoint range, and if not too big a portion of that memory space is modified in between checkpoints. After reaching a high enough modification percentage a full checkpoint becomes cheaper. Unfortunately we do not know the amount of dirty pages before making the decision to checkpoint using *cptfork*(). After taking a hit from the overhead of doing *cptfork*(), it is too late to change our mind.

We could address the problem presented in the previous paragraph by recording page modification information already when the page is modified. Since our checkpointable memory ranges are marked copy-on-write, the operating system takes page faults to copy pages which are being modified. In addition to gaining knowledge on modification statistics before making any expensive decisions such as *cptfork*(), there would be other benefits.

- There would be no need to do a lookup for all the pages in checkpoint memory ranges during *cptfork*(), as the modification information could be already recorded in a simple form, such as a bitmap. This would effectively cut down the checkpoint-time from O(total_pages) to O(pages_modified).
- The scheme would also work on platforms which do not have page modification information in their MMU.

5. Conclusions

This work set out to investigate the possibility of using a checkpointing approach for Hot Spare High Availability in environments where the application is time-critical and freezing it for an arbitrary period during execution for taking the checkpoint is not acceptable.

The key idea in the approach was to make checkpointing the responsibility of the application, since it best knows what it is doing with its state as opposed to an external facility, which must treat all data as opaque. The efficiency of the architecture was enhanced by adding a kernel component, which serves the application-level library by providing information on which pieces (memory pages) have changed since the last checkpoint.

If the application itself contains vast amounts of redundant state, using application-guided checkpointing to carve out the necessary bits will increase performance dramatically. Incremental checkpointing will enhance performance more and more as the ratio of modifications between checkpoints to the entire checkpointable memory area decreases.

[8] Assuming of course small, kilobyte-sized pages. Megabyte-sized "large pages" are right out.

The Hot Spare Library was written to be both portable and flexible. It provides most of the functionality necessary for standard applications, but since checkpointing is application-driven, the application itself is free to handle anything else it needs to checkpoint.

The biggest part of the work for someone who wishes to use an application-driven scheme is of course adapting the application. It was shown that for a small application the work was just a matter of hours. For a large application, the time depends greatly on how familiar one is with the application before starting the modification task, and how the application was written. The task varies from "trivial" to "impossible without rewriting the entire application", and it is impossible to give an accurate estimate without knowing the particular application.

This work did not address the problem that unfortunately makes the approach invalid for most network services: migrating applications which depend on a persistent TCP connection is not possible[9]. There are two ways to fix the problem: either teach the application and protocol that the connection may be broken if migration takes place, or modify TCP on both endpoints to cope with migration. Unfortunately, neither approach is non-intrusive from several perspectives, and the modifications are far from trivial, either logistically or technically.

As a concluding remark it can be said that the application-driven approach was found to be a working one, and under the right circumstances and right software it can be an extremely attractive option for providing Hot Spare High Availability.

[9] I do not know if it is any condolence that the TCP problem makes just about any checkpointing approach inapplicable.

References

1. Antti Kantee, *Using Application-Driven Checkpointing Logic for Hot Spare High Availability,* Master's Thesis, Helsinki University of Technology (September 2004).
2. Yi-Min Wang, Yennum Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala, *Checkpointing and Its Applications,* pp. 22-31, 25th International Symposium on Fault-Tolerant Computing (June 1995).
3. William R. Dieter and James E. Lumpp, Jr., *A User-level Checkpointing Library for POSIX Threads Programs* (1999).
4. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, *Libckpt: Transparent Checkpointing under Unix,* Winter Usenix Conference (January 1995).
5. C. Cranor, *Design and Implementation of the UVM Virtual Memory System,* PhD thesis, Washington University (August 1998).
6. S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX,* pp. 238-247, Summer Usenix Conference, Atlanta, GA (1986).
7. Sun Microsystems, "java.io Interface Serializable," *Java2 Platform, Standard Edition, v1.4.2 API Specification.*
8. Guido van Rossum and Fred L. Drake, Jr., "pickle -- Python object serialization," *Python Library Reference.*

NetBSD and handheld platforms

Valeriy Ushakov <uwe@NetBSD.org>
Alistair Crooks <agc@NetBSD.org>
The NetBSD Project

Abstract

NetBSD has long been known for its portability to other platforms. The lower end of this range of platforms has included thin clients, and, most recently, handheld PCs. These machines are typically powered by a low-power CPU, and have smaller LCD screens than a laptop, and consequently dissipate less power. These machines usually use a touch screen and a stylus for a pointing device, can run for nearly a day on battery power only, and come with some version of Windows in ROM for an operating system.

This paper discusses a number of issues related to porting NetBSD to handheld PCs – the challenges, and the parts which are needed, including the subsystems still to be written. It contrasts the various handheld PC devices on the market, and looks at Linux support for these devices, as well as Windows and NetBSD. It also looks at the challenges of handheld and pocket PCs, from fitting a graphical browser onto a limited size LCD screen, to window managers designed for mouse-equipped environments, to user interface issues when no keyboard is present.

Finally, it looks into the future, and discusses what the ultimate geek gadget PDA would be.

Introduction

The NetBSD operating system has been ported to a large number of processor families and architectures: computers built around those CPUs range from high-end servers to small computers like laptops, notebooks, and, now, PDAs, handheld PCs (H/PC) and pocket PCs (PPC).

There are two aspects to porting NetBSD to a new platform:

- Porting to the new platform itself. This is occasionally referred to as the "Because it's there" question. There are technical challenges to bringing up an operating system on a new platform, and these are often the things that spur developers to take this lonely road.
- Once NetBSD has been ported to a new platform, the next challenge is to run NetBSD on that platform, from porting third-party applications which may only have been written with Linux and IA32/i386 architecture in mind, to the pure delight of debugging some of those same applications. However, this is also the stage in which the proselytizing of the platform, of NetBSD, and of the combination takes place.

At various conferences and trade shows, the handheld platforms have been the ultimate "geek gadget", which has garnered a large amount of interest, especially for its size. Of course, the size, or lack of it, may be the reason for the interest – being able to use on-line services such as IRC and electronic mail from a handheld computer with a wireless card is a tremendous selling point, both for the platform and for NetBSD.

Target Platforms

All of the supported platforms are originally designed to run some version of Microsoft Windows CE. Most future platforms are likely be Windows CE based as well.[1]

Terminology

Microsoft terminology in this area and their different version numbering schemes are very confusing. The primary distinction that matters for our paper is:

- H/PC – Handheld PC. Keyboard, half or full size VGA display.
- PPC – Pocket PC. No built-in keyboard, quarter VGA display.

In general, we shall use the term "Personal Digital Assistants" (PDAs) throughout this paper to cover both HPC and PPC.

Lack of built-in keyboard might prove to be a big problem for standalone interactive use, unless something like QTopia or GPE is ported. However even PPC can be useful, perhaps in combination with a third-party package like xscribble (pkgsrc/x11/xscribble) which allows a user of a touch screen to input characters into X11 applications, using a uni-stroke (graffiti-like) alphabet – it uses the Xtest extension to allow synthesis of characters as though they had been typed on a keyboard.

Uses

HPCs are very useful as a mobile Unix terminal – they provide all the nice features of "big" NetBSD: IPv6, IPSEC, WiFi, SSH, etc... E.g., one of the authors used his Jornada for aiming a polarized WiFi aerial on a building's rooftop. Laptop is just too bulky for that. PPCs, unfortunately, are not usable for this currently (need QTopia/GPE).

PDAs can be used as a development platform – if you develop software for an embedded platform with limited capabilities you can use NetBSD host as a development, debugging, and testing machine. This use is probably most important for SuperH-based PDAs, as "bigger" ARM and MIPS boxes are more easily available. Even for ARM and MIPS it might be cheaper to buy a handheld then a big machine. Even keyboardless PPC are suitable for this purpose.

These handhelds can be used as a PDA, bereft of Microsoft Office software (although Windows CE Office components tend to be simplified and feature-lacking versions of the traditional Microsoft Office components). Calendar, diary/agenda, spreadsheet, presentation graphics, etc can all be done by using open source equivalents. Insert a broad hint to pkgsrc folks here :)

As a musical juke box, an alternative to an iPod or a Creative Nomad, for playing MP3 or ogg files, again using traditional open source software to achieve this (it should be noted that some of the versions of Windows Media Player on PDAs are burned into ROM, and so upgrades are not usually an option – and early WMP have problems with VBR on some MP3 tracks).

By using the onboard Infra-red port, or USB 1.1 client, or onboard modem, the PDA can be used as a GPS receiver, as well as having all of the benefits above.

Booting NetBSD

There are various ways which different operating systems can boot on these devices. Usually, they will involve installing the alternative operating system onto a secondary storage

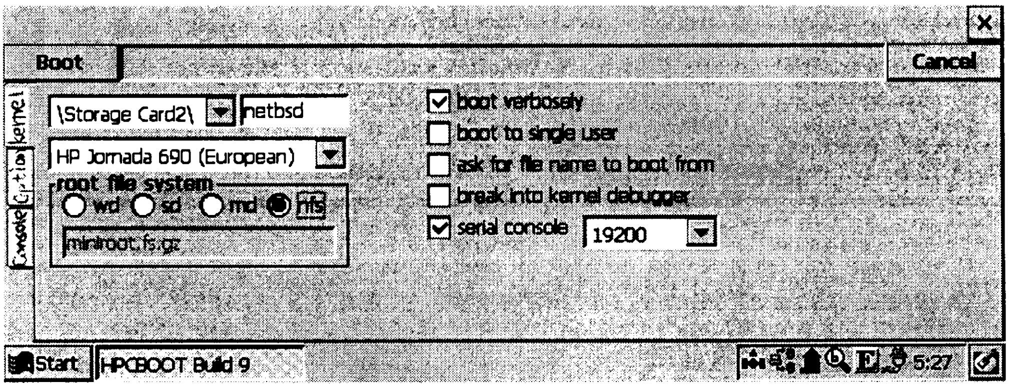
[1] There's also Symbian OS, but devices based on it are not as widespread. Even the recent sub-notebook from Psion runs Windows CE.NET. We will not discuss Symbian based devices in this paper.

medium, such as a Compact Flash (CF) card - these are easy to find, have a large capacity, and are inexpensive; prices have fallen over the last two years.

Typically, HPC and PPC machines have ROM, where the "native" operating system will reside – this is typically Windows CE. Windows CE executes directly from ROM, thereby leaving all available RAM free for applications and data. It is only really "booted" after a hard reset. When suspended, it simply powers down all the devices and only spends a tiny bit of power on refreshing the RAM.

Some of these machines, such as the iPAQ, have flashable ROMs, whilst others have non-rewritable ROMs (such as the Jornada 600 and 700 series). Linux takes advantage of rewritable ROMs, and so it's possible to flash Linux into the ROM on an iPAQ. At the present time, NetBSD does not take advantage of this.

The NetBSD boot loader is, therefore, a separate program, and typically resides on Compact Flash card or some other method of persistent storage, accessed by the native operating system. This boot loader is then executed under the native operating system. The following screenshots show hpcboot, a Windows CE program, being used to boot NetBSD on a Jornada 690.



\Storage Card2\ netbsd

HP Jornada 690 (European)

root file system
○ wd ○ sd ○ md ● nfs

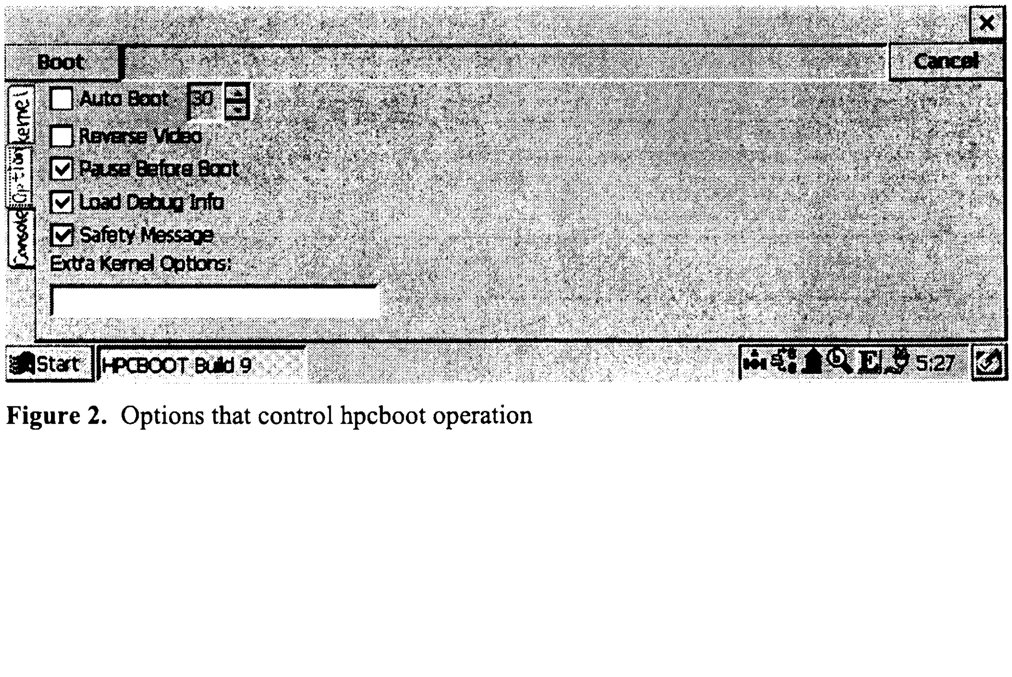
miniroot.fs.gz

☑ boot verbosely
□ boot to single user
□ ask for file name to boot from
□ break into kernel debugger
☑ serial console 19200

Figure 1. Selecting the kernel to boot and boot options

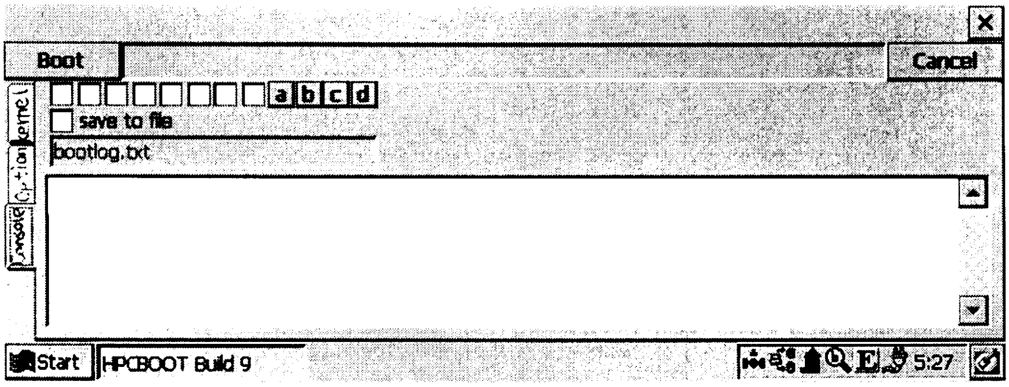
The figure above shows the "Kernel" tab of the hpcboot GUI, where the machine model is selected, the kernel to boot and the type of the root file system are specified, and boot options that can be entered.

The next figure shows the "Option" dialog, that controls hpcboot operation. The options selected on this screenshot are good defaults.



□ Auto Boot 30
□ Reverse Video
☑ Pause Before Boot
☑ Load Debug Info
☑ Safety Message
Extra Kernel Options:

Figure 2. Options that control hpcboot operation



bootlog.txt

Figure 3. Output from hpcboot and debugging options

Finally, the last tab, "Console", features a large text area where hpcboot reports gory details of its progress. There are also some anonymous buttons and checkboxes that are intended to be used for hpcboot debugging.

When NetBSD boots the Windows CE in your ROM is safe, however all your data, installed programs, etc will be lost. When NetBSD is shut down, Windows CE boots back as if after hard reset.

An arrangement that one of the authors finds convenient is as follows:

- Partition the CF card into an MSDOS partition, and a NetBSD partition (you need to do this on a Unix host). Your NetBSD installation will reside in the NetBSD partition of the CF. More on this later.
- After a hard reset (you have already tried to boot NetBSD and lost all your data on the device already, haven't you? ;), install all the Windows CE programs and drivers that you need, configure the system to suit your needs (e.g. dialup settings), then do a full backup. Put that full backup onto the DOS partition of your CF card. Saving another backup copy somewhere else is a good idea as well. Add your PIM (personal information manager) data, and then do a PIM-only backup. Put the PIM backup onto the DOS partition of your CF card as well.

  Make sure that you use the Windows CE backup program that is in your ROM! When NetBSD is shut down, the Windows CE comes up from hard reset and no fancy add-on backup programs that you have installed are available.

- Put the hpcboot.exe onto the DOS partition of the CF – you will need it to boot the NetBSD.
- And the last piece you need on the DOS partition is the NetBSD kernel. Hpcboot can boot kernels from the UFS partition, but having the kernel on the DOS partition is convenient when you want to update it, or try out a new kernel – you can write new kernel to the CF from any operating system, including the Windows CE itself.

So when planning the CF partitioning, you should take into account the space requirements for the things listed above. It is prudent to make a backup in advance to see how much space is used. It is advisable to have space for at least a couple of kernels and a miniroot image, and to reserve some space for files that you want to move between Windows and NetBSD. A partition of 16MB should be sufficient.

Installing NetBSD

So how would you install the NetBSD to the CF card? The standard NetBSD system installer, sysinst, is not yet supported on HPC platforms. Adding HPC support to sysinst is really a SMOP, but nobody has done the legwork yet. Thus, currently the easiest way is to use another NetBSD host to do the installation. Hint: your HPC booted with root file system on NFS qualifies!

If you have an i386 laptop running NetBSD you can connect your CF card to it using either a USB flash card reader, or a CF to PCMCIA adaptor.

Before we proceed with a detailed walkthrough, here is the outline of the process:

- Partition the CF card into DOS and NetBSD partitions (see previous section).
- Disklabel the NetBSD partition.
- Create DOS and UFS file systems.
- Mount both partitions.
- Transfer hpcboot and kernel to the DOS file system.
- Extract installation sets to the NetBSD file system.
- Edit several files in `/etc` to pre-configure your system.
- Move the CF to the HPC and boot!

Installation Walkthrough

This installation walkthrough is based on the transcript taken during a from-scratch installation of NetBSD/hpcsh.

Here is the CF card that we will do our sample installation onto:

```
wdc0 at pcmcia0 function 0
atabus2 at wdc0 channel 0
wd1 at atabus2 drive 0: <Transcend 256M>
wd1: drive supports 1-sector PIO transfers, LBA addressing
wd1: 244 MB, 978 cyl, 16 head, 32 sec, 512 bytes/sect x 500736 sectors
wd1: drive supports PIO mode 4
```

We start by splitting the CF card into a small DOS partition and a NetBSD partition. It's important that the DOS partition comes first. Windows get very confused otherwise.

```
# fdisk -u wd1
Disk: /dev/rwd1d
NetBSD disklabel disk geometry:
cylinders: 978, heads: 16, sectors/track: 32 (512 sectors/cylinder)
total sectors: 500736

BIOS disk geometry:
cylinders: 978, heads: 16, sectors/track: 32 (512 sectors/cylinder)
total sectors: 500736

Do you want to change our idea of what BIOS thinks? [n] <enter>

Partition table:
0: Primary 'big' DOS, 16-bit FAT (> 32MB) (sysid 6)
    start 32, size 500192 (244 MB, Cyls 0-977), Active
1: <UNUSED>
2: <UNUSED>
3: <UNUSED>
Which partition do you want to change?: [none] 0
The data for partition 0 is:
Primary 'big' DOS, 16-bit FAT (> 32MB) (sysid 6)
    start 32, size 500192 (244 MB, Cyls 0-977), Active
sysid: [0..255 default: 6] 1
start: [0..978cyl default: 32, 0cyl, 0MB] <enter>
size: [0..978cyl default: 500192, 977cyl, 244MB] 14MB
```

```
bootmenu: [] <enter>
The bootselect code is not installed, do you want to install it now? [n] <enter>

Partition table:
0: Primary DOS with 12 bit FAT (sysid 1)
    start 32, size 28640 (14 MB, Cyls 0-56), Active
1: <UNUSED>
2: <UNUSED>
3: <UNUSED>
Which partition do you want to change?: [none] 1
The data for partition 1 is:
<UNUSED>
sysid: [0..255 default: 169]  <enter>
start: [0..978cyl default: 28672, 56cyl, 14MB] <enter>
size: [0..922cyl default: 472064, 922cyl, 231MB] <enter>
bootmenu: [] <enter>
The bootselect code is not installed, do you want to install it now? [n] <enter>

Partition table:
0: Primary DOS with 12 bit FAT (sysid 1)
    start 32, size 28640 (14 MB, Cyls 0-56), Active
1: NetBSD (sysid 169)
    start 28672, size 472064 (231 MB, Cyls 56-978)
2: <UNUSED>
3: <UNUSED>
Which partition do you want to change?: [none] <enter>

We haven't written the MBR back to disk yet.  This is your last chance.
Partition table:
0: Primary DOS with 12 bit FAT (sysid 1)
    start 32, size 28640 (14 MB, Cyls 0-56), Active
1: NetBSD (sysid 169)
    start 28672, size 472064 (231 MB, Cyls 56-978)
2: <UNUSED>
3: <UNUSED>
Should we write new partition table? [n] y
```

Now that partitions are ready we need to edit the NetBSD disklabel. We start by using mbrlabel(8) that can update a NetBSD disk label from the Master Boot Record (MBR) label.

```
# mbrlabel wd1
Found MSDOS partition; size 28640 (13 MB), offset 32
  skipping existing MSDOS partition at slot e.
Found 4.2BSD partition; size 472064 (230 MB), offset 28672
  skipping existing unused partition at slot c.

6 partitions:
#        size    offset     fstype  [fsize bsize cpg/sgs]
 c:    472064     28672     unused       0     0        # (Cyl.     56 -    977)
 d:    500736         0     unused       0     0        # (Cyl.      0 -    977)
 e:     28640        32      MSDOS                      # (Cyl.      0*-     55)
 f:    472064     28672     4.2BSD       0     0     0  # (Cyl.     56 -    977)

Not updating disk label.
```

As we can see the disklabel is mostly complete, except that the NetBSD partition is assigned to partition ‘f’. It doesn't make sense to create several UFS partitions (in the disklabel sense) within the NetBSD MBR partition. In particular, note that we do not create any swap partitions, as swapping to a CF card will wear it very fast. Let’s change NetBSD partition letter to ‘a’, as the kernel likes it better that way:

```
# disklabel -e wd1
[rename partition f->a]
```

```
# disklabel wd1
[...]
6 partitions:
#        size    offset     fstype [fsize bsize cpg/sgs]
 a:    472064     28672     4.2BSD      0     0     0  # (Cyl.    56 -    977)
 c:    472064     28672     unused      0     0        # (Cyl.    56 -    977)
 d:    500736         0     unused      0     0        # (Cyl.     0 -    977)
 e:     28640        32      MSDOS                     # (Cyl.     0*-     55)
```

Now we format the two partitions we have just created:

```
# newfs_msdos wd1e
/dev/rwd1e: 28584 sectors in 3573 FAT12 clusters (4096 bytes/cluster)
MBR type: 1
bps=512 spc=8 res=1 nft=2 rde=512 sec=28640 mid=0xf8 spf=11 spt=32 hds=16 hid=32
# newfs wd1a
/dev/rwd1a: 230.5MB (472064 sectors) block size 8192, fragment size 1024
        using 6 cylinder groups of 38.42MB, 4918 blks, 9472 inodes.
super-block backups (for fsck -b #) at:
    [...]
```

Now the CF card is formatted, but is still completely empty. We will mount the freshly-formatted file systems and will start filling them with contents. In the examples below "..." stands for the directory with release sets (the directory you specified as an argument to the `-R` flag of the `build.sh` script if you did the build yourself). The examples below uses "hpcsh".

Important: make sure you mount the DOS file system with the `-l` option (see BUGS in mount_msdos(8)).

```
# mount -o softdep /dev/wd1a /mnt
# mount -o -l /dev/wd1e /mnt2
```

We will start with the DOS file system. As we said above, we want to put hpcboot.exe onto it:

```
# cp .../hpcsh/installation/hpcboot-sh3.exe /mnt2/hpcboot.exe
```

and the NetBSD kernel:

```
# tar -x -p -z -f .../hpcsh/binary/sets/kern-GENERIC.tgz -C /mnt2
```

The hpcboot.exe boot program can boot kernels from UFS, so you can skip this step, and extract the kernel to `/mnt` instead (where your NetBSD root partition of the CF card is mounted).

For now we are done with the DOS partition.

Next we will unpack the NetBSD release sets. The simple loop below extracts all the sets except the kernel ('k') and X11 ('x'). We already extracted the kernel to the DOS partition, so you always want to skip 'k*' sets. As for the X11, if you have a 512MB CF, you can as well extract them.

```
# for f in .../hpcsh/binary/sets/[^kx]*.tgz; do
>   tar -x -p -z -f $f -C /mnt
> done
```

With your CF now fully populated all that remains is some final touches to system configuration.

We create a mount point for the DOS partition:

```
# cd /mnt
# mkdir cf
```

And if you have put the kernel on the DOS partition (as we did in the example above), you create a symlink to the kernel. Strictly speaking, it's not necessary. NetBSD now uses ksyms(4), so programs that traditionally needed access to the NetBSD kernel image to parse its symbol table now use `/dev/ksyms`.

```
# ln -s cf/netbsd
```

Fix your localtime link:

```
# cd /mnt/etc
# rm localtime
# ln ../usr/share/zoneinfo/Europe/Moscow localtime
```

Edit fstab(5). We only have two partitions to add. Note that we mount the root file system with 'noatime' and 'nodevmtime' to reduce CF wear.

```
# cd /mnt/etc
# vi fstab
[...]
# cat fstab
/dev/wd0a       /       ffs     rw,noatime,nodevmtime   1 1
/dev/wd0e       /cf     msdos   -l,rw                   0 0
```

Edit rc.conf(5):

```
# cd /mnt/etc
# vi rc.conf
[...]
# sed -n '/^rc_configured/,$p' rc.conf
rc_configured=YES

# Add local overrides below
#

hostname="nada"

critical_filesystems_local="/cf $critical_filesystems_local"

no_swap=YES
savecore=NO
```

Here `/cf` is added to critical file systems so that `/netbsd` symlink works in case someone needs it, but as mentioned above, it's not strictly necessary any more. Also, if you put the kernel to UFS, you don't need that line either.

With `no_swap=YES` we tell that we intentionally configured the system without swap. Also, savecore is disabled as we don't have `wd0b` anyway, so avoid complains.

Now check `/etc/ttys`, thought the defaults should be sane.

```
# cd /mnt/etc
# vi ttys
[...]
```

And finally populate `/dev`. If you are ok with 1000+ devices in `/dev`, you can just run:

```
# cd /mnt/dev
# sh MAKEDEV all
```

But if you want to avoid all those device nodes for raid and multiport serial cards you can spent just a little bit more time and create only those device nodes that you need. The example below is a baseline that is enough to give you a working system. Add more devices according to your needs.

```
# cd /mnt/dev
# sh MAKEDEV std
# sh MAKEDEV random systrace lkm clockctl
# sh MAKEDEV wscons
# sh MAKEDEV apm
# sh MAKEDEV scif0            # <- sh3 specific serial
# sh MAKEDEV ptm pty0 tty0 tty1
# sh MAKEDEV atabus0 atabus1 wd0 wd1
# sh MAKEDEV bpf0 bpf1
```

Congratulations! At this point your CF card is ready. You can plug it into you HPC and boot from it into multiuser.

Future Work

There are still a number of areas in which the handheld and pocket PC support within NetBSD can be further improved.

- The sound drivers need work – whilst we have VoIP and softphone capabilities within kphone and other packages, the sound drivers could do with an overhaul.
- Support for a windowing environment other than X would be beneficial. Qt/Embedded, Qtopia or GPE or a similar solution would be attractive to some PDA manufacturers
- Microsoft's Windows environments have set the standard for integrated applications, such as Microsoft Office, on PDAs. Whilst Open Office or Star Office is probably too large to reside even on CF, and to run within acceptable limits on a PDA, it is still possible to use other open source applications for the same functions
- Wide-area communications would benefit from better 3G, GPRS, and soft modem support
- Java support in PDAs is becoming an increasing factor – whilst Sun's JDK (versions 1.3 and 1.4) have been ported to i386 and SPARC platforms, there is little support for low-power CPUs and architectures in the standard JDK, although it is almost definitely too large and resource-hungry to fit on a PDA. Whilst there are many open-source Java Virtual machines in existence, such as Wonka, kaffe and sablevm, few have the Windowing Toolkit support to make them attractive propositions for devices with limited input options, and limited screen area. Sun's J2ME is a possibility, more work needs to be done in this area.

The Ultimate Geek PDA

It is interesting to predict what the ultimate geek PDA will look like in a few years. We are seeing PDAs emerge from Japan right now with 2.1 inch screens, yet with VGA resolution. Much has emerged in this area over the last six months, and it is expected that this trend will continue. So it looks like the screen real-estate problem may be being addressed.

Input to the PDA remains a problem. It might be possible that voice recognition will gain ground in the next few years, rendering hunt-and-peck stylus input obsolete. Input is probably the most tortuous aspect to using a PDA these days.

Compact Flash cards may increase in size, and may also decrease in price, allowing the operating system to do more, and for applications to store more data.

Battery life may need to improve, and we may see more takeup, especially in Europe, of services more tightly integrated with what has been viewed as the telephone company domain. Indeed, this crossover into the realm of mobile telephone handsets has some interesting possibilities, and there is a lot of movement in this area.

The consolidation of a number of personal devices – pager, mobile telephone, PDA – into one PDA, Internet-enabled through 3G or GPRS, is an attractive possibility to a number of manufacturers, service providers and software vendors.

Conclusion

Handheld PCs and pocket PCs present a number of challenges for users – their size is what makes them appealing, and yet it is also the cause of some of the major human factors problems, most notably screen real-estate and keyboard and mouse input. Whilst NetBSD can be installed and will run very nicely on such machines, there is a certain amount of future work which needs to be done using NetBSD on these platforms to make the experience even better. These PDAs are not expected to be compute engines – their appeal lies in their portability and size. Cross-building of the operating system and the windowing system is obviously a necessary pre-requisite for developing with PDAs, and NetBSD has these features already. Cross-building of third-party applications for PDAs is also needed – this aspect has been addressed in other papers.

A Portable Packaging System

Alistair Crooks, The NetBSD Project

29th September 2004

Abstract

This paper explains the needs for a portable packaging system, and the approach that was taken by the NetBSD pkgsrc team in order first to port their packaging system to another platform. Having learned from this approach, a different and more scalable approach was tried, and has had tremendous benefits. This scalable approach is explained, along with the problems and solutions in supporting multiple different operating systems, and finally the results of this approach are examined and explained.

1 The Need for Portability

A packaging system is an essential part of an architect or administrator's infrastructure, in order to manage the myriad pieces of third-party software that abound. The proliferation of this software is good, but managing a network of even 100 machines in a secure and professional manner can take a huge amount of time. To ameliorate the problems, administrators tend to work with what they know - learning how to work new systems has never been high on anyone's list of things to achieve.

However, even in 1999, the benefits of using an existing packaging system were recognised - to leverage others' work to keep packages up to date with the latest stable versions, and to use the packaging tools to provide a consistent interface for administrators, and to provide a security vulnerability advisory service.

1.1 An overview of pkgsrc

Before a piece of third-party software can be installed on any system, a number of steps must take place:

1. the distribution files, usually in the form of a gzipped or bzip2ed tar file, must be downloaded
2. its integrity is checked against an SHA1 digest stored in the pkgsrc hierarchy
3. any pre-requisite software is also checked to see that a sufficiently up-to-date version is present

4. the source is extracted from the distribution files
5. any patches necessary are applied, using patch(1)
6. the software is configured
7. the software is built
8. the software is installed

Obviously, most of this work is done by shell commands, driven by a large Makefile, and using standard POSIX utilities. pkgsrc automates this process, ensuring that any pre-requisite dependencies are installed and available.

The infrastructure required for pkgsrc itself is a set of packaging tools, collectively referred to as the pkg_install utilities:

- pkg_create registers all the files, links, directories pertaining to a package - this is done at package install time.
- Binary packages of a package installed on a system can be created, either by using pkg_create, or by using a supplementary package called pkg_tarup.
- pkg_add can be used to install a binary package - it can be given a URL, and pkg_add will download a binary package. pkg_add also has been enhanced to use digital signatures of binary packages, if available, to verify the contents of the binary package.
- pkg_delete will delete a package, and the corresponding files from the file system. pkg_delete will calculate an MD5 digest of the file it is about to delete, and will refuse to delete the file if it does not have the same digest as when it was installed.
- pkg_info shows information on the packages that are installed on a system, or gathered together in a separate place
- pkg_admin performs various administrative tasks

There are also a number of standard NetBSD utilities which are needed:

- bmake(1) - the NetBSD make(1) utility
- tnftp(1) - the NetBSD ftp(1) client, used to download distribution files
- digest(1) - a small program to calculate message digests (this was added subsequent to the use of Zoularis)
- pax(1) - the portable archiving tool
- mtree(8) - a tool to create and manage directory hierarchies
- sed(1) - the stream editor
- pkg_install(1) - the NetBSD package management tools

1.2 Security

A benefit of using a standard third-party packaging system such as pkgsrc is that security vulnerabilities in third-party software can be notified to users and administrators of systems automatically. The administrator can then act to mitigate or remove the effects of an intrusion or exploit via the vulnerable package.

Having this available across a range operating systems is another substantial benefit - for more information, see the **pkgsrc/security/audit-packages** package.

2 Porting the Infrastructure

2.1 The First Approach - Zoularis

NetBSD's pkgsrc [pkgsrc2004] grew out of [Freebsd2004] in 1997, and was customised for NetBSD's immediate needs.

These included

- NetBSD platform independence
- and ELF/a.out agnosticism
- and went on to include deterministic package version number comparison, and many others.

There was considerable pressure on the NetBSD pkgsrc team to port their software to other operating systems - primarily Solaris, but also Linux and others - and this was achieved in 1999 by using a heavy compatibility layer for functionality which was missing in the native operating system. The version of Solaris which was used as the target at that time was 2.6, which did not have such functionality as

- strlcpy(3)
- snprintf(3)
- getfsent(3) etc.

Christos Zoulas had done this work already, and so the compatibility layer was called Zoularis.

Zoularis itself was a compatibility layer on top of the native operating system. It provided the extra functionality that was missing from the native libc, and allowed various NetBSD utilities to be built, and these could then be used to manipulate packages. Zoularis had to be made into a binary package, and that binary package had to be installed on every system that used binary packages (since they were built with Zoularis as a pre-requisite dependency).

At the same time, the main Makefile which defines the way that packages are built, and which provides all the definitions and targets, was modified to support Solaris. Usually this was by means of such statements as

```
    .if ${OPSYS} == "SunOS"
```

One of the main problems was that of the PLIST, which is a file containing the list of files which make up the binary package. pkgsrc already manipulated these PLISTs to support manual pages being gzipped or not. On different operating systems, the package would sometimes install different files, and so the PLIST had to be manipulated accordingly. This was almost always the case with packages which used *imake* in their build process - manual pages became installed with different suffixes on different operating systems.

Another set of packages that quickly showed itself as being different was those which used libtool. Standard Solaris semantics at that time was to create ELF shared libraries, with 3 numeric suffixes. The PLISTs in pkgsrc had all been written for NetBSD's predominantly a.out shared libraries, which commonly used 2 numeric suffixes. Clearly some manipulation of the PLISTs was necessary, as well as modifications to libtool to create shared libraries with 2 numeric suffixes on Solaris. Later still, more manipulation was necessary for Darwin's .dylib libraries.

Zoularis itself was built using a shell script, operating on a checked out version of the NetBSD source tree (for the NetBSD libc functionality, for libedit, and for tnftp, mtree, tar, the pkg_install tools and the other necessary utilities).

It was clear that Zoularis was one solution, and worked very well in practice. What was also clear was that there may well be less intrusive, more lightweight solutions to portable packaging systems.

2.2 Older and Wiser - the GNU autoconf approach

When the request came to the pkgsrc team to port pkgsrc to Darwin, it had become obvious that a new approach was needed. One of the reasons for this is that the only Darwin machine available at the time was located in Atlanta, Georgia at the end of a fairly thin pipe, the engineer doing the porting was in London, UK; and downloading megabytes of NetBSD source over a thin pipe was not the best approach. Zoularis itself was heavily dependent on the NetBSD sources - it used NetBSD header files, and source files from NetBSD's libc, and the list of files needed to compile Zoularis became long and unwieldy. Binary distributions of Zoularis were made available, but they, too, were dependent on NetBSD's libc version. The size of the NetBSD source code at that time was just over 500 MB, and so it was not a particularly practical proposition to port Zoularis to an existing, known-good platform.

2.2.1 The pkgsrc tools

The first step in the new approach was to provide autoconf-ed versions of all the tools needed by the packaging system. This could even mean that cross-compiling and cross-bootstrapping of a pkgsrc environment could be done. The tools which are GNU autoconf-ed are outlined in section 1.1.

Up until now, pkgsrc has managed to use awk scripts which do not contain any GNU-awk specific extensions, and so there has not been a need to provide a portable version of awk(1). However, more configuration and build scripts are using awk over time, and so it is envisioned that we will eventually provide a version of awk, with

GNU extensions, that is appropriately-licensed, but will be able to deal with gawk extensions.

2.2.2 The pkgsrc Infrastructure

The next step was to examine the pkgsrc system itself. Over time, a lot of the support in the main infrastructure had grown machine-dependent .ifdefs (NetBSD make(1)'s equivalent of the C preprocessor's #ifdef, and roughly analogous to gmake's "ifeq"), and these needed to be excised. To do that, OS-specific definitions files were used, which were sourced by make(1) at run-time. So we now have

- defs.AIX.mk
- defs.BSDOS.mk
- defs.Darwin.mk
- defs.FreeBSD.mk
- defs.IRIX.mk
- defs.Interix.mk
- defs.Linux.mk
- defs.NetBSD.mk
- defs.OpenBSD.mk
- defs.SunOS.mk
- defs.Unixware.mk

whilst more - HP/UX, the Hurd, Digital Unix (Tru64) and UWIN - are planned.

2.3 Binary Packages

Most NetBSD users currently build their own packages from source, although we expect that to change over time to be more akin to the situation which is prevalent in the GNU/Linux world, where binary packages are the norm, and people rarely veer from the straight and narrow compiled-in defaults. Even when this does become the norm in the BSD world, some challenges remain:

- package versioning
- library incompatibilities
- tools unpacking one package built for another architecture or operating system

To address these issues, a number of different approaches have been tried:

- "package versioning" is a problem that applies when an attempt is made to install two conflicting packages at any one time. This is a problem for every packaging system, not just pkgsrc, and has been addressed in a number of ways: [Stow2004], [Depot2004] and in package views [Crooks2002].
- library incompatibilities are related to the package versioning problem - if the ABI of a shared library is changed, by convention software authors bump the major version number of a shared library. Packaging systems which allow one package which uses the old ABI to use the new ABI let down their users - the tool will fail to function properly, and there may be knock-on effects, with other shared library needing to have their version numbers bumped. This is addressed in RedHat's versioning system via a PROVIDES and REQUIRES form of export/import information. pkgsrc has adopted this in its build information, and this may be used in future to determine ABI compatibility.
- modifications have had to be made to the NetBSD packaging tools to stop naive extraction of a package built for a different architecture

3 Portability between Operating Systems

3.1 Differences between native tools

It was found very early on that some of the things in the Open Source world that we take for granted are simply not available on other operating systems. There are two obvious approaches to it - firstly, to work around it, and code to the lowest common denominator, or, secondly, to provide the open source tools in a proprietary world. Whilst we had gone down the second route using a "broad stroke" approach with Zoularis, we thought it was still a superior approach to programming to the lowest common denominator.

With Solaris, we found that there were many things that we had to use - the POSIX XPG4 utilities that were available in /usr/xpg4/bin were necessary to have some hope of using modern shell and other constructs - and this theme extended itself further as we ported pkgsrc to more and more platforms.

In total, to obtain consistency across operating systems, it's necessary to use tools which have a common and well-specified interface. For us, this means POSIX and XPG4. In practice, this is still not enough - there are differences between Solaris's version of ln(1) and almost everyone else's ln(1), when given a command line of "ln -fs file1 file2". NetBSD and other operating systems will unlink(2) the "file2" file system entry before attempting to create the symbolic link, whilst Solaris's will attempt the command but will fail, and just not report an error to the caller. The solution in this case, unfortunately, is to do a sweep of pkgsrc and all the package Makefiles in the hierarchy for any occurrences of "ln -fs" or "ln -sf" and put an explicit "rm -f" before the symbolic link is attempted.

The open source operating systems use either GNU or BSD utilities - to abstract away the differences for each package, a "tools.mk" file was created, which will automatically add a build dependency on GNU awk if a package needs it to build, and the platform does not provide it natively. (Adding a build dependency means that the

GNU awk package would be built and installed, if not already present, as part of the pre-requisite checking during a package's build).

3.2 Platform-independent Infrastructure

The main "bsd.pkg.mk" file, which is the main file included by every package Makefile, has no platform-specific ".if" statements in it. A similar scheme to the GNU autoconfiguration mechanism has been used, where the desired feature is defined in the platform-specific files, and then the platform-independent files test for that definition.

This has many advantages, not least amongst them the ability to support many operating systems, often with no changes to any of the platform-independent files.

pkgsrc has a single file called "bsd.prefs.mk" which will set all the default values for make(1) definitions, including setting any command line options, and any options specified in /etc/mk.conf. Sourcing "bsd.prefs.mk" in a package Makefile also has the effect of making various convenience definitions, such as ${OPSYS}, available.

3.3 Inventories and shared libraries

At present, pkgsrc uses static PLISTs - these are packing lists, or inventories, of the files associated with the package. These PLIST files are distributed with pkgsrc itself. (The last release of pkgsrc, pkgsrc-2004Q3, includes modifications for pkgviews, which is a means of having multiple "conflicting" packages installed at any one time - see [Crooks2002]. Package views supports dynamic packing lists, so that PLIST manipulation is no longer a problem).

A lot of packages provide shared libraries, and other packages can link these libraries in at compile-time or run-time.

Different operating systems have different formats for shared libraries:

- Darwin (Mac OS X) uses a dylib name for its shared libraries
- AIX uses a "lib.a" name for its shared libraries
- a.out libraries on older NetBSD systems and OpenBSD have different naming conventions to ELF libraries used by the same operating system
- Interix has some interesting semantics for shared objects and dynamically-linked libraries

The PLISTs in pkgsrc have all been written to use the ELF shared object name, and libtool has been modified to produce shared libraries which correspond to the same naming convention. For each operating system, at install time, a check is made for the format of shared library. If the platform does not support shared libraries, all of the dynamically linked library names are removed form the PLIST. If an a.out library format is used on the platform, the PLIST is manipulated to include simply the ".a" form of the library. A similar manipulation takes place for AIX, and Darwin.

In addition, the PLISTs have all been modified to check for ".la" archives, if LIBTOOLIZE_PLIST is set, and to generate the appropriate PLIST entries for shared objects and libtool archives. This means that a single unified PLIST can be used across multiple operating systems with different shared object names and semantics.

3.4 Manual Pages

In a similar manner to shared libraries, manual pages have different characteristics between operating systems. Some always install pre-formatted manual pages with a ".0" suffix, others include a letter suffix to denote the subsystem being used. pkgsrc recognises the operating system being used, and manipulates the PLIST accordingly.

3.5 Packages Specific to One Architecture or Operating System

There are times when packages, often third-party binary packages, are specific to one version of the operating system. In cases like these, a build should not take place if the relevant criteria are not fulfilled. pkgsrc uses a triple:

```
(Operating System, Architecture, Version)
```

to determine whether or not a package can be built and used. The triples are attached to ONLY_FOR_PLATFORM and NOT_FOR_PLATFORM definitions in the package's Makefile in the pkgsrc hierarchy, and can be specified multiple times for multiple versions or platforms.

This example is taken from the **pkgsrc/net/gkrellm-wireless** platform:

```
ONLY_FOR_PLATFORM=    *BSD-*-* Linux-*-*
```

and this one is taken from the **pkgsrc/net/hping** Makefile

```
NOT_FOR_PLATFORM=     NetBSD-0.* NetBSD-1.[0-4]*
```

3.6 Selecting the correct headers and libraries

One of the situations that arose fairly early on was the situation where a package was installed on the system, and a different version of that package was needed to build, or that a system library should always be used in preference to a library installed via the packaging system. To illustrate this problem, consider a build machine where ncurses is installed - by default a number of packages will detect ncurses's presence in ${LOCALSRC} via the GNU configure scripts, and build and link with ncurses. This may not be wanted - ncurses is quite a large package, and to proscribe its installation in a binary package when all that is wanted is curses functionality is to ensure that ncurses will be installed on every machine.

The buildlink framework was introduced by Johnny Lam as a means of making sure that a package being built was compiled with the right header files, and linked with the right libraries, no matter what libraries and headers were installed on a machine. This is done by populating the desired headers and libraries into the build framework in their own hierarchy, and then convincing the build framework to search that hierarchy before any existing system ones.

The initial buildlink implementation worked well, but had some flaws - each occurrence of /usr/local was caught by the infrastructure and converted to be the buildlink hierarchy, which did not work very well for all cases, especially when installing some files into the regular destination, still containing references to the intermediate buildlink hierarchy.

Buildlink2 was written to address these problems, and involves the extension of this idea to provide wrapper scripts around cc(1), as(1), etc. These scripts fixup the necessary paths and then hand off the correct paths to the real tools. This still has some problems, which are especially relevant here, and buildlink3 has been written to address these. It also includes portability enhancements, which is the reason for its mention here.

Originally, Zoularis mandated gcc as the compiler of choice. When the autoconf-ed tools were introduced, there was no longer any reason to mandate gcc - different compilers could be used, such as the Sun Pro compiler. The trouble with using alternative compilers is that the options are not the same - -Wl,-R${LIBDIR}/lib will do exactly what you expect it to do with a gcc/GNU ld combination - a different construction must be used for the Sun Pro compiler.

Buildlink3 extends the abstraction a bit further - a cc script is provided, and the script is called with the normal gcc arguments. If the compiler being used is actually the Sun Pro compiler, then the arguments are fixed up (as well as the paths), in order to provide portability. The same is true for the MIPS pro compilers, and other platform-specific proprietary compilers.

An additional use of buildlink is to ensure that all necessary pre-requisites have been specified in the package Makefile. The way that buildlink3 works is to create bin, include and lib directories inside the ${WRKDIR} - where the package is being built - and to populate those directories with symbolic links to the utilities, header files, and libraries which it will need. The path in the environment is then modified to search the buildlink directories first. Some of the build utilities are invoked via wrapper scripts, which massage the arguments as necessary, making every compiler appear to take the same arguments as gcc, for example. Additional steps are taken to remove all traces to buildlinked files on package installation. If a utility or library is not found, then the package will not build correctly, and so all necessary pre-requisites are found at package compile time.

In practice, buildlink3 provides a tremendously valuable abstraction, since most open source software assumes gcc to be the compiler.

3.7 GNU config.* files

Packages will often include their own config.guess and config.sub files, as part of their own GNU autoconfiguration mechanism. With old packages, these files can be considerably out of date. pkgsrc has a means of overriding these files within a package, so that, even if a package has no means of configuring itself for NetBSD/sh5 (it is unlikely that the package authors will have even heard of such an architecture or operating system), the package will still be able to be configured appropriately on that platform.

3.8 Native or Packaged Software

Occasionally, an operating system will come with certain utilities as standard, whilst such a facility may only be available as a third-party package on other platforms.

As part of the buildlink work, this whole area has been the target of some abstractions, and this has meant that the whole question of "native versus package" is moot.

If the package being built is already part of the native operating system, by default, pkgsrc will prefer it to the packaged version. This can be changed on a package-by-package basis, or as a complete entity.

- the PREFER_PKGSRC and PREFER_NATIVE definitions can be set to the names of a number of packages
- the PREFER_PKGSRC and PREFER_NATIVE definitions can also be set to "yes" or "no" values.

Preferences are determined by the most specific instance of the package in either PREFER_PKGSRC or PREFER_NATIVE. If a package is specified in neither or in both variables, then PREFER_PKGSRC takes precedence over PREFER_NATIVE.

3.9 Platform-dependent definitions

Having reviewed a number of package Makefiles, we found that it was often the case that we were setting a number of compiler flags in an operating system-dependent manner. Typical package Makefiles would look like:

```
.include "../../mk/bsd.prefs.mk"

.if ${OPSYS} == "SunOS"
LIBS+= -lnsl -lsocket
.endif
```

and that was changed to support

```
LIBS.SunOS+=            -lnsl -lsocket
```

This example was taken from the **pkgsrc/net/jwhois** package Makefile.

4 Related Work

The Solarpack project grew out of some work done by Julien Letissier for Sun in 2001, looked a number of different packaging systems, and ended up choosing pkgsrc. Over summer 2001, various packaging systems were examined, and pkgsrc was eventually selected as the packaging system of choice - one of the early attempts was released as [solarpack2002].

[fink2004] wants to bring the full world of Unix Open Source software to Darwin and Mac OS X, using Debian tools like dpkg.

There was a fledgeling project called OpenPackages which also used the NetBSD packaging tools as its base, but it appears that this project is no longer being actively developed.

5 Future Work

The NetBSD project continues to expand and develop the NetBSD packages collection. In December 2003, a release branch of the CVS repository was created, the branch named "pkgsrc-2003Q4". Since then, a branch has been "cut" every three months, and it is intended that future releases will take place at regular three month intervals.

In order to enhance portability using buildlink3, the buildlink3 infrastructure was merged to the pkgsrc branch prior to the branch being created - we learned from previous CVS branches that many commits were made by the team after a branch had been created, and that the infrastrcuture needed to be in place prior to branching so that any subsequent branch could better be maintained.

It is expected that Sun will use some form of pkgsrc as their packaging system of choice for Solaris, starting with the next release.

Further platforms will be added to pkgsrc over the next few months - patches have already been received, and are stored in the NetBSD GNATS database, for HP/UX and GNU/Hurd. There are also modifications to support Tru64 both in GNATS and available through the NetBSD tech-pkg mailing list archives. The problem with providing support for extra platforms is that pkgsrc developers do not, themselves, have access to any hardware or simulators needed to run the operating system in question.

6 Benefits of a Portable Infrastructure

There are a number of benefits which accrue to a portable infrastructure:

- leverage the work of others in porting the software
- leverage the work of others in maintaining the software
- using standard means of security vulnerability notification and fixing
- using standard tools across a range of different machines, with the same interface and behaviour

References

[Stow2004] http://www.gnu.org/software/stow/stow.html

[Depot2004] http://asg.web.cmu.edu/depot/

[Crooks2002] Package Views - a more flexible structure for third-party software, Proceedings of the European BSD Conference, 2002, Amsterdam

[pkgsrc2004]	http://www.pkgsrc.org/
[solarpack2002]	http://solarpack.sourceforge.net/
[fink2004]	http://fink.sourceforge.net/
[Freebsd2004]	http://www.freebsd.org/ports/

Cross-building packages

Krister Walfridsson
<kristerw@netbsd.org>

29th September 2004

Abstract

It is often desirable to cross-build software, but most software packages have issues in their build process that needs to be corrected before it can be done. This paper describes a mechanism that automatically works around the issues, as implemented in the context of the NetBSD pkgsrc.

1 The basic idea

NetBSD runs on a wide range of architectures, and that cause several problems for the pkgsrc developers, since it is hard to test the packages on all architectures, or make the binary packages available for all of them.

People interested in binary packages often suggest introducing cross-compilation in pkgsrc. Their reasoning is usually that the GNU autoconf-generated `configure` scripts are designed to be used in cross-compiling environments, so at least the autoconf-using packages could be made cross-buildable. They are correct about the autoconf technology, but it is often not used in a cross-friendly way, so most packages would need modifications anyway. And doing even small changes to the 5000+ packages in pkgsrc is a daunting, and error prone, task.

One other approach is to build the packages in an emulator. This produces exactly the same binary packages that would have been built on the real hardware, without any need of modifying the packages. But this approach is slow (although building the package in an emulator running on modern hardware may be faster than building the package on the real machine for some of the older architectures...) There is, however, no need to emulate the complete hardware and operating system – only what is needed to build the packages. In particular, kernel mode does not need to be emulated, since the programs will not see any difference if the emulator does the equivalent action natively. This “as-is” rule can be applied on an even higher level: it is not necessary to run a program emulated if exactly the same result can be obtained by other means. For example, if the emulated environment tries to run

```
/bin/echo "Hello world!"
```

then it is not necessary to run `/bin/echo` emulated, since exactly the same result will be obtained (but much faster!) by running the command natively. And there is no need for the emulator to emulate running `gcc` when it can run a cross-compiler instead.

This makes it possible to cross-build packages relatively efficiently without doing any changes to the package. In fact, many packages build without emulating any program at all (more than possibly small test programs during configuration, but those are usually small enough that it only takes a couple of seconds).

There are however some packages that builds too slowly using this method, and those need to be modified (although it should be noted that they usually need less modifications than would have been the case for traditional cross-building methods). Experience with an actual implementation of these ideas shows that less than 1% of the packages need this kind of change in order to build efficiently.

The rest of this paper describes the implementation of these ideas.

2 How to build packages in pkgsrc

The NetBSD pkgsrc is a framework for building and managing third party software on a variety of operating systems (see http://www.pkgsrc.org/ for a description of its features and inner working). For the rest of this paper it is enough to know that building a binary package (e.g. GNU Emacs) suitable for installing on other systems is as easy as

```
% cd pkgsrc/editors/emacs
% make package
```

This will fetch the source code (using ftp or http), apply pkgsrc specific patches, build the package, run tests to see that it works (if the original source code distribution includes such tests), and tar it up to a binary package.

Pkgsrc does also have a script to build all packages. This mechanism is most often used to find packages that have build problems, or to build the full set of binary packages for download from the NetBSD ftp. This "bulk build" is run by

```
% cd pkgsrc
% sh mk/bulk/build
```

The build script usually build only the packages that has been updated since the last run, and the packages that depends on them.

3 How to cross-build packages

The main goal for the cross-building framework has been to make it as easy to use as possible – both for the pkgsrc developers and the users that want to cross-build packages. Building packages are therefore done exactly as for native packages, although a wrapper is used (in exactly the same way that the standard NetBSD source code is cross-built) instead of the normal make.

```
% cd pkgsrc/editors/emacs
% nbsimmake-shark package
```

There is also a wrapper for sh so that bulk builds can be done:

```
% cd pkgsrc
% nbsimsh-shark mk/bulk/build
```

The file system for the target needs to be set up before starting to cross-build packages. This is done by packing up and configuring the standard NetBSD release in the same way that is done when setting up an NFS-mounted root. The cross-building framework does also need to be set up. These two steps can be done by installing the cross-building package for the target architecture.[1]

The rest of this paper will assume cross-building for NetBSD-1.6/shark, with the shark's file system located at /emulroot/shark/.

[1] These packages are not available from pkgsrc yet, but the software can be downloaded from http://www.df.lth.se/~cato/crossbuild/

4 How the cross-building framework is implemented

4.1 The starting point

The first step in the implementation was to find suitable emulators to build on. The GDB distribution has an extensive collection of emulators (which are called simulators in the GDB terminology) that emulates many of the architectures that NetBSD runs on, so this was seen as a good starting point.

The GDB powerpc simulator can even run NetBSD powerpc binaries by running the system calls natively in the same way as must be done for the cross-building framework! That NetBSD emulation is however rather limited, since it was developed for running the GCC test suite only, and the code has bit-rotted over the years, so the GDB NetBSD emulation code has not been used in this project.

There are some important architectures missing from the GDB distribution (such as m68k). It is however rather easy to add the NetBSD system call emulation to "any" emulator, since it is only the machine instruction responsible for entering system calls that is affected. The only thing that needs to be done is to write a small glue layer between the emulator and NetBSD code, to make it possible to read the parameters, return the result of the system call, and to read/write memory from the emulated machine.

4.2 Basic functionality

Simple system calls, such as close, are easy to run natively:

```
void do_close(void)
{
  int d = get_parameter(0);
  int status = close(d);
  write_status(status);
}
```

The first line gets the parameter that the emulated program provided to the system call. The next line calls the system call natively, and the last line modifies the emulated state so that the emulated program gets the result.

More complex system calls are done in essentially the same way. The only difference is that data may need to be copied between the emulated memory space and the host's memory space:

```
void do_write(void)
{
  int d = get_parameter(0);
  EMUL_ADDR buf = get_parameter(1);
  size_t nbytes = get_parameter(2);

  void* tmp_buf = xmalloc(nbytes);
  read_memory(tmp_buf, buf, nbytes);

  int status = write(d, tmp_buf, nbytes);
  write_status(status);

  free(tmp_buf);
}
```

Care need to be taken so that differences in endianness, or in the width of data types, are compensated for when moving structures between the different memory spaces, so arrays and structures need to be copied one field at a time:

```
void do_socketpair(void)
{
  int tmp_sv[2];
  int d = get_parameter(0);
  int type = get_parameter(1);
  int protocol = get_parameter(2);
  EMUL_ADDR sv = get_parameter(3);

  int status = socketpair(d, type, protocol, tmp_sv);
  write_status(status);
  if (status != -1)
    {
      WRITE_INT32(sv    , tmp_sv[0]);
      WRITE_INT32(sv + 4, tmp_sv[1]);
    }
}
```

Note that the above function takes advantage of fact that both the target and the host are running NetBSD – the `type` and `protocol` would need to be translated if the host were running some other operating system.

4.3 System calls working on files

Not all system calls can be done as straight forward as in the previous section. Consider for example running a program that opens a file through an absolute path:

```
open("/usr/include/machine/types.h", O_RDONLY);
```

It should open the file from the emulated machine's file system, and not the one from the host's file system. This means that the emulator must transform all absolute paths by appending `/emulroot/shark` before calling the native system call. Modifications are needed in the other direction too; system calls like `getcwd` does return a path of the form `/emulroot/shark/foo` that must have the prefix stripped before the control is returned to the emulated program.

4.4 execve

One other special case is the `execve` system call. A call like

```
execve("foo", argv, envp);
```

cannot be run natively as-is, but need to be transformed to

```
execve("emulator", new_argv, envp);
```

where `new_argv` is the `argv` where the arguments needed for the emulator to run the original program have been prepended.

But all programs do not need to be run emulated. Consider for example `/bin/echo`. This cannot do any "dangerous operations" like starting new programs or accessing paths not present on the command line that need to be transformed, so `execve` of `/bin/echo` may as well be run natively. This is true for many programs, although they may need some modifications of arguments. For example `/bin/cat` is safe too, but it may take file names as arguments, so

```
cat /path/foo
```

must be transformed to

```
cat /emulroot/shark/path/foo
```

before it is executed.

And some programs may safely be run, unless some special arguments are used. For example

```
/usr/bin/awk -F "'" '/^PACKAGE_VERSION=/ {print $2}' file
```

is safe, but

```
/usr/bin/awk -f foo file
```

is not. This means that a small parser must be implemented for each system binary so that the dangerous constructs can be transformed, or the command run emulated, as appropriate.

One important program in this category is `gcc`. We may greatly decrease the time needed to build a package by `execve` a cross-compiler instead of running `gcc` in the emulator.

One other thing to look out for is environment variables that may affect how the program works. For example, the `/usr/bin/install` may strip binaries when installing them, and the program to use for stripping is provided in the `STRIP` environment variable. The emulator must therefor check `STRIP` to see if it points to a program that can be run natively. The `/usr/bin/install` need to be run emulated otherwise.

Yet another issue that needs care are symbolic links. Note that all symbolic links made in the emulator must point to the full path within the real file system in order for the following to work

```
% ln -s /tmp/foo.c .
% gcc foo.c
```

This has the effect that some programs that otherwise would be able to run natively cannot do that. The most important example is `tar`. The good thing is that `tar` is rather efficient, so we do not lose much by running it emulated.

4.5 Removing one bottleneck

One of the goals of the cross-building framework is that it should be easy to maintain, so the original idea was to avoid building custom versions of components provided by the NetBSD distribution. Profiling did however show that a big amount of the time spent cross-building was spent in emulating `sh` and `make`, so it was decided to create special versions of those two programs to make them safe to run natively. The only difference from the original versions are that the cross-versions have modified all library calls with file parameters so that they are correctly transformed, and modified all calls to the exec-family of calls to use the same mechanism the emulator uses for `execve`.

This has the effect that there is a risk of the real and emulated `make` and `sh` binaries being out of sync, but the goal is to eventually get these changes into the real NetBSD distribution, so this is hopefully just a temporary problem.

4.6 Shortcuts

The goal of the cross-building framework is to cross-build packages – not to be able to run arbitrary programs from another architecture. This means that the emulation does not need to be perfect, as long as it does not affect the building of packages. It may even in some cases be easier to modify a specific package to build with a limited emulator instead of implementing all needed functionality in the emulator.

One trivial example of a limitation that is unlikely to affect the resulting packages can be seen by:

```
% nbsimsh-shark
$ mkdir /bin/foo
mkdir: /emulroot/shark/bin/foo: Permission denied
```

Observe that the error message contains the full path from the host operating system instead of the path that the emulated environment passed to mkdir. It is possible to prevent this kind of path leakage by improving the parameter checking done before running the native mkdir, to make sure that native execution always succeeds (or alternatively, parse and transform the output). But this kind of situation is not common when building packages, so it is much easier to change the affected packages, if such packages are found.

Many packages looks at the file and line information that gcc -E outputs, to e.g. build the dependency information for make. This cause problems when the cross-compiler is used, since absolute paths gets an /emulroot/shark prefix. This could of course be solved by always running gcc -E emulated, but that results in an unnecessary increase in build time for many packages. It is much better to solve this by treating paths such as /emulroot/shark/foo as /foo within the emulated environment, although this has the effect that it is not possible to have a directory called /emulroot/shark within the emulated file system...

5 Future work

The bulk build does currently need to be run with root privileges because many packages need to set user and group etc. on the files they create. It is however possible to use the emulator infrastructure to build packages as an unprivileged user.

The idea is to do all file operations as an unprivileged user, and to record the side effect of the "privileged" system calls (such as chown) in a log file. The log file is consulted every time a file system call, such as stat is called, so that the the correct information is returned to the caller. This will ensure that the resulting binary packages get the correct owner etc. for its files.

Some packages may however need to run e.g. SUID programs during its build process, and some of those may fail when not being run with the "real" user ID. Such packages need to be modified to build as an unprivileged user, but it is believed that only a few packages are affected.

Note that this could be used for unprivileged native builds too, by e.g. emulating an i386 target on an i386 host.

Fighting the Lemmings

Martin Husemann
martin@NetBSD.org

From the NetBSD byteorder.3 manual page cvs log:

```
revision 1.8
date: 2001/11/29 22:55:57;  author: ross;  state: Exp;  lines: +1 -6
Delete the old BUGS section entry:
> On the VAX bytes are handled backwards from most everyone else in
> the world.  This is not expected to be fixed in the near future.

Multiple levels of irony there...
```

Abstract

"All the world is a VAX" has been a famous proverb in the UNIX community when discovering code or design that has no easy, rational justification for existence or style.

Nowadays everyone codes for Linux and everything runs on i386 machines. People porting to other (vastly different) architectures are faced with lots of - let's say – "suboptimal" decisions, and usually hack around it with lots of casts or similar band-aid solutions. This does not work well in the long term. (Un-)fortunately, improvements in gcc turn a lot of these short-term patches into real bugs later.

Non-portable code is simpler at first sight, sometimes. The same is true for ad-hoc design. Both often fall in the long run.

This paper will look into some pitfalls on the way to portable code and try to improve awareness of possible portability problems. Many of these problems are obvious, when pointed at – but nevertheless regularly appear in real world code.

Why Do We Care About Portability?

Portability is no objective measurable property of source code. It is not a one-dimensional scale at all. We can measure the effort it took to port some software after doing so and try to asses the amount of obfuscation it added to the source. Still it will be hard to estimate how long the next port will take.

Today many projects have a fixed target architecture. It is tempting to ignore portability while creating the software – often by calling the lack of portability "optimization" for the given target system. But target systems change, by installation of a newer version of the operating system, an upgrade of another related software system, a new X version, or by being replaced by another operating system and/or hardware.

Having portable code in the first place makes it robust against this kind of basically inevitable changes.

Portability is Relative

Example from a famous IOCCC (International Obfuscated C Code Contest) entry – this code is portable between a VAX and a PDP-11:

```
short main[] = {
    277, 04735, -4129, 25, 0, 477, 1019, 0xbef, 0, 12800,
    -113, 21119, 0x52d7, -1006, -7151, 0, 0x4bc, 020004,
    14880, 10541, 2056, 04010, 4548, 3044, -6716, 0x9,
    4407, 6, 5568, 1, -30460, 0, 0x9, 5570, 512, -30419,
    0x7e82, 0760, 6, 0, 4, 02400, 15, 0, 4, 1280, 4, 0,
    4, 0, 0, 0, 0x8, 0, 4, 0, ',', 0, 12, 0, 4, 0, '#',
    0, 020, 0, 4, 0, 30, 0, 026, 0, 0x6176, 120, 25712,
    'p', 072163, 'r', 29303, 29801, 'e'
};
```

(Winning entry 1984 by Sjoerd Mullender and Robbert van Renesse, see http://www.de.ioccc.org/1984/mullender.c)
This data array happens to be a set of valid VAX and PDP-11 assembler code that prints a message on the screen. The first word is a PDP-11 branch instruction jumping to the PDP-11 specific code, while on the VAX the C startup code uses the "calls" instruction, which expects a bit mask of registers to save at the first address of the subroutine, so effectively the VAX code starts on the second word.

Portability is Expensive

It is rarely a primary project goal, and is sometimes perceived as an obstacle to fast results. Of course this is a fallacy. Porting is expensive, if portability has not been considered during coding... but early replacement / migration is even more expensive!

Portability is the result of real porting

You can plan for it in advance, but without actual ports your code will always find some pitfalls.

Is there a single regression test for portability?

Unfortunately – no. Though porting something to NetBSD/sparc64 is close ;-)

Aspects of Portability

GUI Differences

Complete books could be written about this. Differences in GUI design, implementation and APIs are obvious. Sometimes even on one machine/operating system – consider Gnome vs. KDE. There are various libraries available, like wxGTK, that try to provide a common API for different GUI back ends.

Since there are basically no hidden traps in this area this paper completely ignores this topic.

Operating System/API Differences

Depending on the range of operating systems you consider porting to, there are POSIX standards to allow you portability to at least Unix-like systems. If you are careful to only use POSIX-blessed APIs, your code should be portable. But you can not avoid to deal with "text mode" for ascii file output if you include windows in your portability profile. CreateProcess() [a win32 API to start a new process] differs from fork/exec and pthread_create. There is nothing you can do about this - #ifdef is your friend, and code obfuscation is the price you pay for it. To avoid it, sometimes creating (or using an existing) wrapper library makes sense.

Compiler Differences

Using language extensions beyond the C/C++ standard breaks portability. However, it is often easy to hide this compiler differences behind macros.
Luckily these differences usually hit at compile time, which makes them really easy to spot.

Assembler Code

Using assembler code limits portability to the machine it is written for – often even to the assembler/compiler in use. The easy solution is to use conditional compilation and makefile magic to choose between different assembler sources or a portable C implementation of the algorithm. This leads to vastly different runtime behavior – compare the (assembler optimized) versions of zlib or OpenSSL for i386 vs. some other processor of comparable speed using the C version. Both are examples of portability not interfering with performance -- on i386 machines.

Byte Order

Multi byte values are stored in different orders in memory, depending on the CPU type, and sometimes the selected byte order (where the selection is done by the firmware, the mainboard or the operating system). Most programs do not care, as long as no external binary representation (a file or a network connection) is needed. It is easy to explicitly code the binary representation on a byte-by-byte basis, or use some existing conversion macro to "fix" the byte order and then dump the result.

Example for the byte-by-byte approach:

```
#define PUT_32BIT(f, val) \
       fputc((val >> 24) & 0xff , f);    \
       fputc((val >> 16) & 0xff , f);    \
       fputc((val >> 8) & 0xff , f);     \
       fputc(val & 0xff , f)

PUT_32BIT(f, origVal);
```

Example for the "fix-in-place-then-dump" approach:

```
int32_t val;
val = htole32(origVal) ;
fwrite(&val, sizeof val, 1, f) ;
```

The latter approach is used a lot in the BSD networking code. It is not always more clear than the byte-by-byte approach, but it allows to convert complete structs in place and have structure definitions that resemble the on-wire/on-disk format.

The problem with byte order is, that sometimes values slip through the byte order fix when the CPU and the binary representation "accidentally" have the same byte order.

Machine Dependent Integer Type Sizes

Different processors have different requirements for "good" representations of data types. The C language definition is intentionally weak to allow implementations (the compiler) to choose the optimal representation for most of the predefined data types. This makes writing certain kind of code pretty hard, and later versions of the C standard recognized this and added fixed sized and fast integer types.

There are various alias names for types, defined by the C standard (or Posix). Examples are time_t, size_t, off_t. Some machines use the same size for two types like int and long on i386 – explicitly allowed by the C standard. The compiler will not warn if you pass a pointer to a int argument to a function expecting a pointer to a double on i386. It will warn (and the code will break) on machines that use different sizes for these types, like all 64bit architectures currently known.

Some CPUs prefer unsigned characters (ARM, PowerPC), so the default "char" type is "unsigned char", while others use "signed char" as "char" (i386). Again, the C standard explicitly allows this. Usually this is no problem, unless code erroneously uses char instead of int or tries to test properties of characters with homegrown, arithmetic expressions instead of relying on the ctype character classification macros:

```
char c;
if (c < 0) { … }
```

Of course on a `char == unsigned char` machine this condition will never evaluate to true (and gcc warns about this). Another example of code that works with signed chars but does fail with unsigned chars is:

```
char c;
FILE *f;
while ((c = fgetc(f)) != EOF) { … }
```

Clearly the intend is to read characters from f until the end of file. EOF is, unfortunately, defined as –1. No problem, as long as c can store a –1. On machines with `char == unsigned char` this just does not work. Note that fgetc() is defined as

```
int fgetc(FILE *);
```

and the return value is an integer that either is EOF or a value represent able as `char`.

A slightly more involving example is based on a long time bug in NetBSD: `struct timeval` (used for example in the `gettimeofday(2)` system call) is defined as:

```
struct timeval {
      long  tv_sec;            /* seconds */
      long  tv_usec;           /* and microseconds */
};
```

There are two bugs here. First, `struct timespec`, defined by POSIX.1b, featuring nanosecond resolution, should be used instead nearly everywhere. This will eventually happen. The second bug is the usage of `long` for the `tv_sec` member. Of course, and Posix even requires this, it should be a `time_t`. Unfortunately NetBSD does care about binary compatibility, which means a great effort and versioning lots of functions when this changes – so it has not been fixed yet. Now consider code that does

```
struct timeval tv;
time(&tv.tv_sec);
```

This code looks good at first glance, and on i386 it will actually compile and work. On sparc64 it will not work (and gcc will at least warn about it). What is wrong here? The sparc and sparc64 port of NetBSD share a lot of code, and sparc64 provides binary compatibility, i.e. it can execute sparc (32bit) binaries. To ease this compatibility, `time_t` has been kept the same size on sparc and sparc64 – a 32 bit value, making it an `int` on sparc64. The above code passes a pointer to a long (64bit value) to a function expecting a pointer to a 32bit value. While gcc only warns about it, the runtime result of the call will not be as expected - the result is written into the 4 high bytes of the long value, effectively shifting the time left by 32bits, followed by 32bit of more or less random noise.

A whole class of problems related to pointer-to-different-sized-integer problems originates from the family of functions taking printf()-like format strings. Fortunately gcc nowadays checks these format strings pretty well against the actual parameters, so this class of problems has become a compile time problem. Before, wrong argument sizes led to stack corruption, crashes and exploits. Consider you scan some input that has time_t values as decimal strings. On i386 you could do:

```
time_t t;
FILE *f;
fscanf(f, "%ld", &t);
```

and it will work. On sparc64 it will mangle your stack – expecting a long (8 byte) value at the address passed to fscanf, when only a 4 byte value is pointed too. You might be lucky and run into alignment trouble (see below), or the function call will overwrite four bytes on the stack. Newer C standard version define additional modifiers for some of the alias types, you can use `"%zd"` for size_t and `"%td"` for ptrdiff_t – but no direct way exists to scanf/printf a time_t. The above code should do:

```
time_t t;
long temp ;
FILE *f;
fscanf(f, "%ld", &temp);
t = temp;
```

Printing is a bit easier, a simple cast is enough:

```
time_t t;
printf("%ld\n", (long)t) ;
```

If you use a fixed size integer type, the standard defines corresponding format specifier macros since it is unclear which “native” integer type is related to the fixed size type.

So instead of

```
int64_t t;
printf("%lld\n", t);        /* works on i386 */
```

Better do :

```
#include <inttypes.h>
int64_t t;
printf("%" PRId64 "\n", t);        /* portable */
```

Machine Dependent Types

Strictly speaking there can not be any portable code using floating point values in externally visible contexts. Luckily today there are basically only IEEE 754 floating point format machines out there in the wild – famous counterexamples are the VAX, older Cray machines, many mainframes and some ARM implementations. The alpha CPU uses IEEE floating point, but has additional load and store instructions that convert to and from some VAX formats – probably only used in OpenVMS.

Using floating point values in binary files or communication streams is hard to do portably. Valid options often used are strings, fixed point formats – or just relying on a known-size IEEE format (thus limiting portability, but with known, small impact).

There are multiple sizes of IEEE floating point values, the most used ones being 32bit (on i386 known as "float") and 64bit (on i386 "double"). Some 64bit machines use 128 bit floats, and the C standard avoids nailing down sizes. Some (RISC) machine architectures define 128bit float formats and instructions for them, but implementations do not actually support them – relying on software emulation by the OS (or avoiding them altogether by the compiler). The UltraSparc is an example of this.

For internal calculations, you simply can not depend on properties of a particular floating point implementation. Luckily most algorithms are not critical in this regard, and if they are, the corner cases have been studied extensively by mathematicians, leading to well known workarounds in the few critical cases. Therefore real world portability problems, besides the binary "on-disk" representation mentioned above, are rarely caused by floating point issues.

Alignment

Since type sizes vary, the size of structure and the offset of members in a structure may vary when porting from one CPU to another. Some CPUs require or recommend layout of data in memory at certain addresses. A common alignment rule is: everything should be aligned to an address dividable by its own size. For example, a 32 bit integer value should live on an address dividable by four. Some CPUs have even bigger alignment rules, or honor greater alignment by better performance (like 16 byte alignment for SSE2 instructions on modern i386 CPUs).

The way CPUs penalize disrespect of this alignment rules varies:

- need more time for the unaligned access (i386)
- ask the operating system to emulate the access (alpha)
- silently get the results wrong (some ARM variants)
- signal a bus error (sparc and most other RISC)

The most forgiving (i386) makes a bad test platform for alignment issues.

The compiler handles all alignment issues, in the typical cases. Again the exceptions are structured designed for on-disk (or on-wire) layout, using compiler directives to pack structure members. This structures need to be handled carefully, and typically need byte order treatment too. Note that the "byte-by-byte" approach to byte order also handles any alignment issues as a side effect.

Once you start playing tricks on the compiler, it can not handle alignment any more. The most common trick is a pointer cast, but this deserves its own chapter.

Pointer Casts

```
void encode_packet(unsigned char *packet, size_t len, uint32_t tag)
{
        *((uint32_t*)packet) = htole32(tag);
        packet += sizeof tag;
        len -= sizeof tag;
        ...
}
```

Code like this happened, in the real world, in 2004.
What is the problem? It will work on i386, so there can not be anything wrong with this!

Once the unsigned char pointer (which has no special alignment constraints) is casted to a pointer to a type with different (larger) alignment constraint, the compiler gives up and leaves us alone with the responsibility to do the right thing. But what is the right thing to do? A CPU with strict alignment requirements (like sparc) can not access a 32 bit value at three out of four addresses pointed to by an unsigned char pointer. It needs to read them byte by byte and create the 32 bit value – similar to what the PUT_32BIT macro did above for byte order treatment. So the code may as well do it explicitly that way, if the alignment of the packet pointer is unknown – or just access the address, if we know for sure that it is aligned. The compiler could not tell.

The other way to write this is:

```
void encode_packet(unsigned char *packet, size_t len, uint32_t tag)
{
        uint32_t le32tag = htole32(tag);
        memcpy(packet, &le32tag, sizeof le32tag);
        packet += sizeof le32tag;
        len -= sizeof tag;
        ...
}
```

Another bad pointer cast example, this time on the right hand side:

```
void decode_packet(unsigned char *packet, size_t len)
{
        struct header *h = (struct header)packet;
        switch (h->tag) {
                ...
        }
}
```

The alignment constraints of h might not match the address in packet. The C standard guarantees that a cast does not change the address stored in a pointer variable [in this case – the exact rule is slightly more involved], so the cast will not fix the alignment.
Trying to avoid the unaligned access by using memcpy:

```
void decode_packet(unsigned char *packet, size_t len)
{
	struct header h, *p = (struct header*)packet;
	memcpy(&h, p, sizeof h);
	switch (h.tag) {
		...
	}
}
```

This example is still bad; the C standard allows the compiler to assume proper alignment for a struct header pointer, which we do not guarantee here. The compiler might optimize the memcpy by using multiple 64bit load/stores instead of a byte-by-byte copy, resulting in a bus error.

Good version:

```
void decode_packet(unsigned char *packet, size_t len)
{
	struct header h;
	memcpy(&h, packet, sizeof h);
	switch (h.tag) {
		...
	}
}
```

(Obvious error checking and asserting that len is big enough is left as an exercise to the reader)

Rule of thumb: left hand side casts are pure evil. There never is an excuse for them.

Other pointer casts are evil too. They might prevent the compiler from warning for real problems or using optimizations. The C standard basically says that two pointers can only alias if they have the same type (or signed/unsigned/qualified variants thereof), or one of them is a char pointer. You are allowed to cast to the "right" type of an object when assigning via a pointer, so the compiler assumes all variables of the casted-to type might have changed. Playing bad games with casts may prevent the compiler from applying optimizations it might have done for properly designed (cast-less) interfaces.

Ioctl Calls

The ioctl(2) API is an example of functions that deliberately take void pointers as a placeholder for something else, leaving no chance for the compiler to verify the call. The actual data type passed as the third argument to ioctl(2) depends on the value of the second argument. Mismatching them is hard to detect at compile time. Unfortunately sometimes operating systems differ in small details there.

Some time ago a long standing bug in the X source code has been discovered: the FIONREAD ioctl, used to determine if some input is immediately readable from a file descriptor, takes a pointer to an int as its third argument (according to POSIX and the NetBSD code). Some other operating systems, namely windows, and apparently IRIX in 64 bit mode, use an unsigned long pointer there. The X source used code like this:

```
long arg;
ioctl(fd, FIONREAD, (char *)&arg);
return (int)arg;
```

On NetBSD/sparc64 this led to the number of bytes to read being placed in the upper 4 bytes of the long arg – and then only the lower 4 byte being used. Of course it worked fine on alpha – by chance, due to the different byte order.

Conclusion

NetBSD claims to be the worlds most portable operating system. NetBSD runs on fifty four different system architectures (ports), featuring seventeen machine architectures across fifteen distinct CPU families, and is being ported to more. All this happens from a single code base, which has gone a long way to achieve this portability. Sometimes there are still portability bugs found, and there is still, despite a lot experience, new code added that shows bugs like the portability problems described in this paper. Portability is one of the major project goals for NetBSD.

It would be nice if more and more application software (that is, the software our users like to run on our operating system) would follow this lead.

You can not expect to get portability for free, but if you are aware of possible problems and the easy solutions/workarounds, portability is painless. Porting early helps – so everyone get a NetBSD/sparc64 system and once your software runs on your primary target, port it there ;-}

But I'm not a developer...how can I contribute to Open Source?

Dru Lavigne <dru@theopensourceadvocate.org>

October 11, 2004

1 Abstract

The BSDs and other Open Source projects have made great strides in the past decade. But it doesn't take a marketing analyst to see that we've still a long way to go. We've all seen friends and family fight their way through viruses and spyware because their operating system of choice is "easier to use". We've all had customers or bosses who chose solutions involving hefty license fees over free software because of the "support". And we've all at one point or another had to maintain a non-BSD install because there wasn't an Open Source application available that provided a required functionality.

The question is, "what can be done about this state of affairs?" As a developer, your role in Open Source is fairly straightforward: write good code, add new features, fix bugs. However, the non-developer's role is less clearly defined. This often leaves the end-user feeling isolated and intimidated; not only are they unsure how to contribute, they may not even see their contribution as worthwhile.

This talk will address the importance of non-developers to the success of Open Source software as well as what you, as a developer, can do to assist in this process. We'll do this by taking a look at both the developer's and non-developer's points of view which will bring to light various misconceptions. Once these are dealt with, concrete roles can be defined.

2 Introduction

Those who stay in Open Source tend to be technically minded. For the developer, sysadmin, and power user, the ability to look under the hood and play with the guts of a system holds a powerful attraction. Yet, history has proven time and time again that the success of a project isn't based solely upon technical merit. There is a danger in only approaching things from the technical perspective.

This presentation provides an opportunity to temporarily step away from the technical aspects of Open Source in order to view the larger picture–which includes the "softer" side of computing. As we explore different perspectives, take the time to consider the questions which are raised and to examine your own attitudes and actions.

3 The Novice User:

Being a new user can be overwhelming at times–and that initial learning curve often seems insurmountable! While you may be brimming with enthusiasm over your new discoveries, you probably also feel the twang of inexperience. How can you possibly contribute when you feel surrounded by gurus?

Start by becoming aware of the resources available to you. Browse the available mailing lists and subscribe to those that look interesting. Find an IRC channel. Go to your project's documentation and FAQ pages; bookmark them and start reading. Pick up a cheap system to try things out on so you don't ruin your main system. Become good friends with Google and look for tutorials and how-tos. Scour your local library or friends' bookshelves for quality books. Above all, join a local user group! Even if you're a shy person, hanging out with like-minded people does wonders for softening that learning curve. If you can't find a group that concentrates on your particular operating system or project, join a similar group. Keep your eye out for installathons, conferences and local events which you can attend.

Once you find the resources, don't be afraid to use them. However, do yourself and others a big favour by doing your research first. For example, are you considering posting a question to a mailing list? Before shooting off that email, use Google to see if the answer to your question is already documented somewhere. If you can't find an answer, have you researched the best mailing list to post the question? And more importantly, have you read the posting rules for that list? Have you included the details of your situation? Other users may be gurus, but they're not psychics. And if the rules say "don't email the developer directly", don't email the developer directly!

Never underestimate the value of your perspective. Remember, you're not the only new user out there. If you receive an answer and still don't understand something, ask for clarification. You'll not only help yourself, you'll help the other users who were afraid to ask. And, while documentation is steadily improving, there are still pockets of non-userfriendly information out there that assume a lot of previous knowledge. If you have to struggle to figure something out, write a clearer explanation and post it for the benefit of others. Keep that documentation ball rolling: whenever you find a useful tutorial or article, take a minute to send an appreciative email to the author.

Before you know it, you may find that you've matured into...

4 The Experienced, Non-developer

As your technical skills improve, try not to forget what it was like to be a new user. For now the coin has turned and newer users are looking to you for advice and encouragement. Try not to let a bad or harried day result in a rude or negative answer to a question. For example, while you may be tempted to respond with "RTFM", something as simple as a "You're in luck. There's a really good tutorial on this very subject at URL" maintains the user's dignity while pushing them off into their own explorations. If you don't have the time or patience to find suitable documentation, you don't have to answer the post!

Remember, it's not so much what you say as how you say it. What you say in a public forum continues to have a ripple effect long after you've forgotten about it. You can probably remember at least one incident when you were made to feel stupid for asking a question. Don't repeat that cycle for someone else.

Attitude has impact. Watch for negativity when answering posts and for fanaticism when promoting your project over others. Watch the tone of your posts. Do you find yourself promoting Open Source successes or negatives? Take a step back if you find yourself repeatedly saying "that won't work" or "things were always that way".

Now's the time to start taking a closer look at the prevalent atmosphere of your community. Does it promote exclusivity? Are "outsiders" welcome or does it feel more like a "techie boy's club"? Is there an obvious gender skew? A recent survey published in Software Development magazine (1) probed the reasons why Open Source suffers from a greater lack of female participation than the IT industry as a whole. Does your community tolerate inappropriate remarks or turn a blind eye to discriminatory behaviour? Does it encourage female users to actively participate and promote the successes of those that do? As a female, are you lurking in the woodwork or are you an active participant?

What about your project's goals? Does it require users to maintain an all or nothing attitude? Are users denigrated if they continue to use an alternate operating system instead of converting over to your operating system of choice? After all, isn't open source about choice?

Have you found that your initial enthusiasm has devolved into a general apathy? How many times in the past year have you:

- found a bug and neither reported it nor submitted a patch?
- scoured the Internet in order to to cobble together a solution yourself without sharing it so the next person didn't have to reinvent the wheel?

Will Open Source continue to get better if noone:

- contributes feedback,
- files bugs,
- submits feature requests,
- donates money or hardware,

- creates patches,
- writes missing documentation,
- provides examples, support, or advocacy?

NO!

Take a look at your own systems. On top of the operating system you're probably running literally hundreds of Open Source applications. When's the last time you visited the websites of those applications, or for that matter, Sourceforge or Freshmeat? You won't have to look very far to find a project in dire need of exposure, beta testers or documentation. *Pick a project and contribute!*

5 Using your Pet Peeves

Sometimes you see so much need you don't know where to start. Well, anything that bugs you is a good indicator of potential action on your part. For example:

- *Do grammatical errors, undocumented switches and out-dated manpages drive you batty?* Sounds like you should investigate the submission process of your community's documentation project.
- *Frustrated by the lack of documented material available in your native language?* You may be a much needed translator.
- *Bugged that your place of work or school doesn't use Open Source?* Install what you can on the systems available to you. Show others what they're missing. Start an informal group which meets at lunch or after hours. Volunteer to setup and maintain an Open Source computer lab at your child's school or at the local seniors residence.
- *Tired of reading bad press regarding Open Source?* Write opinion pieces and product reviews which showcase the positives of Open Source. This is a largely neglected area as most writing efforts concentrate on technical how-tos. The irony is that we live in a world where artists and musicians who spend obsessive amounts of time honing their craft are "creative"; likewise, athletes are "driven". Yet the technically adept are relegated to "geek" or "nerd", both of which have connotations of social ineptness and fall pretty low on the "cool" scale. Negative terms like these do damage, but often this damage is unseen as potential users are turned off and go elsewhere. Media pieces are needed showing just how cool and fun it is to be involved

in Open Source. Or perhaps you'd prefer to speak to students at your local high school or community college.

- *Don't like the look of your project's website?* Volunteer your HTML talent.

Perhaps you're more in tune with your likes than your dislikes:

- *If your dream job would involve installing and playing with software,* find a project that is looking for beta testers or install current on a spare system and join the current mailing list.
- *Are you an avid reader?* Most publishers offer free books if you'll write and post a review. When you're finished, donate the book to your local user group's lending library.
- *Love to talk and help others?* Find a forum, list, or IRC channel and look for opportunities to assist new users.
- *Were you born to be around other people?* Organize and/or volunteer at a local installfest. Even if the venue's main event isn't specifically about your project, see if you can get a booth.
- *Do you wish to see Open Source promoted in business?* Research and create a list of vendors in your area that support, use or promote open source. Find or create an association that promotes Open Source in your community.
- *Just don't have the time?* Use your coffee money to buy a CD subscription for yourself and a friend. Commit yourself to make a dent in a project's donation page. Sponsor someone's admission fee to a conference.
- *Do you want to see more articles and how-tos but feel your writing isn't up to par?* Online ezines such as Daemonnews may still be interested in your submission. Do you find that your favourite ezine isn't always published on time? It's probably because the ezine is short on proofreaders and formatters.

Finally, don't be intimidated when you're...

6 Dealing with Developers

As a developer, you may not realize just how intimidating you are to non-developers. There is a definite mystique regarding the ability to decipher what appears to be so much mumbo-jumbo (to the non-programmer) in order to solve a problem. You really do speak another "language" (pun intended). Developers also tend to attract the spotlight, much more so than the average Open Source user. These factors discourage many users from reporting bugs or making feature requests.

There are several things you can do as a developer to promote participation within your project. First, ensure there is a supportive infrastructure including mailing list(s), FAQs, to-do lists and a bug reporting system such as Bugzilla (2). Make it clear on your website what a user can do to contribute and the steps they should follow when doing so. If you really don't have the time to respond to individual requests, say so and give users an alternative. If you do respond to individual requests, please don't patronize the user or trivialize their request. What's obvious to you is probably something they've never even heard about.

When dealing with a developer, the usual rules regarding researching your request apply, if not more so. When you send an email to a mailing list it is seen by thousands of users, some of which probably have the time, inclination and knowledge to reply. When you send an email to a developer, you're subject to one personality and one person's time constraints. Make sure the developer is willing to receive personal email and that the email includes the information required to respond in a helpful manner. Finally, don't take it personally if the developer doesn't immediately respond.

7 Food for Thought

I'd like to leave you with some additional points to consider. Not every user of Open Source is interested in becoming a technical guru. This is actually a good thing and such users should be encouraged to contribute to Open Source in non-technical ways. Imagine the boost to Open Source if it could benefit from the talents of those who:

- are involved in media or have promotional skills.
- work in government, understand governmental process and have contacts within government.
- are participants within the legal process.
- are educators who are familiar with the process of creating and submitting curricula.
- are successful business persons.

Remember, successful "networking" has nothing to do with cabling or switches. Our ability to make contacts and to encourage others to contribute their talents directly affects the success of Open Source.

8 Additional Resources:

http://www.theopensourceadvocate.org

9 Footnotes:

1. http://www.sdmagazine.com
2. http://www.bugzilla.org

Track B Saturday

Notes:

Mac OS X binary compatibility on NetBSD: challenges and implementation

Emmanuel Dreyfus

September 2004

Abstract

Binary compatibility is the ability to run binaries from foreign Operating Systems (OS) with a minimal performance penalty. It is limited to binaries built for the same processor family.

In this paper, we describe how binary compatibility works in NetBSD and then we concentrate on the challenges we need to overcome in order to execute Mac OS X binaries. Finally we present the current status of the project, together with the roadmap for the future.

It is assumed that the reader is familiar with Unix system programming.

1 Binary compatibility

NetBSD has a long record in binary compatibility with other operating systems. Provided a program was built for the same processor, NetBSD is able to execute it despite the fact that the program was not built for NetBSD but for Linux, FreeBSD, Solaris, IRIX, or many others. In this part, we will have a quick look at how binary compatibility works.

1.1 kernel and user mode

UNIX systems have two distinct mode of operation: user mode, and kernel (or system) mode. In user mode, the operating system executes code provided by users. This code is run with restricted privileges. It has limited access to the computer's memory, and usually no access at all to the hardware.

When running in kernel mode, the OS is only executing trusted code, which was loaded at boot time. This code is known as the OS kernel. The kernel has full access to the memory and hardware. It is here to provide services to user programs: giving access to the hardware, scheduling processes, and enforcing resource allocation and protection.

The transition from user mode to kernel mode occurs on events called traps. A trap is a hardware or software exception that suspends user process execution, and gives control to kernel code. The kernel will handle the exception, after which it may return to user mode and resume the execution of the user process, or it may destroy it. Example of traps are division by zero, memory faults (accessing any virtual addresses where no physical memory is mapped), timer interrupts (that are used to switch between user processes), or system calls. System calls are software traps called by user processes to request access to resources controlled by the kernel. They can be seen as functions called by the user process executing with kernel privilege.

System calls are used to perform a lot of different tasks, ranging from reading from a file to creating a new process, or setting network communication options. The behavior of each system call is documented in section 2 of the manual. The operation described above are therefore documented in `read(2)`, `fork(2)`, and `setsockopt(2)`.

Each system call has a number, typically ranging from 0 to a few hundred. On many processors, system calls are invoked by loading CPU registers with the system call numbers and parameters and by calling a CPU instruction that cause trap. Here is an example of PowerPC assembly that call the `fork()` system call on NetBSD:

```
li      r0,2        # 2 is the system call number for fork()
                    # r0 is the register holding the system call number
sc                  # sc is the CPU instruction that causes the trap.
```

There is a clean separation between user mode and kernel mode. User processes run on top of the kernel with very little knowledge of what is inside a system call. They just expect a behavior documented by kernel developers in a set of man pages. Most programs do not care about kernel internals and will just work if you change the kernel, as long as the system call behavior is left unchanged. This is how binary compatibility works: by emulating the kernel behavior. The user program runs at full speed and is fooled into thinking it runs on the kernel it was built for, whereas it is really running on the NetBSD kernel.

1.2 System call tables

As we explained earlier, when a trap is encountered, control is transfered to the kernel. The kernel calls a function known as the trap handler to take care of the exception.

When the trap is a system call, a particular trap handler – the system call handler – is invoked. The system call handler looks in a table for the function implementing the system call – this is the system call table. The system call number is used as an index in the system call table.

Here is an excerpt of the file that defines the system call table in NetBSD. This file, which is named syscalls.master, is not written in C language. A shell script uses syscalls.master as input to produce several files in C language.

```
1      STD     { void sys_exit(int rval); }
2      STD     { int sys_fork(void); }
3      STD     { ssize_t sys_read(int fd, void *buf, size_t nbyte); }
4      STD     { ssize_t sys_write(int fd, const void *buf, size_t nbyte); }
5      STD     { int sys_open(const char *path, int flags, ... mode_t mode); }
```

The first idea behind binary compatibility is to have multiple system call tables: the native system call table for native processes, the Linux system call table for Linux processes, and so on.

As emulated OSes usually provide equivalent functionality as NetBSD does, the system call tables used to emulate them tend to mostly contain wrappers that call native kernel functions after doing some argument translation. When the native code succeeds the reverse argument translation takes place, or if the system call fails, the appropriate error code is returned. In some rare situations, an emulated system call has no native counterpart and it must be completely re-implemented.

Of course, the kernel must have a clear idea of what system call table must be used for a given process. It does that by keeping track of the emulated OS for each process. At process launch time, in the execve() system call, the OS emulation is discovered and the appropriate system call table is selected. This system call table will then be used for any system call the process will issue.

There are a few other problems that must be taking into account, including the transitions from kernel to the user process: when the process is first launched, or when it catches a signal, the kernel must setup the stack and registers in the same way the emulated kernel would have do. This is implemented in NetBSD by having emulations hooks for that operations, so that a given binary compatibility layer can perform its particular setup.

Another major problem is dynamic linking. If an emulated program is dynamically linked, it will open shared libraries, which must be built for the same OS. The program usually expects these foreign libraries to be in the same place and with the same names as NetBSD native libraries.

The solution to that problem is to have a shadow root directory for foreign processes, where files are first looked up before looking at the real root. For instance, when a Linux binary attempts to open /usr/lib/libncurses.so, the NetBSD kernel will first attempt to open /emul/linux/usr/lib/libncurses.so and if that fails, it will try /usr/lib/libncurses.so.

The shadow root directory is also extremely useful to store configuration files for foreign binaries, as the file names often clash with their native counterparts.

2 Mac OS X binary compatibility challenges

Binary compatibility is an old feature in NetBSD, and the kernel now contains enough compatibility code to easily emulate another Unix flavor. But when we came to Mac OS X binary compatibility, we hit several new challenges that had never been encountered while working on binary compatibility layers in NetBSD.

Here is a list of the most challenging problems:

- Mac OS X uses an executable format called Mach-O. NetBSD knew nothing about Mach-O. It was only able to run ELF, ECOFF, and a.out binaries.
- Mac OS X is an hybrid system, based on a Mach kernel and a BSD kernel. It features a dual system call table: positive system call numbers refer to the BSD system call table, and negative system call numbers refer to the Mach system call table. NetBSD never had to handle such an odd setup.

  While Mac OS X's kernel BSD interface is very close to NetBSD kernel interface, and therefore can be very easily emulated, the Mach interface has just nothing to do with a Unix system. This is a complete set of system calls with no NetBSD equivalents that must be completely re-implemented.
- NetBSD provides user programs with hardware device access through traditional Unix device files, in the /dev directory. Mac OS X provides such a feature for a limited subset of devices: mostly storage units and terminals. Other devices, such as video, keyboard and mouse, are made available to user programs through the IOKit, an extended device driver interface specific to Mac OS X.
- And last but not least, NetBSD uses the X Window System for the graphic user interface, whereas Mac OS X uses Quartz, a PDF based display system. The two systems are similar but incompatible: they are both based on a client/server scheme, which a display server handling the video output and graphic user interface applications behaving as client. The protocol used between the clients and the server is different.

3 Running Mach-O binaries

Binary compatibility layers are usually developed in an incremental way. The developer attempts to run a simple binary built for the target OS, and of course that fails. The failure is caused by a feature of the target OS kernel that the NetBSD kernel does not emulate properly. The developer fixes that by implementing the missing feature emulation, retry running the binary, finds another failure, and things repeat until the emulation is good enough to transparently run the foreign OS binary.

When trying to implement a feature emulation, a few sources of informations are useful: the man pages and other system documentation, and the system include files. For some OSes, such as Linux or FreeBSD, a possible source of information is the kernel sources, but contrarily to a popular belief, having access to the target OS source code does not help very much. What we are interested in is the target system behavior, not its implementation. It tends to be much more productive to write unit test programs, to run them on the target OS, and to see what they do, rather than reading the target OS kernel sources to understand what it should do.

Usually, the work is done on system calls emulations. But when trying to run a Mac OS X binary, the first show stopper was not occurring on a system call. The NetBSD kernel was unable to actually load and launch the foreign binary. The reason for that failure was that Mac OS X uses an executable format called Mach-O, which was not known to NetBSD.

NetBSD knew about the legacy a.out and the newer ELF executable formats. Migration from a.out to ELF occurred a long time ago, but NetBSD retained the ability to run a.out binaries, for the sake of backward compatibility. The support for running binaries in two different executable formats on the same kernel helped a lot when introducing Mach-O executable format support. Let us have a closer look on how it was done.

The `execve()` system call is responsible for running a new binary. It uses a `struct execsw` table called the exec switch to perform its duties. Each entry in the exec switch defines the operations used to load the binary for a given OS emulation and executable format. For example that switch contains entries for NetBSD ELF binaries, NetBSD a.out binaries and Linux ELF binaries, among many others. The first part of the job was to add an entry for Mac OS X Mach-O binaries.

This entry defines a few operations for loading Mach-O binaries, which had to be implemented:

First, the probe function. Each entry in the exec switch defines a probe function whose task is to tell if the entry is able to run a given binary or not. The probe function looks at the executable header to decide if it is a binary it can handle. In `execve()`, the kernel first walks the exec switch, using the probe functions to discover which entry should be used to execute the binary.

Once the kernel knows which entry in the exec switch should be used, it uses another function – the `makecmds()` function – to load the executable. This function is responsible for setting up the process virtual memory space, loading the text and data sections from the executable, and setting up stack space.

Things are more complicated when implementing the `makecmds()` command for Mach-O binaries than for ELF binaries. Mach-O binaries can be fat, that is, they can contain text segments for different architectures. Of course that needed to be taken into account so that the right text section would be loaded.

Another difference is in object loading. When loading an ELF executable, the duty of the kernel is just to load the executable, and possibly a dynamic linker. With Mach-O binaries, the kernel also has to load any dynamic library needed by the executable. The kernel duty stops there, as it is not required to load the libraries used by the libraries used by the program: the dynamic linker will do that from userland.

Finally the kernel needs to setup the stack and populate it with arguments and other information the userland startup code expects. There are some Mac OS X oddities here: On many systems, the stack of a newly created program starts by the argument count and a pointer to the argument vector (the famous `argc` and `argv` arguments to the `main()` function of a program written in C). Mac OS X processes' stack starts by a pointer to a copy of the Mach-O executable header, the argument count and the pointer to the argument vector.

In NetBSD kernel source, all the code for running Mach-O binaries was written by Christos Zoulas. It can be found in `src/sys/kern/exec_macho.c`. The exec switch is defined in `src/sys/sys/exec.h` and `src/sys/kern/exec_conf.c`. The code used to setup the stack of Mac OS X processes is available in `src/sys/compat/darwin/darwin_exec.c`.

4 Mach system calls

As we explained earlier, the Mac OS X kernel is an hybrid system, featuring two sets of system calls. Apple used the following scheme: positive system call numbers are used for the BSD interface, whereas negative system call numbers are used for the Mach interface. Mac OS X user processes mix system calls to both parts of the kernel: BSD and Mach. The BSD part features a well known Unix kernel interface, while the Mach part's interface has just nothing common with Unix.

Mach is a microkernel, implementing virtually anything as processes called servers.

some servers run in user mode, other run in kernel mode. The kernel only provides two services: process scheduling and Inter-Process Communication (IPC). IPC is extensively used in a Mach-based system, because a process that need a system resource will send a request to a server process instead of sending it to the kernel as it does on a Unix system.

Most of the Mach kernel interface is therefore devoted to Mach IPCs. The Mach microkernel implements a message passing system, which uses objects called messages, ports, and rights.

A Mach message is a packet of data with a 24 bytes header and a payload that can carry any information.

A Mach port has nothing to do with TCP or UDP ports. It is a message queue maintained in the kernel. A single process reads from it whereas multiple processes may write to it. Each message is sent with a destination port and a source port, so that the server can answer the request to the right process.

A Mach right determines a process access right on a port, such as a send right or a receive right. The right is a kernel resource that the process acquires, uses and releases, just like files in Unix. For the process, a right is handled through a 32 bits integer, which is usually called a port name. You can think of port names as Unix file descriptors. Rights can be obtained through some Mach system calls, or they can be carried by messages. For instance, when a process receives a message from another process, the message normally carries a send right to the source port so that the receiver can reply to the message.

The Mach system call table can be found in `src/sys/compat/mach/syscalls.master` in NetBSD kernel sources. The most frequently used system call is `msg_trap()`, which is used to send and receive Mach messages. This system call was quite complicated to implement since it has to handle both sending and receiving, asynchronous reception, timeouts, and other features. Moreover, it has to juggle with port and right lists. `msg_trap()` is implemented in `src/sys/compat/mach/mach_message.c`.

In order to make debugging easier, the `ktrace` command on NetBSD was modified to record Mach system calls, a feature which is not available in Mac OS X's `ktrace`. NetBSD `ktrace` is even able to dump Mach messages. Here is an excerpt of the kernel trace for the Mac OS X's `ls` command running on NetBSD. It gives a good insight on the way Mach IPC works.

```
   541 ls        CALL  host_self_trap
   541 ls        RET   host_self_trap 35454977/0x21d0001
(...)
   541 ls        CALL  reply_port
   541 ls        RET   reply_port 35454979/0x21d0003
   541 ls        CALL  msg_trap(0xbffedd70,3,0x18,0x30,0x21d0003,0,0)
   541 ls        MMSG  host_page_size [202]
       000       00001513   00000000   021d0001   021d0003   ................
```

```
     010        00000000  000000ca                               ........
 541 ls         MMSG  host_page_size reply [302]
     000        00001200  00000028  00000000  021d0003  .......(........
     010        00000000  0000012e  00000000  00000000  ................
     020        00000000  00001000  00000000  00000008  ................
 541 ls         RET   msg_trap 0
```

host_self_trap() gives a send right to the host port, which is used to request host-specific configuration. reply_port() allocates a receive right to a new port. Then msg_trap() is used to send a message to the host port and get a reply.

The Mach message header is defined in src/sys/compat/mach/mach_message.h. It contains 6 words of 32 bits:

```
typedef struct {
        mach_msg_bits_t msgh_bits;         /* flags */
        mach_msg_size_t msgh_size;         /* message size */
        mach_port_t     msgh_remote_port;  /* destination port */
        mach_port_t     msgh_local_port;   /* source port */
        mach_msg_size_t msgh_reserved;     /* unused */
        mach_msg_id_t   msgh_id;           /* Message Id */
} mach_msg_header_t;
```

The message Id is used to characterize the message meaning and the payload type. Here, a message Id of 202 sent to the host port requests the machine memory page size. The message payload is void. The server listening behind the host port responds by a message with a 24 bytes payload. That message ends with a 32 bits word containing the requested value (here 0x1000, or 4096), followed by a 64 bits message trailer.

It is interesting to note that using the Unix kernel interface, the same operation can be done by doing a single call to sysctl(), requesting the hw.pagesize variable.

5 Mach kernel servers

The Mac OS X kernel implements many Mach servers inside the kernel. They can be reached through three ports: the host port, the task port and the thread port. The host port is used to request configuration about the machine the caller is running on, whereas the task and thread ports, are used to request system resources on behalf of the calling task (a task is a Unix process in Mach terminology) or thread.

Because binary compatibility takes place at the kernel boundary, NetBSD had to implement all the Mach kernel servers. Instead of implementing different kernel threads servicing the requests, the NetBSD kernel services each request in the process context of the caller. The message Id is used to lookup the function that will get the request message and produce the reply.

The NetBSD implementation of msg_trap() uses a table defined in src/sys/compat/mach/mach_services.master to select the appropriate function. Like the syscalls.

master file, this file is not written in C, and a shell script is used to produce various C files from it. Here is an excerpt from that file:

```
200     STD     host_info
201     UNIMPL  host_kernel_version
202     STD     host_page_size
203     UNIMPL  memory_object_memory_entry
204     UNIMPL  host_processor_info
205     STD     host_get_io_master
206     STD     host_get_clock_service
207     UNIMPL  kmod_get_info
208     UNIMPL  host_zone_info
209     UNIMPL  host_virtual_physical_table_info
210     UNIMPL  host_ipc_hash_info
```

Here we find the information that a Mach message with message Id 202 must be handled by the host_page_size() function. This function is defined in src/sys/compat/mach/mach_host.c. Here is its complete implementation:

```
int
mach_host_page_size(args)
        struct mach_trap_args *args;
{
        mach_host_page_size_request_t *req = args->smsg;
        mach_host_page_size_reply_t *rep = args->rmsg;
        size_t *msglen = args->rsize;

        *msglen = sizeof(*rep);
        mach_set_header(rep, req, *msglen);

        rep->rep_page_size = PAGE_SIZE;

        mach_set_trailer(rep, *msglen);

        return 0;
}
```

mach_host_page_size_request_t and mach_host_page_size_reply_t are the request and reply Mach messages, defined in src/sys/compat/mach/mach_port.h:

```
typedef struct {
        mach_msg_header_t req_msgh;
} mach_host_page_size_request_t;

typedef struct {
        mach_msg_header_t rep_msgh;
        mach_ndr_record_t rep_ndr;
        mach_kern_return_t rep_retval;
        mach_vm_size_t rep_page_size;
        mach_msg_trailer_t rep_trailer;
} mach_host_page_size_reply_t;
```

host_page_size() call mach_set_header() and mach_set_trailer() to fill the Mach header and trailer for the reply packet, and it sets the requested value: the page size.

There are many other Mach kernel services, most of them being unused by Mac OS X binaries and therefore left unimplemented in NetBSD. The most used services deal with port, task, thread, and memory management. For example a task can spawn a new thread or request a memory mapping by sending a Mach message to its own task port.

And of course there are services implemented as user-level daemons. We do not have to do anything special for them: the Mac OS X binary can be executed on NetBSD on the top of the Mac OS X binary compatibility layer, and they will provide the adequate service to other Mac OS X processes running on NetBSD.

6 Mach IPC bootstrap

The Mach IPC is at the core of Mac OS X, it is used intensively everywhere. As a result, Mac OS X processes have a recurrent problem: how to obtain a send right on the port of a given server?

For kernel-level servers, Mac OS X processes use the host, task and thread ports, obtained by the `host_self_trap()`, `mach_task_self()`, and `mach_thread_self()` system calls.

For user-level servers, a bootstrap mechanism is needed. The `mach_init` daemon is responsible for providing this service.

`mach_init`, always running with PID 2, is the first user-level process spawn on Mac OS X. The reader used to Unix will probably wince: Usually, `init` is the first process spawn, and it has PID 1. Mac OS X even happens to have an `init` process with PID 1.

In fact, `mach_init` is really the first user process spawn, with PID 1. It then forks, and the father, with PID 1, uses `execve()` to run `init`, while the child, with PID 2, continue executing `mach_init`. This odd move is there to ensure that `init` still gets the PID that a lot of Unix programs expect, while `mach_init` is launched first.

Once it has forked the traditional Unix's `init`, `mach_init` behaves as a directory service. It registers to the kernel as being the bootstrap process, thus making one of its ports available to all processes through another special port any process can access: the bootstrap port.

Each time a server process starts, it uses the bootstrap port to send a register message to `mach_init`, giving the ports on which it is servicing and the service name. And when a client process needs a send right to a server port, it uses the bootstrap port to send a message containing the service name. `mach_init` will reply by a message carrying a send right to the server port.

Implementing support for this was easy. We just had to implement the Mach service used by `mach_init` to register its port: `task_set_special_port`. The NetBSD kernel maintains a global variable called `mach_bootstrap_port`, and once `mach_init` registers, any process requesting a send right to the bootstrap port will get a right to the registered port. That way things work as expected.

But there was one small problem: mach_init checks its PID, and will only behave as the bootstrap process if it is started with PID 1. On NetBSD, PID 1 is always used by init, so it is not possible to book it for mach_init. It is not possible either to spawn mach_init at system startup instead of init.

The solution was finally to fool mach_init into thinking it has the PID 1 whereas it is not the case. mach_init uses the BSD system call getpid() to obtain its PID. We just had to recognize that mach_init was started and have getpid() answering 1 instead of the real PID.

But the kernel has no way of recognizing mach_init. We therefore use the help of the system administrator, which tells the kernel that it runs mach_init using a sysctl variable.

This is done with the following shell command:

```
sysctl -w emul.darwin.init.pid=$$ && exec /emul/darwin/sbin/mach_init
```

Using sysctl, the system administrator informs the kernel that a Mac OS X process running with this PID (remember that $$ is the shell's PID) should be fooled into thinking that it's PID is 1. Then we use exec to run mach_init without forking a new process, thus retaining the same PID.

That way, mach_init thinks it is the first process spawned, with PID 1, and it behaves as the Mach bootstrap process.

Of course, Mac OS X being an open source OS, it would have been possible to patch mach_init sources so that it does not check its PID and always behave as the Mach bootstrap process, but the goal of binary compatibility is to run unmodified binaries from the foreign OS, therefore the sysctl hack choice.

7 Handling binaries built for Mac OS X.3

Unix processes tend to use a a few library functions such as memcpy() or bzero very often. In a dynamically linked executable, calling a library function means walking various tables, which is time consuming. Starting with Mac OS X.3, Apple introduced a nifty optimization: the kernel maps a few pages of code containing various utility functions at the end of each processes' address space. Theses pages are called the comm pages.

The functions can be reached at an absolute memory address that is carved into the stone. Calling such a function is blazingly fast because it just involve branching to a well known address, there is no more performance loss caused by dynamic linking.

Another advantage of this approach is that the kernel can map optimized versions of the functions that make use of some particular optional hardware feature, such as Altivec on the PowerPC G4. The user process does not have to deal with checking the hardware ability and/or the kernel version, nor does it have to include multiples versions of some function to match various optional optimization.

For the binary compatibility layer developer, this optimization caused surprising failures. As most of Mac OS X.2 command line programs worked on NetBSD, any binary built for Mac OS X.3 quickly died with a segmentation fault. After some investigation, it became obvious that something weird was taking place: the segmentation fault was caused by the program jumping at a fixed absolute address where nothing was mapped. Testing on Mac OS X with `gdb` did show that a page of memory containing code was mapped at that fixed address. Because it was mapped before the process did any system call, it was obvious that the kernel had to do it.

Fortunately, when running on Mac OS X, `gdb` displayed symbols when dumping the memory in the comm pages, so it was not that difficult to understand the purpose of the code they contained. The last part of the job was to actually implement the functions in the comm pages. It had to be done in assembly, as some functions had to fit in a small slot of memory, a constraint that a C compiler is not able to understand.

The assembly code for the comm pages can be found in `src/sys/arch/powerpc/powerpc/darwin_commpage_machdep.S`. Peter Grehan, Srinivasa Kanduru, and Wolfgang Solfrank helped a lot writing it.

8 Running Aqua application and emulating the IOKit

By implementing the Mach IPC and a collection of kernel services, we have been able to run most command-line binaries from a Mac OS X system. Programs using the X Window graphic interface are also likely to work, though this has not been tested. But what we are really aiming for is running programs using the native Mac OS X graphical user interface, known as Aqua.

Aqua is based on a display system called Quartz, which is similar but incompatible with the X Window system. Quartz uses a display server and Aqua applications are clients that talk to the display server. In Mac OS X.2, the display server is called `WindowServer`. In Mac OS X.3, it was replaced by `QuartzDisplay`.

The main problem with running Aqua applications is to have a Quartz display server so that they can actually display something. We have several ways of obtaining a Quartz Display server running on NetBSD.

- Write a Quartz Display server from scratch. That solution is clearly not the way to go, since it means re-implementing the code for managing various video boards.
- Write a Quartz to X11 bridge, in order to reuse X11 video hardware support. The problem with that solution is that we need to discover how Quartz clients talk with the Quartz server in order to implement the bridge. Moreover, it is not certain that this bridge is posible to implement, because Quartz seems to have many more features compared to X Window.
- Use Mac OS X's Quartz display server and run it on the top of our binary compatibility layer. We chose that path, with the idea that it will be easier to reverse engineer the Quartz client/server interface and work on a Quartz to X11 bridge once we will have the Quartz display server running on the top of NetBSD.

We therefore tried to run `WindowServer`, and later `QuartzDisplay`, on NetBSD. The big problem we encountered was to provide an emulation for the IOKit, which is the device interface used by the Quartz display server to access the video board, the keyboard and the mouse.

Traditional Unix device interface is rather simple, not to say poor. Devices are available through special files in the `/dev` directory, which can be opened for reading or writing. Any operation that cannot be implemented through a read or write is implemented through the `ioctl()` system call, a general purpose I/O function used for virtually anything.

Mac OS X uses the traditional Unix device interface for disks and terminals, but most of the hardware is not available through that interface. Video boards, keyboards and mice are only available through a big object oriented framework known as the IOKit. The IOKit provides mechanisms based on Mach IPC for discovering and accessing hardware. It defines device classes, and device drivers are objects within the classes.

The display server uses two device classes: `IOFramebuffer`, to access the video, and `IOHIDSystem`, for the input systems (keyboard and mouse). In order to run the display server, we needed to implement an `IOFramebuffer` driver and an `IOHIDSystem` driver, plus enough Mach services from the IOKit interface so that things just work.

The IOKit interface is quite complex and not exciting enough to be covered here. The two drivers were more tricky.

Drivers in the `IOFramebuffer` class implement access to a framebuffer. There are a few Mach services used to read and write configuration information about the framebuffer, such as pixel depth, color palette, gamma table, screen size, and so on.

`IOFramebuffer` drivers must also make two memory mappings available to a user process that would request them:

- The framebuffer itself.
- A page of memory shared between kernel and userland where cursor related configuration is stored. A user process can use that area to read the cursor position or to modify the cursor visibility, for instance.

The biggest problem was to provide access to a framebuffer while the NetBSD kernel does not know about all the various video boards that may be present in a machine. Fortunately the Power Macintosh boot environment, known as Open Firmware, provides a framebuffer to NetBSD. It is slow and not configurable, but it is available.

The `IOFramebuffer` driver in the Mac OS X compatibility layer maps the framebuffer from the wscons console driver. On a Power Macintosh, we know that this framebuffer will be at least the Open Firmware framebuffer. If the NetBSD kernel supports accessing the framebuffer of the video board in a more efficient way (for instance, NetBSD kernel's `machfb` device is able to use the ATI Mach64 framebuffer), then the `IOFramebuffer` driver will automatically use it.

As we do not support hardware accelerated cursors yet, the shared memory page used for the cursor configuration is not emulated beyond a simple mapping of zero-filled memory.

All the code for the `IOFramebuffer` driver is located in `src/sys/compat/darwin/darwin_ioframebuffer.c`. The code that implements the IOKit interface is located in `src/sys/compat/mach/mach_iokit.c`

The `IOHIDSystem` driver was the most difficult part. Like `IOFramebuffer`, it works by mapping a shared page of memory between the kernel and the user program. The kernel will write keyboard and mouse events to a queue located in that page. The display server will read them from the queue. This mechanism saves a lot of system calls for reading user input.

In our implementation, when the display server maps the page of shared memory, the kernel spawns a new kernel thread that opens the wscons console driver. This kernel thread, called `iohidsystem`, reads wscons input events, converts them to `IOHIDSystem` events, and writes them to the queue. That way, the keyboard and mouse events are made available to the display server, without the need to hack some hooks in the NetBSD native input drivers for keyboard and mouse.

The code for the `IOHIDSystem` driver can be found in `src/sys/compat/darwin/darwin_iohidsystem.c`

Working with the Quartz display server was not easy, as the IOKit interface is really complex and the user program is not open source. Fortunately, XFree86 provides a X server called XDarwin, which can use the IOKit. Working with XDarwin did help a lot, since it was possible to poke debug `printf()` in it to understand how things were going on.

We are now able to run a fully functional XDarwin on NetBSD/macppc. This means that the IOKit emulation, the `IOFramebuffer` and `IOHIDSystem` drivers are good enough to be actually used. Unfortunately, as of today, the Quartz display server won't work yet. Debugging the problems that prevent it from working is the next item on the project TODO list.

Conclusion

Mac OS X binary compatibility in NetBSD grew quite large. It now features more than 20.000 lines of C and assembly code. For now support has only been written for NetBSD PowerPC ports. NetBSD could also run Darwin/i386 binaries, provided the machine dependant part is ported (it accounts for 5% of the code).

As of today, NetBSD-current is able to run most command line tools from Mac OS X. Mac OS X Programs using the X11 graphical user interface such as Matlab or Open Office should work too, though nobody explored that area yet. It is difficult to give an idea of when NetBSD will be able to run Aqua applications, because we do not really know the issues we are going to encounter and solve. Moreover, some event could shorten the delay: if some Quartz display server become available for NetBSD (for instance as an open source Mac OS X remote desktop project), it would remove the major problem.

The biggest and hardest part of the work so far was to implement the Mach IPC and various Mach services related to task, threads, ports, memory, and many others resources. The IOKit was one of the most complex part of the picture.

On the performance front, a comparison of Mac OS X and emulated Mac OS X on NetBSD would be interesting. Binary compatibility does not cause major performance loss. If the native implementation of a feature is much more efficient than the target OS implementation, then the emulation can even be faster than the original for that feature. Such a comparison will probably be the subject of a future paper.

Acknowledgements

Thanks to Christos Zoulas for reviewing this paper.

The flaf filesystem

Søren Jørvang

October 12, 2004

Abstract

Flaf ("First Load All Files") is a new filesystem that tries to achieve modern features like crash robustness and high metadata performance but small code size by adapting to the ever-widening gap between sequential and seek speeds of modern disks.

Contemporary filesystems like SGI xfs and Veritas VxFS that provide crash robustness by way of journalling, and good metadata performance using indexed directories, are complicated and massive in terms of code size. Much of this complexity deals with the synchronization of heavy state between memory and disk.

It turns out that because the relationship between (memory size, sequential disk bandwidth, number of files stored) very often tends to stay within certain bounds, it is feasible to sidestep the complexity of the likes of xfs by having virtually no interdependencies on-disk (no directory lists, no block bitmaps), but instead building the structure at mount time.

The basis of flaf was the idea of spending "5-10 seconds at mount time and 5-10 percent of physical memory" in return for small code size in combination with modern features like crash robustness and good metadata performance.

In flaf, all metadata resides in a single disk block per file and everything else follows from that.

The final version of this paper did not meet the deadline for inclusion in the printed proceedings and will be published on the conference website after the conference.

Integrating ALTQ QoS into FreeBSD

by ADRIAN PENIŞOARĂ*

Abstract

Most of the modern operating systems of our times, either commercial or open source, have developed support for network quality of service (QoS). The BSD camp makes no exception thanks to the ALTQ framework developed by Mr. Kenjiro Cho from Sony Computer Science Labs, Japan. The OpenBSD and NetBSD folks have already integrated this framework into their core distributions, while for the FreeBSD project the integration is just about to start.

ALTQ, standing for "ALTernate Queueing", is a framework of (network) queueing disciplines and related components required to offer resource sharing and Quality of Service capabilities. It is thought to be a development framework, although it has been successfully used in production environments. ALTQ development is now taking place in the KAME repository where it has been recently imported.

Some of the biggest challenges that ALTQ poses are the modifications to the network drivers needed to enable the shaping of the packet flows and figuring out the locking changes needed for the new fine-grained architecture of the -CURRENT branch. The standard BSD network architecture, inherited from the original BSD distribution, makes it hard to integrate alternative queueing disciplines. Instead, hooking points had to be devised in order to place the packets in controlled network queues. The task becomes even more difficult if you have to take into account the newly introduced locking semantics.

Recently work has been started and evolved quite rapidly towards the integration of the "pf" packet filter from OpenBSD; the nice point is that the queueing functionality has been achieved using the ALTQ package proposed for integration. This offers an alternative hooking point for the packet flows and queueing disciplines management.

In the long term, I believe that a new approach is needed in the design of the networking stack which would permit using alternative queueing mechanisms and a unified method for packet tagging for various networking services (traffic shaping, firewalling, etc). Although the *BSD family is renown for its networking performance, the modern IT community demands more network versatility from the nowadays' operating systems.

Note: the full content of this article will be available on the conference's website.

*You can contact the author at *ady@freebsd.ady.ro* or see his webpage on *www.ady.ro*

Lightweight FreeBSD package cluster in a jail

Version 2004-09-22

Dirk Meyer
dinoex@freebsd.org
http://people.freebsd.org/~dinoex/

How to setup a `jail' to ensure clean package builds on FreeBSD, making it easy to distribute customized packages for more than one machine. Also, how to use this scripts as a test environment to compile new or modified ports.

1. Generation of the packages

1.1. Motivation to start this project

When the ports I maintained became more complex, I needed a clean environment to test existing and new ports.

1.2. Differences to the `bento' cluster system

A. It works in a `perl' free environment

FreeBSD packages are built on the `bento' or `pointyhat' cluster. The infrastructure for this is designed for parallel package building on more than two machines. It makes heavy use of `perl' scripts, which was the main reason I have dropped the idea of using this as a base for testing my own ports.

I had many problems with `perl' scripts. The transition from `perl4' to `perl5' was difficult, with many portability problems and script incompatibility.

B. It runs inside or outside a `jail'

Running `chroot' as on the cluster will not protect your applications on the host. Some ports try to be very smart by stopping services while programs install or uninstall, e.g. the `cups' port still kills all running `cupsd' processes. By running in a `jail', this and other side effects can be prevented.

I didn't want to endanger my base machine, and I wanted to get rid of the many packages that are installed as `build only' dependencies. In a clean `jail', I don't have to worry about a lot of side effects, such as auto detection of extra installed libs, which quickly become an unexpected dependency. For example, the `samba' port sucked in the `popt' lib without registering it as dependency. So, when I removed `rpm' and the unused `popt' from the system, `samba' refused to start. This problem has been fixed already, but there are several similar pitfalls.

You can start one `jail' IP-address for only fetching the ports you need, and use a different `jail' IP-address for the build. This allows you to block any access from your build jail in your firewall, giving you additional protection from Trojan programs.

C. KISS - Keep it small and simple

Only the tools in the base system, and `cvsup', are used. I use `make' to extract all vital information, and use `shell' to get everything working together. For processing the `plist', I used a few lines of `awk'.

It is designed for a single jail building all packages you need. Then, you can install the ready made packages quickly on every machine. One `nfs mount' of `/usr/ports' contains all files you need to install or upgrade a system. This is the reason why I chose `/usr/ports/local/update/' as the default location for my tools.

1.3. `INDEX' problems, Customizing

In the FreeBSD ports tree, the `INDEX' files are very large, caching a lot of information from 11739 ports (FreeBSD 5.3 BETA). It weights about 5.5M for each branch. These files are only up to date when a RELEASE is generated, and in the meantime, you have to rebuild the INDEX each time a `Makefile' in the ports tree changes. A lot of package tools use these files, but the information may not be consistent with the environment the user has chosen. Dependencies may be changed at any time by options and settings in `/etc/make.conf'. So you waste a lot of effort rebuilding the `INDEX' files and keeping the information in there up to date. If using the target `fetchindex', you get a snapshot that is more recent, but you can't trust that the information in there is still correct.

Instead, I save bandwidth by ignoring all `INDEX' files and rely only on the current information provided by the ports `Makefiles'. The trade off in computing time is smaller than I estimated, because only the needed ports are processed, but the lack of caching shows on the big dependency trees like in `gnome' or `kde' and will slow you down again.

Parts of my ideas have been discussed on the ´freebsd-ports' mailing list, and I believe in avoiding the `INDEX' files at all will be a step in the right direction.

1.4. Preserving good builds and reusing them

The FreeBSD package cluster builds all ports on each run. Having limited amount of computer power, I decided to reuse packages untill they become obsolete. Therefore, each dependency is built as a package, so it can be reused for new builds of a port and for other ports depending on it.

The current ports system supports this too by setting `/etc/make.conf'.
I recommend the following settings for each host you want to install packages on.

```
DEPENDS_TARGET=package
USE_PACKAGE_DEPENDS=yes
```

1.5. How to decide if a port needs to be rebuilt

Instead of using the package name, I decided to use only one key to represent a port or dependency: the path to the port's directory. This allows fast reference to the originating `Makefile' and access to all the current information we need.

Step 1.
I determine the affected ports by path name. This is easy because my scripts can use the standard target `all-depends-list'. There is no need to distinguish between build, lib and run depends, as building the package needs them all.

```
root@jail:/usr/ports/archivers/arj# make all-depends-list
/usr/ports/converters/libiconv
/usr/ports/devel/autoconf253
/usr/ports/devel/gettext
/usr/ports/devel/gmake
/usr/ports/devel/libtool13
/usr/ports/devel/libtool15
/usr/ports/devel/m4
/usr/ports/devel/p5-Locale-gettext
/usr/ports/misc/help2man
/usr/ports/textproc/expat2
```

Step 2.
I check each dependency for the exact version. I get this by using the target `package-name'. If the exact version is not installed, we have to add or build the dependency. If a required package is newer than an already existing package of the port we want, a rebuild is triggered. This is very useful to catch any changes in the `build only' dependency.

```
root@jail:/usr/ports/devel/autoconf253# make package-name
autoconf-2.53_3
```

1.6. Build only the packages you need

Several operation modes are supported. When you call the script with the directory of a port, it will only check and build the port and its dependencies. This can be used for emergency patches and for simply testing a new port.

Calling the script with a file will process the directories listed in that file for rebuild. This is useful to build a couple of related ports.

Finally, if you omit the file, then a stored host specific list will be used.

1.7. Performance in FreeBSD4 vs. FreeBSD5

Running the `jail' on FreeBSD 4.x is about 4 times faster than with FreeBSD 5.x. There are two reasons for this: the `bzip2' compressed packages takes much longer to access or extract, and the `gcc3.4' compiler takes up much more time. You can regain a bit of time by reverting the compression scheme for the packages to `gzip'.

1.8. Caveats: e.g. Linux emulations

There are some ports that won't build in the `jail'. The `linux' emulation is a prominent example of this. In this case, you can build it outside with `chroot', and the resulting package can be used to build dependent ports in the `jail' again, until it becomes obsolete.

2. Keeping up to date

2.1. Installation and Layout

You extend your ports tree by creating the directory:

```
root@host# mkdir -p /usr/ports/local/update
```

Download the files from `http://people.freebsd.org/~dinoex/batch/'
and don't forget the `README'.

You may add extra ports under `/usr/ports/local', I create a slave port there when I need more than one package from the same port.

Inside the `jail' I use the minimal `/etc/make.conf':

```
USA_RESIDENT=NO
WRKDIRPREFIX=/image
PACKAGES=/usr/ports/packages
BATCH=yes
SUP_UPDATE?=yes
SUP?=/usr/local/bin/cvsup
SUPFLAGS?=-g -L 2
SUPHOST?=cvsup.de.FreeBSD.org
PORTSSUPFILE?=/usr/share/examples/cvsup/ports-supfile
```

2.2. Create a list of installed packages

You can create a starting configuration easy like this:

```
root@host# pkg_info -qao > /usr/ports/local/update/\
data/make-packages.hostname
```

This is sufficient, but you may optimize the order of ports to match your needs. Feel free to remove dependencies, so they won't be built once they are no longer in use, or put them in any order you prefer.

2.3. Stetting up the jail

Extract a release in a directory of your choice, or use a current or stable build with `make DESTDIR=/jail5 installworld'. I recommend you keep the `jail' up to date with each `buildworld'. In the host system, you should enable the `IPC' for the `jail'. This is needed for some databases and for the `sendmail' ports. I set this in `/etc/sysctl.conf':

FreeBSD 4.x:

```
jail.sysvipc_allowed=1
```

FreeBSD 5.x:

```
security.jail.sysvipc_allowed=1
```

In FreeBSD 5.x you have to mount the `devfs' in the jail. Some ports may require the `prcofs' mounted as well, if you do this you should mount `procfs' read only, I use read only `procfs' on the hosting system as well.

2.4. Running each day

The update cycle is easy. It can be run unattended, so you have the new packages at hand when you decide to update. I run `cvsup' to update the ports tree, then I have to get rid of obsolete packages. After removing log files from aborted builds, I am ready to rebuild all the missing packages.

2.5. Find outdated packages

I have to check each package for its origin and the registered dependencies. In case we detect a difference, we move the package aside, so we have the latest package around in case the new build might fail.

2.6. Running through the dependency tree

The scripts are very picky about each version, older or newer don't matter. You can even do a clean `downdate' if necessary, because an exact matching version is enforced on each run.

2.7. How to clean your `distfiles' directory

Once in a while you like to clean up your `distfiles' directory. This was much easier than I thought. I look for each `distinfo' file in the whole ports tree and compare the list with the list of downloaded `distfiles'. Each file that is no longer listed in the `distinfo' in any port can be safely moved away.

2.8. Small and full upgrades

Each run will generate the newest packages you need. Mostly it will need less than half an hour, but when a major dependency has changed it can run up to more than one day.

If you upgrade your base system, I recommend you move all packages away. Some ports have paths that change with the FreeBSD version, or includes and libraries in the base change, and if you keep the old packages, other problems might occur.

2.9. Naming problems when ports using auto detection

Some ports change their package name while build, and this will be recorded as a failure, but the built package will kept around. Setting build options ahead for the port will give you a clean build.

To set options for a specific port when settings in `Makefile.local' are not working.
I was successful to place this as a conditional in `/etc/make.conf':

```
.if ${.CURDIR} == "/usr/ports/multimedia/mplayer"
WITH_GUI=yes
WITH_GTK1=yes
WITHOUT_ESOUND=yes
CFLAGS+=-O
.endif
```

When this is more than 3 lines, I strongly recommend to build with a simple slave port. I found an example for this `/usr/ports/local/mplayer-extra/Makefile':

```
PKGCATEGORY?=   local
MASTERDIR=      /usr/ports/multimedia/mplayer

WITH_GUI=yes
WITH_SDL=yes
WITH_VORBIS=yes
WITH_XANIM=yes
WITH_FREETYPE=yes

WITHOUT_RUNTIME_CPUDETECTION=yes
WITHOUT_3DNOW=yes

.include "${MASTERDIR}/Makefile"
```

2.10. Errors

Well in active development of the ports, logfiles are preserved in the log subdir. Looking at current sample you can see three different types of files.

```
-rw-r--r--  1 root  wheel      24890 Sep 15 18:13
build,local,gnumail
-rw-r--r--  1 root  wheel        120 Sep 15 18:14 plist,
local,gnumail
-rw-r--r--  1 root  wheel      14925 Sep 15 18:23 build,
local,projectcenter.app
-rw-r--r--  1 root  wheel      11309 Sep 15 18:37 build,
local,preferences.app
-rw-r--r--  1 root  wheel        161 Sep 15 18:38 plist,
local,preferences.app
-rw-r--r--  1 root  wheel      57388 Sep 15 18:56 build,
local,gworkspace
-rw-r--r--  1 root  wheel        198 Sep 15 18:57 plist,
local,gworkspace
-rw-r--r--  1 root  wheel       5290 Sep 16 06:25 err,
local,gnumail_112
```

First there are successful build logs from ports with a name as: `build,<category>,<port>'. They stay around until the next successful build. In case of problems I found the `configure' output there is very helpful.

Second we may have files with a name such as: `plist,<category>,<port>'.
This indicates that after building this package and deleting its dependency, additional files or directories were found. Directories can be mostly ignored, but missing files can indicate a problem with the port or its dependency. If you are a maintainer, you can fix this by updating the `pkg-plist' of your port.

Last we have a log named `err,<category>,<port>'. This can be a build log in progress or an aborted build. In this case, a `diff' between the new log file and the last successful build log can be helpful to find the cause.

2.9. Improvements

This project was started in 2002, and has been tested at different locations. It's development is stable, the history can be reviewed via `cvsweb'.

DHCP: Unexplored Capacities

Yannick Cadin

October 12, 2004

Abstract

No current SOHO router comes without DHCP. Within a few years, DHCP became one of the most popular network protocols.

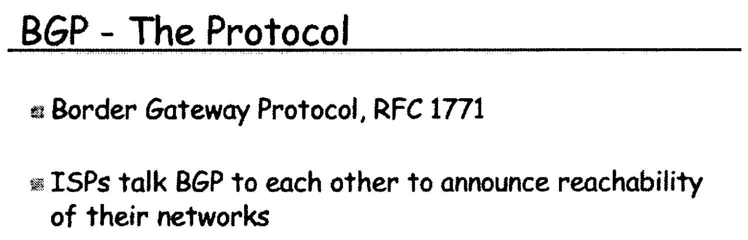
An insidious side effect of the sudden widespreading of an embedded version of DHCP is that many administrators never realized that this service offers many resources, most of which remain unused. DHCP is also prone to evolve and be extended.

There would be a lot to say about client-side customization, communication between DHCP servers and dedicated directories like DNS, or even specific uses by some operating systems.

The final version of this paper did not meet the deadline for inclusion in the printed proceedings and will be published on the conference website after the conference.

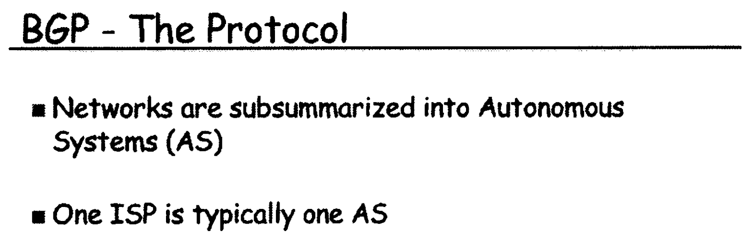
A Secure BGP Implementation

Henning Brauer <henning@openbsd.org>



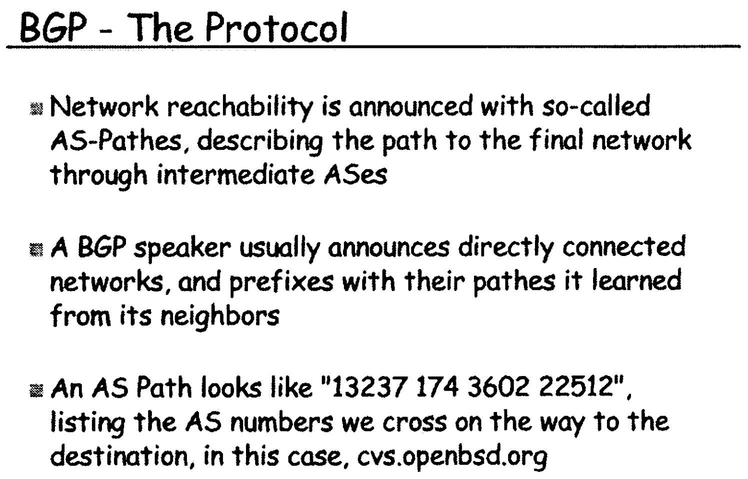
BGP - The Protocol

- Border Gateway Protocol, RFC 1771
- ISPs talk BGP to each other to announce reachability of their networks



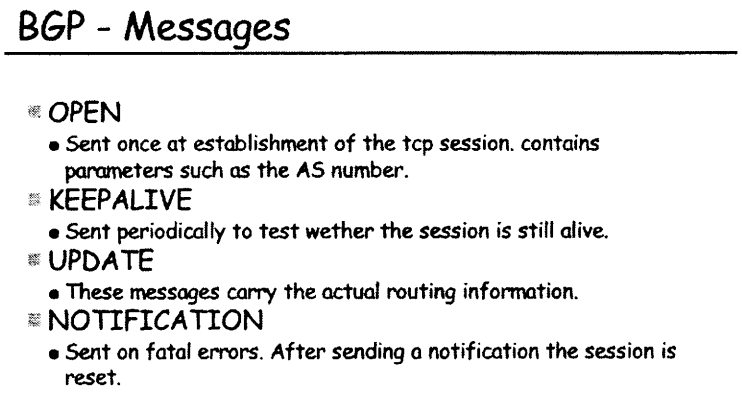
BGP - The Protocol

- Networks are subsummarized into Autonomous Systems (AS)
- One ISP is typically one AS



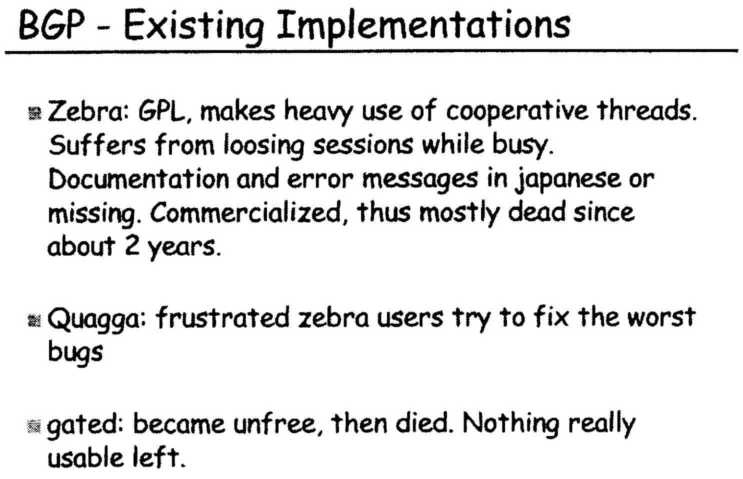
BGP - The Protocol

- Network reachability is announced with so-called AS-Pathes, describing the path to the final network through intermediate ASes
- A BGP speaker usually announces directly connected networks, and prefixes with their pathes it learned from its neighbors
- An AS Path looks like "13237 174 3602 22512", listing the AS numbers we cross on the way to the destination, in this case, cvs.openbsd.org



BGP - Messages

- OPEN
  - Sent once at establishment of the tcp session. contains parameters such as the AS number.
- KEEPALIVE
  - Sent periodically to test wether the session is still alive.
- UPDATE
  - These messages carry the actual routing information.
- NOTIFICATION
  - Sent on fatal errors. After sending a notification the session is reset.

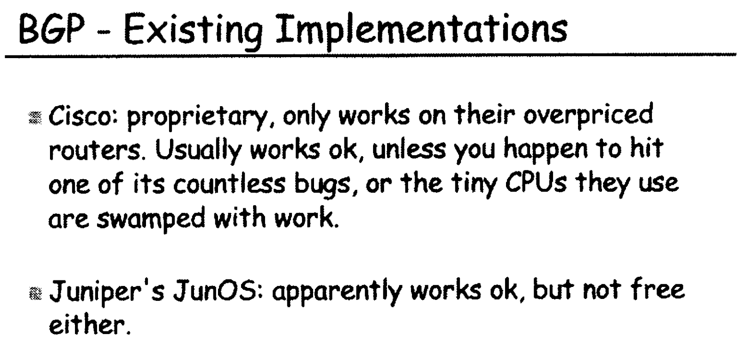


BGP - Existing Implementations

- Zebra: GPL, makes heavy use of cooperative threads. Suffers from loosing sessions while busy. Documentation and error messages in japanese or missing. Commercialized, thus mostly dead since about 2 years.

- Quagga: frustrated zebra users try to fix the worst bugs

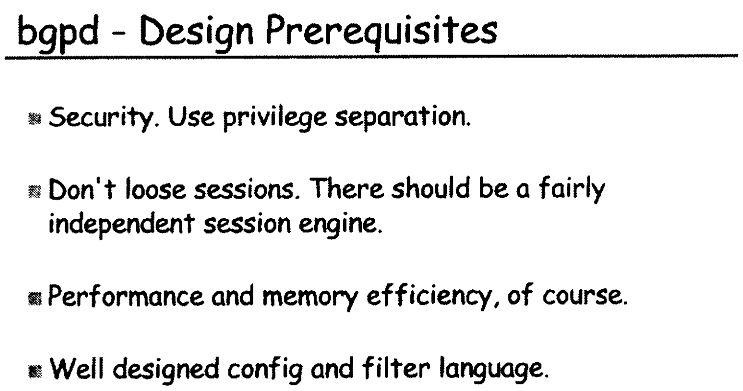
- gated: became unfree, then died. Nothing really usable left.



BGP - Existing Implementations

- Cisco: proprietary, only works on their overpriced routers. Usually works ok, unless you happen to hit one of its countless bugs, or the tiny CPUs they use are swamped with work.

- Juniper's JunOS: apparently works ok, but not free either.



bgpd - Design Prerequisites

- Security. Use privilege separation.

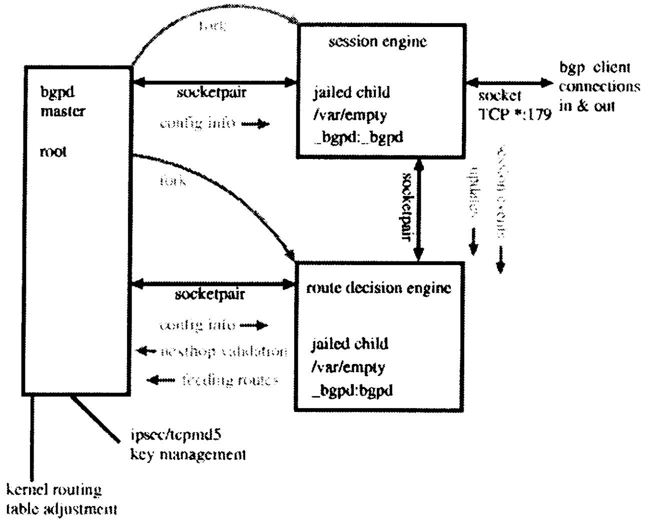
- Don't loose sessions. There should be a fairly independent session engine.

- Performance and memory efficiency, of course.

- Well designed config and filter language.

bgpd - Design

- 3 processes
  - Session Engine (SE): manages bgp sessions
  - Route Decision Engine (RDE): holds the bgp tables, takes routing decisions
  - Parent: enters routes into the kernel, starts SE and RDE

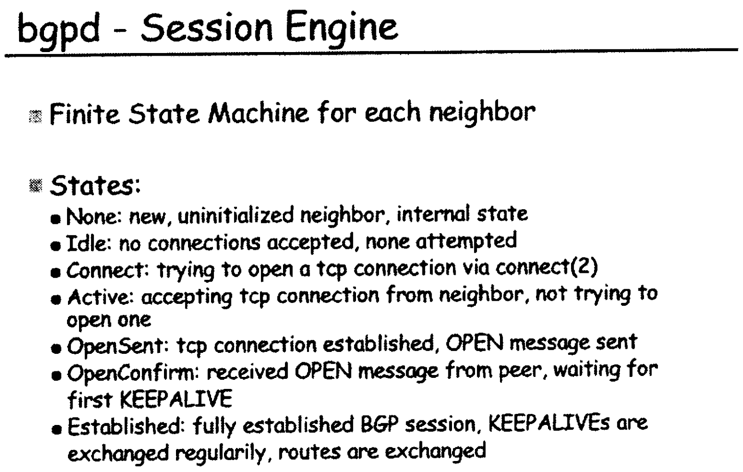


bgpd - Design

- Obviously, the Session Engine needs to be nonblocking, and use nonblocking sockets.
  - We need to handle all buffering ourselves.
- Invent an easy to use Buffer API
- For the internal messaging, invent an "imsg" API as well.
  - internal messaging is a core component in privilege separation
  - 40 message types now

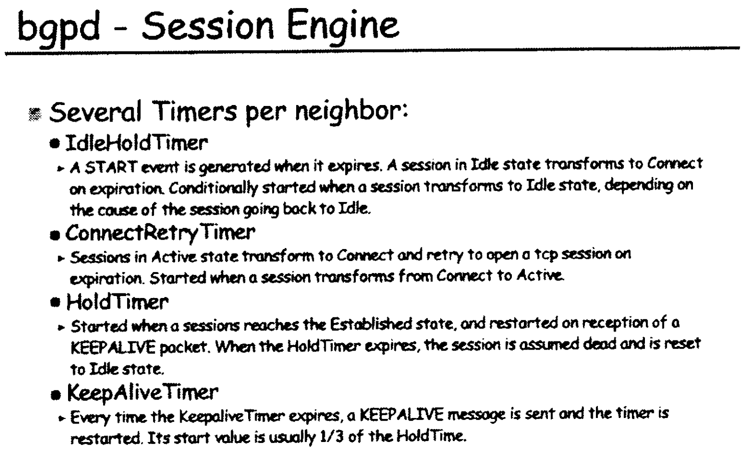
bgpd - Session Engine

- Maintains a listening tcp socket
- Opens tcp connections to neighbors
- Negotiates parameters with neighbors via OPEN messages
- Once a session is established, it sends KEEPALIVE messages regularily, and receives ones from the neighbors



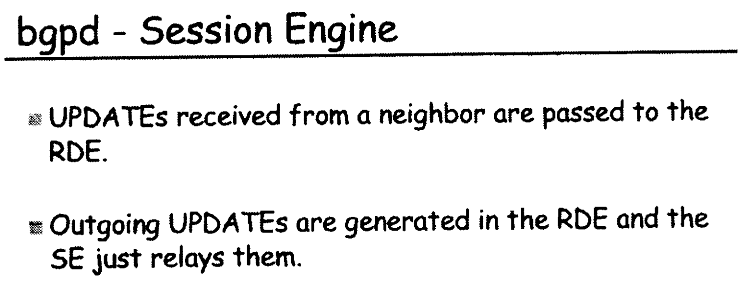
bgpd - Session Engine

- Finite State Machine for each neighbor
- States:
  - None: new, uninitialized neighbor, internal state
  - Idle: no connections accepted, none attempted
  - Connect: trying to open a tcp connection via connect(2)
  - Active: accepting tcp connection from neighbor, not trying to open one
  - OpenSent: tcp connection established, OPEN message sent
  - OpenConfirm: received OPEN message from peer, waiting for first KEEPALIVE
  - Established: fully established BGP session, KEEPALIVEs are exchanged regularily, routes are exchanged



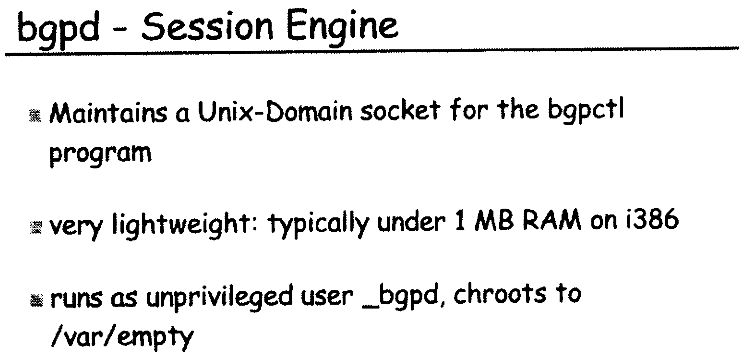
bgpd - Session Engine

- Several Timers per neighbor:
  - IdleHoldTimer
    - A START event is generated when it expires. A session in Idle state transforms to Connect on expiration. Conditionally started when a session transforms to Idle state, depending on the cause of the session going back to Idle.
  - ConnectRetryTimer
    - Sessions in Active state transform to Connect and retry to open a tcp session on expiration. Started when a session transforms from Connect to Active.
  - HoldTimer
    - Started when a sessions reaches the Established state, and restarted on reception of a KEEPALIVE packet. When the HoldTimer expires, the session is assumed dead and is reset to Idle state.
  - KeepAliveTimer
    - Every time the KeepaliveTimer expires, a KEEPALIVE message is sent and the timer is restarted. Its start value is usually 1/3 of the HoldTime.



bgpd - Session Engine

- UPDATEs received from a neighbor are passed to the RDE.
- Outgoing UPDATEs are generated in the RDE and the SE just relays them.



bgpd - Session Engine

- Maintains a Unix-Domain socket for the bgpctl program
- very lightweight: typically under 1 MB RAM on i386
- runs as unprivileged user _bgpd, chroots to /var/empty

bgpd - Route Decision Engine

- Maintains the Routing Information Base (RIB)
  - prefix table
  - AS path table
- BGP Filters run here
- Calculates the best path per prefix
- Generates UPDATE messages as needed

bgpd - Route Decision Engine

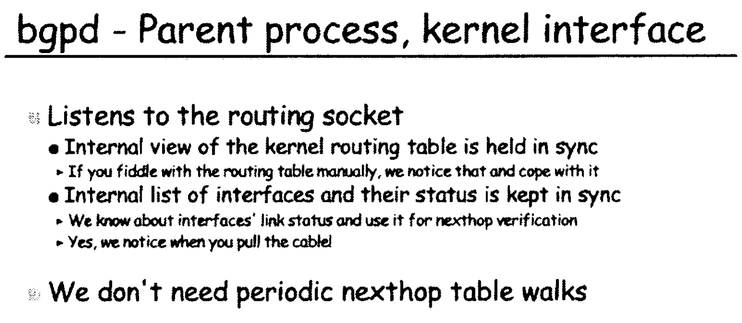
- RIB Layout
  - Split into many tables
  - Heavily linked
  - Avoid table walks
- UPDATE messages are processed to completion
- Generated UPDATEs are queued to use piggy-back optimization
- RIB Table and sessions can be dumped to mrt files

bgpd - Route Decision Engine

- Memory efficent
  - 1 full view needs around 20 MB
  - 2 full views need around 25 MB
- Fast
  - Around 10s to load a full view on a PIII 1GHz
  - Less than 5s to dump a full view to another router
- Runs as unprivileged user _bgpd, chroots to /var/empty

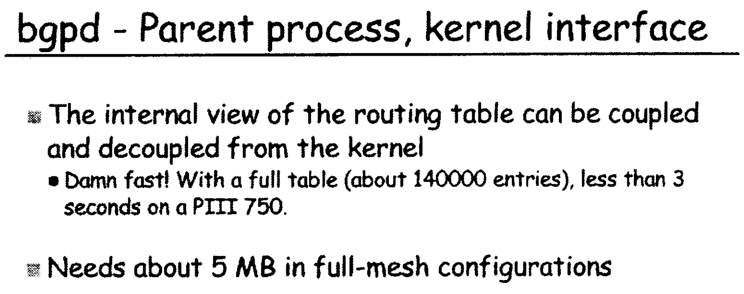
bgpd - Parent process, kernel interface

- Responsible for getting the routes into the kernel
- Maintains its own copy of the kernel routing table
- Fetches the kernel routing table and interface list on startup
- Does nexthop validation for the RDE



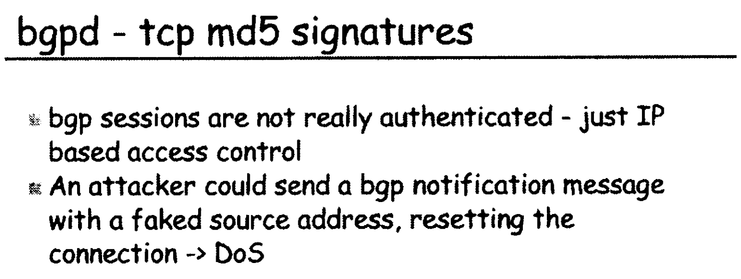
bgpd - Parent process, kernel interface

- Listens to the routing socket
  - Internal view of the kernel routing table is held in sync
    - If you fiddle with the routing table manually, we notice that and cope with it
  - Internal list of interfaces and their status is kept in sync
    - We know about interfaces' link status and use it for nexthop verification
    - Yes, we notice when you pull the cable!
- We don't need periodic nexthop table walks



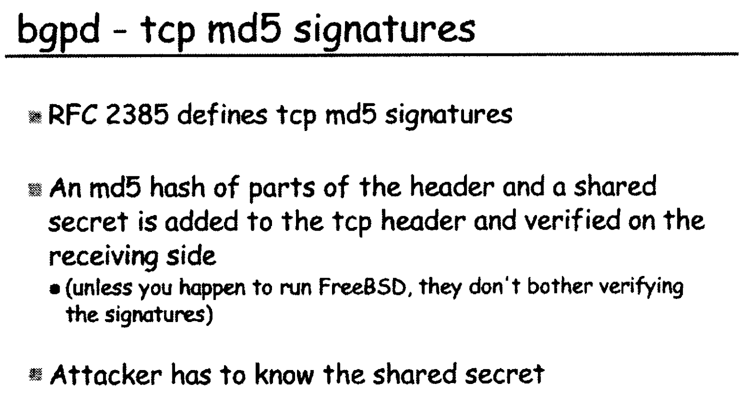
bgpd - Parent process, kernel interface

- The internal view of the routing table can be coupled and decoupled from the kernel
  - Damn fast! With a full table (about 140000 entries), less than 3 seconds on a PIII 750.
- Needs about 5 MB in full-mesh configurations



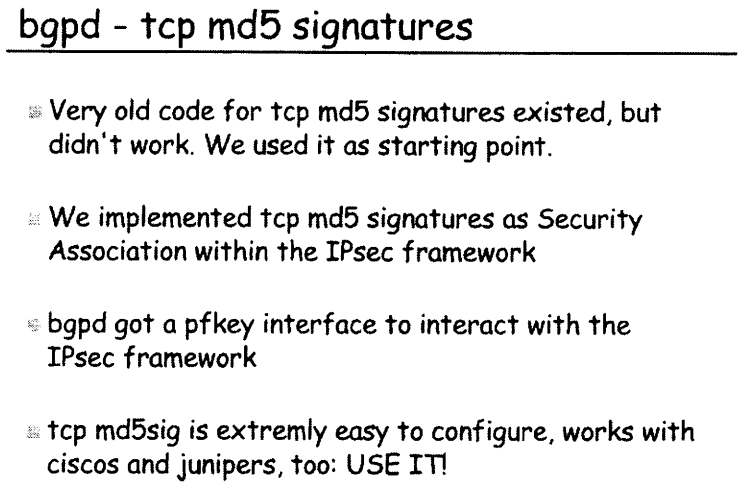
bgpd - tcp md5 signatures

- bgp sessions are not really authenticated - just IP based access control
- An attacker could send a bgp notification message with a faked source address, resetting the connection -> DoS



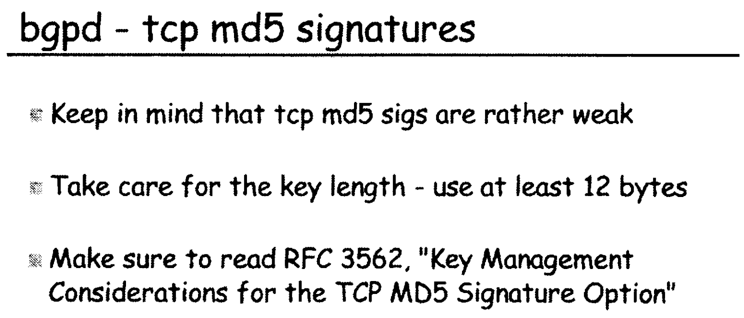
bgpd - tcp md5 signatures

- RFC 2385 defines tcp md5 signatures
- An md5 hash of parts of the header and a shared secret is added to the tcp header and verified on the receiving side
  - (unless you happen to run FreeBSD, they don't bother verifying the signatures)
- Attacker has to know the shared secret



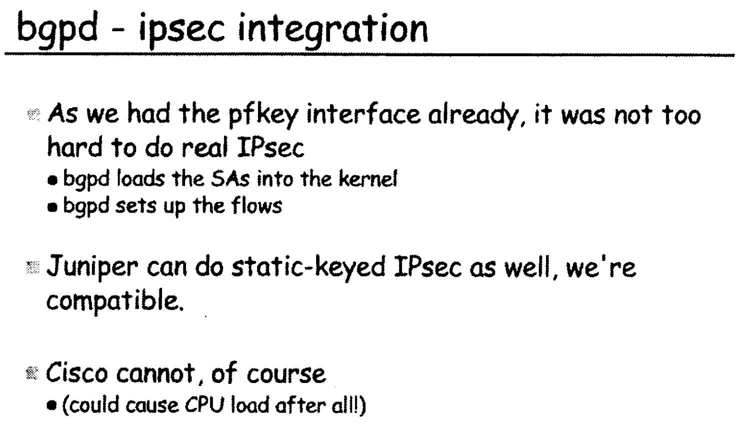
bgpd - tcp md5 signatures

- Very old code for tcp md5 signatures existed, but didn't work. We used it as starting point.
- We implemented tcp md5 signatures as Security Association within the IPsec framework
- bgpd got a pfkey interface to interact with the IPsec framework
- tcp md5sig is extremly easy to configure, works with ciscos and junipers, too: USE IT!



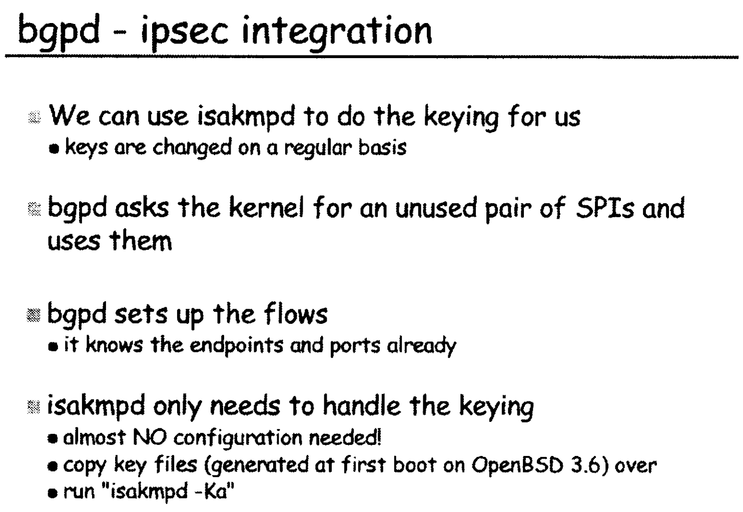
bgpd - tcp md5 signatures

- Keep in mind that tcp md5 sigs are rather weak
- Take care for the key length - use at least 12 bytes
- Make sure to read RFC 3562, "Key Management Considerations for the TCP MD5 Signature Option"



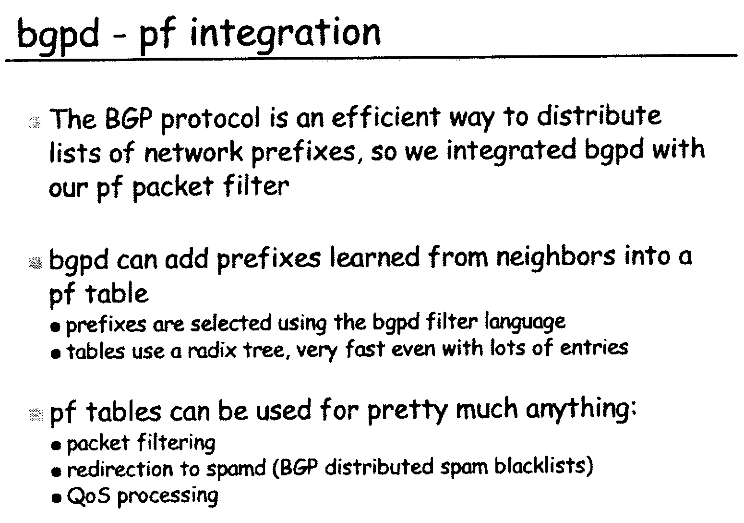
bgpd - ipsec integration

- As we had the pfkey interface already, it was not too hard to do real IPsec
  - bgpd loads the SAs into the kernel
  - bgpd sets up the flows
- Juniper can do static-keyed IPsec as well, we're compatible.
- Cisco cannot, of course
  - (could cause CPU load after all!)



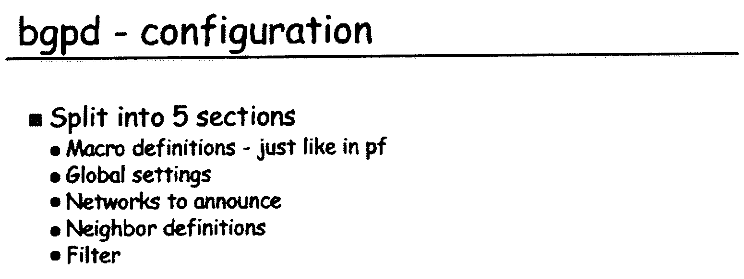
bgpd - ipsec integration

- We can use isakmpd to do the keying for us
  - keys are changed on a regular basis
- bgpd asks the kernel for an unused pair of SPIs and uses them
- bgpd sets up the flows
  - it knows the endpoints and ports already
- isakmpd only needs to handle the keying
  - almost NO configuration needed!
  - copy key files (generated at first boot on OpenBSD 3.6) over
  - run "isakmpd -Ka"



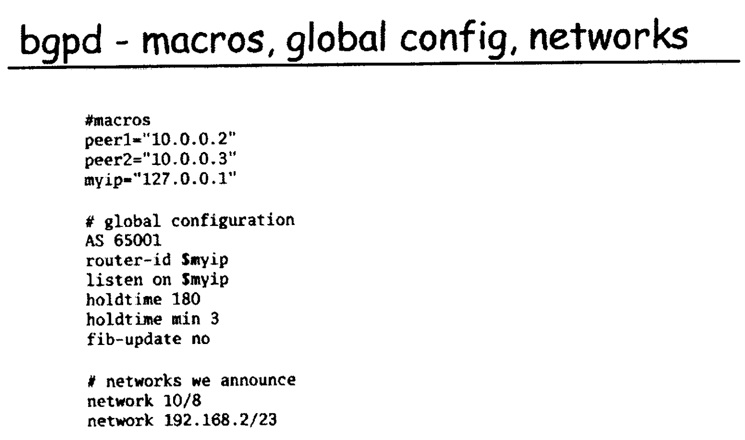
bgpd - pf integration

- The BGP protocol is an efficient way to distribute lists of network prefixes, so we integrated bgpd with our pf packet filter
- bgpd can add prefixes learned from neighbors into a pf table
  - prefixes are selected using the bgpd filter language
  - tables use a radix tree, very fast even with lots of entries
- pf tables can be used for pretty much anything:
  - packet filtering
  - redirection to spamd (BGP distributed spam blacklists)
  - QoS processing



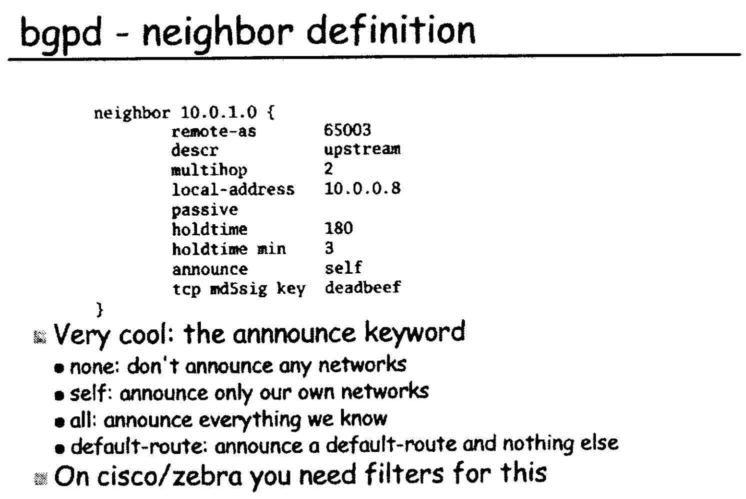
bgpd - configuration

- Split into 5 sections
  - Macro definitions - just like in pf
  - Global settings
  - Networks to announce
  - Neighbor definitions
  - Filter



bgpd - macros, global config, networks

```
#macros
peer1="10.0.0.2"
peer2="10.0.0.3"
myip="127.0.0.1"

# global configuration
AS 65001
router-id $myip
listen on $myip
holdtime 180
holdtime min 3
fib-update no

# networks we announce
network 10/8
network 192.168.2/23
```



bgpd - neighbor definition

```
neighbor 10.0.1.0 {
        remote-as       65003
        descr           upstream
        multihop        2
        local-address   10.0.0.8
        passive
        holdtime        180
        holdtime min    3
        announce        self
        tcp md5sig key  deadbeef
}
```

- Very cool: the annnounce keyword
  - none: don't announce any networks
  - self: announce only our own networks
  - all: announce everything we know
  - default-route: announce a default-route and nothing else
- On cisco/zebra you need filters for this

bgpd - neighbor groups

```
group "peering AS65002" {
        remote-as       65002
        passive
        holdtime        180
        holdtime min    3

        neighbor $peer1 {
                descr    "AS 65001 peer 1"
                announce self
                tcp md5sig password mekmitasdigoat
        }
        neighbor $peer2 {
                descr    "AS 65001 peer 2"
                announce all
        }
}
```

bgpd - ipsec configuration, static keying

```
neighbor 10.2.1.1 {
        remote-as 65023
        local-address 10.0.0.8
        ipsec esp in  spi 10 \
                sha1 0a4f1d1f1a1c4f3c9e2f6f0f2a8e9c8c5a1b0b3b \
                aes 0c1b3a6c7d7a8d2e0e7b4f3d5e8e6c1e
        ipsec esp out spi 12 \
                sha1 0e9c8f6a8e2c7d3a0b5d0d0f0a3c5c1d2b8e0f8b \
                aes 4e0f2f1b5c4e3c0d0e2f2d3b8c5c8f0b
}
```

bgpd - ipsec configuration, using IKE

```
neighbor 10.2.1.1 {
        remote-as 65023
        local-address 10.0.0.8
        ipsec esp ike
}

neighbor 10.2.1.2 {
        remote-as 65024
        local-address 10.0.0.8
        ipsec ah ike
}
```

filter language

```
# filter out prefixes longer than 24 or shorter than 8 bits
deny from any
allow from any prefixlen 8 - 24

# do not accept a default route
deny from any prefix 0.0.0.0/0

# filter bogus networks
deny from any prefix 10.0.0.0/8 prefixlen >= 8
deny from any prefix 172.16.0.0/12 prefixlen >= 12
deny from any prefix { 192.168.0.0/16 169.254.0.0/16 } \
        prefixlen >= 16
deny from any prefix 192.0.2.0/24 prefixlen >= 24
deny from any prefix { 224.0.0.0/4 240.0.0.0/4 } prefixlen >= 4
```

filter language

```
allow from $someuplink transit-as { $dfn }     set localpref 114
allow from $someuplink source-as  { $viag }    set localpref 112
allow from $someuplink transit-as { $telekom } set localpref 110

allow from group peerings set localpref 200

allow to $someuplink set community 13129:1911
allow to group uplinks set prepend-self 1
```

filter language

- Last match
- Rule consists of 3 parts:
  - Action: allow, deny or match
  - Match: based on prefix, prefixlen or AS path
  - Set: add prepends, modify localpref, metric etc

bgpctl

- Client connecting to bgpd via unix domain socket
  - query runtime information
  - reload configuration
  - (de-)couple kernel routing table
  - take specific sessions up/down

bgpctl

```
Neighbor          AS    MsgRcvd  MsgSent  OutQ  Up/Down  State/PrefixRcvd
192.168.133.46    64639    4333     4332     0 3d00h10m 1/100
192.168.133.47    64686   33618    33585     0 5d19h57m 12/100
192.168.133.85    64847    6768     6756     0 1d18h28m 3/100
192.168.133.86    64847    8693     8689     0 6d00h46m 3/100
192.168.133.49    64918    9096     9582     0 6d15h40m 80/200
192.168.133.28    64586    9113     9581     0 2d23h00m 1/100
192.168.133.65    64902   19158    19161     0 6d15h40m 2/100
192.168.133.48    64956   35310     9588     0 1d15h41m 45/100
192.168.133.95    64727    9585     9582     0 6d15h40m 5/100
192.168.133.22    65126    9589     9585     0 6d13h28m 1/100
192.168.133.17    64562    9405     9582     0 6d15h40m 142
192.168.133.173   64785  361006     9582     0 6d15h40m 143006
192.168.133.169   64562   77441     9582     0 6d15h40m 35987
```

bgpctl

```
<henning@cr10>  $ bgpctl show fib connected static
        flags: * = valid, B = BGP, C = Connected, S = Static
               N = BGP Nexthop reachable via this route

        flags destination         gateway
        *C    80.86.162.24/30     link#2
        *SN   80.86.164.16/32     80.86.162.25
        *S    80.86.181.0/24      80.86.183.4
        *S    80.86.182.0/23      80.86.183.4
        *C    80.86.183.0/28      link#5
        *C    80.86.183.16/28     link#7
        *S    80.86.183.30/32     127.0.0.1
        *S    81.209.180.0/22     80.86.183.4
        *S    81.209.196.0/22     80.86.183.4
        *C    127.0.0.1/8         link#0
        *S    127.0.0.1/32        127.0.0.1
        *S    192.168.214.0/24    80.86.183.17
        *SN   212.20.158.0/30     212.20.158.201
        *C    212.20.158.200/29   link#3
     [ ... ]
```

bgpctl

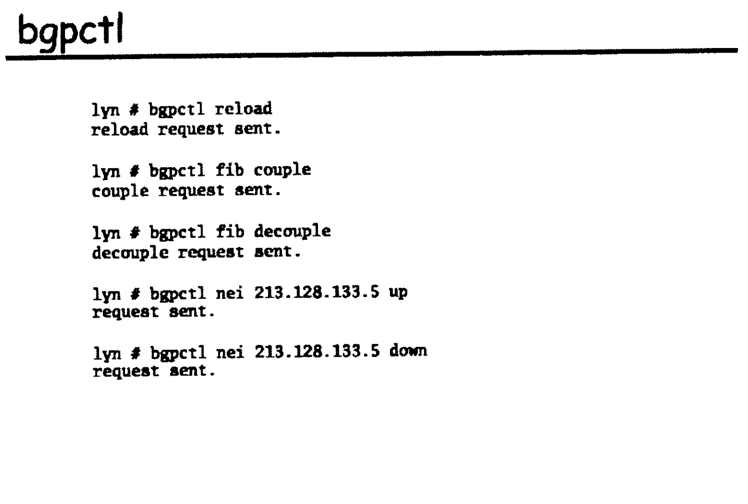
```
<henning@cr10>  $ bgpctl s nei 10.0.0.16
BGP neighbor is 10.0.0.16, remote AS 13237
 Description: lnc
  BGP version 4, remote router-id 10.0.0.16
  BGP state = Established, up for 02:53:52
  Last read 00:00:15, holdtime 90s, keepalive interval 30s
  Neighbor capabilities:
    Multiprotocol extensions: IPv4 Unicast
    Route Refresh

  Message statistics:
                  Sent       Received
  Opens              1          1
  Notifications      0          0
  Updates            1      78260
  Keepalives       348          1
  Route Refresh      0          0
  Total            350      78262

  Local host:           10.0.0.26, Local port:    179
  Remote host:          10.0.0.16, Remote port:  2667
```

bgpctl

```
<henning@cr10>  $ bgpctl s nei 10.0.0.16 timers
BGP neighbor is 10.0.0.16, remote AS 13237
 Description: lnc
  BGP version 4, remote router-id 10.0.0.16
  BGP state = Established, up for 02:57:16
  Last read 00:00:18, holdtime 90s, keepalive interval 30s
  Neighbor capabilities:
    Multiprotocol extensions: IPv4 Unicast
    Route Refresh

  IdleHoldTimer:       not running          Interval:    30s
  ConnectRetryTimer:   not running          Interval:   120s
  HoldTimer:           due in 00:01:12      Interval:    90s
  KeepaliveTimer:      due in 00:00:30      Interval:    30s

  Local host:          10.0.0.26, Local port:    179
  Remote host:         10.0.0.16, Remote port:  2667
```

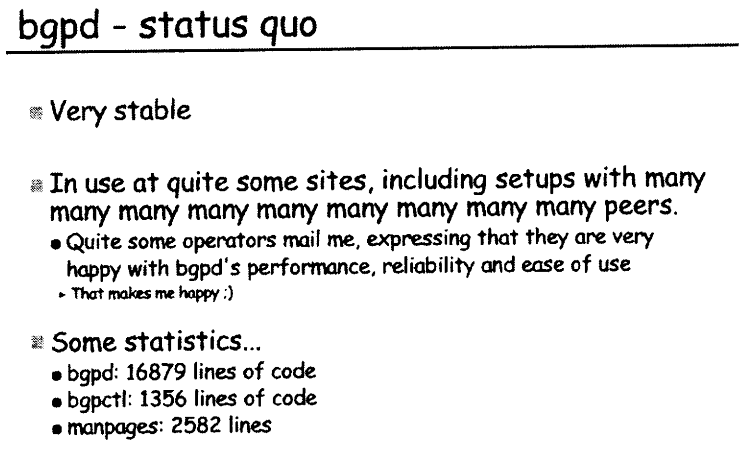
bgpctl

```
<henning@cr10>  $ bgpctl sh nex
Nexthop              State
80.86.164.16         valid
213.128.128.6        valid
212.20.158.2         valid
```



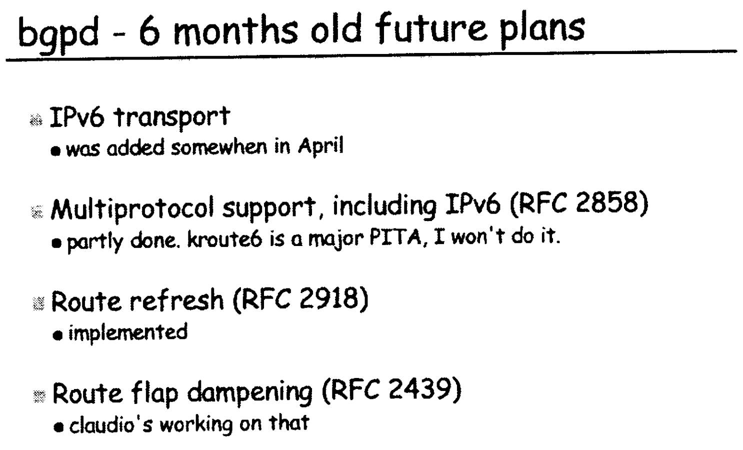
bgpctl

```
lyn # bgpctl reload
reload request sent.

lyn # bgpctl fib couple
couple request sent.

lyn # bgpctl fib decouple
decouple request sent.

lyn # bgpctl nei 213.128.133.5 up
request sent.

lyn # bgpctl nei 213.128.133.5 down
request sent.
```



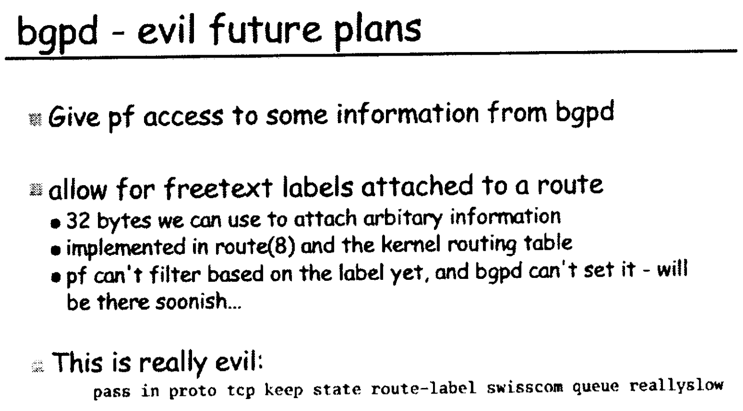
bgpd - status quo

- *Very stable*
- In use at *quite some sites*, including setups with many many many many many many many many many many peers.
  - Quite some operators mail me, expressing that they are very happy with bgpd's performance, reliability and ease of use
    - That makes me happy :)
- Some statistics...
  - bgpd: 16879 lines of code
  - bgpctl: 1356 lines of code
  - manpages: 2582 lines



bgpd - 6 months old future plans

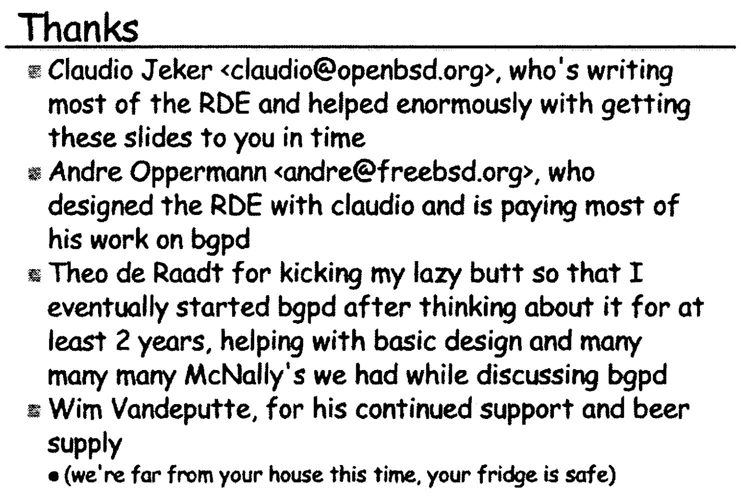
- IPv6 transport
  - was added somewhen in April
- Multiprotocol support, including IPv6 (RFC 2858)
  - partly done. kroute6 is a major PITA, I won't do it.
- Route refresh (RFC 2918)
  - implemented
- Route flap dampening (RFC 2439)
  - claudio's working on that



bgpd - *evil* future plans

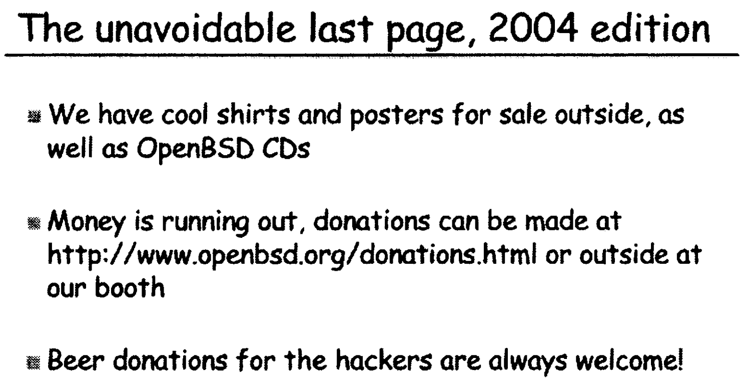
- *Give pf access to some information from bgpd*
- allow for freetext labels attached to a route
  - 32 bytes we can use to attach arbitary information
  - implemented in route(8) and the kernel routing table
  - pf can't filter based on the label yet, and bgpd can't set it - will be there soonish...
- This is *really* evil:

```
pass in proto tcp keep state route-label swisscom queue reallyslow
```



Thanks

- Claudio Jeker <claudio@openbsd.org>, who's writing most of the RDE and helped enormously with getting these slides to you in time
- Andre Oppermann <andre@freebsd.org>, who designed the RDE with claudio and is paying most of his work on bgpd
- Theo de Raadt for kicking my lazy butt so that I eventually started bgpd after thinking about it for at least 2 years, helping with basic design and many many many McNally's we had while discussing bgpd
- Wim Vandeputte, for his continued support and beer supply
  - (we're far from your house this time, your fridge is safe)



The unavoidable last page, 2004 edition

- We have cool shirts and posters for sale outside, as well as OpenBSD CDs
- Money is running out, donations can be made at http://www.openbsd.org/donations.html or outside at our booth
- Beer donations for the hackers are always welcome!

Track A Sunday

Notes:

EuroBSDCon 2004 Karlsruhe, Germany:

An Introduction to Sysadmin Training in the Virtual Unix Lab

Hubert Feyrer <hubert@feyrer.de>

October 31st, 2004

Abstract

The Virtual Unix Lab (vulab) is an interactive course system which allows students to do Unix system administration exercises. Machines are installed on which students can do their assignments with full "root"-access. At the end, the system checks which parts were done correctly, and gives a feedback on the exercise result. Access to the lab is via the Internet via a web-browser as well as standard Unix clients (ssh, telnet, ftp). Some detail on exercise-verification are outlined in this paper.

Contents

1 Introduction & Background 2

2 The Virtual Unix Lab 2

3 A Tour trough the Virtual Unix Lab 2
3.1 User Area 2
3.2 Admin Area 11
3.3 Creating New Exercises 18

4 Setup 28
4.1 Hardware 28
4.2 Lab Machine Installation 30
4.3 Restricting Access to Lab Machines 31
4.4 Software 31

5 Current Status 32

6 Future Perspectives 32

References 32

1 Introduction & Background

A problem in teaching Unix system administration is the lack of machines available on which students can practice with full system administrator privileges. Without system administrator (root) privileges, many things cannot be practiced. On the other handside, when handing out root privileges, the lab machines are in an unknown state, requiring reinstallation of the lab machines for future students to get a known safe & well-configured environment.

The Virtual Unix Lab was created to solve this problem.

2 The Virtual Unix Lab

The Virtual Unix Laboratory was started in the "Praktikum Unix Cluster Setup" project in the "Hochschul- und Wissenschaftsprogramm (HWP)" initiative at the University of Applied Sciences of Regensburg (FH Regensburg). It was designed as an interactive course system for system administration training in general, and with a focus on training installation and configuration of the Network File System (NFS) and Network Information System (NIS) on Unix-based systems in particular.

The Virtual Unix Lab fulfills this purpose today. After sign-up, machines are installed on demand, and students can do their assignments with full "root"-access. Users can book exercises for a certain time, and all machines are setup identically. Exclusive access to the lab machines during exercises is guaranteed, with access to the lab being realized via the Internet via a web-browser as well as standard Unix clients (ssh, telnet, ftp). At the end of the assignment, the system checks if/which parts were done correctly, and gives a report containing feedback on the success of the exercise to the student.

After that, machines are re-installed from scratch for next user and exercises.

3 A Tour trough the Virtual Unix Lab

The tour through the Virtual Unix Lab covers both the parts that the users will see while using the system as well as a few areas which cover administrative actions plus an overview of the steps for updating and creating a new exercise.

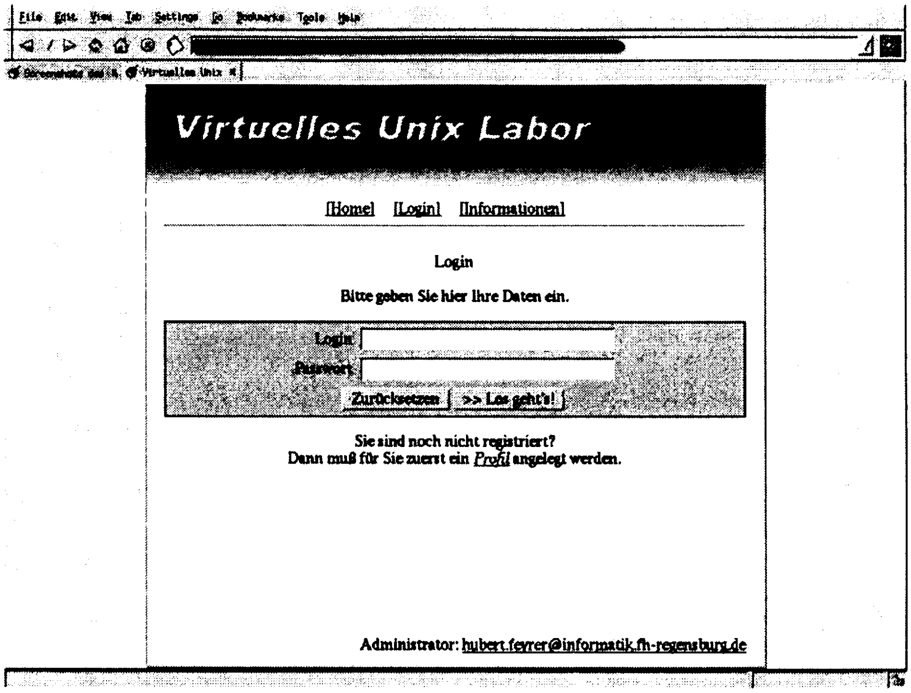
3.1 User Area

This tour through the user area of the Virtual Unix Lab covers the following areas:

- Login and account creation
- List of exercises
- Booking an exercise
- Taking an exercise
- Retrieving feedback afterwards

The tour itself will consist of a number of screenshots displaying the various parts of the web based user interface that the Virtual Unix Lab presents.

1. Access to the user interface of the Virtual Unix Lab is through a web browser, which allows accessing all facilities provided, except performing exercises themselves (see


File Edit View Tab Settings Go Bookmarks Tools Help
[redacted]
Virtuelles Unix
[Home] [Login] [Informationen]
Login
Passwort
Zurücksetzen
>> Los geht's!

Virtuelles Unix Labor

Login

Bitte geben Sie hier Ihre Daten ein.

Sie sind noch nicht registriert?
Dann muß für Sie zuerst ein *Profil* angelegt werden.

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 1: Logging into the Virtual Unix Lab

below). Language of the user interface is German (only) right now – internationalisation is on the list of items to do in the future.

When accessing the webpage, the first thing students encounter is a mask to login as displayed in figure 1.

2. If a student doesn't have a login yet, he can create a new login ("Profile") using the form displayed in figure 2.

   The student will have to give his student ID number ("Matrikel-Nummer"), first and last name, an email address where he can be reached and a password (twice). Upon registering, an email will be sent to the given email address, which contains an authentication token that the user has to enter to permanently enable his account. Accounts not enabled that way will be deleted after 7 days. This allows instant access to the lab, but ensures that people provide at least a valid email address if they want to keep using the lab.

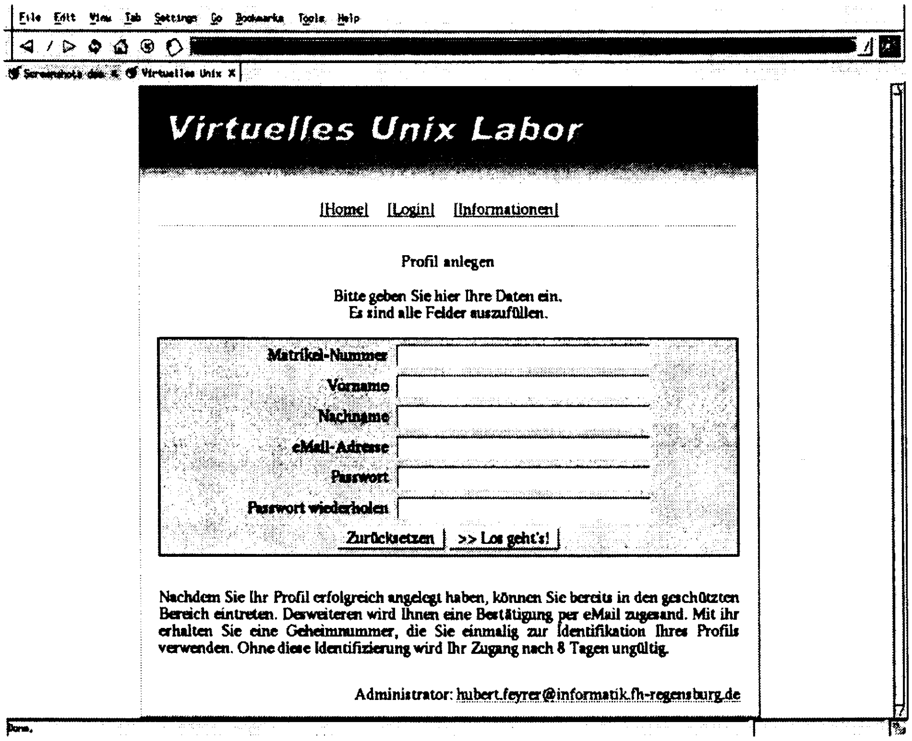
3. After successful login into the Virtual Unix Lab, the welcome screen shown in figure 3 is displayed and users can choose from several actions they want to do: Update their user settings ("Benutzerdaten"), get a list of available exercises ("Übungen auflisten"), book an exercise for a certain time & date ("Buchung vornehmen"), get a list of past and future exercises, delete future exercises and retrieve feedback on past ones ("Buchungen einsehen") as well as logout of the web site:

4. Figure 4 shows a list of exercises available in the Virtual Unix Lab, including the exercise name ("Übung"), a one-line description of the exercise ("Bezeichnung") and duration of the exercise for the user ("Dauer"):

5. Each of the exercises in the list can be clicked on to retrieve the exercise text as shown when actually taking the exercises, see figure 5 for an example.

   This allows preparing the exercises, and learning all the items necessary to successfully perform the exercise in the Virtual Unix Lab.

6. After these preparations, an exercise can be booked by selecting the "übung buchen"



Virtuelles Unix Labor

[Home] [Login] [Informationen]

Profil anlegen

Bitte geben Sie hier Ihre Daten ein.
Es sind alle Felder auszufüllen.

Matrikel-Nummer
Vorname
Nachname
eMail-Adresse
Passwort
Passwort wiederholen

Nachdem Sie Ihr Profil erfolgreich angelegt haben, können Sie bereits in den geschützten Bereich eintreten. Desweiteren wird Ihnen eine Bestätigung per eMail zugesand. Mit ihr erhalten Sie eine Geheimnummer, die Sie einmalig zur Identifikation Ihres Profils verwenden. Ohne diese Identifizierung wird Ihr Zugang nach 8 Tagen ungültig.

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 2: Entering data for a new login

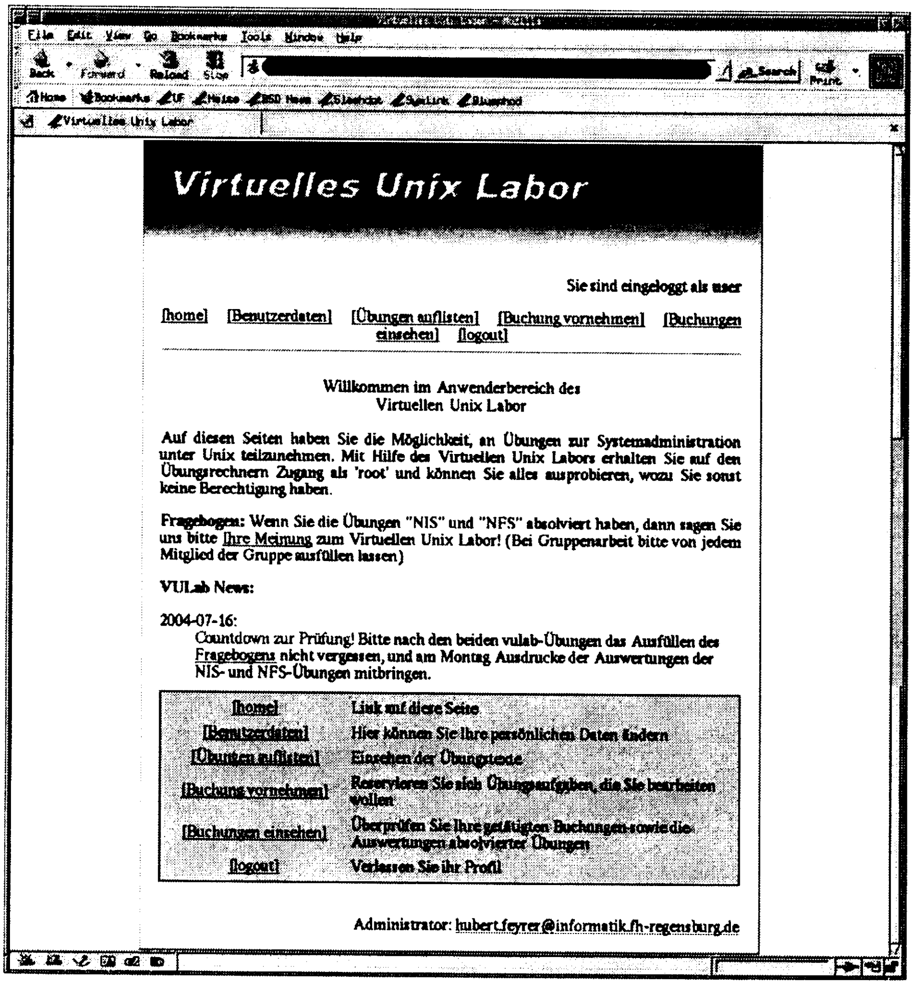
menu item. The first step in booking an exercise consists of deciding at which date and time to take the exercise, which is displayed in figure 6.

Exercises are available in three-hour intervals (1.5 hours for the exercise, plus about one hour for preparation of the lab machines and some time for postprocessing), slots already booked by other users are not displayed as available. In the above screenshot, the exercises at 0am, 3am, 6am, 9am and 12am are not available because of that.

7. After deciding on the date and time for the exercise to take place, the next step is to choose which actual exercise to take, from the list of available exercises. As in the previous list of available exercises, the exercise name, description and duration are displayed, and the user has to decide for one as displayed in figure 7.
8. After selecting date & time and which course to take, a final confirmation shown in figure 8 has to be made before the exercise is booked.
9. The exercise is booked now, and the system will know when to prepare the lab machines for the exercises, using an at(1) job.

   The student can walk away, prepare for the exercise or whatever, and as for a real test, he should come back to the lab a few minutes before the selected time of the exercises, logging in again, see figure 9.
10. After the user has logged in, the system will tell him that an exercise was prepared for him (and which one), and that he can already start preparing the exercise by following the provided link ("bitte hier klicken" in the red text) as displayed in figure 10.
11. Before starting the exercise, the student has to enter the IP address of the machine from which he wants to access the lab machines. This process, shown in figure 11, is needed to restrict access so other students cannot disturb the exercise.

    What happens when the user has entered his IP number here is that appropriate rules will be injected into the firewall covering the lab machines to allow access to the lab machines only from the given hostname when the exercise starts.



Virtuelles Unix Labor

Sie sind eingeloggt als user

[home] [Benutzerdaten] [Übungen auflisten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Willkommen im Anwenderbereich des
Virtuellen Unix Labor

Auf diesen Seiten haben Sie die Möglichkeit, an Übungen zur Systemadministration unter Unix teilzunehmen. Mit Hilfe des Virtuellen Unix Labors erhalten Sie auf den Übungsrechnern Zugang als 'root' und können Sie alles ausprobieren, wozu Sie sonst keine Berechtigung haben.

Fragebogen: Wenn Sie die Übungen "NIS" und "NFS" absolviert haben, dann sagen Sie uns bitte Ihre Meinung zum Virtuellen Unix Labor! (Bei Gruppenarbeit bitte von jedem Mitglied der Gruppe ausfüllen lassen)

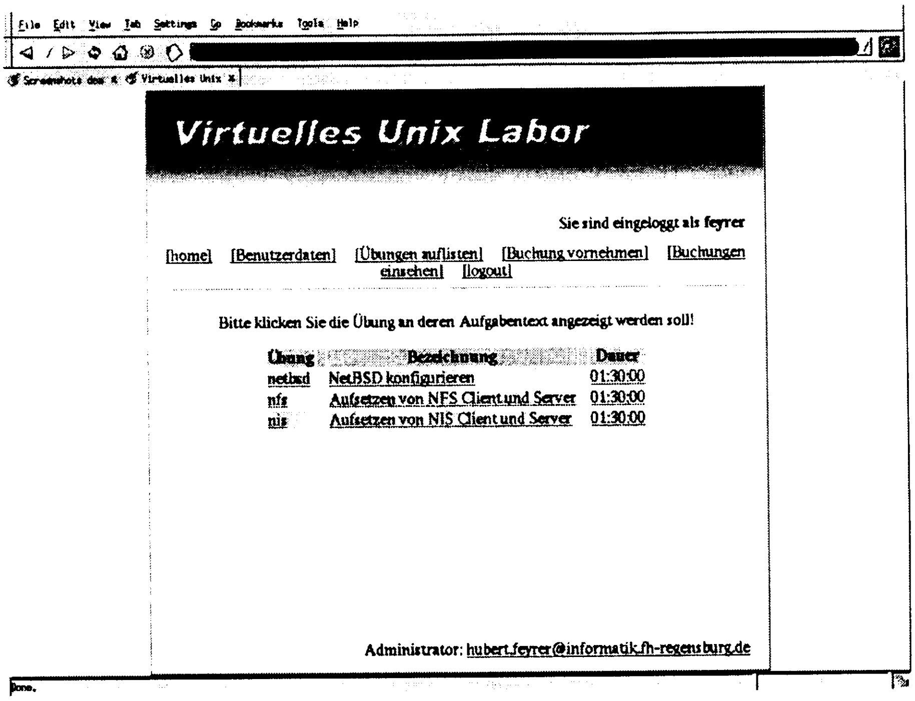
VULab News:

2004-07-16:
Countdown zur Prüfung! Bitte nach den beiden vulab-Übungen das Ausfüllen des Fragebogens nicht vergessen, und am Montag Ausdrucke der Auswertungen der NIS- und NFS-Übungen mitbringen.

[home]	Link auf diese Seite
[Benutzerdaten]	Hier können Sie Ihre persönlichen Daten ändern
[Übungen auflisten]	Einsehen der Übungstexte
[Buchung vornehmen]	Reservieren Sie sich Übungsaufgaben, die Sie bearbeiten wollen
[Buchungen einsehen]	Überprüfen Sie Ihre getätigten Buchungen sowie die Auswertungen absolvierter Übungen
[logout]	Verlassen Sie ihr Profil

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 3: Welcome to the Virtual Unix Lab



Virtuelles Unix Labor

Sie sind eingeloggt als feyrer

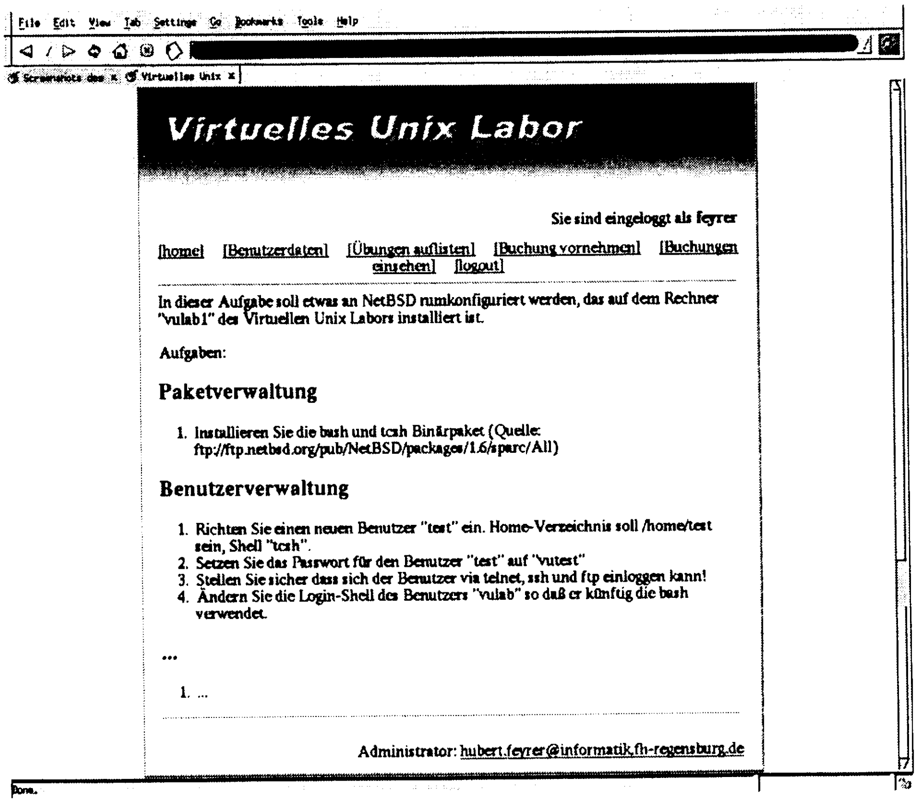
[home] [Benutzerdaten] [Übungen auflisten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Bitte klicken Sie die Übung an deren Aufgabentext angezeigt werden soll!

Übung	Bezeichnung	Dauer
netbsd	NetBSD konfigurieren	01:30:00
nfs	Aufsetzen von NFS Client und Server	01:30:00
nis	Aufsetzen von NIS Client und Server	01:30:00

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 4: List of available exercises



Virtuelles Unix Labor

Sie sind eingeloggt als feyrer

[home] [Benutzerdaten] [Übungen auflisten] [Buchung vornehmen] [Buchungen einsehen] [logout]

In dieser Aufgabe soll etwas an NetBSD rumkonfiguriert werden, das auf dem Rechner "vulab1" des Virtuellen Unix Labors installiert ist.

Aufgaben:

Paketverwaltung

1. Installieren Sie die bash und tcsh Binärpaket (Quelle: ftp://ftp.netbsd.org/pub/NetBSD/packages/1.6/sparc/All)

Benutzerverwaltung

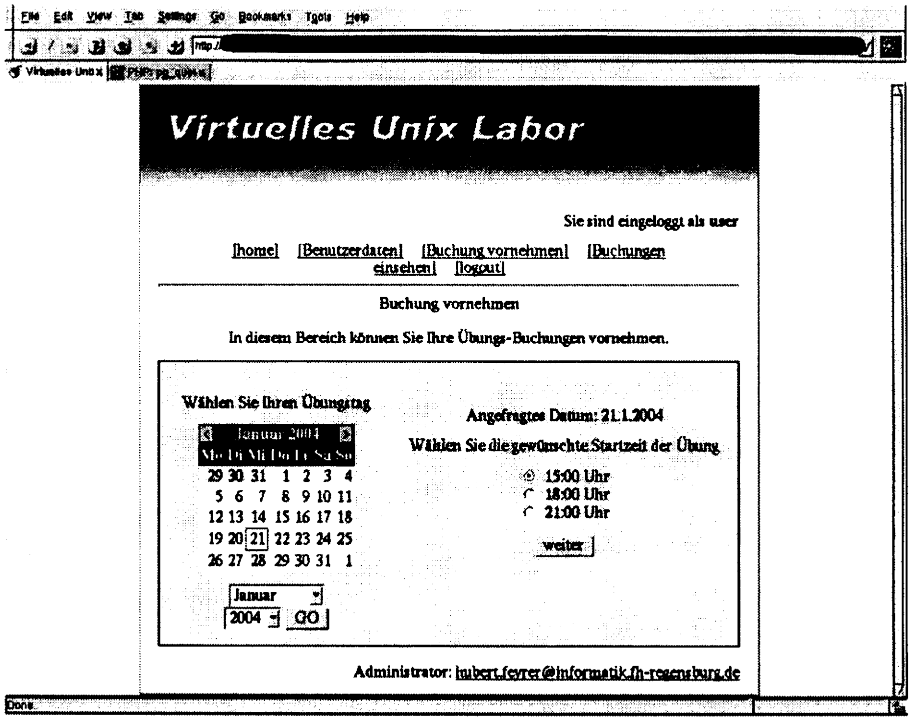
1. Richten Sie einen neuen Benutzer "test" ein. Home-Verzeichnis soll /home/test sein, Shell "tcsh".
2. Setzen Sie das Passwort für den Benutzer "test" auf "vutest"
3. Stellen Sie sicher dass sich der Benutzer via telnet, ssh und ftp einloggen kann!
4. Ändern Sie die Login-Shell des Benutzers "vulab" so daß er künftig die bash verwendet.

...

1. ...

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 5: Exercise text preview



Virtuelles Unix Labor

Sie sind eingeloggt als user

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Buchung vornehmen

In diesem Bereich können Sie Ihre Übungs-Buchungen vornehmen.

Wählen Sie Ihren Übungstag

Januar 2004

Mo	Di	Mi	Do	Fr	Sa	So
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1

Januar
2004 GO

Angefragtes Datum: 21.1.2004

Wählen Sie die gewünschte Startzeit der Übung

● 15:00 Uhr
○ 18:00 Uhr
○ 21:00 Uhr

weiter

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 6: Booking an exercise: selecting date & time

12. Just as in a real test, the student can come into the lab and sit down, but the test won't start until a certain time. In a real lab test, this would be when the teacher passes out sheets of paper with the exercises printed on them. In the Virtual Unix Lab, the student has to wait for the start of the exercise too, as displayed in figure 12.

13. When exercise time is reached, the firewall protecting the lab systems will be opened to allow access to the lab systems, and the exercise will be displayed as e.g. in figure 13.

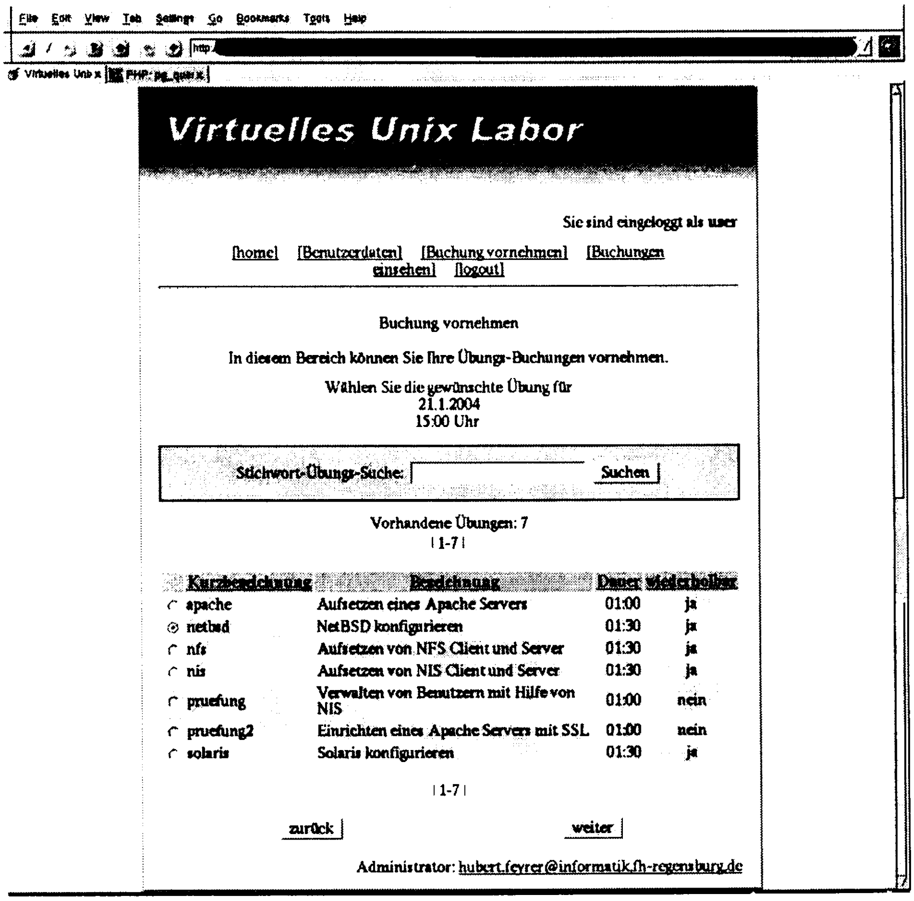
    The text displayed here is the same as was available for looking at before, so students were able to prepare properly, with a few small additions. First, a link with help for accessing the lab systems is placed under the exercise text, so students not familiar with the lab (yet) can make themselves familiar how to access the lab machines, giving proper syntax for telnet, ftp and ssh. Below this link, the time remaining for the exercise is printed on the lower left ("Verbleibende Zeit"), and if the user decides she has finished the exercise before the time runs out, this can be stated by pressing the "Fertig!"-button.

14. For accessing the lab machines, separate terminal windows have to be opened to access the lab machines and perform the tasks needed to successfully solve the tasks given in the exercise text as shown in figure 14.

    Figure 14 shows access to a Solaris/sparc (left xterm) and NetBSD/sparc (right xterm) system. For each system, the user knows a "normal" user account and password (without any system privileges) as well as the system administrator (root) password, given in the instructions on how to access the lab machines.

    The student can the use any measures he seems appropriate to solve the tasks, using the full administrative privileges he has available. If one of the lab machines has to be rebooted, this can be done as with any normal remotely administrated machine (i.e.: there is no access to the console, right now).

15. After the exercise has ended – either because time ran out, or because the student



Virtuelles Unix Labor

Sie sind eingeloggt als user

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Buchung vornehmen

In diesem Bereich können Sie Ihre Übungs-Buchungen vornehmen.

Wählen Sie die gewünschte Übung für
21.1.2004
15:00 Uhr

Stichwort-Übungs-Suche: Suchen

Vorhandene Übungen: 7
| 1-7 |

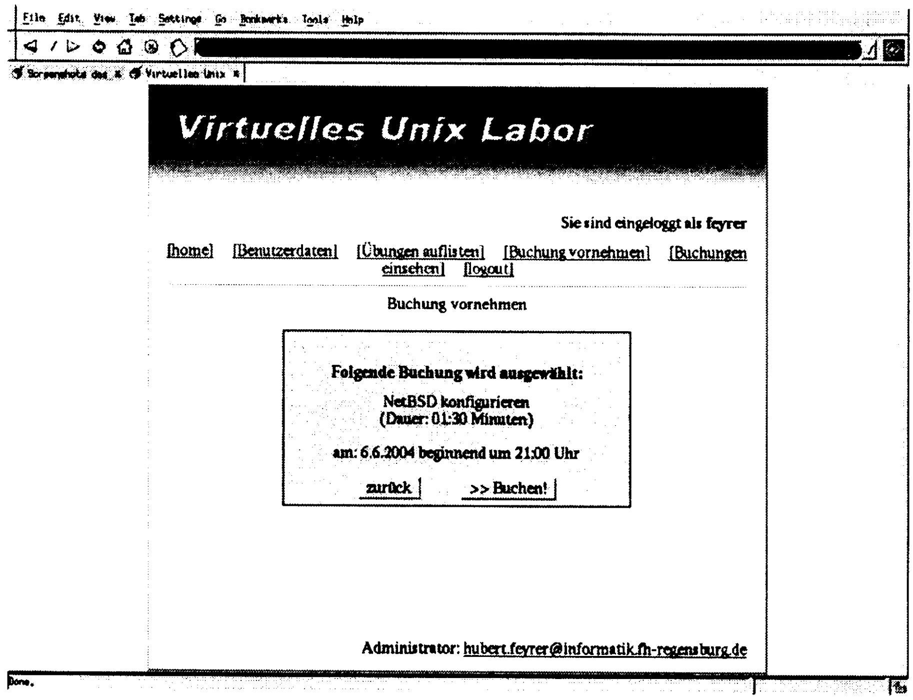
Kurzbezeichnung	Bezeichnung	Dauer	wiederholbar
○ apache	Aufsetzen eines Apache Servers	01:00	ja
◉ netbsd	NetBSD konfigurieren	01:30	ja
○ nfs	Aufsetzen von NFS Client und Server	01:30	ja
○ nis	Aufsetzen von NIS Client und Server	01:30	ja
○ pruefung	Verwalten von Benutzern mit Hilfe von NIS	01:00	nein
○ pruefung2	Einrichten eines Apache Servers mit SSL	01:00	nein
○ solaris	Solaris konfigurieren	01:30	ja

| 1-7 |

zurück weiter

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 7: Booking an exercise: selecting the exercise



[redacted]

Virtuelles Unix Labor

Sie sind eingeloggt als feyrer

[home] [Benutzerdaten] [Übungen auflisten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Buchung vornehmen

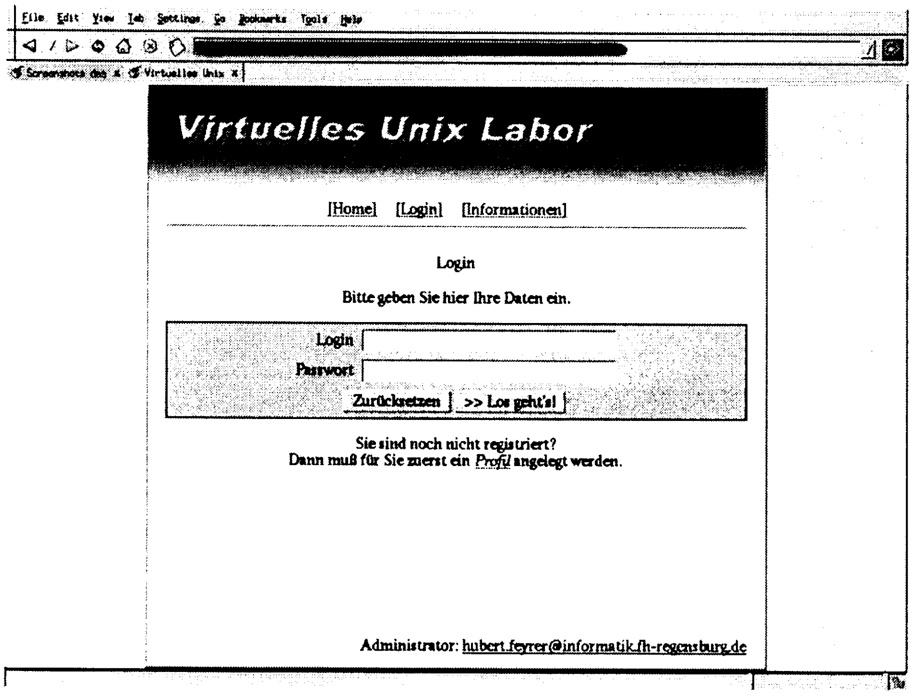
Folgende Buchung wird ausgewählt:

NetBSD konfigurieren
(Dauer: 01:30 Minuten)

am: 6.6.2004 beginnend um 21:00 Uhr

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 8: Booking an exercise: confirmation



[redacted]

Virtuelles Unix Labor

[Home] [Login] [Informationen]

Login

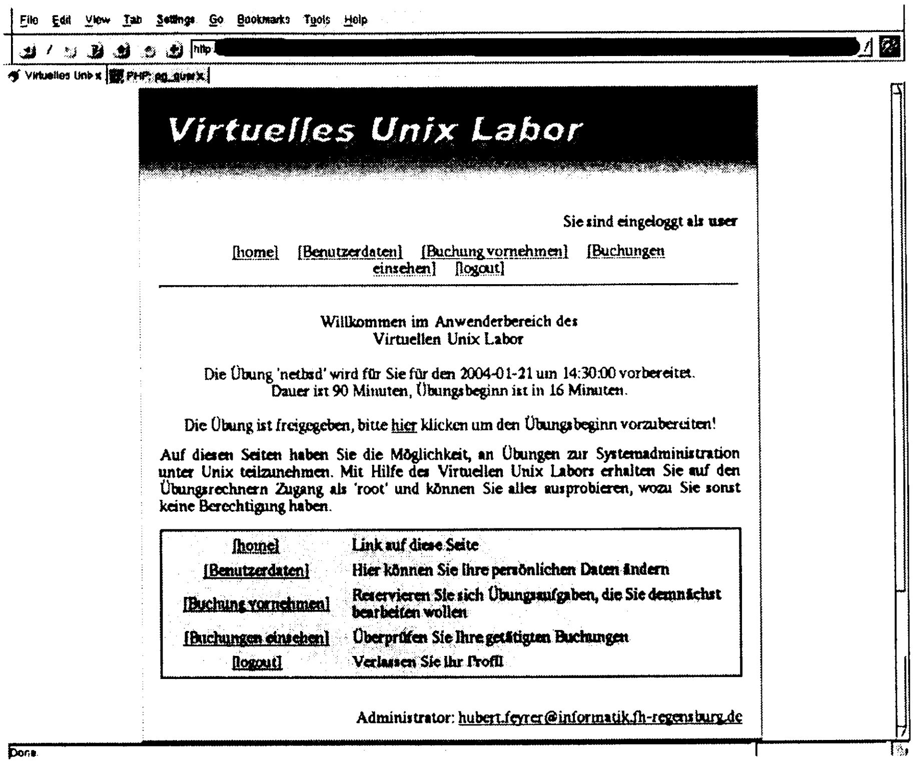
Bitte geben Sie hier Ihre Daten ein.

Login
Passwort

Sie sind noch nicht registriert?
Dann muß für Sie zuerst ein *Profil* angelegt werden.

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 9: Logging in for a booked exercise



Sie sind eingeloggt als **user**

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Willkommen im Anwenderbereich des
Virtuellen Unix Labor

Die Übung 'netbsd' wird für Sie für den 2004-01-21 um 14:30:00 vorbereitet.
Dauer ist 90 Minuten, Übungsbeginn ist in 16 Minuten.

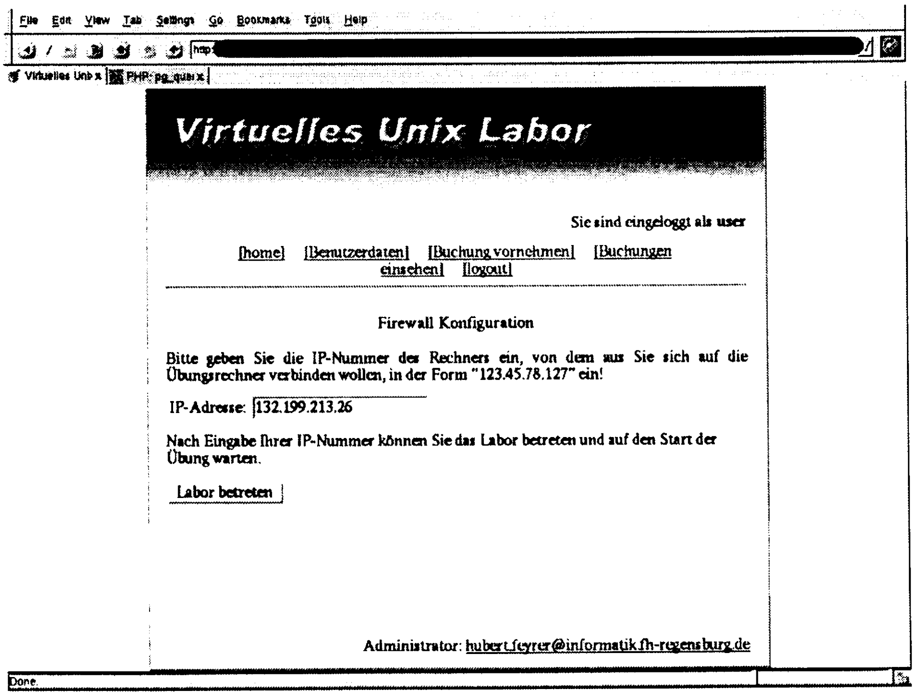
Die Übung ist freigegeben, bitte hier klicken um den Übungsbeginn vorzubereiten!

Auf diesen Seiten haben Sie die Möglichkeit, an Übungen zur Systemadministration unter Unix teilzunehmen. Mit Hilfe des Virtuellen Unix Labors erhalten Sie auf den Übungsrechnern Zugang als 'root' und können Sie alles ausprobieren, wozu Sie sonst keine Berechtigung haben.

[home]	**Link auf diese Seite**
[Benutzerdaten]	**Hier können Sie Ihre persönlichen Daten ändern**
[Buchung vornehmen]	**Reservieren Sie sich Übungsaufgaben, die Sie demnächst bearbeiten wollen**
[Buchungen einsehen]	**Überprüfen Sie Ihre getätigten Buchungen**
[logout]	**Verlassen Sie Ihr Profil**

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 10: An exercise is prepared & waiting



Sie sind eingeloggt als **user**

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Firewall Konfiguration

Bitte geben Sie die IP-Nummer des Rechners ein, von dem aus Sie sich auf die Übungsrechner verbinden wollen, in der Form "123.45.78.127" ein!

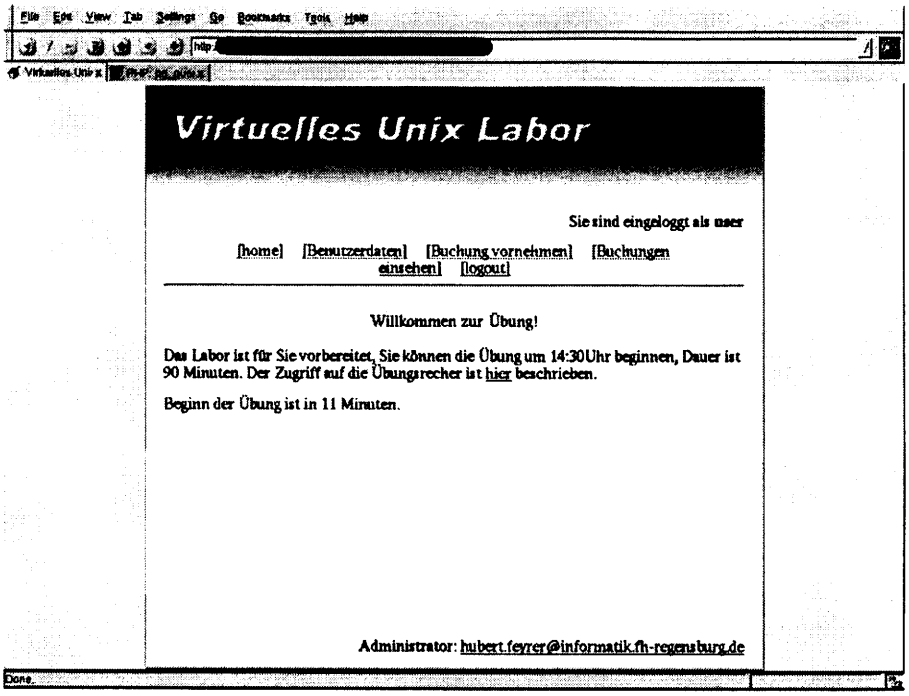
IP-Adresse: 132.199.213.26

Nach Eingabe Ihrer IP-Nummer können Sie das Labor betreten und auf den Start der Übung warten.

Labor betreten

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 11: Configuring access to the lab machines



Virtuelles Unix Labor

Sie sind eingeloggt als user

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Willkommen zur Übung!

Das Labor ist für Sie vorbereitet, Sie können die Übung um 14:30Uhr beginnen, Dauer ist 90 Minuten. Der Zugriff auf die Übungsrecher ist hier beschrieben.

Beginn der Übung ist in 11 Minuten.

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 12: Waiting for start of exercise time

signalled he's done by pushing the "Fertig!"-button – the system will revoke access to the lab systems by re-enabling the firewall, and print a message that the exercise is over and that feedback on the exercise can be retrieved from the database within a few minutes as shown in figure 15.

At this point, the lab systems are analyzed in the background by a number of scripts, which know what configuration steps are necessary for successful performance of the exercise, and which will report their findings in the database for later retrieval (see below for more on this).

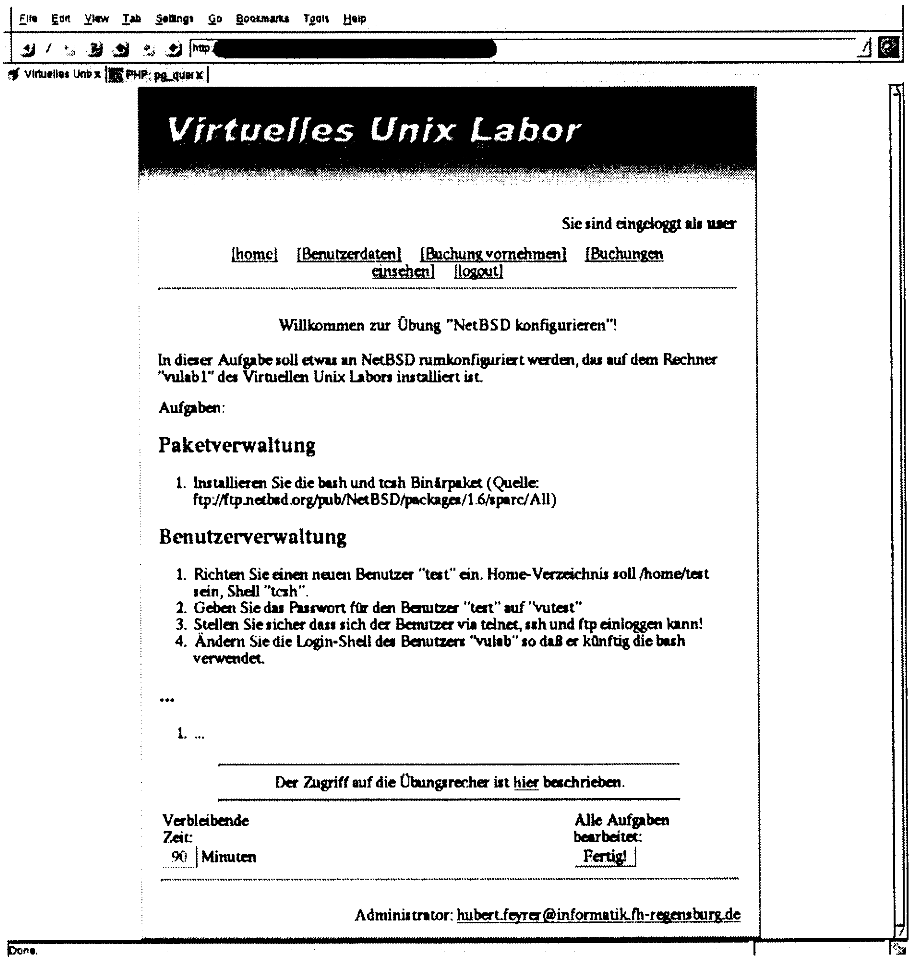
16. After an exercise has finished, students can retrieve feedback on an individual exercise via the main menu ("Buchungen einsehen"). They will see the exercise text again, with the various tasks containing comments on what checks were done (green text) , and if the particular check was done successfully ("OK") or not ("Nein"). See figure 16 for an example.

3.2 Admin Area

After a walkthrough of the functions provided by the Virtual Unix Lab to it's users and students, this section concentrates on some of the administrative actions that are available to administrators of the Virtual Unix Lab.

Again, here are the various impressions to show these aspects, in screenshot format.

1. Users with administrative privileges in the Virtual Unix Lab have to log into the system like "normal" users too, the system will know they have admin status, and display available actions as appropriate. Available actions are updating & changing user settings ("Benutzerdaten"), editing or creating new exercises ("Uebungs-Setup"), looking at past and future exercises booked by all users ("Buchungen") and seeing what users said in the survey ("Feedback", not covered here). Figure 17 shows the admin



Virtuelles Unix Labor

Sie sind eingeloggt als user

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Willkommen zur Übung "NetBSD konfigurieren"!

In dieser Aufgabe soll etwas an NetBSD rumkonfiguriert werden, das auf dem Rechner "vulab1" des Virtuellen Unix Labors installiert ist.

Aufgaben:

Paketverwaltung

1. Installieren Sie die bash und tcsh Binärpaket (Quelle: ftp://ftp.netbsd.org/pub/NetBSD/packages/1.6/sparc/All)

Benutzerverwaltung

1. Richten Sie einen neuen Benutzer "test" ein. Home-Verzeichnis soll /home/test sein, Shell "tcsh".
2. Geben Sie das Passwort für den Benutzer "test" auf "vutest"
3. Stellen Sie sicher dass sich der Benutzer via telnet, ssh und ftp einloggen kann!
4. Ändern Sie die Login-Shell des Benutzers "vulab" so daß er künftig die bash verwendet.

...

1. ...

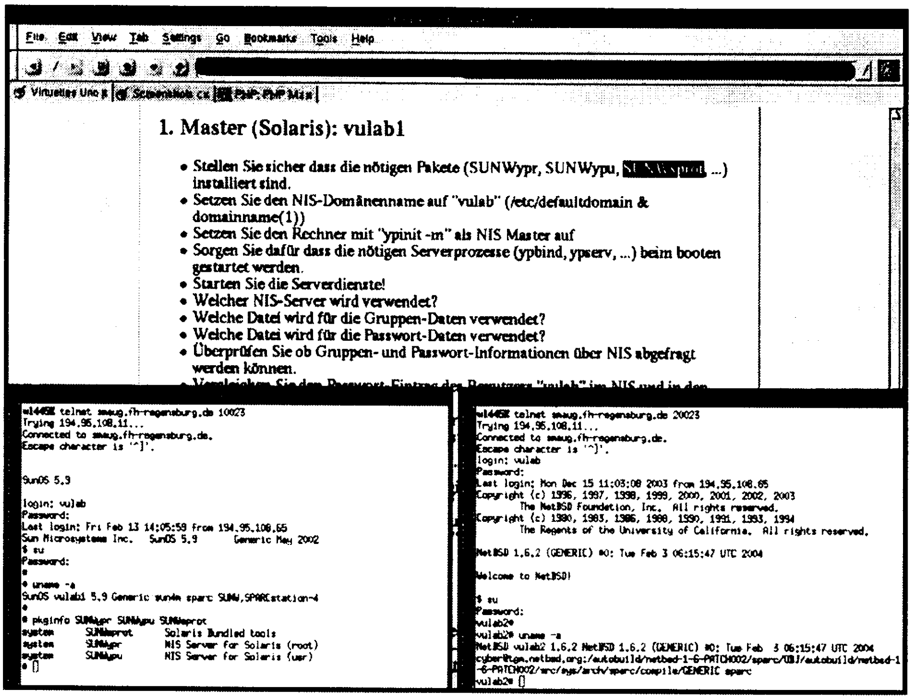
Der Zugriff auf die Übungsrecher ist hier beschrieben.

Verbleibende Zeit: 90 Minuten

Alle Aufgaben bearbeitet: Fertig!

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 13: Display of the exercise text



[redacted]

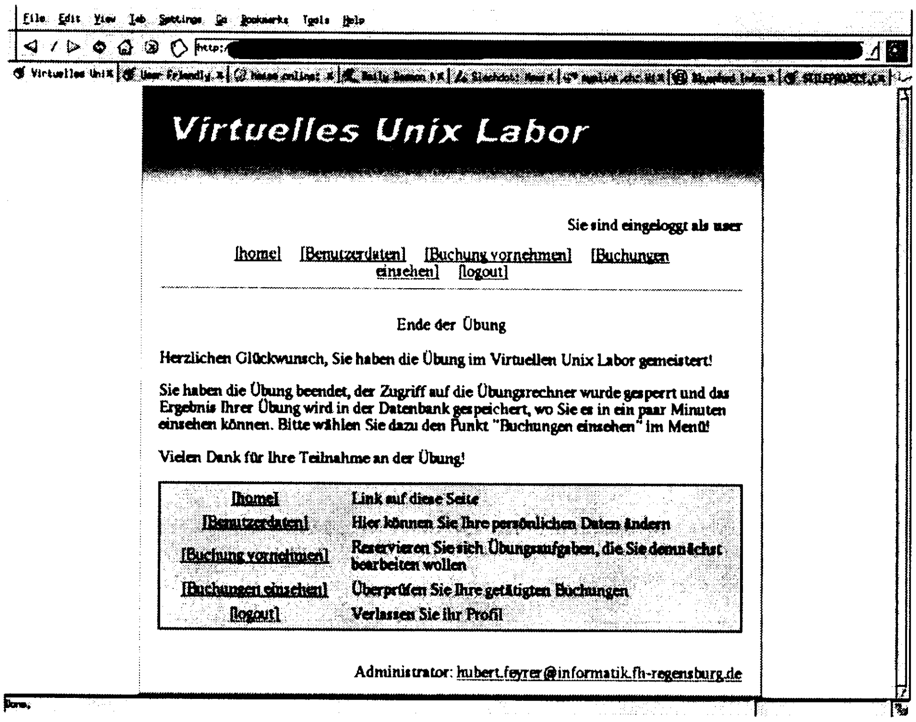
1. Master (Solaris): vulab1

- Stellen Sie sicher dass die nötigen Pakete (SUNWypr, SUNWypu, SUNWsprot, ...) installiert sind.
- Setzen Sie den NIS-Domänenname auf "vulab" (/etc/defaultdomain & domainname(1))
- Setzen Sie den Rechner mit "ypinit -m" als NIS Master auf
- Sorgen Sie dafür dass die nötigen Serverprozesse (ypbind, ypserv, ...) beim booten gestartet werden.
- Starten Sie die Serverdienste!
- Welcher NIS-Server wird verwendet?
- Welche Datei wird für die Gruppen-Daten verwendet?
- Welche Datei wird für die Passwort-Daten verwendet?
- Überprüfen Sie ob Gruppen- und Passwort-Informationen über NIS abgefragt werden können.
- Vergleichen Sie den Passwort-Eintrag des Benutzers "vulab" im NIS und in den

```
wl446% telnet smaug.fh-regensburg.de 10023
Trying 194.95.108.11...
Connected to smaug.fh-regensburg.de.
Escape character is '^]'.


SunOS 5.9

login: vulab
Password:
Last login: Fri Feb 13 14:05:59 from 194.95.108.65
Sun Microsystems Inc.   SunOS 5.9       Generic May 2002
$ su
Password:
#
# uname -a
SunOS vulab1 5.9 Generic sun4m sparc SUNW,SPARCstation-4
#
# pkginfo SUNWypr SUNWypu SUNWsprot
system      SUNWsprot      Solaris Bundled tools
system      SUNWypr        NIS Server for Solaris (root)
system      SUNWypu        NIS Server for Solaris (usr)
# []
```

```
wl446% telnet smaug.fh-regensburg.de 20023
Trying 194.95.108.11...
Connected to smaug.fh-regensburg.de.
Escape character is '^]'.
login: vulab
Password:
Last login: Mon Dec 15 11:03:08 2003 from 194.95.108.65
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003
        The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1980, 1983, 1986, 1988, 1990, 1991, 1993, 1994
        The Regents of the University of California.  All rights reserved.

NetBSD 1.6.2 (GENERIC) #0: Tue Feb 3 06:15:47 UTC 2004

Welcome to NetBSD!

$ su
Password:
vulab2#
vulab2# uname -a
NetBSD vulab2 1.6.2 NetBSD 1.6.2 (GENERIC) #0: Tue Feb  3 06:15:47 UTC 2004
cyber@tgm.netbsd.org:/autobuild/netbsd-1-6-PATCH002/sparc/OBJ/autobuild/netbsd-1
-6-PATCH002/src/sys/arch/sparc/compile/GENERIC sparc
vulab2# []
```

Figure 14: Logging into lab machines for the exercise



[redacted]

Virtuelles Unix Labor

Sie sind eingeloggt als user

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Ende der Übung

Herzlichen Glückwunsch, Sie haben die Übung im Virtuellen Unix Labor gemeistert!

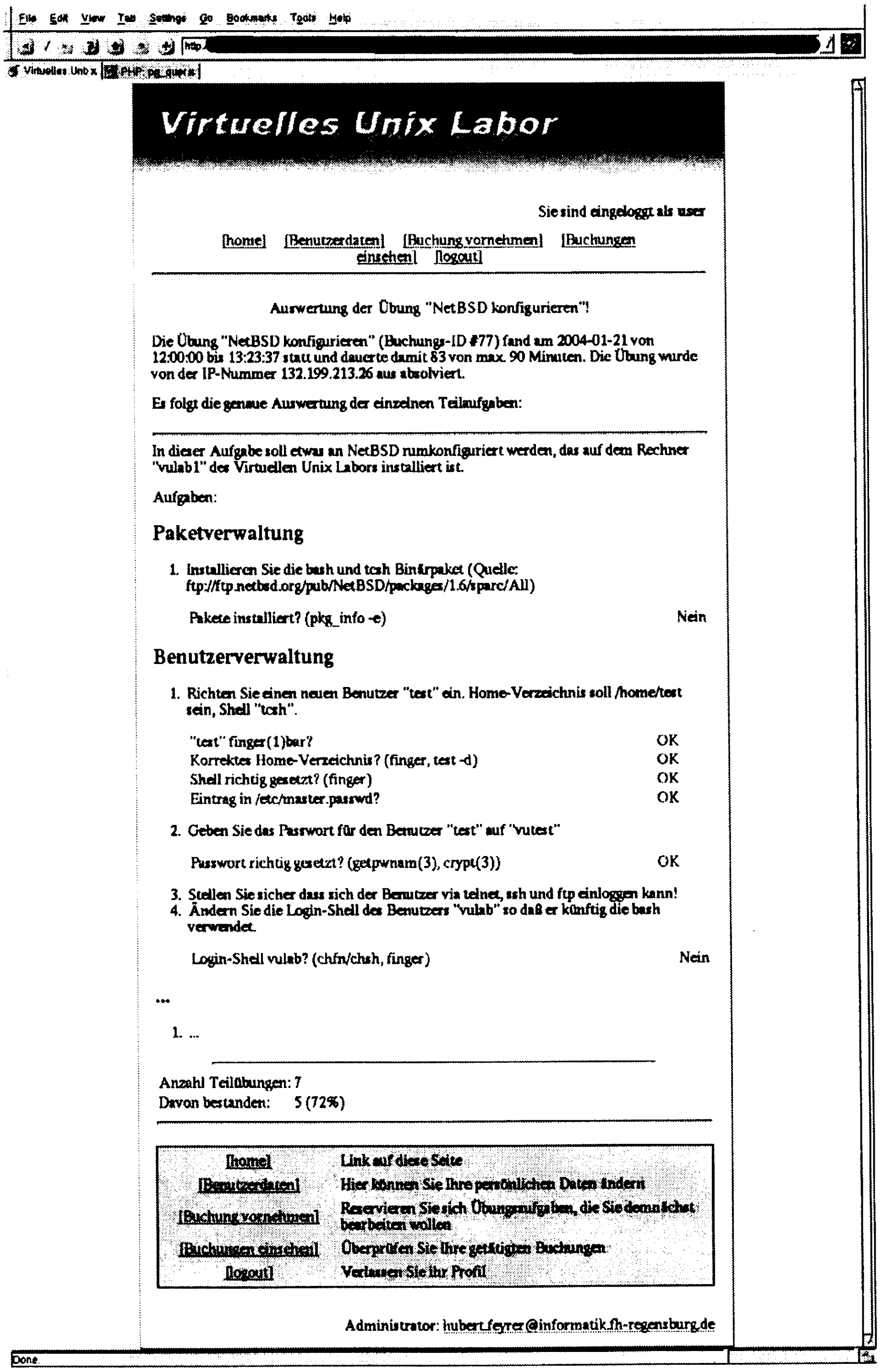
Sie haben die Übung beendet, der Zugriff auf die Übungsrechner wurde gesperrt und das Ergebnis Ihrer Übung wird in der Datenbank gespeichert, wo Sie es in ein paar Minuten einsehen können. Bitte wählen Sie dazu den Punkt "Buchungen einsehen" im Menü!

Vielen Dank für Ihre Teilnahme an der Übung!

[home]	Link auf diese Seite
[Benutzerdaten]	Hier können Sie Ihre persönlichen Daten ändern
[Buchung vornehmen]	Reservieren Sie sich Übungsaufgaben, die Sie demnächst bearbeiten wollen
[Buchungen einsehen]	Überprüfen Sie Ihre getätigten Buchungen
[logout]	Verlassen Sie ihr Profil

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 15: End of exercise



[redacted]

Virtuelles Unix Labor

Sie sind eingeloggt als user

[home] [Benutzerdaten] [Buchung vornehmen] [Buchungen einsehen] [logout]

Auswertung der Übung "NetBSD konfigurieren"!

Die Übung "NetBSD konfigurieren" (Buchungs-ID #77) fand am 2004-01-21 von 12:00:00 bis 13:23:37 statt und dauerte damit 83 von max. 90 Minuten. Die Übung wurde von der IP-Nummer 132.199.213.26 aus absolviert.

Es folgt die genaue Auswertung der einzelnen Teilaufgaben:

In dieser Aufgabe soll etwas an NetBSD rumkonfiguriert werden, das auf dem Rechner "vulab1" des Virtuellen Unix Labors installiert ist.

Aufgaben:

Paketverwaltung

1. Installieren Sie die bash und tcsh Binärpaket (Quelle: ftp://ftp.netbsd.org/pub/NetBSD/packages/1.6/sparc/All)

   Pakete installiert? (pkg_info -e) Nein

Benutzerverwaltung

1. Richten Sie einen neuen Benutzer "test" ein. Home-Verzeichnis soll /home/test sein, Shell "tcsh".

   "test" finger(1)bar? OK
   Korrektes Home-Verzeichnis? (finger, test -d) OK
   Shell richtig gesetzt? (finger) OK
   Eintrag in /etc/master.passwd? OK

2. Geben Sie das Passwort für den Benutzer "test" auf "vutest"

   Passwort richtig gesetzt? (getpwnam(3), crypt(3)) OK

3. Stellen Sie sicher dass sich der Benutzer via telnet, ssh und ftp einloggen kann!
4. Ändern Sie die Login-Shell des Benutzers "vulab" so daß er künftig die bash verwendet.

   Login-Shell vulab? (chfn/chsh, finger) Nein

...

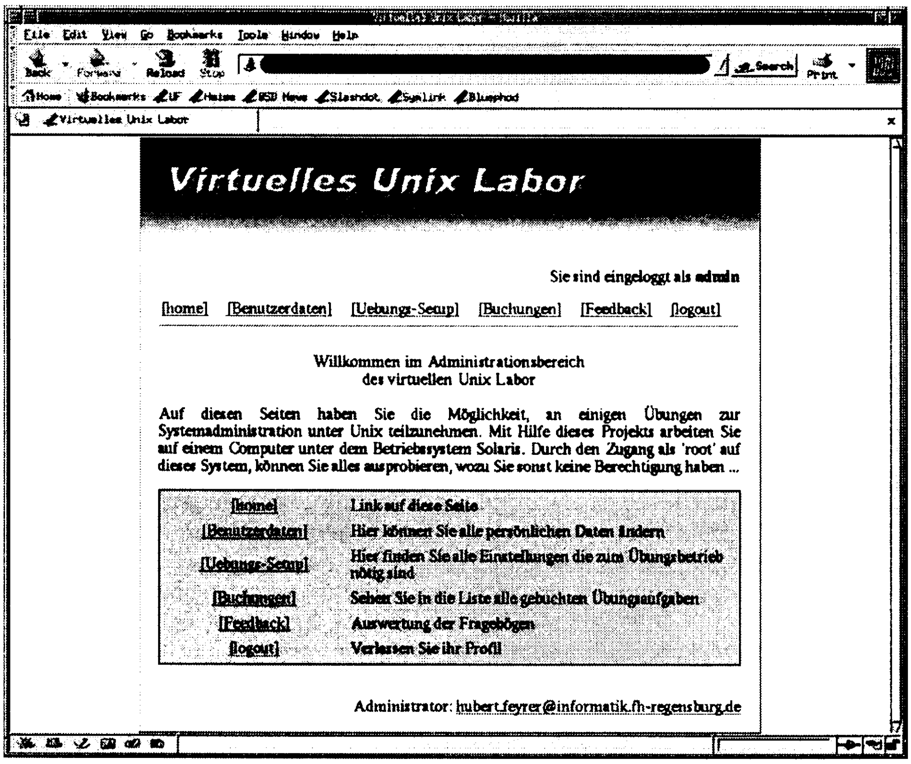
1. ...

Anzahl Teilübungen: 7
Davon bestanden: 5 (72%)

[home]	Link auf diese Seite
[Benutzerdaten]	Hier können Sie Ihre persönlichen Daten ändern
[Buchung vornehmen]	Reservieren Sie sich Übungsaufgaben, die Sie demnächst bearbeiten wollen
[Buchungen einsehen]	Überprüfen Sie Ihre getätigten Buchungen
[logout]	Verlassen Sie Ihr Profil

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 16: Feedback on an exercise taken



Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]

Willkommen im Administrationsbereich
des virtuellen Unix Labor

Auf diesen Seiten haben Sie die Möglichkeit, an einigen Übungen zur Systemadministration unter Unix teilzunehmen. Mit Hilfe dieses Projekts arbeiten Sie auf einem Computer unter dem Betriebssystem Solaris. Durch den Zugang als 'root' auf dieses System, können Sie alles ausprobieren, wozu Sie sonst keine Berechtigung haben ...

[home]	Link auf diese Seite
[Benutzerdaten]	Hier können Sie alle persönlichen Daten ändern
[Uebungs-Setup]	Hier finden Sie alle Einstellungen die zum Übungsbetrieb nötig sind
[Buchungen]	Sehen Sie in die Liste alle gebuchten Übungsaufgaben
[Feedback]	Auswertung der Fragebögen
[logout]	Verlassen Sie ihr Profil

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 17: Administrator's login & menu

login screen.

All the menu items available in this menu will be presented in the following paragraphs.

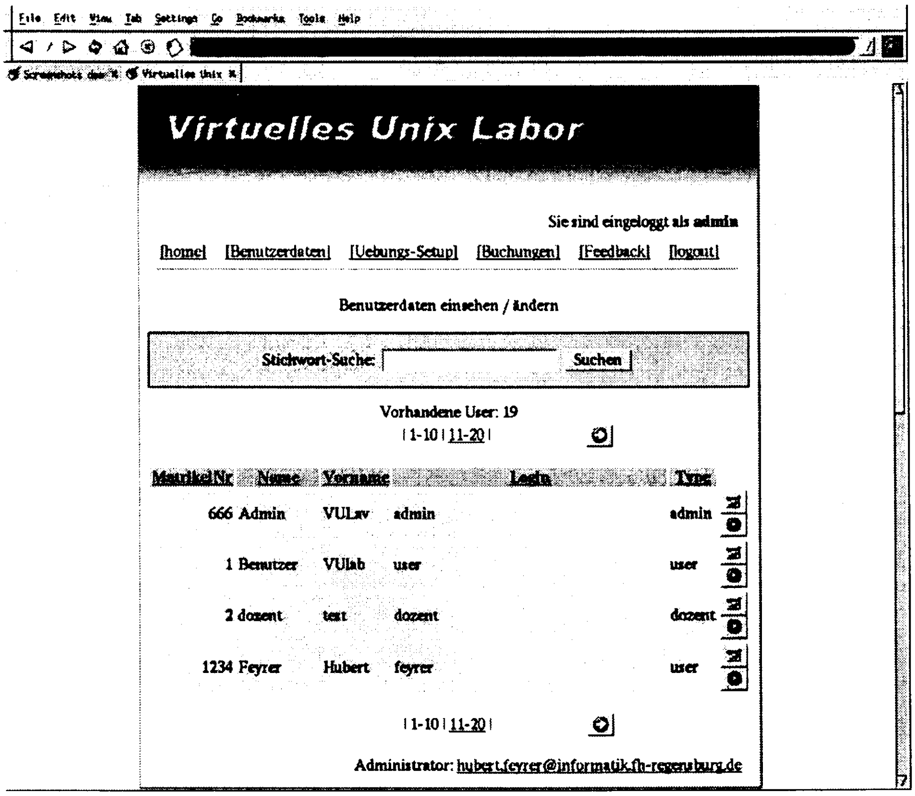
2. Selecting "Benutzerdaten" displays a list of all users of the Virtual Unix Lab as shown in figure 18, including their student ID ("MatrikelNr"), last and first name, login (which defaults to the email address entered when signing up, but can be changed by users later) as well as the type of a user. Types are defined for users and administrators, with an additional type reserved for teachers of the system ("dozent") for future additions.

   Two buttons are available right to each user, the lower button to delete the user from the system (and all his associated data, like exercises he took), and the upper one to edit the data of a certain user.

3. When pressing the button to edit a user's data, the form displayed in figure 19 will be presented to change a users' data.

   Interesting items here are the user type ("Benutzer-Typ"), which is usually "User" for students, and "Admin" for administrators of the Virtual Unix Lab. If a new user's login has not been confirmed yet by entering the authentication token mailed to the user when registering, the "Anmeldung bestätigt?" field will be set to "Nein" to indicate the login hasn't been confirmed, with the authentication token being displayed for confirmation & verification.

   If a user can't remember his password, a new one can be filled in (twice, for confirmation) to set a new one. During tests with a group of 40 computer science students over the duration of one semester in summer 2004, it was amazing how many students managed to forget their password – "I have forgotten my password" apparently isn't a user-support myth (only)!



Virtuelles Unix Labor

Sie sind eingeloggt als admin

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]

Benutzerdaten einsehen / ändern

Stichwort-Suche: Suchen

Vorhandene User: 19

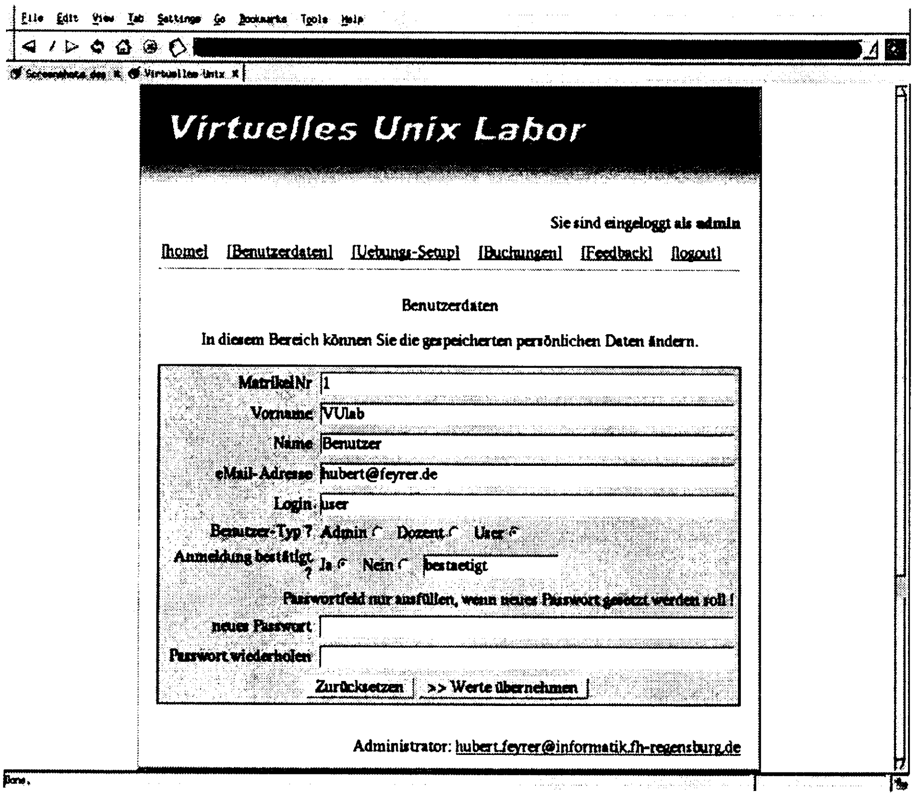
| 1-10 | 11-20 |

MatrikelNr	Name	Vorname	Login	Typ
666	Admin	VULav	admin	admin
1	Benutzer	VUlab	user	user
2	dozent	test	dozent	dozent
1234	Feyrer	Hubert	feyrer	user

| 1-10 | 11-20 |

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 18: Managing users



Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]

Benutzerdaten

In diesem Bereich können Sie die gespeicherten persönlichen Daten ändern.

MatrikelNr	1
Vorname	VUlab
Name	Benutzer
eMail-Adresse	hubert@feyrer.de
Login	user
Benutzer-Typ ?	Admin ○ Dozent ○ User ●
Anmeldung bestätigt ?	Ja ● Nein ○ bestaetigt

Passwortfeld nur ausfüllen, wenn neues Passwort gesetzt werden soll !

neues Passwort	
Passwort wiederholen	

Zurücksetzen | >> Werte übernehmen

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 19: Editing user specific data

4. Listing all exercises booked by all users in the past and future can be achieved by selecting "Buchungen" from the menu. The list, displayed in figure 20, contains login name of the user and what exercise he booked, including date & time and also an indicator if the exercise is still to be taken ("freigegeben") or already done.

   To the right of each exercises, two buttons are placed, just as in the for for editing user data. The lower button can be used to delete all traces of a past exercise (not available to normal users to prevent them from destroying evidence), the upper button can be used to retrieve feedback on a particular exercise for both admin and student users.

5. When an administrator selects feedback on a certain exercise, he will get similar feedback as normal users (see figure 21), containing data if single tasks of the exercise were solved successfully, and some hints what the system did to test (in green font).

   In addition to normal users, administrators will get an overview on how all students taking the same exercise performed, giving numbers on how many did ("Bestanden") or did not ("Nicht bestanden") manage to solve the task successfully, as well as an overall number of students who did the exercise. In addition to absolute numbers and percentage, bars of "o"s are printed to give sort of a graphical overview, making it more visible how the overall group performed.

6. When selecting the "Uebung-Setup" menu item, various settings for new and existing exercises displayed in figure 22 can be changed. First, a new menu row will appear which allows managing the list of lab machines available ("Rechner verwalten"), the list of harddisk images available for them ("Images verwalten") as well as creating a new exercise ("neue Übung erstellen)". Furthermore, a list of all existing exercises is given, with two buttons on the right. The rightmost button can be used to delete an exercise (including all data available on that exercise, esp. past exercises taken by students - use with care!). The other button can be used to change various settings of the exercise, using the same menus as when creating a new exercise, see below.

3.3 Creating New Exercises

1. When creating new exercises or editing existing ones, a list of lab machines needs to be known. This list can be edited by selecting "Rechner verwalten" in the previous menu, and using the user interface shown in figure 23.

   Buttons besides the entries are used to delete entries for lab machines, or edit properties, with hostnames being the only property right now.

2. A similar list can be retrieved for all harddisk images available, that can be deployed on the lab machines.

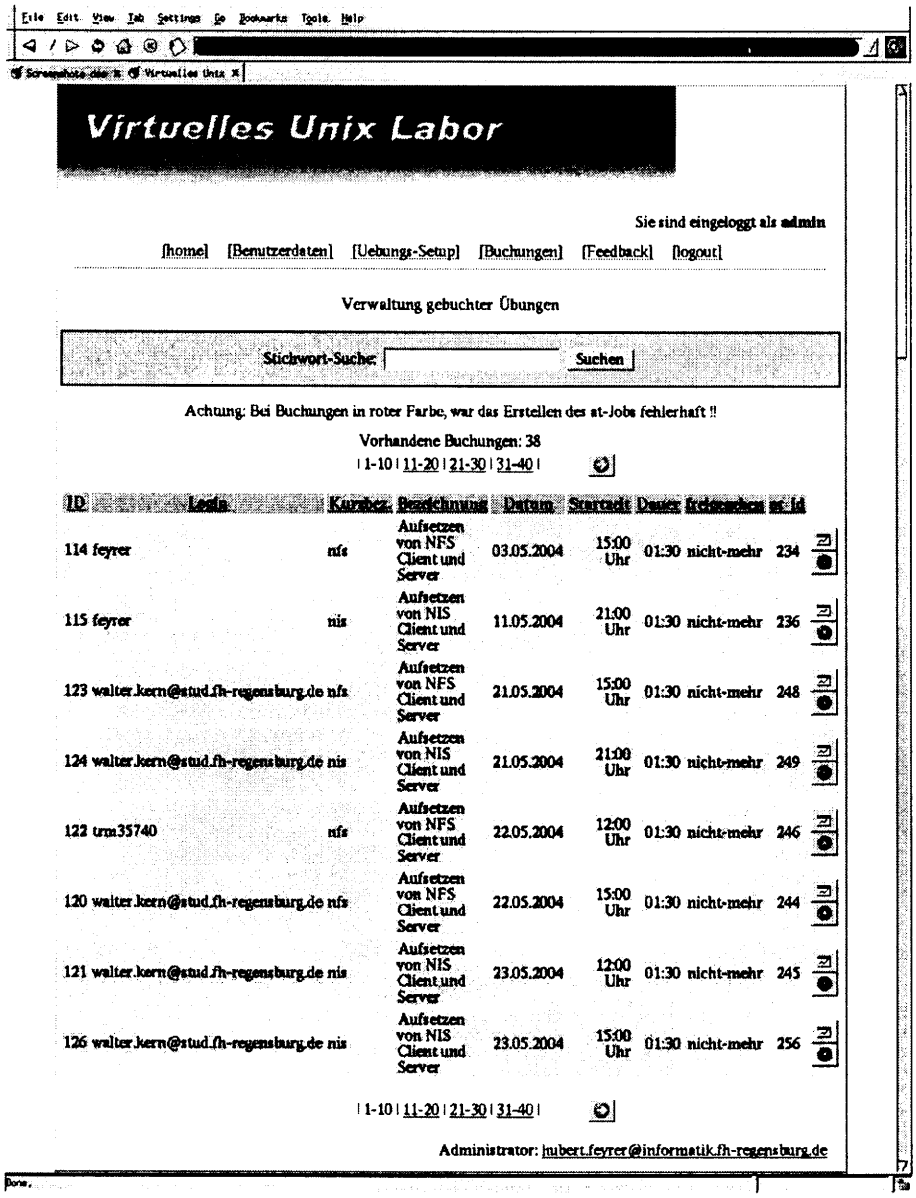
   As can be seen in figure 24, harddisk images are available for NetBSD 1.6.2/sparc and Solaris 9/sparc right now. Adding other harddisk images for other operating systems like Linux, or special setups like troubleshooting would be easily possible.

   At this point, only one kind of lab machine is available (two Sun SPARCstation 4), and thus no additional checks are needed if an image can be deployed on a certain machine. This may change in the future!

3. When choosing to create a new exercises, three screens have to be filled with data.

   The first one asking for general information on the new exercise, like a short description ("Kurzbezeichnung"), one-line description ("Bezeichnung") and a username that should be allowed exclusive access to the exercises ("Nur für", used for administrative exercises, see below). Time for preparing the lab machines ("Vorlauf"), duration of the exercise ("Dauer") as well as time for analyzing the lab machines ("Nachlauf") is needed next, each given in hours and minutes.

   If an exercise should not be repeatable like for a real test, not just a lab exercise, this can be done by setting the exercise to be not repeatable ("Wiederholbar?"). Next item needed is file placed in the filesystem of the Virtual Unix Lab which contains the exercise text ("Pfad auf die Textdatei") and which needs to contain some special PHP calls to allow giving feedback (not covered here).



Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]

Verwaltung gebuchter Übungen

Stichwort-Suche: Suchen

Achtung: Bei Buchungen in roter Farbe, war das Erstellen des at-Jobs fehlerhaft !!

Vorhandene Buchungen: 38

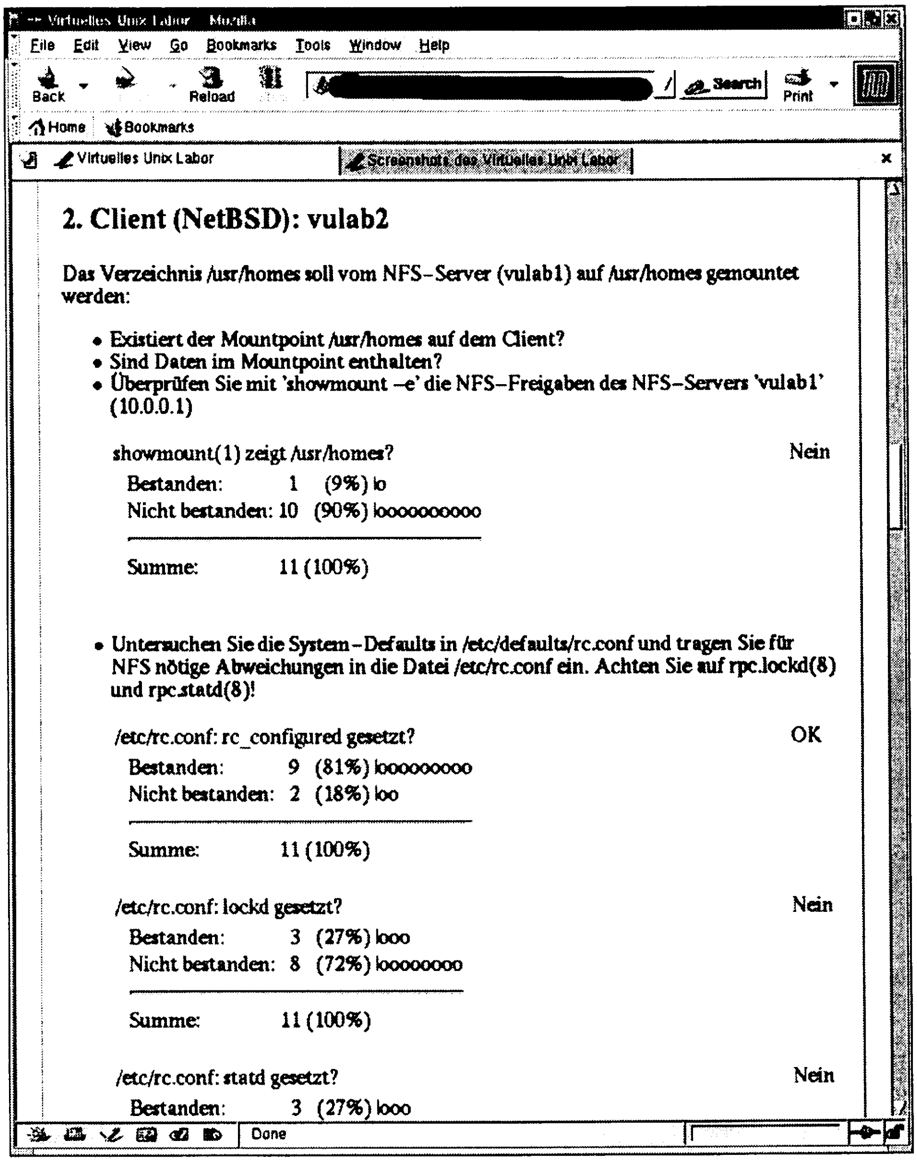
| 1-10 | 11-20 | 21-30 | 31-40 |

ID	Login	Kurzbez.	Bezeichnung	Datum	Startzeit	Dauer	freigegeben	at-Id
114	feyrer	nfs	Aufsetzen von NFS Client und Server	03.05.2004	15:00 Uhr	01:30	nicht-mehr	234
115	feyrer	nis	Aufsetzen von NIS Client und Server	11.05.2004	21:00 Uhr	01:30	nicht-mehr	236
123	walter.kern@stud.fh-regensburg.de	nfs	Aufsetzen von NFS Client und Server	21.05.2004	15:00 Uhr	01:30	nicht-mehr	248
124	walter.kern@stud.fh-regensburg.de	nis	Aufsetzen von NIS Client und Server	21.05.2004	21:00 Uhr	01:30	nicht-mehr	249
122	trm35740	nfs	Aufsetzen von NFS Client und Server	22.05.2004	12:00 Uhr	01:30	nicht-mehr	246
120	walter.kern@stud.fh-regensburg.de	nfs	Aufsetzen von NFS Client und Server	22.05.2004	15:00 Uhr	01:30	nicht-mehr	244
121	walter.kern@stud.fh-regensburg.de	nis	Aufsetzen von NIS Client und Server	23.05.2004	12:00 Uhr	01:30	nicht-mehr	245
126	walter.kern@stud.fh-regensburg.de	nis	Aufsetzen von NIS Client und Server	23.05.2004	15:00 Uhr	01:30	nicht-mehr	256

| 1-10 | 11-20 | 21-30 | 31-40 |

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 20: Managing booked exercises, past & future


Virtuelles Unix Labor Mozilla
File Edit View Go Bookmarks Tools Window Help
Back
Reload
[redacted]
Search
Print
Home
Bookmarks
Virtuelles Unix Labor
Screenshots des Virtuelles Unix Labor
Done

2. Client (NetBSD): vulab2

Das Verzeichnis /usr/homes soll vom NFS-Server (vulab1) auf /usr/homes gemountet werden:

- Existiert der Mountpoint /usr/homes auf dem Client?
- Sind Daten im Mountpoint enthalten?
- Überprüfen Sie mit 'showmount -e' die NFS-Freigaben des NFS-Servers 'vulab1' (10.0.0.1)

showmount(1) zeigt /usr/homes? Nein

Bestanden: 1 (9%) lo
Nicht bestanden: 10 (90%) loooooooooo

Summe: 11 (100%)

- Untersuchen Sie die System-Defaults in /etc/defaults/rc.conf und tragen Sie für NFS nötige Abweichungen in die Datei /etc/rc.conf ein. Achten Sie auf rpc.lockd(8) und rpc.statd(8)!

/etc/rc.conf: rc_configured gesetzt? OK

Bestanden: 9 (81%) loooooooooo
Nicht bestanden: 2 (18%) loo

Summe: 11 (100%)

/etc/rc.conf: lockd gesetzt? Nein

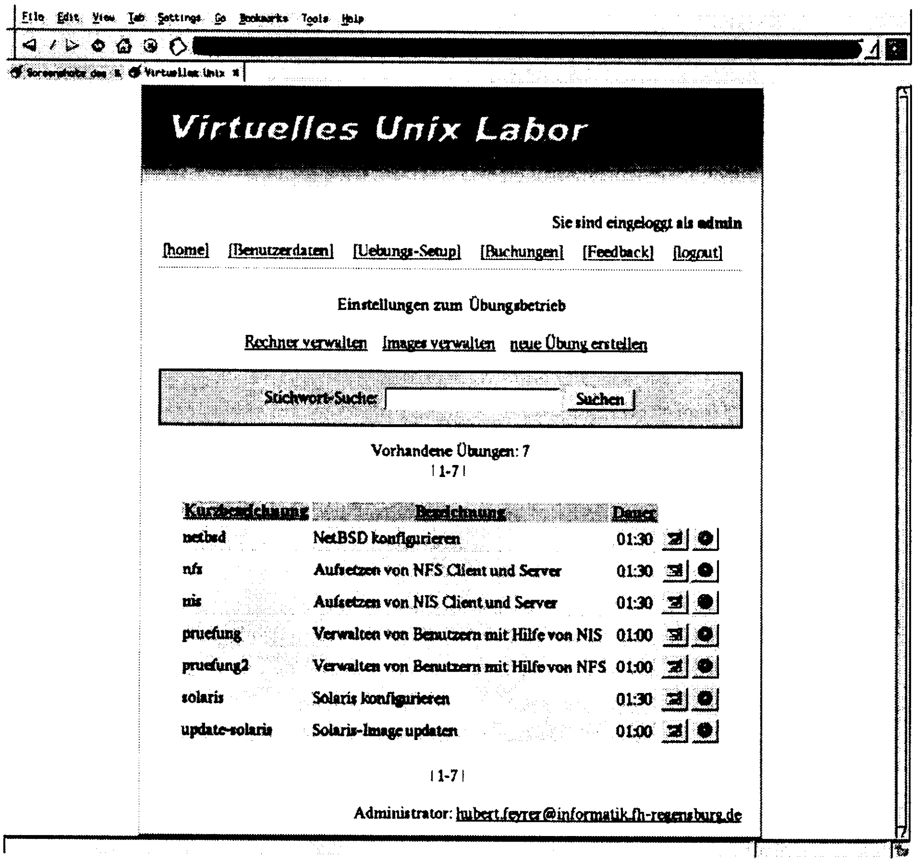
Bestanden: 3 (27%) looo
Nicht bestanden: 8 (72%) loooooooo

Summe: 11 (100%)

/etc/rc.conf: statd gesetzt? Nein

Bestanden: 3 (27%) looo

Figure 21: Retrieving feedback on a group's performance



Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]

Einstellungen zum Übungsbetrieb

Rechner verwalten Images verwalten neue Übung erstellen

Stichwort-Suche: Suchen

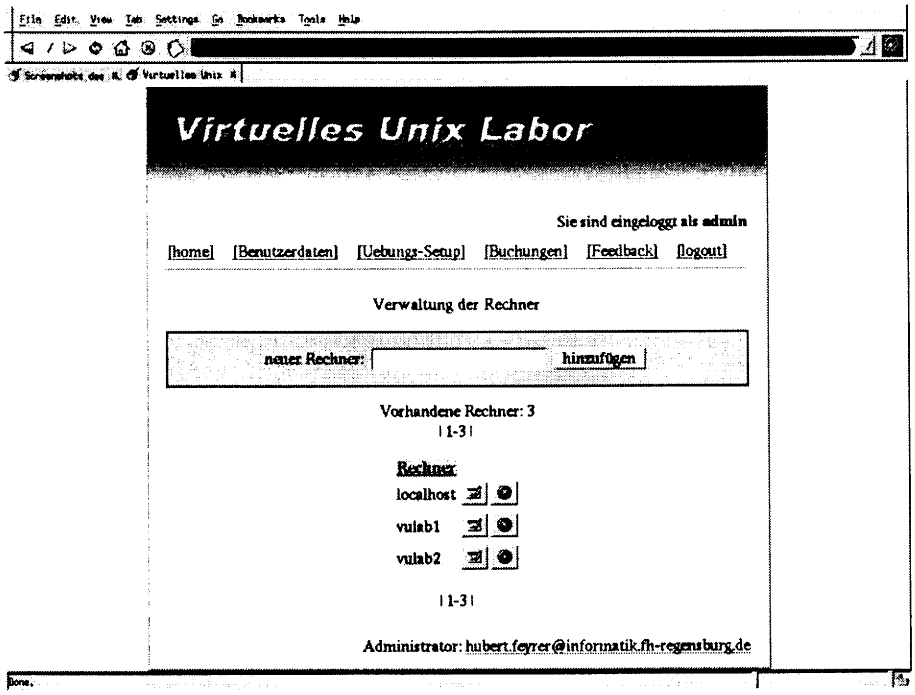
Vorhandene Übungen: 7
| 1-7 |

Kurzbezeichnung	Bezeichnung	Dauer		
netbsd	NetBSD konfigurieren	01:30		●
nfs	Aufsetzen von NFS Client und Server	01:30		●
nis	Aufsetzen von NIS Client und Server	01:30		●
pruefung	Verwalten von Benutzern mit Hilfe von NIS	01:00		●
pruefung2	Verwalten von Benutzern mit Hilfe von NFS	01:00		●
solaris	Solaris konfigurieren	01:30		●
update-solaris	Solaris-Image updaten	01:00		●

| 1-7 |

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 22: Editing various properties of an exercise



File Edit View Tab Settings Go Bookmarks Tools Help
Screenshots des ... Virtuelles Unix
Virtuelles Unix Labor
[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]
hinzufügen
Done.

Sie sind eingeloggt als **admin**

Verwaltung der Rechner

neuer Rechner:

Vorhandene Rechner: 3
| 1-3 |

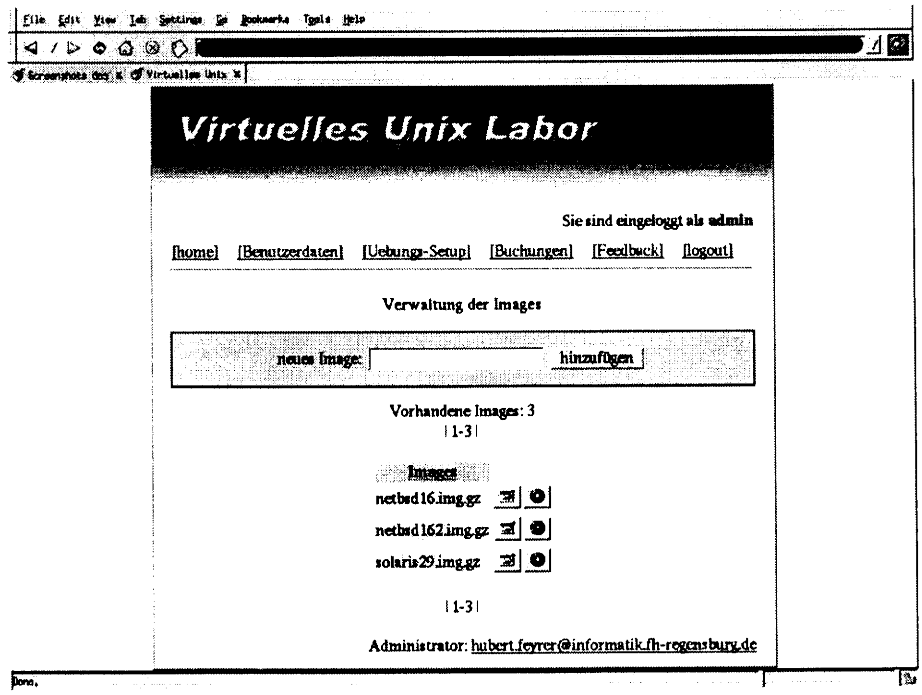
Rechner

localhost
vulab1
vulab2

| 1-3 |

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 23: Managing lab machines



File Edit View Tab Settings Go Bookmarks Tools Help
Screenshots des ... Virtuelles Unix
Virtuelles Unix Labor
[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]
hinzufügen
Done.

Sie sind eingeloggt als **admin**

Verwaltung der Images

neues Image:

Vorhandene Images: 3
| 1-3 |

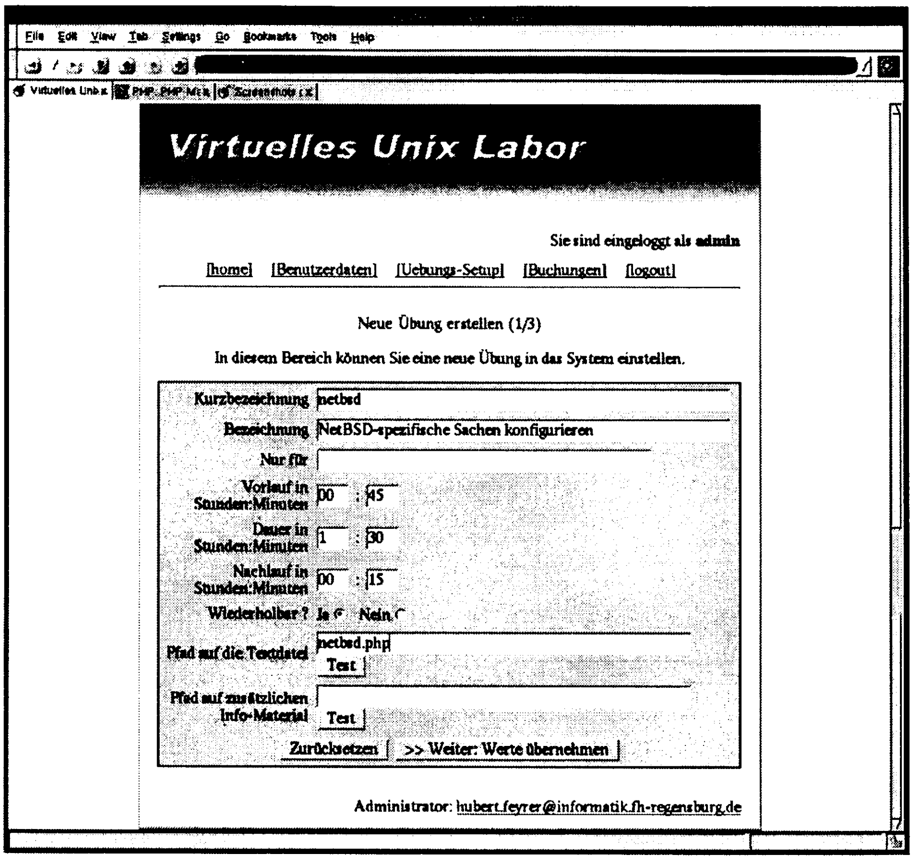
Images

netbsd16.img.gz
netbsd162.img.gz
solaris29.img.gz

| 1-3 |

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 24: Managing disk images for lab machines


File Edit View Tab Settings Go Bookmarks Tools Help
Virtuelles Unix x
PHP: PHP Mix
Screenshot x
[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [logout]
Test
Test
Zurücksetzen
>> Weiter: Werte übernehmen

Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

Neue Übung erstellen (1/3)

In diesem Bereich können Sie eine neue Übung in das System einstellen.

Kurzbezeichnung	netbsd
Bezeichnung	NetBSD-spezifische Sachen konfigurieren
Nur für	
Vorlauf in Stunden:Minuten	00 : 45
Dauer in Stunden:Minuten	1 : 30
Nachlauf in Stunden:Minuten	00 : 15
Wiederholbar ?	Ja ● Nein ○
Pfad auf die Testdatei	netbsd.php
Pfad auf zusätzlichen Info-Material	

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 25: Entering basic data for a new exercise

Finally, design plans for the Virtual Unix Lab include a tutorial component which can give help on demand, which is what the filename with additional information ("Pfad auf zusätzliches Info-Material") is intended for, but not used right now.

Figure 25 displays the first screen asking for all this data.

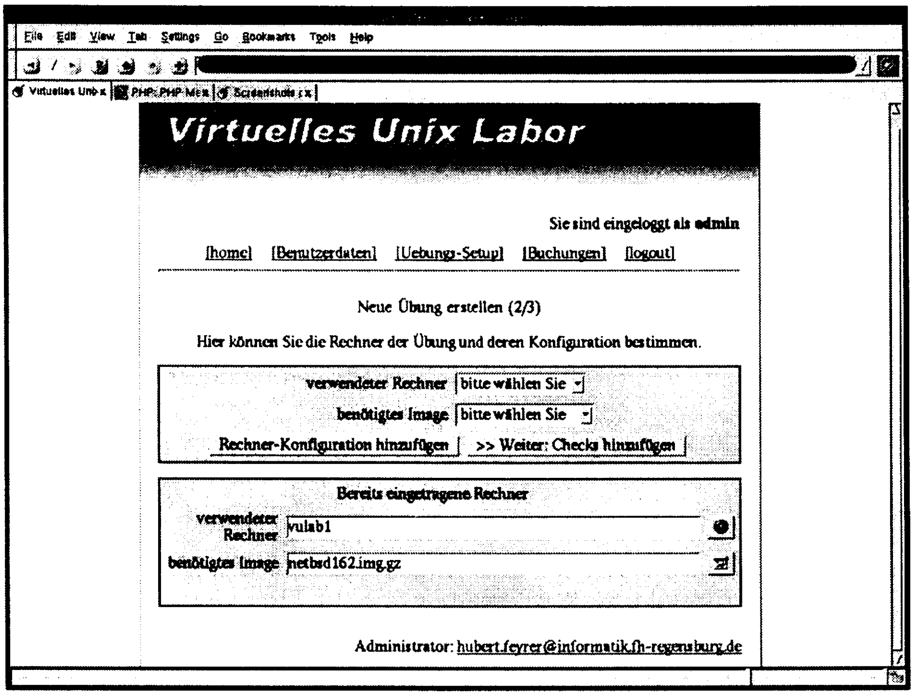
4. The second part of creating a new exercise consists of deciding which lab machines are needed for the exercise ("verwendeter Rechner"), and which harddisk image they should get installed ("benötigtes Image"). Pressing the "Rechner-Konfiguration hinzufügen" button will add the machine/image-combination to the exercise. A list of machines & images already part of the image are printed below that, with buttons available to delete the machine/image combination or to edit it. To make sure several lab machines get setup for an exercise, they must be added with an appropriate exercise here.

   In the example screenshot displayed in figure 26, only one lab machine running NetBSD 1.6.2 would be prepared for the exercise (unless others were added).

5. The third and last step of creating a new exercise consists of defining which tests are made at the end of the exercise to determine if the (parts of) the exercise were performed successfully or not.

   Testing can be done via a number of so-called "Check-Scripts", which exist to test a number of aspects of a system. Parameters can be passed to the scripts to adjust what they do (see below). As an exercise can involve several lab machines, and as each lab machine can run a different configuration (client, server, ...), it's important to define on which machine a check script runs ("Läuft auf Rechner").

   The selected check script will be ran on the named machine with the given parameters at the end of of an exercise, and store the result of the script – success or failure –


File Edit View Tab Settings Go Bookmarks Tools Help
Virtuelles Unix | PHP: PHP Men | Screenshots

Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [logout]

Neue Übung erstellen (2/3)

Hier können Sie die Rechner der Übung und deren Konfiguration bestimmen.

verwendeter Rechner bitte wählen Sie

benötigtes Image bitte wählen Sie

Rechner-Konfiguration hinzufügen | >> Weiter: Checks hinzufügen

Bereits eingetragene Rechner

verwendeter Rechner vulab1

benötigtes Image netbsd162.img.gz

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 26: Defining machines & configuration for a new exercise

in the database for giving feedback later. As a simple display of "success" or "you failed" may not be too helpful[1], an additional description of what exactly was tested and was either passed or failed can be given ("Bezeichnung für Auswertung").

As with the selection of lab machines used for an exercise, several checks can be added to an exercise ("Check hinzufügen"), completion of the new exercise can be indicated by pressing the "Fertig" button of the form displayed in figure 27.

6. All scripts available to perform checks on the lab machines are stored on the filesystem of the Virtual Unix Lab server. A list which allows easy selection of a check-script is made available through the PHP framework displayed in figure 28.

   Scripts are available to test various aspects of a system, either being independent of the operating system ("`check-*`"), work on all Unix systems ("`unix-check-*`") or only on a particular Unix flavour ("`netbsd-check-*`", "`solaris-check-*`"). If a certain subsystem allows various properties to be changed, that's also encoded in the check script's filename ("`unix-check-user-*`", "`check-file-*`"). Future incarnations of the Virtual Unix Lab may also include check scripts to run on Windows systems.

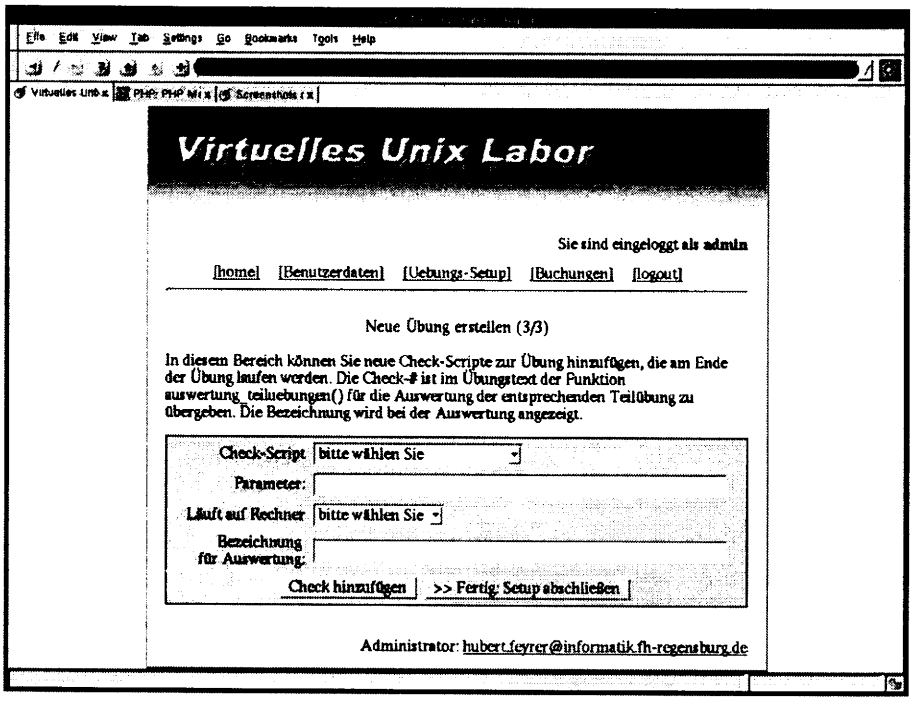
7. Figure 29 shows an exercise that has some check scripts defined to be ran upon completion of the exercise:.

   For each check, buttons are available to remove it from the list of checks to run, and to edit the data stored for the particular check.

8. When choosing to edit a particular check, all the parameters from the previous form can be changed using the form displayed in figure 30: script, parameters, description for feedback, and on which machine to run the script.

   In addition, information on what the script does is printed in addition to a list of parameters that the script takes, including name of the variable, the default value (if applicable), and a description of the parameters.

[1] [Schulmeister, 1997] p. 111



Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [logout]

Neue Übung erstellen (3/3)

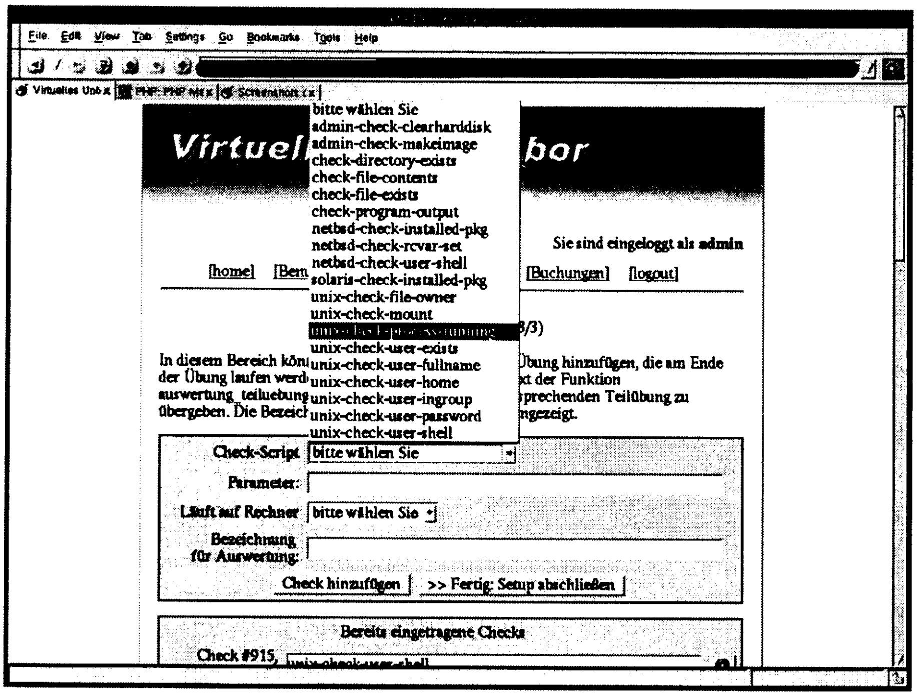
In diesem Bereich können Sie neue Check-Scripte zur Übung hinzufügen, die am Ende der Übung laufen werden. Die Check-# ist im Übungstext der Funktion auswertung_teiluebungen() für die Auswertung der entsprechenden Teilübung zu übergeben. Die Bezeichnung wird bei der Auswertung angezeigt.

Check-Script	bitte wählen Sie
Parameter:	
Läuft auf Rechner	bitte wählen Sie
Bezeichnung für Auswertung:	

Check hinzufügen | >> Fertig: Setup abschließen

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 27: Adding checks for a new exercise



Virtuel... bor

Sie sind eingeloggt als **admin**

[home] [Ben... [Buchungen] [logout]

bitte wählen Sie
admin-check-clearharddisk
admin-check-makeimage
check-directory-exists
check-file-contents
check-file-exists
check-program-output
netbsd-check-installed-pkg
netbsd-check-rcvar-set
netbsd-check-user-shell
solaris-check-installed-pkg
unix-check-file-owner
unix-check-mount
unix-check-process-running
unix-check-user-exists
unix-check-user-fullname
unix-check-user-home
unix-check-user-ingroup
unix-check-user-password
unix-check-user-shell

3/3)

In diesem Bereich kön... Übung hinzufügen, die am Ende der Übung laufen werd... xt der Funktion auswertung_teiluebung... sprechenden Teilübung zu übergeben. Die Bezeich... ngezeigt.

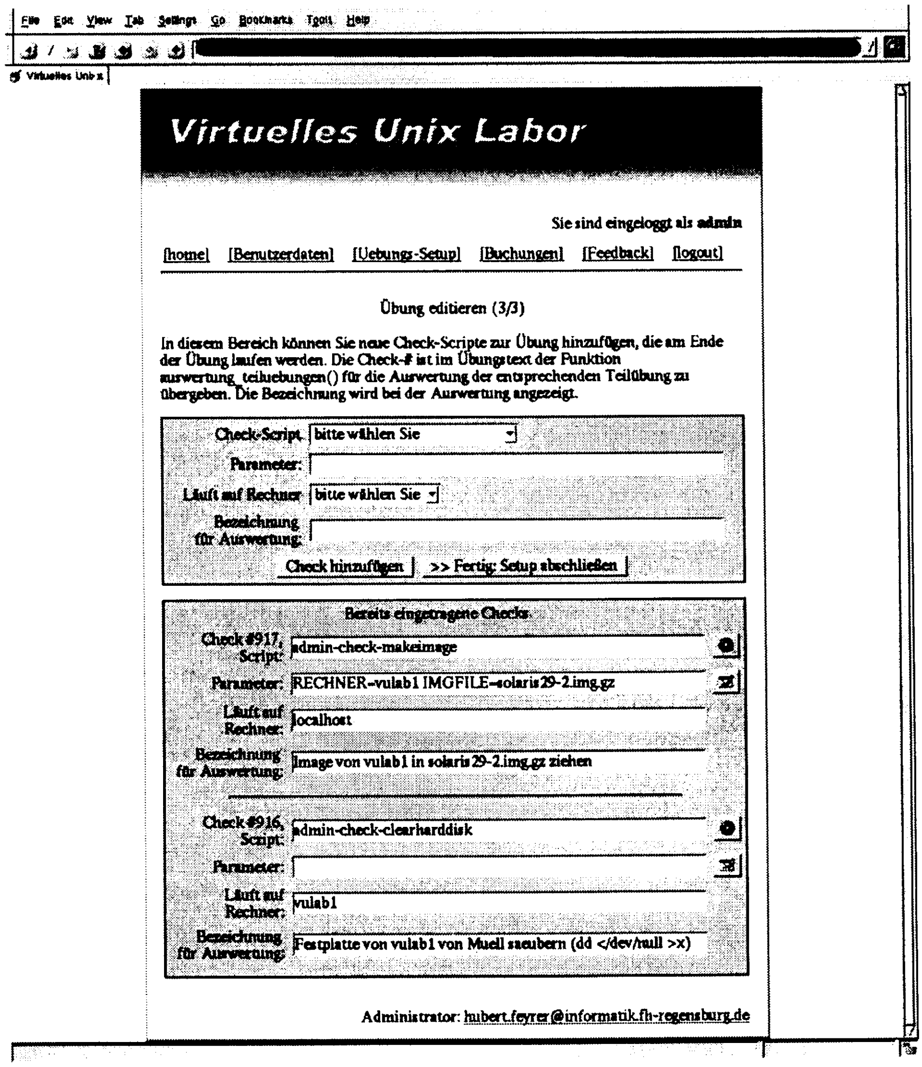
Check-Script	bitte wählen Sie
Parameter:	
Läuft auf Rechner	bitte wählen Sie
Bezeichnung für Auswertung:	

Check hinzufügen | >> Fertig: Setup abschließen

Bereits eingetragene Checks

Check #915, unix-check-user-shell

Figure 28: A list of available check-scripts



Virtuelles Unix Labor

Sie sind eingeloggt als **admin**

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]

Übung editieren (3/3)

In diesem Bereich können Sie neue Check-Scripte zur Übung hinzufügen, die am Ende der Übung laufen werden. Die Check-# ist im Übungstext der Funktion auswertung_teiluebungen() für die Auswertung der entsprechenden Teilübung zu übergeben. Die Bezeichnung wird bei der Auswertung angezeigt.

Check-Script: bitte wählen Sie
Parameter:
Läuft auf Rechner: bitte wählen Sie
Bezeichnung für Auswertung:

Check hinzufügen | >> Fertig: Setup abschließen

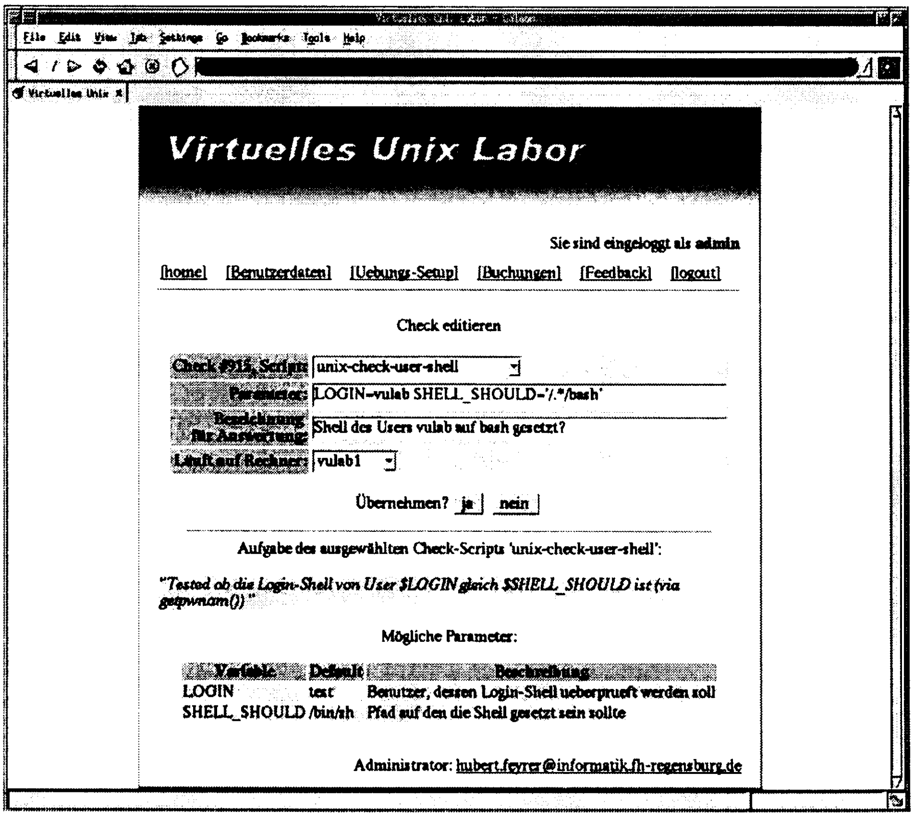
Bereits eingetragene Checks:

Check #917, Script: admin-check-makeimage
Parameter: RECHNER=vulab1 IMGFILE=solaris29-2.img.gz
Läuft auf Rechner: localhost
Bezeichnung für Auswertung: Image von vulab1 in solaris29-2.img.gz ziehen

Check #916, Script: admin-check-clearharddisk
Parameter:
Läuft auf Rechner: vulab1
Bezeichnung für Auswertung: Festplatte von vulab1 von Muell saeubern (dd </dev/null >x)

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 29: An exercise with two checks defined



Virtuelles Unix Labor

Sie sind eingeloggt als admin

[home] [Benutzerdaten] [Uebungs-Setup] [Buchungen] [Feedback] [logout]

Check editieren

Check #915, Script: unix-check-user-shell

Parameter: LOGIN=vulab SHELL_SHOULD='/.*/bash'

Bezeichnung für Auswertung: Shell des Users vulab auf bash gesetzt?

Läuft auf Rechner: vulab1

Übernehmen? ja nein

Aufgabe des ausgewählten Check-Scripts 'unix-check-user-shell':

"Tested ob die Login-Shell von User $LOGIN gleich $SHELL_SHOULD ist (via getpwnam())"

Mögliche Parameter:

Variable	Default	Beschreibung
LOGIN	test	Benutzer, dessen Login-Shell ueberprueft werden soll
SHELL_SHOULD	/bin/sh	Pfad auf den die Shell gesetzt sein sollte

Administrator: hubert.feyrer@informatik.fh-regensburg.de

Figure 30: Editing data of a certain check

The information on purpose of a check-script and list of parameters including their description is retrieved from the check scripts – all check scripts are expected to allow querying them for these informations.

9. A typical exercise consists of about 30-40 single checks to run at the end. Entering all the data for all these checks via the web frontend is possible, but tiresome. To solve this problem, an alternative way to enter data on check scripts into the database had to be constructed[2].

   As the text of an exercise already contains calls to PHP functions to retrieve success of the particular check (using the check numbers shown in the previous screenshots), putting data near these calls was an obvious solution. So, instead of putting

```
Do task #1

<?php auswertung_teiluebungen( 916, 917);
 ?>

Do task #2
...
```

   into the exercise text's PHP file to describe task #1 and associate checks numbered 916 and 917 with them as e.g. defined in figure 29, the layout can first be a bit changed (no functional change):

```
Do task #1

<?php auswertung_teiluebungen(
            916,
            917
        );
 ?>
```

   and after this layout change, comments can be made in the PHP code to add the data for the checks as comments as shown in figure 31.

   With an appropriate preprocessor, the data can be extracted and stored in the database. In the above example, the check-numbers are not known when writing the exercise text, and left as "XXX". When running the preprocessor, it will extract data from the PHP comments, and store them into the database.

   Figure 32 shows running the preprocessor, which fills in the now-known check-numbers into the PHP calls for giving feedback as shown in figure 33.

   Using this framework, it is possible to keep all the data for an exercise – text, which check script, on which lab machine to run it, any possible parameters as well as text for feedback – in one file, which is a lot easier to maintain. The web interface can still be used to edit existing exercises.

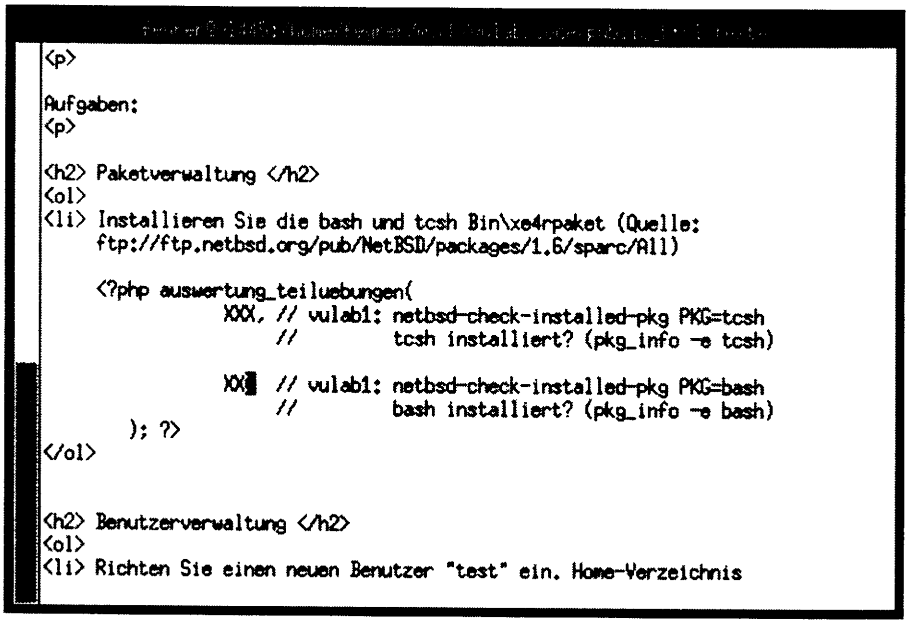
   More words could be spent here on exercise verification, stereotypes and language design, system front-ends, domain specific languages and design patterns, but it would be beyond the introductional character of this paper and be published elsewhere.

4 Setup

4.1 Hardware

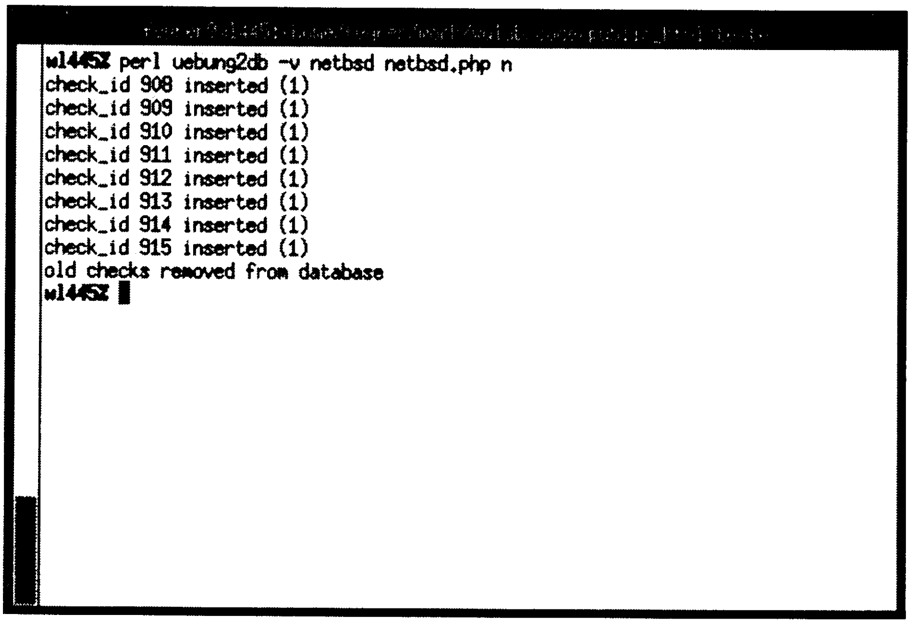
The following machine is used as server machine:

[2] [Spinellis, 2001] p. 96



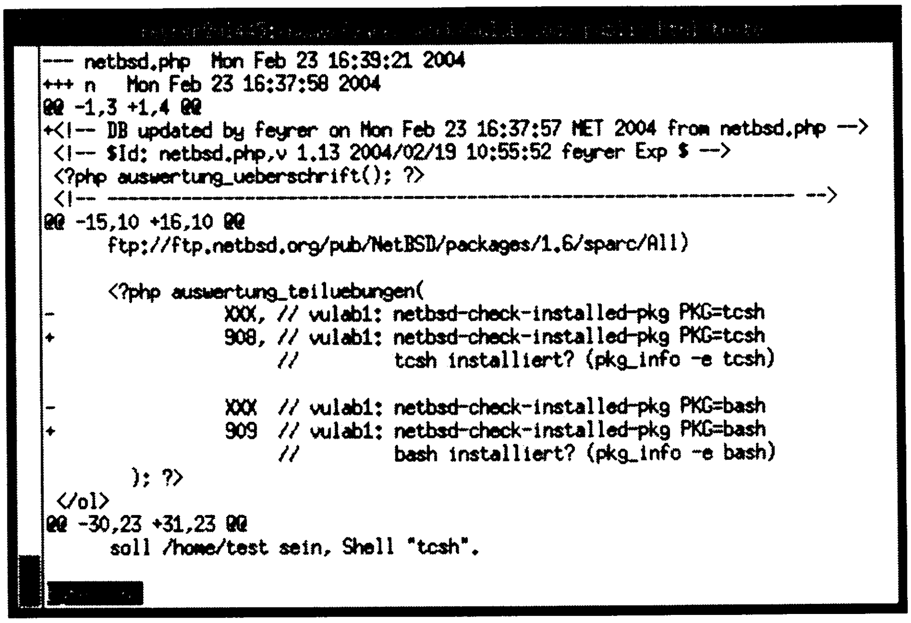
```
<p>

Aufgaben:
<p>

<h2> Paketverwaltung </h2>
<ol>
<li> Installieren Sie die bash und tcsh Bin\xe4rpaket (Quelle:
     ftp://ftp.netbsd.org/pub/NetBSD/packages/1.6/sparc/All)

     <?php auswertung_teiluebungen(
                XXX, // vulab1: netbsd-check-installed-pkg PKG=tcsh
                     //         tcsh installiert? (pkg_info -e tcsh)

                XX█  // vulab1: netbsd-check-installed-pkg PKG=bash
                     //         bash installiert? (pkg_info -e bash)
        ); ?>
</ol>


<h2> Benutzerverwaltung </h2>
<ol>
<li> Richten Sie einen neuen Benutzer "test" ein. Home-Verzeichnis
```

Figure 31: Embedding check-data into exercise text comments



```
wl445% perl uebung2db -v netbsd netbsd.php n
check_id 908 inserted (1)
check_id 909 inserted (1)
check_id 910 inserted (1)
check_id 911 inserted (1)
check_id 912 inserted (1)
check_id 913 inserted (1)
check_id 914 inserted (1)
check_id 915 inserted (1)
old checks removed from database
wl445% █
```

Figure 32: Extracting check data into the database



```
--- netbsd.php  Mon Feb 23 16:39:21 2004
+++ n   Mon Feb 23 16:37:58 2004
@@ -1,3 +1,4 @@
+<!-- DB updated by feyrer on Mon Feb 23 16:37:57 MET 2004 from netbsd.php -->
 <!-- $Id: netbsd.php,v 1.13 2004/02/19 10:55:52 feyrer Exp $ -->
 <?php auswertung_ueberschrift(); ?>
 <!-- ------------------------------------------------------------ -->
@@ -15,10 +16,10 @@
       ftp://ftp.netbsd.org/pub/NetBSD/packages/1.6/sparc/All)

       <?php auswertung_teiluebungen(
-                XXX, // vulab1: netbsd-check-installed-pkg PKG=tcsh
+                908, // vulab1: netbsd-check-installed-pkg PKG=tcsh
                     //          tcsh installiert? (pkg_info -e tcsh)

-                XXX  // vulab1: netbsd-check-installed-pkg PKG=bash
+                909  // vulab1: netbsd-check-installed-pkg PKG=bash
                     //          bash installiert? (pkg_info -e bash)
         ); ?>
 </ol>
@@ -30,23 +31,23 @@
       soll /home/test sein, Shell "tcsh".
```

Figure 33: The preprocessor has filled in the check-script numbers

- Sun SPARCstation 5, 85MHz
- 192 MB RAM
- 3* external SCSI disk
- additional SBus ethernet card
- Runs NetBSD 1.6.2/sparc

The following machines are used as lab clients:

- Two
- Sun SPARCstation 4, 110MHz
- 64 MB RAM
- 1 GB internal SCSI disk
- Run NetBSD 1.6.2/sparc or Solaris 9/sparc

The ultimate goal here is to use virtual machines instead of real hardware. When the Virtual Unix Lab project was started, no hardware for running virtual machines was available.

4.2 Lab Machine Installation

Installation of the lab machine is done by using the server to act as DHCP, RARP and NFS server to lab-internal network. When a new exercise is to be prepared, the lab clients are netbooted, so they are independent of any operating system (and it's possible damaged state). From the netbooted environment, a new harddisk image is written to the lab client's harddisk. The image-deployment techniques used here were developed in the g4u[3] project.

[3] [Feyrer, 2004]

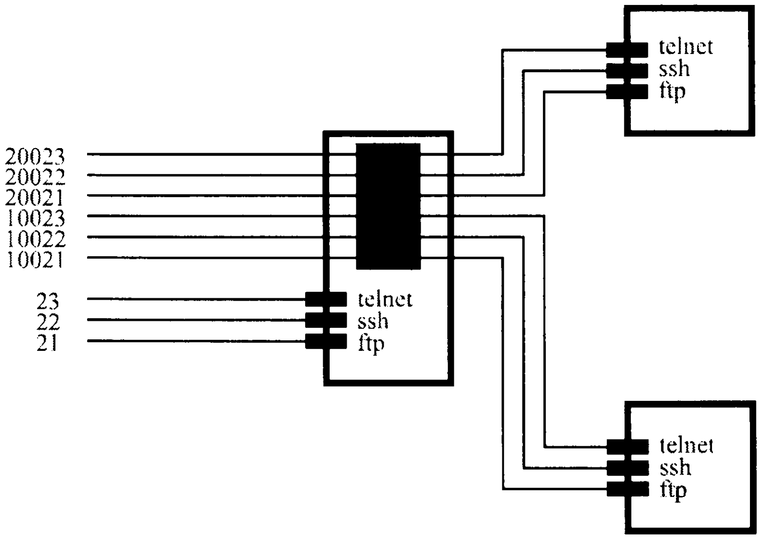


Figure 34: Accessing the lab clients

4.3 Restricting Access to Lab Machines

Protecting the lab clients from unauthorized access during exercises was one of the design goals of the Virtual Unix Lab, but it was also considered important to not hook up lab clients directly to the production network of the University of Applied Sciences of Regensburg (FH Regensburg) to prevent students having access to the lab machines abusing their admin privileges.

As a result, the lab clients were placed inside their own network, with only the lab machines and the Virtual Unix Lab server. The Virtual Unix Lab server acts as firewall, and access to the clients is realized by redirecting access to certain ports on the server to the client machines, as can be seen in figure 34.

The firewall is configured dynamically to allow access from a single client when an exercise starts (see figure 11), and access is disabled at the end of an exercise either when time runs out or the user indicates he's done, just before verification of the exercise results starts.

4.4 Software

The following software was used to create the Virtual Unix Lab:

Apache: Web server for the user interface and exercise text

Postgres: Database engine; MySQL didn't compile on NetBSD/sparc, and Postgres has worked very fine

IPfilter: Firewalling software by Darren Reed [Reed, 2004]

NetBSD: Operating system for the Virtual Unix Lab server and clients

Solaris: Fine operating system, for clients of the Virtual Unix Lab

PHP: Scripting engine for the web-based user interface

Perl: Scripting engine for result verification and some internals

Bourne shell: Scripting engine for result verification, client deployment and more internals

5 Current Status

Current status of the Virtual Unix Lab is that it works(!).

Two full-length exercises have been worked out and are available for students:

- Network Information System (NIS)
- Network File System (NFS)

The system was tested successfully in summer semester 2004 by 40 students during course "System Administration" at the department of computer science at the University of Applied Sciences (Fachhochschule) Regensburg.

An upgrade of the server hardware to a modern PC is pending.

6 Future Perspectives

Many ways are possible to improve the system on one end, but there are also a number of perspectives that the system in it's current incarnation can be used for. Here's an itemized list:

- **Funding** badly needed, for keeping the system running, and any of the following items
- Define **more exercises**:
  - Web- and Mail server, including spamfilters etc.
  - DNS, DHCP, LDAP, Samba
  - Database setup & tuning
  - System and network troubleshooting
  - Security post-morten analysis and prevention
- Add more options for **lab machines**:
  - Real hardware (PCs, Sun E15000)
  - Emulated (virtual) hardware: VMware, Xen, ...
  - More operating systems: Linux, Windows
- Internationalisation
- Implement a tutoring system
- Think about user modeling
- Do a lot of polishing and internal restructuring for the above items
- Funding! Very very badly needed!

References

[Feyrer, 2004] Feyrer, H. g4u - Harddisk Image Cloning for PCs [online]. (2004) [cited 2004-07-16]. Available from: `http://www.feyrer.de/g4u/`.

[Reed, 2004] Reed, D. IP Filter [online]. (2004) [cited 2004-09-14]. Available from: `http://coombs.anu.edu.au/~avalon/`.

[Schulmeister, 1997] Schulmeister, R. (1997). *Grundlagen hypermedialer Lernsysteme.* Oldenbourg Verlag, München, Germany.

[Spinellis, 2001] Spinellis, D. (2001). Notable design patterns for domain-specific languages. *The Journal of Systems and Software*, 56(1):91–99.

NetBSD/Desktop: Scalable Workstation Solutions

Jan Schaumann – Stevens Institute of Technology

Abstract

As a mature operating system with emphasis on code quality and standards compliance as well as an abundance of third-party applications readily available, NetBSD offers an easily deployable and user-friendly desktop solution. Managing large numbers of identical workstations can be facilitated through the use of a dedicated build server and an IPSec'd server-push strategy to update the clients; a strategy which has proven to be reliable, secure and scalable.

1 Introduction

While BSD in general and NetBSD in particular has had a reputation for being primarily a reliable and secure operating system for servers, it is (in the mainstream, at least) still not considered "user-friendly enough" for the desktop. Yet aside from the sheer number of available applications there are distinct advantages to choosing NetBSD, a complete open source operating system with emphasis on code quality and standards compliance, *especially* when it comes to managing a large number of identical desktop workstations.

This paper will elaborate on these advantages as well as present techniques and strategies to install, maintain and update large installations, accounting for the various needs of many hundreds of users and the implied complexities of thousands of third-party applications.

The infrastructure presented has been in production use in the Department of Computer Science at Stevens Institute of Technology for over 3 years, where it is used to manage several public laboratories as well as most of the faculty workstations. It consists of a dedicated build server and an IPSec'd server-push strategy to update the clients, providing scalable, secure and easy updates of the operating system as well as of all third-party application.

2 Why NetBSD?

In order to see why NetBSD would be a good solution for a regular desktop system, we should investigate what exactly is required of such a workstation and how it can be easily duplicated and deployed in a large environment. In order to address the first point, we need to look at the needs of the target users:

Not very surprisingly, given the obvious heritage of UNIX and BSD, Unix-like Operating Systems (OS) have long been the preferred choice in academic institutions in general and the field of Computer Science in particular. All users in such an environment (faculty, researchers, students) have high expectations of the robustness of the OS and its performance. At the same time, the widely different research interests require a large number of at times specialized applications to be made available to them.

2.1 What does "user-friendly" mean, anyway?

Many people expect a desktop OS to be "user-friendly", without clearly understanding the meaning of this phrase. As should become immediately clear from the varying needs and expectations of a multifaceted user-base, it can mean different things to different people, based on their background knowledge, experience and interest. It would be prudent to characterize any OS that allows these users to get their work done efficiently to be "user-friendly".

Oddly enough, the phrase "user-friendly" is often quoted in the context of OS installation, software installation and upgrades, security patches, and general maintenance. The procedures involved in these tasks are not considered *user*-friendly. This reveals the common misconception that a "desktop" or a "workstation" is a single, autonomous machine used (and maintained!) by a single user.

Actually, the tasks most often described as not being "user-friendly" are not – and *should* not be – performed by the *user*, but rather by the administrator. Maintaining the complex dependency-tree of installed applications, tweaking the OS to remain performant under duress and keeping dozens or hundreds of workstations in sync as well as available to all users are not tasks for a regular user and pose an entirely different set of requirements to the OS.

In the end, it boils down to this: The user of the desktop must get work done. The administrator must be able to maintain such desktop systems easily and efficiently. What we need is an "*admin*-friendly" OS with "*user*-friendly" applications!

2.2 Requirement: User-friendly

From the user's point of view, all interactions with the OS are characterized by the applications available on this platform. As long as the right tools for the job are available, the user may remain oblivious as to what exactly goes on "behind the curtain".

While commercial vendors release their software for only a limited number of unix-like OS, the vast majority of the available Open Source Software (OSS) – ranging from small command-line tools to entire desktop environments or office suites – is written for one or another version of Unix and can (often easily) be ported to another flavor.

In a multi-OS environment, it is beneficial to the user if she can use the same general desktop software and tools on one host as on another. Maintaining the same set of installed applications at the same version across even different OS helps avoid confusion when it comes to the user-interface or a user's configuration files.

Another important and often neglected aspect of user-friendliness is the combination of, on the one hand, hiding unnecessary complexities from one set of users while, at the same time, catering to the expertise of another set:

Since the target user-base includes people with very specific interests in the field of networking, operating system design, and system security, as well as students who learn to understand these concepts and develop software, the OS provided to them should allow them to use applications that are not traditionally part of a so-called desktop system. This includes access to the software development tools such as different compiler toolchains and various scripting languages, as well as the more typical desktop applications. In addition, it is often beneficial to provide the full sources for the OS in use to allow these users to look "under the hood" and understand how the OS, and the tools provided by it, work.

2.3 Requirement: Admin-friendly

As explained in section 2.1, the user-friendliness of an OS and its applications is necessary, but not *sufficient* to make a sensible choice for your environment. We also need to take a look at the qualities of an OS from the other point of view. In fact, given that the majority of the software in use is available for most unix-like OS, it seems that the admin-friendliness of an OS might be the more important aspect of all!

A suitable OS should be easy to install, maintain and update. A dependency on a specific hardware or software vendor is equally undesirable as too inconsistent a base system. On the one hand, we would want a *complete* OS with all base applications (ie the *userland*) from the same source tree; this allows for easy system maintenance and OS upgrades. At the same time it's important to ensure that all required software applications are available; finally, we need to anticipate future requirements.

With respect to hardware requirements, we must be able to support state-of-the-art equipment without having to resort to "bleeding edge" development snapshots. This, together with the need for the necessary security patches, requires an OS with a regular and reliable release engineering cycle. A mature and well thought-out third-party application framework or package management system is equally important since the vastly different needs of the many users imply a complex set of software dependencies. Finally, we must be able to run a few commercial applications that may not be available for all OS.

2.4 So... why NetBSD?

Looking at the requirements and the target user base as analyzed and described above, we see that NetBSD is a near perfect match for this environment.[1] NetBSD has a strong following among system administrators: it provides a *complete* OS, easily maintained in a single source tree and has an outstanding security track record. No need to try to track down and follow different development branches for each and every single userland binary, nor is it necessary to adhere to one commercial vendor's idea of release engineering and hope that other software vendors will adapt their packages accordingly.

Through the use of binary emulation it is possible to run a large number of commercial applications which may only have been released for another OS.[2]

[1] This is not to say that other OS choices would not be justifi able; however, the intimate familiarity with and knowledge of NetBSD by the administrators clearly make it the easier choice. It is also worth noting that over the years, experience has taught us that other, more popular choices would likely have caused us more headaches.

[2] At Stevens, the most important applications in this category are Sun's JDK's, MathWorks' MATLAB and Maplesoft's Maple; all industry standard applications which perform without performance penalty using NetBSD's Linux emulation layer.[3]

Finally, installation of third-party software and maintenance of all installed software is simplified by using the NetBSD Packages Collection[2] (aka *pkgsrc*), a source based software package management system. Through consistent use of pkgsrc and its cross-platform features, not only are we able to track over 1000 different applications and their interdependencies, but we are also able to provide the same number of applications in the same versions across different OS such as IRIX or Linux when this is necessary.

All currently used hardware – from regular single-processor desktop workstations to multi-processor, SATA-RAID enabled servers with support for fibre-channel or optical gigabit network devices – is fully supported by NetBSD. Its conservative release cycle allows us to rely on the hardware being fully functional and well-tested rather than having to make use of development features in unstable drivers. At the same time, the progressive nature of NetBSD in areas such as support for AMD's 64bit processors or IPv6 networking and its focus on standards compliance ensures that we will be able to add new hardware as it becomes available while at the same time continuing to provide our researchers with the most suitable reference platform available.

3 Infrastructure

At Stevens Institute of Technology, NetBSD is now used throughout the Department of Computer Science and the Department of Mathematics to maintain the Unix desktops and public laboratories in addition to a number of administrative machines.[3]

In order to maintain these machines, we have developed a set of administrative scripts that make up an infrastructure consisting of a dedicated build server and an IPSec'd server-push strategy to update the clients, providing scalable, secure and easy updates of the OS as well as of all third-party application. This setup, which has been in production use for over three years, reduces the administrative overhead involved in managing a large number of virtually identical hosts immensely and makes a trivial task integrating new machines into the framework.

All workstations have similar hardware, which allows us to run a common yet tailored kernel. The kernel contains the drivers for all available hardware throughout the system, thus allowing us to replace, for example, a network or a graphics card without having to recompile the kernel, while at the same time not suffering the performance penalty of a GENERIC kernel that includes support for a multitude of devices that are not required.

Since all workstations contain an identical software image, we can easily minimize downtime of a single system: if a machine goes down due to a hardware failure, we can quickly and easily replace the failed component, bring the machine back up and thus allow the user to continue to use the resources while we troubleshoot the failed device.

Statistical details about the infrastructure presented here can be found in Figure 1; the general setup is described in Section 3.1.

# of administrative scripts	7
total LOC of administrative scripts	388
# of users	approx. 2900
# of workstations	70
# of third-party packages not under pkgsrc	7
# of third-party packages under pkgsrc	1054
size of workstation image	9.3 GB

Figure 1: The system in numbers

3.1 Server Configuration

The build server, known by its hostname, *amstel*, used to maintain the workstation image, is a dual-processor i386 machine with 2 GB of RAM, powerful enough to build all required binary packages and the NetBSD kernel and userland in an acceptable timeframe. It has enough diskspace in a RAID 5 disk array to host the pkgsrc and regular src trees used, the workstation image (itself described in more detail in section 3.2) and provides enough temporary scratch space for the build process.

The machine hosts a total of three pkgsrc trees: one for the latest stable branch of pkgsrc (I), one for the HEAD of pkgsrc (II) and a third tree which incorporates parts of the HEAD with the latest stable branch as well as some local modifications (III). The first two are updated regularly from anonymous CVS sources and are used as a reference for the third one, which is itself used to build packages for the workstation. Trees (I) and (II) are also exported via NFS through a local gigabit network to the other non-NetBSD servers that use pkgsrc. Tree (III) is mounted via a null-mount from within two specific chroots.

The first chroot on the server is located in `/new` and represents the workstation image in use on all clients. The second is located in `/sandbox`, which represents the "playpen" for the workstation image. That is, the workstation image is duplicated in this directory and all software updates or additions are performed in this directory first. Packages are built and installed here, then turned into binary packages which can then be added in `/new` using the `pkg_*` tools. This setup makes it possible to update the userland using the standard NetBSD build.sh build mechanism (see "Using the build.sh Front End" in [5]) by simply defining `DESTDIR=/sandbox` in `/etc/mk.conf`.

[3] The Department of Computer Science also maintains a state-of-the-art clustered High Performance Computing Facility suitable for research in areas of computer science and engineering that may require substantial computational effort.[4] A second, similar cluster within the Department of Physics is currently being converted to NetBSD as well.

In general, /new is not modified directly and only binary packages are handled in this chroot, but every now and then it is necessary to perform one of the various package-related administrative activities in the /new chroot (after carefully testing it in /sandbox, of course), which is why the second null-mount is necessary.

A nightly cronjob checking the set of installed packages (in each of the server's base, the sandbox and the production workstation image) against a list of known vulnerabilities[4] ensures that we are alerted of any security issues as soon as a vulnerability is known.

3.2 Client Configuration

The disk image of the workstations is, as mentioned previously, stored in amstel:/new. It consists of a standard NetBSD/i386 installation together with a large number of third-party applications in order to cater to the different needs of the different users. Among those applications are most common window managers and desktop environments (including WindowMaker, KDE and GNOME), various browsers (including Netscape, Firefox, Mozilla and Opera), numerous software development tools (ranging from autoconf and automake to full-featured IDEs like eclipse), and programming languages (such as Sun's JDKs, Python, Perl, Ruby and Scheme), specialized commercial applications (including MathWorks' MATLAB and Maplesoft's Maple) and office suites and applications (including OpenOffice, gnumeric, gnucash, abiword and KOffice). In other words, it includes a complex and full desktop environment for a multitude of different users.

In order to allow us to swap parts easily among all the different workstations or even replace an entire machine with a minimum downtime for the user, all workstations are kept identical with respect to the software installed. Since some of the hardware differs (most notably graphics and network cards), there are a few differences in a limited number of configuration files, which therefore need to be kept exclusive to each host in addition to the obviously security-related files:

```
etc/X11/XF86Config
etc/master.passwd
etc/racoon/psk.txt
etc/rc.conf
etc/spwd.db
etc/ssh/ssh_host_dsa_key.admin
etc/ssh/ssh_host_dsa_key.admin.pub
etc/ssh/ssh_host_key.admin
etc/ssh/ssh_host_key.admin.pub
etc/ssh/ssh_host_rsa_key.admin
etc/ssh/ssh_host_rsa_key.admin.pub
etc/printcap
```

The list of files is rather self-explanatory, though it might be worth pointing out that all machines share the same default ssh-configuration and -keys. This is due to the fact that the hostname "lab.cs.stevens-tech.edu" is a round-robin of a number of these workstations, allowing users to connect to a generic name, yet spread the load across multiple machines.[5] Using a common key for all machines, however, necessitates running a second ssh dæmon on an alternate port for administrative purposes, which explains the distinct files in etc/ssh.

In order to keep these files separate from the rest of the workstation image, we created one directory for each client's etc directory in <hostname>-etc as well as one directory amstel:/new/etc which contains all other files usually found under /etc on a NetBSD host. Section 5.1 explains how the script used to update a workstation processes these directories.

4 Software installation

All software installed on the workstations that is not part of the NetBSD base system is installed, if possible, from the NetBSD Packages Collection (see [1] and [2] for detailed documentation). Any required piece of software that is not part of pkgsrc must be carefully investigated: if it is OSS or otherwise publicly available, we create the appropriate package and feed the changes back into the NetBSD pkgsrc CVS tree. If it is not suitable for inclusion in the Packages Collection, then the software is installed in amstel:/new/usr/local. As can be seen from Figure 1, we are fortunate enough to only have a very small number of applications that cannot be controlled through pkgsrc; these applications include mostly commercial, specially licensed software and a number of homegrown applications and scripts.

The pkgsrc trees in use are updated via CVS from the local anoncvs mirror on a regular basis. Due to the large number of applications installed and the resulting, rather complex dependency-tree we are taking a more conservative approach with the production use pkgsrc tree. This tree (III in Section 3.1) is carefully merged and updated on an as-needed basis only. That is, if a newer version of a piece of software is required, it is first updated to the latest stable branch of pkgsrc. If any security fixes have not yet been pulled up to the stable tree or a newer version is required, the tree is updated to the HEAD or patches backported. A small number of packages have some local modifications applied as well.

The build process itself is done in the

[4] using pkgsrc's *audit-packages*

[5] Of course this would also be possible with distinct ssh-keys, but we have found that especially among novice users and windows clients, multi-homed hosts with different ssh-keys often cause some confusion and generate warnings about the keys not matching the hostname. Rather than encouraging users to ignore such warnings, we decided to distribute a single key to all machines.

amstel:/sandbox chroot to ensure that the package (and all its dependencies and/or other packages that had to be updated as a result) can successfully be built, installed, cleanly deinstalled, reinstalled and tested. Binary packages are then created from within that chroot and the production use image is updated accordingly.

5 Update procedure

Updating all the workstations from the build server is done by means of an IPSec'd server-push performing several rsync-passes on the remote side (see Section 5.1 for details). The administrative scripts used to initiate the push allow for synchronization of all workstations or certain subsets based on department or laboratory in parallel or pushing just a single machines.

If the changes require additional testing, individual machines can be synchronized with the sandbox to allow for the installation or upgrading of large parts of the workstation image without risking breaking all clients in production use. In these cases a note explaining the currently tested update is placed into a special file (called "dont"; see Appendix A.2), the existence of which prevents a full push to all clients from taking place – a security precaution that has often proved useful.

Similarly, the setup also allows for synchronizing only parts of the filesystem: each pass can be performed individually. If, for example, a general system upgrade is still being tested on some individual workstations but an important update of one of the applications in /usr/local is pending, then only the update of that part of the filesystem can be pushed out to all clients, retaining the full update of the base system until it has been sufficiently tested.

If parts of the filesystem need to be (temporarily) excluded from being synchronized, they can be specified in specific configuration files as regular rsync(1) exclude patterns.

A push-strategy rather than a pull-strategy was chosen to prevent accidental deployment of not fully tested updates. Since a push is proactive, it forces the administrators to think about what changes are ready for deployment and carefully evaluate when a complete push of all workstations should take place. A client-pull mechanism might easily be forgotten and cause serious downtime if any software updates were accidentally left incomplete (either as a result of human negligence or of a software failure during the upgrade process). Also, as can be seen from the above, a server-push allows for much finer control, as a client-pull would have to be automated and thus complete.

5.1 The push script

The shell script push.sh (the full script can be found in Appendix A.1) is used to update a single workstation after changes have been made to the workstation image. It uses rsync(1) to update the remote host (see section 5.2 for security considerations). To summarize, push.sh performs the following steps:

1. **Enter new information in /new/etc/updates.** All changes to the workstation image are briefly noted in the file /usr/local/stevens/UPDATES. This file helps us keep track of what changes were made and, more importantly, why they were made. The push.sh script timestamps this file and places a copy into /new/etc/updates, so that users can always review the latest changes and the administrator can easily determine if any given workstation is currently up-to-date.

   This file is also (manually) emailed in a PGP-signed message to a local mailing list. This, too, allows users to keep up to date with changes on the system while at the same time providing some log of the changes.

2. **Set up exclusions.** It may be desirable or necessary to exclude certain files on a specific host from being synchronized with the rest. Any such files can be entered in a separate file which is parsed by push.sh in order to create a list of exclusions.

   A reason for such an exclusion might be a machine that requires a different kernel from all the others, but that is otherwise identical to the normal workstations.

   Note that these exclusions are set up on a per-host basis. If general site-wide exclusions need to be set up, they can be entered in a different file using standard rsync(1) exclude patterns.

3. **Run any remote commands if necessary.** Depending on the changes pushed out, it may be necessary to first run a specific command on the remote host, for example to stop a service or to unmount a partition. If push.sh finds the file beforesync in the local directory, it is copied to the remote host where it is executed as a shell script.

4. **Do passes as desired.** The actual push process is divided into several *passes* which may be specified individually. If no pass is explicitly specified, all default passes are performed consecutively. This allows for fine-grained control over which parts of the filesystem are updated and is necessary to set up the appropriate exclusions.

   Each pass inspects a special file containing these exclusions as a list of pathnames (directories or files) which should be ignored when running rsync(1).

5. **Do absolute copies if necessary.** Similar to the exclusion set up in step two, this step allows special files to be

copied to a specific absolute destination on the remote host.

6. **Run any remote commands if necessary.** Depending on the changes pushed out, it may be necessary to run a specific command on the remote host after all changes have been pushed out; for example, to restart a service after the configuration file was changed.

   If `push.sh` finds the file `aftersync` in the local directory, it is copied to the remote host where it is executed as a shell script.

As mentioned above, this script only updates a single host. A number of other scripts are used to update all available workstations or a subset thereof (see Appendix A.2). In combination with these scripts, `push.sh` provides a sufficient amount of flexibility for most situations. For example, if a major package build is still in progress (for example, KDE needs to be updated) but a new release of one of the commercial applications installed is available, it would be trivial to push out the `/usr/local` hierarchy to all clients, while retaining `/usr/pkg` for another time to allow for more testing.

On the other hand, it may be desirable to simply run a specific command on all hosts or to update a single file. The script would be able to achieve this, but it would involve some performance overhead. Therefore we have added a few simple scripts to perform just these tasks (Appendices A.3, A.4).

5.2 Security considerations

Since the entire filesystem for each client is transferred over the network, there are a number of security aspects to be considered. First, we need to ensure that only machines that are known are allowed to synchronize with the server. Second, we need to make sure that sensitive files are not transferred in the clear.

As explained in Section 5, we chose a server-push strategy, which partly addresses the first problem: no client can initiate the update, so that it's not possible for a Trojan to connect to the network, steal a known IP address and request an update. However, it would still be possible for an adversary to pose as a normal workstation and wait for the update to be pushed out. Fortunately, that, too, is not possible, as the only two ways we allow the `rsync(1)` processes to perform is either tunneled through `ssh(1)` or by use of `rsh(1)` over mandatory IPsec.

Both of these approaches require authentication, either by use of the private ssh-key for the dæmon listening on an alternate port or by use of IKE, IPSec'd dynamic encryption key exchange protocol. In our setup, we use `racoon(8)` with a pre-shared secret key stored in `/etc/racoon/psk.txt`. The permissions on this file – just like on the private ssh keys in `/etc/ssh/` – do not allow regular users to read these, so it should be impossible for the adversary to pose as the real client when a push is initiated by the server. Similarly, we have a specific ssh-key set in `/root/.ssh/authorized_keys` (the private key for which is only on the non-publicly accessible build server) that is used for synchronizing the machines if ssh is used as the remote shell, allowing access by this key only from the build server.

type	remote shell	time
minor updateem +	rsh + ipsec	332.27s → 5.5m
minor update	ssh	459.48s → 7.6m
major update*	rsh + ipsec	474.42s → 7.9m
major update	ssh	494.91s → 8.25m
full update#	rsh + ipsec	1307.71s → 21.8m
full update	ssh	1426.22s → 23.8m

+ update of a few files or a small package

* update of at least three large packages (such as mozilla, KDE etc.)

# update of the entire userland and several large packages

Figure 2: Scalability of a single push

It might be possible to run nearly the entire push unencrypted using regular `rsh(1)` for `rsync`, but clearly there are a few files that must not be transmitted in the clear (`/etc/master.passwd` and the files from the previous paragraph, for instance). However, if this approach were pursued, then the push script would need to switch the mechanism used to log in the remote machine based on the files to be transferred. This does not prove to be scalable, especially since packages are added that might also require encryption during file transfer (for example the `sudo(8)` package). In addition, we would loose the authentication mechanism provided by IPsec, which is why in the end we decided to make use of IPsec for all `rsh`/`rlogin` processes mandatory. Once we enabled IPsec, we quickly realized that another beneficial side-effect was that it allowed us to let `syslogd(8)` log all events to *amstel* in a secure fashion as well.

5.3 Scalability

When expanding the network of clients under control of this system, it is important to consider how well the system scales, how long each process takes, and what the penalties are of choosing one approach over another. The main factors with respect to scalability are:

- the time it takes for a single complete update
- the time it takes for all clients to be updated

The time it takes for a complete update is influenced by the number of changes since the last update, as well as which push-strategy (ssh or rsync over IPsec) is chosen. In production use there may often be miniscule changes (i.e. changes of only a few files) just as there may be rather large updates (such as a rebuild of a significant portion of the binary packages) that need to be pushed out. Figure 2 shows the time in seconds for a few example changes including the worst-case scenario: an update of the entire base system as well as a number of large third-party updates.

Another factor in the calculation of these numbers is the amount of RAM available on the client. The majority of the workstations have 512 MB RAM, but a number of them still have only 256, while a few have as much as 1 GB RAM. The numbers in Figure 2 and Figure 3 are based on the average of repeated pushes of machines with 256 MB RAM, with the highest and lowest results being dropped.

The time it takes for all clients to be updated obviously depends on the time for each individual client, but also on how many hosts can be pushed out in parallel without saturating the network connection or overloading the server. Figure 3 shows the results of updating several sets of machines.

6 New workstation installation

Adding a new workstation into this setup is trivial. After the hostname of the new machine and its intended physical location (which determines the subnet the new host will be on) has been specified, we generate a new configuration for this client by using the script `gen-conf.sh` (see Appendix A.5). To complete the setup on the server side, all we need to do is add the new hostname to the appropriate collection of machines that allow us to push subsets based on location.

The actual installation process is initiated by booting off a NetBSD install floppy or CD-ROM. Instead of performing the regular installation that NetBSD's `sysinst` tool would usually initiate, we abort the installation process and `disklabel(8)` the hard drive by hand, configure the network and mount the workstation image via NFS from the build server. Finally, we run the script `getit.sh` (see Appendix A.6), which completes the installation.

6.1 Security considerations

While the installation procedure as described above is simple enough, we must to again pay attention to the security aspects of performing a network install. The same files that necessitate encryption during the update process (as explained in Section 5.2) must not be transmitted in the clear during the installation process and are therefore omitted at this point.

type	remote shell	time
minor update	rsh + ipsec	36m
minor update	ssh	49m
major update	rsh + ipsec	47m
major update	ssh	51m
full update	rsh + ipsec	149m
full update	ssh	155m

Figure 3: Scalability of a full push

Since we do not intend to transmit these sensitive files, they are not stored under the `amstel:/new` directory. This in turn allows us to make the `/new` directory accessible via NFS – clearly, this would be impossible if these files resided therein: the adversary could presumably connect a rogue machine to the network and mount the directory via NFS by spoofing one of the IP addresses which are granted access. Without any sensitive information under this hierarchy, the adversary can only gain access to the same files she could otherwise retrieve from any public workstation.

On the other hand, the fact that we must not transfer the pre-shared secrets and other important files in the clear leaves us with a bit of a conundrum: the installation cannot be completely unattended, as both the sshd keys and the pre-shared secret for IKE must not go across the network unencrypted and the installation process cannot be encrypted, as the pre-shared secrets are not yet on the to-be-installed client. We will look at possible solutions to this problem in Section 7.2. For the time being, this forces us to add manually one of the pre-shared secrets after the installation process and run a partial push (to synchronize the other sensitive files) before deployment of the new host.

6.2 Scalability

As we have seen above, installing a single host does not involve much effort, but how long does it actually take, and how well does this system scale if a large number of hosts need to be installed? The process of generating a new configuration for a new host is obviously simple and will take negligible time, as only a single short shell script needs to be executed. For a large number of new hosts, this can be wrapped into a for-loop and still not take much longer than it takes to actually enter the data the script requires. Should it be necessary to integrate a significant number of new hosts into this setup, it might be wise to collect the required information beforehand and tweak the script to run non-interactive.

The most time is spent actually installing the software on the new machine. Due to the large size of the workstation image based on the huge number of packages required, it quickly becomes clear that at this time the bottleneck lies in the network installation procedure. Most installations are currently done over the regular 10/100 campus network, as gigabit networking is not currently available in all parts of the campus. On average, the installation process from start to finish (including the creation of a new configuration on the build host, booting the new host from installation media, `disklabel(8)`ing the hard drive, running the install script over NFS) takes approximately 67 minutes. In Section 7.3 we will consider ways to improve this process.

Installing multiple hosts at the same time faces the same limitations as running multiple pushes at the same time: the more clients install in parallel, the higher the load on the build server and the higher the saturation of the network connection. Fortunately, at the moment there rarely is a need to perform more than just a handful of installations at once.

7 What's next?

While the current configuration allows for convenient and easy installation and maintenance of the workstations, like any other piece of software or administrative setup, there is, of course, always room for improvement. In this section, we will try to look at solutions for the problems mentioned in previous sections, as well as consider new features that might be desirable.

7.1 Improving the general setup

Given the large number of applications under pkgsrc control, it is crucial that the package management system works flawlessly when updating packages. Until the beginning of the year, the ever-changing nature of pkgsrc was somewhat of a problem: trying to follow a fast-moving target, our pkgsrc trees would have to be updated frequently to keep up with the infrastructure changes under `pkgsrc/mk`, but on the other hand it was risky to update the entire tree: it would then always contain the latest release of each version of software, so that building one package may pull in an update of another, already-installed package even though that is not strictly required. Rebuilding large parts of the installed software could occasionally lead to a broken dependency, leaving the workstation image in a non-stable state or – in the worst case – without a particular required package.

Fortunately, the NetBSD Packages team introduced the concept of stable pkgsrc branches, which made a lot of system administrators very happy by allowing for much easier tracking of the installed packages and keeping up with security issues. While this has improved the situation immensely, from time to time we still encounter instances of package dependencies[6] that are not strictly necessary, which is why we must maintain our own pkgsrc tree alongside the stable branch, merging changes and updates by hand. In addition, it is occasionally necessary to install conflicting packages on the same system.[7] This currently requires more modifications to the packages in question which must be remembered when updating the pkgsrc trees.

The *pkgviews* framework, introduced to pkgsrc early this year, promises to solve a number of these problems (refer to [6] for details), but unfortunately it will require starting with a clean slate, so to speak: the need to uninstall every single package and rebuild them using pkgviews is a project that we originally intended to approach during the summer of 2004, but for which, in the end did not have time. The conversion to pkgviews therefore remains on our list of improvements of the system.

As mentioned previously, all software is compiled in a sandbox and binary packages built, which are then used to maintain the actual workstation image. This is done only on an as-needed basis at the moment, allowing for the potential risk of encountering a failure at a crucial time. To avoid this scenario, it might be desirable to create a system that periodically builds binary packages from scratch, so that newer versions of binary packages compiled for our systems are *immediately* available. It might be feasible to perform complete bulk-builds to anticipate new software requests, or we might consider only building what is currently installed. Frameworks for both approaches do already exist within pkgsrc and we should be able to implement such a solution with relative ease.

While we use a problem report system at Stevens Institute of Technology to track and analyze software requests and complications, this system often relies on the user to provide an accurate analysis of the situation or detailed feedback. When changes to the software are made, they are not always annotated in great detail in the PR system, so that several weeks or months later it is not always clear why exactly a change was made, or what change was made to fix a particular problem. Maintaining a detailed changelog of the entire workstation image, possibly through the use of a revision control system, such as *CVS*, would prove useful.

Similarly, the scripts that maintain the system presented in this paper are currently not under any revision control either. Importing these files into a CVS repository would similarly allow us to better track the files as they are changed as well as quickly and easily determine why a certain change was made.

Finally, our system did not provide detailed documentation on how workstation installations or updates are done. Fortunately, this paper easily solves this problem, but it will require us to update it regularly to keep the documentation in sync with reality.

7.2 Improving Security

It might be worth considering identifying clients not only by hostname or IP address, but to also require their MAC address to match to introduce one more barrier for a MITM based attack. However, this approach introduces a significant administrative overhead while not gaining much more security: on the one hand, it is relatively easy for the adversary to spoof a

[6] *buildlink bumps* have proven to be particularly dangerous

[7] An example: we currently have both *print/teTeX1* and *print/teTeX* installed to allow our users to continue to use their older TeX files.

MAC address[8] just like she could spoof the IP address; on the other hand, it would require the careful logging of each machine's ethernet interface change. Since we routinely swap hardware among different clients, such a MAC-address-to-hostname mapping could easily become outdated.

Of more urgency would be the development of an encrypted installation mechanism, which would allow us to move to an entirely unattended install process. Considering the chicken-and-the-egg problem eluded to in Section 6.1, in this context one possible solution might be to use asymmetric cryptography rather than the current symmetric approach, which necessitates the existence of a pre-shared secret on both sides. An alternative solution might involve a custom install CD containing a special "install-key" and to perform encryption and decryption for the few sensitive files using PGP or OpenSSL. Finally, we might decide to pay the price of convenience[9] and perform installations through a private network only.

7.3 Improving Scalability

As mentioned above, the Stevens campus network does not currently provide gigabit networking. However, our build server is equipped with a gigabit network card and connected to a small private gigabit switch through which it performs regular backups of various other servers. In order to allow for a faster installation, we could perform this process over the private gigabit network, taking advantage of a more secure and much faster connection to the server.

Another way to cut down on the time taken for a full installation would be to install only what is necessary to deploy the system and then update the rest on the next push. For example, the `/usr/src` hierarchy does not need to be immediately available and could be excluded during installation. On the other hand, this increases the load during the first push, so we really do not gain very much.

The numbers in Figure 3 represent a push of all machines while synchronizing approximately 20 hosts at the same time. This number was chosen without much empirical evidence of a performance increase, but was rather based on the increase of processing speed and memory in the server compared after an upgrade. It might be desirable to perform more accurate benchmark tests and determine the appropriate number of parallel pushes to optimize CPU-, memory- and network utilization.

8 Conclusion

In this paper I hope to have debunked the common misconception that NetBSD is "not ready for the desktop" and stressed the importance of an "admin-friendly" operating system for production use in an environment that at the same time demands a sophisticated environment for computer specialists as well as a more traditional, user-friendly setup for novices. The desktop workstation control system presented allows, as we have seen, for simple yet scalable maintenance of a large number of identical workstations from a central build server.

The update and installation processes provide enough flexibility to allow for multiple configuration differences among hosts and subnets while at the same time ensuring that security relevant files are not compromised. As mentioned above, the infrastructure has been in production use for several years and has in addition been used as a basis of a very similar system for the maintenance of one of our High Performance Computing Facilities (also entirely based on NetBSD).

Like all software systems, there is room for improvement, and I have elaborated on a few shortcomings and considered ways to improve the system. The most important scripts of this setup can be found in the Appendix and have been placed in the public domain – any corrections, suggestions or questions are most welcome and can be directed at the author.

9 Author Information

Jan Schaumann is a System Administrator in the Department of Computer Science at Stevens Institute of Technology in Hoboken, NJ, USA, where he manages a large, nearly homogenous NetBSD environment, ports and maintains NetBSD pkgsrc tools and packages on non-NetBSD platforms such as IRIX and Linux, and teaches classes in UNIX programming and System Administration.

Jan holds Bachelor's and Master's degrees in Computer Science from Stevens Institute of Technology; he joined the NetBSD Project as a developer in January of 2002 and enjoys living with his wife Paula in New York City. The non-computer related activities he enjoys usually involve a board, often in combination with some form of H_2O. Jan can be reached at *jschauma@cs.stevens.edu*.

[8] It is noting that since many of our workstations are in public laboratories, it would even be possible for the adversary to simply steal the actual ethernet device of one of the machines.

[9] I.e. we no longer would be able to install a new machine from any department.

A Code Listings

A.1 The push.sh script used to update a single host

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original authors are Thor Lancelot Simon and Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

#set -x
#DONTDOIT=echo

LOCALFILES=/usr/local/stevens
RSYNC_RSH=rsh
PASSES=all
ROOT=/new
RSYNC_OPTS="-aSH"

usage()
{
    echo "Usage: $0: [-p <pass>] [-r <root>] [-s]"
    exit 1;
}

args=`getopt p:r:s $*`
if [ $? -ne 0 ]; then
    usage
fi
set -- $args

while [ $# -gt 0 ]; do
    case "$1" in
        -p)
            PASSES=$2; shift
            ;;
        -r)
            ROOT=$2; shift
            ;;
        -s)
            RSYNC_RSH=${LOCALFILES}/rsync-ssh
            ;;
        --)
```

```
            shift; break
            ;;
    esac
    shift
done

export RSYNC_RSH

for i in $*; do

    echo "NEW HOST: $i"

    # show when we last updated
    date > ${ROOT}/etc/updates
    echo >> ${ROOT}/etc/updates
    cat ${LOCALFILES}/UPDATES >> /new/etc/updates

    # set up exclusions.  Rather ghastly.
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\") &&  \
    (\$3 == \"/\")) print \$4}" < special.files > /tmp/x0.$i.$$
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\") &&  \
    (\$3 == \"/etc\")) print \$4}" < special.files > /tmp/x1.$i.$$
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\")  &&  \
    (\$3 == \"/usr/pkg\")) print \$4}" < special.files > /tmp/x2.$i.$$
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\") &&  \
    (\$3 == \"/var/db/pkg\")) print \$4}" < special.files > /tmp/x3.$i.$$

    # start the bombardment
    if [ -e ./beforesync ]; then
        $DONTDOIT rsync ./beforesync $i:/tmp
        $DONTDOIT $RSYNC_RSH $i chmod u+x /tmp/beforesync
        $DONTDOIT $RSYNC_RSH $i /tmp/beforesync
    fi

    # note that these "passes" correspond to the exclusion statements above
    if [ "$PASSES" = "/" -o "$PASSES" = "all" ]; then
        echo "pass 0: /"
         $DONTDOIT rsync ${RSYNC_OPTS} --delete --one-file-system      \
            --exclude-from=${LOCALFILES}/exclude-from          \
            --exclude-from=/tmp/x0.$i.$$ ${ROOT}/ $i:/
    fi

    if [ "$PASSES" = "/etc" -o "$PASSES" = "all" ]; then
        echo "pass 1: /etc"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete                          \
            --exclude-from=${LOCALFILES}/exclude-from-etc     \
            --exclude-from=/tmp/x1.$i.$$ ${ROOT}/etc/ $i:/etc
        $DONTDOIT rsync ${RSYNC_OPTS} ${LOCALFILES}/client-etcs/common-files/ $i:/etc
        $DONTDOIT rsync ${RSYNC_OPTS} ${LOCALFILES}/client-etcs/$i-etc/ $i:/etc
    fi

    if [ "$PASSES" = "/usr/pkg" -o "$PASSES" = "all" ]; then
        echo "pass 2: /usr/pkg"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete --exclude-from=/tmp/x2.$i.$$     \
            ${ROOT}/usr/pkg/ $i:/usr/pkg

        echo "pass 3: /var/db/pkg"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete --exclude-from=/tmp/x3.$i.$$     \
            ${ROOT}/var/db/pkg/ $i:/var/db/pkg
    fi


    if [ "$PASSES" = "/usr/local" ]; then
        echo "pass: /usr/local"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete ${ROOT}/usr/local/ $i:/usr/local
    fi

    if [ "$PASSES" = "/usr/src" -o "$PASSES" = "all" ]; then

      if [ ! `grep $i small` ]; then
          echo "pass 3.5: /usr/src"
          $DONTDOIT rsync --exclude-from=${LOCALFILES}/exclude-from-src \
              --delete-excluded ${RSYNC_OPTS} --delete ${ROOT}/usr/src/ $i:/usr/src
      fi
    fi

    if [ "$PASSES" = "absolute" -o "$PASSES" = "all" ]; then
        # "pass 4": special "absolute" copies (shudder in fear!)
        echo "pass 4: absolute copies (if any)"
        $DONTDOIT eval `awk "/^ABSCOPY/ {if (\\$2 == \\"$i\\")        \
            print \\"rsync ${RSYNC_OPTS} \\" \\$4 \\" \\" \\$2 \\":\\"  \
            \\$3 \\";\\"}" < special.files`
    fi

    if [ -f specials/aftersync.$i ]; then
        $DONTDOIT rsync specials/aftersync.$i $i:/tmp
        $DONTDOIT $RSYNC_RSH $i chmod u+x /tmp/aftersync.$i
        $DONTDOIT $RSYNC_RSH $i /tmp/aftersync.$i
    fi

    if [ -e ./aftersync ]; then
        $DONTDOIT rsync ./aftersync $i:/tmp
        $DONTDOIT $RSYNC_RSH $i chmod u+x /tmp/aftersync
        $DONTDOIT $RSYNC_RSH $i /tmp/aftersync
    fi

    rm /tmp/x0.$i.$$
    rm /tmp/x1.$i.$$
    rm /tmp/x2.$i.$$
    rm /tmp/x3.$i.$$

done
```

A.2 The push.batch.sh script used to update all hosts in parallel

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

BASE=/usr/local/stevens

if [ -f ${BASE}/dont ]; then
    echo "You probably don't want to do this right now..."
    cat ${BASE}/dont
    exit 1
fi

MACHINES=`egrep -v ^# ${BASE}/machines.${1:-all}`

if [ $# -gt 0 ]; then
    unset MACHINES
    for SET in $@; do
        ADDMACH=`egrep -v ^# ${BASE}/machines.${SET}`
        MACHINES="$ADDMACH $MACHINES"
    done
fi

echo $MACHINES | xargs -n 3 ${BASE}/bgsync.sh
```

A.3 The pushfile.sh script used to update a single file

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED
# WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

#DONTDOIT=echo
file=${FILE:?"No file?"}

if [ -z ${RSYNC_RSH} ]; then
    RSYNC_RSH=rsh
fi
#export RSYNC_RSH=/usr/local/stevens/rsync-ssh
export RSYNC_RSH

for i in $*; do

    echo "NEW HOST: $i"

    $DONTDOIT rsync /new/$file $i:/$file
done
```

A.4 The runcmd.sh script used to run a command on the remote host

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED
# WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

DONTDOIT=echo
cmd=${CMD:-"uname -a"}

if [ -z ${RSYNC_RSH} ]; then
     RSYNC_RSH=rsh
fi

export RSYNC_RSH

for i in $*; do

    echo "NEW HOST: $i"
    $DONTDOIT $RSYNC_RSH $i $cmd

done
```

A.5 The gen-conf.sh script used to add a new host

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original authors are Thor Lancelot Simon and Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


ENVBASE=/new

if [ $1 ]; then
    NEWHOST=$1
    echo using host $1
    shift
else
    read -p "Name of new host? " NEWHOST
fi

LOCALFILES=/usr/local/stevens/
NEWETC=${LOCALFILES}/client-etcs/${NEWHOST}-etc

if [ -e ${NEWETC} ]; then
  echo "I'm sorry, ${NEWETC} already exists.  You can't name a host ${NEWHOST}." >/dev/st
  exit 1
fi

if [ $1 ]; then
    FQDN=$1
    echo using FQDN $1
    shift
else
    read -p "Fully-qualified domain name of new host? " FQDN
fi

read -p "Enter name of primary ethernet interface (e.g. fxp0): " ETHERTYPE
read -p "Enter IP address of primary ethernet interface: " INETADDR

SUBNET=`echo ${INETADDR} | awk 'BEGIN {FS="."; OFS="."} {print $3}'`
DEFROUTER=`echo ${INETADDR} | awk 'BEGIN {FS="."; OFS="."} {print $1,$2,$3,"1"}'`

mkdir -p ${NEWETC}/ssh ${NEWETC}/racoon ${NEWETC}/ssh
cp ${LOCALFILES}/rc.conf.default ${NEWETC}
```

```
echo hostname\=${FQDN} >> ${NEWETC}/rc.conf
echo ifconfig_${ETHERTYPE}\=\"inet ${INETADDR} netmask 255.255.255.0\" >> ${NEWETC}/rc.co
echo defaultroute\=${DEFROUTER} >> ${NEWETC}/rc.conf

ln -s printcap.${SUBNET} ${NEWETC}/printcap

echo "Now you must select an X-Windows configuration for the new machine."

while [ ! -f ${NEWETC}/X11/${WHICHX} ]; do
  echo "These are the configurations available.  Please select one."
  (cd ${NEWETC}/X11; ls XF86*)
  read -p "Which configuration file? " WHICHX
done

ln -s ${WHICHX} ${NEWETC}/X11/XF86Config


echo Now you must set a root password for the new machine.

# get default, just to make sure
cp ${LOCALFILES}/master.passwd.default ${ENVBASE}/etc/master.passwd

chroot ${ENVBASE} /usr/bin/passwd -l root

# put in place
mv ${ENVBASE}/etc/master.passwd ${ENVBASE}/etc/spwd.db ${NEWETC}/

# overwrite, just to make sure
cp ${LOCALFILES}/master.passwd.default ${ENVBASE}/etc/master.passwd


RACKEY=‘hexdump -n 16 -e \"%08x%08x%08x%08x\\\n\" /dev/urandom‘;
echo -e "${INETADDR}\t${RACKEY}" >> /etc/racoon/psk.txt
echo -e "155.246.89.68\t${RACKEY}" > ${NEWETC}/racoon/psk.txt
chmod 0600 ${NEWETC}/racoon/psk.txt


echo Making SSH host keys for ${NEWHOST}...

umask 022
/usr/bin/ssh-keygen -t rsa1 -b 1024 -f ${NEWETC}/ssh/ssh_host_key -N ''
/usr/bin/ssh-keygen -t dsa -f ${NEWETC}/ssh/ssh_host_dsa_key -N ''
/usr/bin/ssh-keygen -t rsa -f ${NEWETC}/ssh/ssh_host_rsa_key -N ''
/usr/bin/ssh-keygen -t rsa1 -b 1024 -f ${NEWETC}/ssh/ssh_host_key.admin -N ''
/usr/bin/ssh-keygen -t dsa -f ${NEWETC}/ssh/ssh_host_dsa_key.admin -N ''
/usr/bin/ssh-keygen -t rsa -f ${NEWETC}/ssh/ssh_host_rsa_key.admin -N ''


echo I have set up ${NEWETC} for you, but you will probably need to
echo check the following files: ${NEWETC}/rc.conf,
echo ${NEWETC}/inetd.conf, $NETWETC/ipsec.conf, ${NEWETC}/racoon/psk.txt
echo
echo Remember to restart racoon and ipsec as well.
```

A.6 The getit.sh script used to install a new host

```
#!/bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

#DONT="echo"

for i in `cat /mnt2/getit.excludes`; do
        export EXCLUDES="-s ,^./$i,,p ${EXCLUDES}"
done

PAX="pax ${EXCLUDES} -rwvp e "
DEVICE="wd0a"

echo Stage 1: Creating filesystem...
echo ------------------------------
$DONT /sbin/newfs /dev/r${DEVICE}

echo Stage 2: Mounting filesystem...
echo ------------------------------
$DONT /sbin/mount -o async /dev/${DEVICE} /mnt

$DONT mkdir -p /mnt/tmp
$DONT chmod 1777 /mnt/tmp
$DONT export TMPDIR=/mnt/tmp

echo Stage 3: Starting the bombardment...
echo -----------------------------------

cd /mnt2 && $DONT $PAX . /mnt/

cd /mnt2/usr/mdec
$DONT ./installboot -v biosboot.sym /dev/$DEVICE

echo "Remember to:"
echo "    - manually install etc/racoon/psk.txt"
echo "    - reboot, ifconfig and push"
```

References

[1] *Documentation on the NetBSD Package System* Hubert Feyrer, Alistair Crooks et al, pkgsrc/Packages.txt, October 2004

[2] *The NetBSD Packages Collection*
`http://www.NetBSD.org/Documentation/software/packages.html`

[3] *Binary emulation in NetBSD*
`http://www.NetBSD.org/Documentation/compat.html`

[4] *High Performance Computing at Stevens Institute of Technology*
`http://www.cs.stevens.edu/~jschauma/hpcf/`

[5] *The NetBSD Operating System* Federico Lupi et al, October 2004
`http://www.NetBSD.org/guide/en/`

[6] *Package Views - a more flexible infrastructure for third-party software*, Alistair Crooks, November 2002
`http://www.NetBSD.org/Documentation/software/pkgviews.pdf`

The Challenges of Dynamic Network Interfaces

Brooks Davis
The FreeBSD Project
Seattle, WA
brooks@{aero,FreeBSD}.org

Abstract

On early BSD systems, network interfaces were static objects created at kernel compile time. Today the situation has changed dramatically. PC Card, USB, and other removable buses allow hardware interfaces to arrive and depart at run time. Pseudo-device cloning also allows pseudo-devices to be created dynamically. Additionally, in FreeBSD and Dragonfly, interfaces can be renamed by the administrator. With these changes, interfaces are now dynamic objects which may appear, change, or disappear at any time. This dynamism invalidates a number of assumptions that have been made in the kernel, in external programs, and even in standards such as SNMP. This paper explores the history of the transition of network interfaces from static to dynamic. Issues raised by these changes are discussed and possible solutions suggested.

1 Introduction

In the early days of UNIX, network interfaces were static. The drivers were compiled into the kernel along with their hardware addresses. The set of devices on each machine changed only when the administrator modified the kernel. Those days are long gone. Today devices, hardware and virtual, may come and go at any time. This dynamism creates a number of problems for both kernel and application developers.

This paper discusses how the current dynamism came about in FreeBSD, documents the problems it causes, and proposes solutions to some of those problems. The following section details the history of dynamic devices from the era of purely static devices to the modern age of near complete dynamism. Following this history, the problems caused by this dynamism are discussed in detail. Then solutions to some of these problems are proposed and analyzed, and advice to implementers of userland applications is given. Finally, the issues are summarized and future work is discussed.

2 History

In early versions of UNIX, the exact set of devices on the system had to be compiled in to the kernel. If the administrator attempted to use a device which was compiled in, but not installed, a panic or hang was nearly certain. This system was easy to program and efficient to execute. Unfortunately, it was not convenient to administer as the set of available interfaces grew.

Published in the Proceedings of EuroBSDCon 2004

4.1BSD, released June, 1981, included a solution to this problem called autoconfiguration [McKusick2]. Autoconfiguration is a process by which devices are detected at boot time [McKusick1]. Under autoconfiguration, each system bus is probed for devices and those devices that have drivers in the system are enabled. The process of identifying devices is called probing and devices which are found in probing are attached. The procedure used to probe devices varies by bus. On some buses, such as non-PnP ISA, compiled-in addresses are probed and if they respond as expected the device is assumed to be there. With more advanced buses such as PCI or SCSI, devices are self-identifying. PCI devices are identified by an ID number composed of a vendor portion and a vendor allocated product portion. SCSI devices are identified by a device class and a free form string. With autoconfiguration, all devices to be used still had to be compiled in to the kernel, but a super set could be used enabling one kernel to work with multiple system configurations.

In FreeBSD 2.0, the LKM (Loadable Kernel Modules) system modeled after the facility in SunOS 4.1.3 was implemented by Terry Lambert [man4-2]. The LKM system freed administrators from the requirement that all device drivers be compiled into a single static kernel. Now devices could be loaded at run time. This enabled support for a number of new features. Devices manufactured after the kernel was first built could be supported without a full rebuild. Pseudo devices could be added on the fly. And some development testing could be done without a reboot[1]. With LKM came the possibility of devices coming and going during run time. Generalized support for detaching interfaces was not implemented until Doug Rabson replaced LKM with the dynamic kernel linker (KLD) and newbus in FreeBSD 3.0. As part of this process, he implemented a primitive version of `if_detach()`. The KLD interface is an enhanced module system based on dynamic linking of ELF binaries.

While modules introduced the possibility of dynamic interfaces, dynamic interfaces were first used by ordinary people with the introduction of PC Card (PCMCIA) devices. The PC Card standard supports hot insertion and removal of cards. Through use of short pins, a few milliseconds of warning that a device is departing are provided, but otherwise, they come and go at will. In the 2.x time frame, PAO[2] was developed to provide PC Card support to FreeBSD [PAO]. Fairly functional support was available based on the work in PAO in FreeBSD 3. With the release of FreeBSD 4, PAO development ceased because all major changes had been incorporated into the FreeBSD tree. CardBus support was added in FreeBSD 5.0 and is currently fairly mature (roughly speaking, CardBus is to PCI what PC Card is to ISA.)

In FreeBSD 4.0, support for USB Ethernet devices was added. Like PC Card devices, USB devices may be attached and detached at will.

Since the introduction of USB networking devices, a number of new types of removable networking devices have appeared. The fwe(4) [man4], Ethernet over FireWire driver appeared in 5.0 and was later merged in to 4.8. The fwip(4), IP over FireWire interface, implements RFC 2734 [RFC2734], IPv4 over IEEE1394, and RFC 3146 [RFC3146], IPv6 over IEEE1394; it was introduced in 5.3. Bluetooth support was introduced in FreeBSD 5.2. Being wireless, Bluetooth devices are inherently dynamic. While not currently supported, hot plug PCI, Compact PCI, PCI Express, and Express Card all support some form of hot insertion and removal. Compact PCI is of particular interest because it provides the administrator with a button to press to indicate their intention to remove a device and a light the OS can use to notify the administrator that the device in inactive and removal is now safe.

I anticipate that as bus standards evolve, an increasing number will support hot

[1] Assuming that the developer was fortunate enough to make an error that did not result in a kernel panic.

[2] According to the PAO FAQ, "PAO stands for nomads."

plug devices. Further, I expect that in the not too distant future, hot plug interfaces will be the norm with the exception of integrated interfaces on motherboards.

In early autoconfiguration implementations, a finite number of units was statically allocated when the kernel or module was compiled. Devices implementing these sorts of preallocations are referred to as count devices due to the kernel configuration directive used to declare them. Today, this sort of hard coding is frowned upon unless there can only be a small, fixed number of devices (usually one). In FreeBSD 6.x, support for count devices will be removed from the config program. This will require that device counts either be fully dynamic or specified at boot time. With physical devices, new driver instances are allocated and destroyed as hardware is added and removed so the removal of count devices is no hardship.

In FreeBSD 3.4 the netgraph system was introducednetgraph(4). Netgraph allows dynamically configured nodes implementing networking functions to be configured into arbitrary graphs. One of the standard nodes is ng_iface(4) which appears as a virtual network interface. Since netgraph nodes are configured dynamically using ngctl(8) [man8], this is another source of dynamic interfaces.

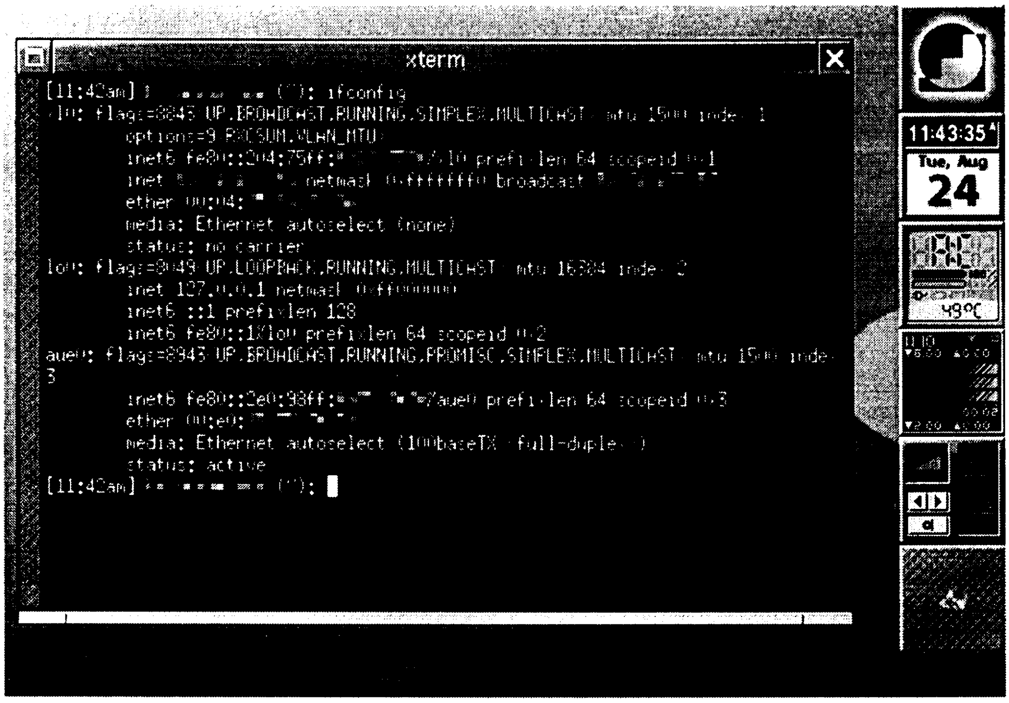
For pseudo-devices such as the loopback interface, lo(4), most devices are created by the network interface cloning code, referred to as if_clone. Network interface cloning was introduced in FreeBSD 4.4. Typically, a cloned device is created with a command like "`ifconfig vlan create`" which creates a new vlan interface and prints its name. This creates a number of interesting opportunities such as an IPv6 tunnel server that creates gif(4) devices on demand. The initial cloning code in FreeBSD was obtained from NetBSD and has since been extended to allow cloned devices to match more complex names in the ifconfig(8) create request. Initially, cloned devices could only be created with a command of the form "`ifconfig <drivername>[<unit>] create`". With enhanced cloning support, devices may support more complex names such as `<ethernet_interface>.<vlan_tag>` for vlan(4).

A new feature of FreeBSD 5.2 and DragonFly BSD is the ability to rename network interfaces. This can be useful to allow an administrator to hide the details of interface types or to easily identify the purpose of a dynamically created interface. Returning to the example of a gif(4) based tunnel server, tunnels could be named after registered users; so instead of `gif42`, the interface could be named `gif-<user>-<host>`, a much more meaningful name. Another way this feature can be useful is to give logical names to physical devices, allowing them to be upgraded or replaced without changing most system configuration. Eventually, devd(8) will support the ability to make decisions based on attributes such as slot number which could allow interfaces to be named based entirely on their location in the system.

As I have shown in this section, significant interface dynamism is present in today's network stack. With hot-pluggable and wireless hardware, kernel modules, netgraph interfaces, pseudo interface cloning, and interface renaming, virtually no device can safely be assumed to be static.

3 Problems

With network interface dynamism come a variety of problems of two major types. First are those having to do with userland, typically network management or monitoring applications. Most of the userland issues revolve around the fact that applications have not adapted to modern dynamic systems. In some cases, the applications have been partially adapted, but trip over problems such as the concept of a *different* interface. Second are problems in the kernel which are typically hardware race conditions or stale references to freed data.



```
[11:42am] [illegible] ("): ifconfig
[illegible]l0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500 index 1
        options=9<RXCSUM,VLAN_MTU>
        inet6 fe80::204:75ff:[illegible]%[illegible]l0 prefixlen 64 scopeid 0x1
        inet [illegible] netmask 0xffffff00 broadcast [illegible]
        ether 00:04:[illegible]
        media: Ethernet autoselect (none)
        status: no carrier
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384 index 2
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
aue0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500 index
3
        inet6 fe80::2e0:98ff:[illegible]%aue0 prefixlen 64 scopeid 0x3
        ether 00:e0:[illegible]
        media: Ethernet autoselect (100baseTX <full-duplex>)
        status: active
[11:42am] [illegible] ("): █
```

Figure 1: The xterm shows the correct interface listing, but wmnd shows wi0 which has been removed and replaced with aue0.

To facilitate this discussion of the problems caused by dynamic network interfaces, I will present three example systems and the problems they face:

1. A laptop with removable wired and wireless interfaces.

2. A server with hot swappable network interfaces.

3. An IPv6 tunnel server.

These systems are sufficient to expose most of the issues of dynamic network interfaces.

The laptop I will consider has a USB Ethernet interface supported by the aue(4) driver and a PC Card wireless device supported by the wi(4) driver. This situation presents several challenges. These challenges derive from the fact that these interfaces may come and go at arbitrary times. From the user's perspective there will be three significant problems. First, most of the simple network monitoring tools used in this type of environment do not detect the arrival or departure of interfaces. This causes them to only show interfaces that were attached when they were started. Second, because devices may come and go in arbitrary order, the indexes of those devices may not be consistent. Since many monitoring applications assume that the kernel index of an interface is unique for an instance of the application, they may confuse the wired and wireless interfaces with each other. Figure 1 shows an example of conflicting ifconfig(8) and wmnd [WMND] output. While the index is *the* valid handle to an interface, the life of that index is only the life of the driver attachment; when the device is detached, the index may be freely reused by another interface. Thus, it is not safe to assume the index will remain associated with the same interface unless the interface is being monitored for departure. The third and final userland problem is caused by interface renaming. Since interfaces may be renamed at any time, the name must not be used as a handle for an interface unless the interface list is somehow monitored for changes or some external assurance is provided that the name will not change. Automated monitoring systems should therefore assume that interfaces may change their names, but it is perfectly reasonable to use names for start up configuration or as part of an ifconfig(8) command line.

In addition to these userland problems, there are two classes of kernel issues. The first is stale pointers to the `struct ifnet`. When an interface is removed, its `struct ifnet` is destroyed, but sometimes references to the interface in the form of pointers to that structure remain. The `struct ifnet` is the device independent interface to a network interface within the kernel. A number of structures including `struct mbuf` may contain a pointer to the `struct ifnet` of an interface. Since the `struct ifnet` is destroyed immediately when the interface is detached, using these references may result in accessing random data or, worse, an unmapped page. Today little is done to prevent this race from occurring. Fortunately, since the interface is marked as down and the queues are drained early in the detach process, the system will generally not retain stale references when the `struct ifnet` is destroyed. There are however some situations that will prolong this race making it more likely it will be lost in a way that causes a kernel panic. The main one is use of the dummynet(4) system. Dummynet is a "traffic shaper, bandwidth manager, and delay emulator." While dummynet is holding packets, it currently stores a pointer to the destination interface which is used to send the packet once the desired delay has occurred. This increases the race window to the point that it will almost certainly be triggered if a significant delay is configured. This generally will not affect a typical laptop user, but could easily affect a server user.

The second kernel issue is hardware races on eject. These occur when a function that manipulates the hardware runs during or after the physical removal of a device. With devices such as PC Cards, drivers manipulate the hardware in ways that may cause system hangs or panics if the device is removed, due to issues such as corrupted reads or writes to nowhere. Complete solutions to these problems often require hardware modifications. These problems can be avoided by powering down the device in an orderly manner before removing it. My understanding is that an eventual result of modernization of the device code will be the addition of a devcontrol(8) program that allows such operations on all devices that support them. In addition to this solution, there are some workarounds to reduce the risk, but I will leave their discussion to others.

Having discussed the laptop case, I will move on to the case of a server with hot swappable interfaces. The server case has most of the problems of the laptop with two major differences. First, if hot-plug PCI or compact PCI devices are used, they close the hardware removal race by providing a mechanism for the administrator to notify the OS that the hardware is going to be detached. In theory this mechanism could also close the reference races. Unfortunately, these technologies are not currently supported by FreeBSD.

The second way the server scenario differs from the laptop scenario is that servers are often monitored by SNMP [RFC1157]. Dynamic interfaces present problems when dealing with SNMP. Today, most SNMP agents use the kernel interface index as the `ifIndex` variable. In MIB-II (RFC 1213 [RFC1213]) the `ifIndex` variable for each interface is defined to be:

> A unique value for each interface. Its value ranges between 1 and the value of `ifNumber`. The value for each interface must remain constant at least from one re-initialization of the entity's network management system to the next re-initialization.

MIB-II defines, `ifNumber` to be:

> The number of network interfaces (regardless of their current state) present on this system.

This means that interfaces may not have sparse indexes in SNMP. This in turn will not work if interfaces are dynamic. In RFC 2233 [RFC2233], "The Interfaces Group MIB using SMIv2", section 3.1.5 attempts to revise the interface numbering constraints to allow for dynamic interfaces. They do so by removing the constraint that `ifIndex` be less than or equal to `ifNumber` which allows the index space to be sparse, and by adding the constraint that the same `ifIndex` may not be reused by a *different* dynamic interface.

Unfortunately, the concept of a different interface is complicated and application specific. RFC 2233 simply states that the following constraints must be observed:

1. a previously-unused value of `ifIndex` must be assigned to a dynamically added interface if an agent has no knowledge of whether the interface is the "same" or "different" to a previously incarnated interface.
2. a management station, not noticing that an interface has gone away and another has come into existence, must not be confused when calculating the difference between the counter values retrieved on successive polls for a particular `ifIndex` value.

In the simplest case of the server with hot-plug interfaces, the current system mostly works because interfaces are typically added but not removed except to be replaced by a different device serving the same function. However, the second constraint above may not be handled correctly in this case because the counters are attached to the interface and will be reset. A slight modification to the agent to allow detection of this case and setting the `ifCounterDiscontinuityTime` object for the interface when its removal is detected would correct this issue.

The more complex case of a server with frequent interface arrivals and departures is typified by the IPv6 tunnel server scenario. This tunnel server has hardware similar to that in the previous scenario, but has a vastly different mode of operation. Registered users may request a tunnel for one or more hosts. When requested, a gif(4) interface is created using cloning. When the user requests that the tunnel be torn down or a specified timeout passed, the interface is destroyed. Because interfaces are created on demand, the automatically assigned kernel interface indexes should not be used for SNMP `ifIndex` values as is. The problem is that the only tunnel interfaces that may be considered the *same* are those which share the same user, host[3] pair. Thus, since kernel interface indexes will be allocated in a manner which attempts to limit the sparseness of the index space, kernel indexes will frequently reference *different* interfaces once a few interfaces have been destroyed. Ideally, the tunnel server should allocate `ifIndex` values and inform the SNMP agent when interfaces are created, but this is easier said than done, as most agents simply assume that the kernel index is the correct value for `ifIndex`. Since the user and host are not known to the kernel, there is no current mechanism for the kernel to choose a correct value for `ifIndex`. Additionally, there is no easy way to control the index from userland.

The problems posed by these three example systems cover most of the issues caused by dynamic network interfaces. Kernel and hardware races present challenges for kernel developers, and the complexity of maintaining consistent references to interfaces causes problems in userland.

[3] By host we mean the machine itself, not the IP address in most cases, e.g. a laptop might move about, but would be the same host.

```
struct ifindex_entry {
    struct ifnet *ife_ifnet;
    struct ifaddr *ife_ifnet_addr;
    struct cdev *ife_dev;
};

#define ifnet_byindex(idx) \
    ifindex_table[(idx)].ife_ifnet
#define ifaddr_byindex(idx) \
    ifindex_table[(idx)].ife_ifnet_addr
#define ifdev_byindex(idx) \
    ifindex_table[(idx)].ife_dev
```

Figure 2: ifnet_byindex() and related macros from sys/net/if_var.h

4 Solutions

In this section I propose and evaluate solutions to some of the problems presented in the previous section. In particular, I discuss two possible solutions to the problem of stale struct ifnet references, as well as the kernel framework for a partial solution to the problems of inconsistent indexes to the *same* interface.

At first glance, the problem of stale struct ifnet references would seem to be solvable through the simple addition of reference counts. After all, the problem is that references to the interface's struct ifnet are still held when the structure is freed. Unfortunately, there are significant problems with this approach. The first is simply that reference counts are expensive to maintain. Incrementing or decrementing a reference count requires either obtaining a lock or using another atomic operation. This is especially problematic when code is in the fast path since ever moment counts and many atomic operations take over a hundred cycles to complete. Since the struct ifnet references in dummynet and struct mbuf are used in the fast path, atomic or mutex operations should be avoided there if possible. The second problem is that the struct ifnet is part of the softc of physical interfaces which is destroyed when the device is detached. This means that a reference count might not prevent the destruction of the struct ifnet. The struct ifnet could be moved to separate storage to be managed by the networking system, but doing so would required modifications to virtually every one of the approximately 100 network drivers in the system plus all the externally maintained ones. Not only would this be difficult, but it would fail to resolve the hardware races, so the effort is unlikely to be worthwhile. Due to these problems, reference counting struct ifnet is unlikely to work.

There is a second possible solution, which is referencing the interface by index instead of a direct pointer to struct ifnet. In this case, each long lived struct ifnet pointer would be replaced with the interface's index. The pointer dereferences would be replaced with ifnet_byindex() called. To avoid null-pointer dereferences in this case, ifnet_byindex() would be modified to return a pointer to a special dead_if interface which has no-op functions in place of driver specific ones. Where possible, the dead_if struct ifnet would be filled with values that will not provoke panics. In general, kernel code should be modified to check the return value of ifnet_byindex() for dead_if and abort processing unless the check is more expensive than completing the operation and the expense matters. This solution avoids the need for an explicit (and expensive) atomic operation because assignment to pointers is atomic on all architectures supported by FreeBSD. If the modifications to ifnet_byindex() are done by insuring that the array used to implement it has all empty entries filled with pointers to `dead_if`, there will be no performance impact on `ifnet_byindex()`. There will be some performance impact on code that previously referenced `struct ifnet` directly since an additional look up will be needed. Since `ifnet_byindex()` is simply a macro as shown in Figure 2, this should be relatively cheap, but performance testing will be needed to precisely quantify the extent of the impact. If the performance impact is deemed too high, it may be possible to use macros to choose between these solutions at compile time so that environments such as dummynet systems with dynamic interfaces could optionally enable this extra level of indirection. It is worth noting that while using indirect references to `struct ifnet` will shrink the race, it will not completely close it.

The problem of SNMP agents assuming that the kernel interface index is a good value for `ifIndex` is difficult to solve, short of rewriting the agent to remove this assumption and forcing the agent to manage its own application specific `ifIndex` space. The kernel will assign the same indexes to interfaces across reboots and some effort is made to preserve indexes across module reloads, but since the allocator attempts to avoid sparse allocations, the indexes are inherently unsuited to the requirements of SNMP agents in applications such as an IPv6 tunnel server. One possible solution to this problem is to enable userland programs to set the kernel index of interfaces.

I propose an implementation of this functionality as follows: setting the index will only be allowed when the interface is not in the `IFF_UP` state. The actual change will take place by detaching the interface, changing the index, clearing the interface statistics, and reattaching the interface. From the perspective of userland applications, the interface will be destroyed and a new interface will be created with the desired index. A tunnel server controller process could use this functionality to create interfaces with userland managed indexes, thus allowing SNMP agents to work with fewer modifications. The agent will need to set `ifCounterDiscontinuityTime` appropriately. To aid in setting it, it may be useful to add a new per-interface variable indicating the epoch of the interface. The epoch would be reset any time the interface statistics were reset.

There are a few potential issues with this approach. First, the `if_index` variable in `struct ifnet` is a signed short so the useful range of index values is 1 to 32767 ($2^{15}-1$) which is not very large for some applications. This could be solved by increasing `if_index` to an int or long, but that would raise other issues. Specifically, there are some arrays that are currently required to be at least as long as the highest index. If the index is allowed to grow to `INT_MAX`, these arrays would be larger than the system address space of a 32-bit system. As such, these interfaces would have to be modified to use more complex structures such as trees or hashes. This would cause some operations such as `ifnet_byindex()` to change from constant time to $O(logn)$. This could severely impact system performance if indirect references were used in the fast path as suggested earlier in this section.

The second issue with expanding `if_index` is related to the problem of sparse indexes. With the current limit on the maximum index, storage concerns are not insoluble, but there are efficiency concerns. In the kernel this should not be a major issue as there is no reason to search for interfaces one index at a time when the interface list can be walked directly. In userland things are more difficult. The correct way to access interface information is via sysctl(3), but sysctl(3) does not provide the equivalent of SNMP's `GetNext` functionality. This means that walking the list of interfaces by index could take 32767 syscalls with the existing implementation. This is probably not acceptable overhead for each update of a monitoring interface. Even without expanding `if_index`, it will probably be necessary to provide better sparse access support to userland. Some options for this include adding a `GetNext` equivalent to sysctl(3), adding a next interface pointer to the sysctl(3) output, or publishing a list or bitmap of allocated indexes. A `GetNext` equivalent for sysctl(3) or the addition of a next interface pointer would allow applications to only make syscalls for information that actually exists. A list would be easy to produce and cheap to process in userland, but a bitmap would be smaller and could be maintained at virtually no cost. A bitmap is probably the easiest option.

I have presented possible solutions to two of the problems of dynamic interfaces. The solution of adding a layer of indirection to long-lived, stored references to `struct ifnet` shows some promise if performance is acceptable. Allowing userland applications to control kernel interface index allocation may or may not be useful in practice. It would allow tunnel servers to work with more or less unmodified SNMP agents, but it would not provide a full solution. A full solution will probably require application specific agents or better generalization of generic agents to allow application specific `ifIndex` management.

5 Advice to Application Implementers

Other than the problems with SNMP agents and indexes, most userland issues with dynamic network interfaces are problems of application design. Most simple interface monitoring tools such as wmnd are written with the assumption that once the application is started, the set of interfaces will remain constant. Since this is not the case with modern versions of FreeBSD, these applications behave in unexpected (though generally harmless) ways.

To prevent this problem, applications should use appropriate APIs to access interface data, and should use those APIs in ways that allow detection of changes to the list of interfaces. In particular, applications need to detect the arrival, departure, and renaming of interfaces. In this section, three possible ways to do so are presented. The first way is to periodically rescan the entire list of interfaces. In environments with few interfaces this may be done for every application refresh or it may be done less frequently if scanning the whole list is too expensive and delayed detection of changes in the list is acceptable. Another method is to monitor the `/dev/net` directory for the comings and goings of device nodes. This can be accomplished with the kqueue(2) [man2] mechanism or by scanning the directory with readdir(3) [man3]. A third approach (applicable only to programs that run as root outside a jail) is to monitor the routing socket for arrival and departure notifications.

There are two related complications with the third approach. If an interface is destroyed and then replaced between update cycles, the application needs to detect this some way. This isn't an issue with routing sockets or kqueues on `/dev/net` because notices will be sent for for both arrival and departure, but since the list is monitored via sysctl or by using opendir on `/dev/net`, there may be continuity problems where an interface appears to still exist, but in fact has been replaced with another. In the case of the routing socket there is an issue that a rename is modeled as a detach and attach which means a application may need a heuristic to detect this situation. To aid in solving this problem, I have added an interface epoch variable to FreeBSD. The epoch helps with both the problem of detecting interfaces that replace removed ones in the same cycle and interfaces that were renamed rather then removed. In this context, an interface is the same if and only if both its index and epoch are the same. In the routing socket case, replacing the current detach and attach notifications with a rename notification would be the ideal solution.

Once applications have been modified or written to notice new interfaces, the author may wish to consider ways to bring these new interfaces to the user's or administrator's attention. Exactly how this should be done is application specific. For example, in a WindowMaker dock application on a laptop, bringing new interfaces to the front may be the best approach, but that certainly wouldn't be appropriate to a tunnel server.

In addition to monitoring for added or removed interfaces, application designers should avoid the following two practices. First, many current applications refer to interfaces by name internally. Since interfaces can now be renamed at any time, this is no longer considered good practice. Instead, applications should refer to interfaces by index and convert that to a name when needed. Second, many monitoring or status applications currently obtain interface information via the kvm(3) interface which provides direct access to kernel memory. This is bad practice for a number of reasons. First, requiring that applications be suid kmem is dangerous from a security perspective as it is nearly always possible to leverage kmem access to obtain root access in the case of a programming error. Second, since the `ifnet_list` is now dynamic, walking it without a lock is not reliable. Third, perfectly good sysctl interfaces exist to access this information, so there is no actual need to put up with the first two drawbacks.

6 Conclusions and Future Work

As I have shown above, dynamic network interfaces present a number of challenges to developers of network device drivers and network monitoring and management applications. In the kernel these challenges are divided between hardware races which may be reduced by careful programming, but may only be eliminated with external signaling mechanisms, and races involving freeing of `struct ifnet` instances before all references to them have been removed. The problem of stale `struct ifnet` references may be reduced by replacing long lived references to `struct ifnet` with interface indexes, allowing a special no-op interface to be substituted when the interface is removed. Further exploration of this idea is needed before it can be put into common use. Performance impacts will need to be quantified and it will be necessary to determine whether or not the solution reduces the race sufficiently to warrant the overhead.

In userland the challenges are generally issues with outdated assumptions in userland applications. The most common problem today is network monitoring applications that assume the set of network interfaces is static. Solving this problem requires modifying the applications to monitor for changes in the interface list such that attaches, detaches, and renames are all correctly detected. The addition of an epoch variable on each interface should help detection of some of these cases.

A secondary userland problem is specific to SNMP agents. SNMP agents need to maintain an `ifIndex` which is unique for each *different* interface. Prior to the introduction of dynamic interfaces, agents were able to use the kernel interface index for `ifIndex`. This no longer works because allocation of kernel indexes is done in a manner which minimizes sparse allocation and RFC 2233 requires that allocations be sparse in a dynamic system. Allowing userland applications to control kernel indexes may provide a workaround in some circumstances, but enhancement of SNMP agents to allow external management of `ifIndex` variables for applications such as tunnel servers will probably ultimately be necessary.

Going forward, I intend to implement a sample network interface monitoring application demonstrating best practices in this area. Blind copy-and-paste from outdated applications is probably the single most significant cause of monitoring tools that do not correctly handle network interface dynamism. An up-to-date example should help this situation significantly.

Today nearly all network interfaces are potentially dynamic and in the future I believe dynamic interfaces will be the rule rather then the exception. Give this state of affairs, future kernel and application programmers should keep the dynamic nature of network interfaces in mind when they write interface related code. In fact, programmers dealing with kernel or application level management of any hardware or virtual devices should keep dynamism in mind to avoid the sort of problems we see with network interfaces today.

References

[McKusick1] K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, Boston, MA, 1996.

[McKusick2] K. McKusick. *Twenty Years of Berkeley Unix: From AT&T-Owned to Freely Redistributable*, Open Sources, O'Reilly and Associates, January 1999. http://www.oreilly.com/catalog/opensources/book/kirkmck.html

[man2] The FreeBSD Project, *FreeBSD System Calls Manual*, FreeBSD 5.3, 2004.

[man3] The FreeBSD Project, *FreeBSD Library Functions Manual*, FreeBSD 5.3, 2004.

[man4-2] The FreeBSD Project, *FreeBSD Kernel Interfaces Manual*, FreeBSD 2.0, 1995. http://www.freebsd.org/cgi/man.cgi?query=lkm&manpath=FreeBSD+2.0-RELEASE

[man4] The FreeBSD Project, *FreeBSD Kernel Interfaces Manual*, FreeBSD 5.3, 2004.

[man8] The FreeBSD Project, *FreeBSD System Manager's Manual*, FreeBSD 5.3, 2004.

[PAO] http://www.jp.freebsd.org/PAO/

[RFC1157] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *A Simple Network Management Protocol (SNMP)*, RFC1157, IETF Network Working Group, May 1990.

[RFC1213] K. McCloghrie and M. Rose, editors, *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*, RFC1213, IETF Network Working Group, March 1991.

[RFC2233] K. McCloghrie and F. Kastenholz, *The Interfaces Group MIB using SMIv2*, RFC2233, IETF Network Working Group, November 1997.

[RFC2734] P. Johansson, *IPv4 over IEEE 1394*, RFC2734, IETF Network Working Group, December 1999.

[RFC3146] K. Fujisawa and A. Onoe, *Transmission of IPv6 Packets over IEEE 1394 Networks*, RFC3146, IETF Network Working Group, October 2001.

[WMND] WindowMaker Network Devices. http://www.yuv.info/wmnd/

NetBSD Status Report

Ignatios Souvatzis
University of Bonn, CS Dept., Chair V
<ignatios@cs.uni-bonn.de>

29th September 2004

1 Introduction

NetBSD is an Unix-like open source operating system based on the 4.4BSDlite code and other contributions. One goal is compatibility with the POSIX and SUS specified interface definitions. The complete kernel sources and big parts of libraries and programs are BSD-style licensed, thus easily adoptable for commercial third-party applications.

This paper outlines the major changes in NetBSD-2.0 and beyond.

2 Project Structure

When NetBSD was founded in 1993, it was just a few developers cooperating informally.

In the meantime, The NetBSD Foundation was founded to formally represent the project. The NetBSD Foundation is formal owner of (part of the) software, and administers material contributions and the project servers (ftp, cvs, www, ...). Members of the foundation are the project developers, some 300 world-wide, communicationg mostly via the Internet.

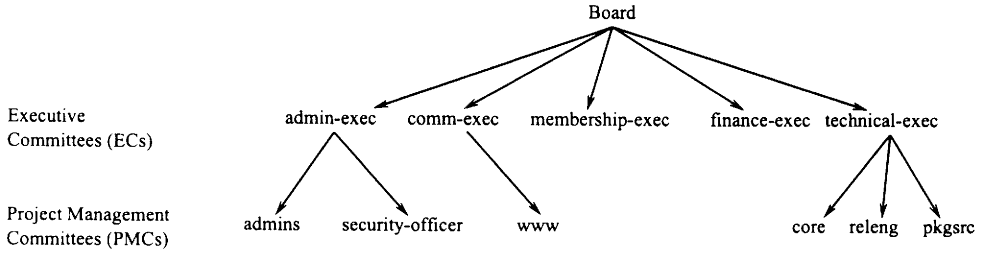


Figure 1: Structure of the NetBSD foundation offices

The NetBSD Foundation is a non-profit organization and has (in the USA) IRC 501(c)(3) status. It has registered the “NetBSD” and the “pkgsrc” trademarks in the USA.

3 The Operating System

3.1 Releases and Development

The NetBSD release numbering scheme was changed this year.

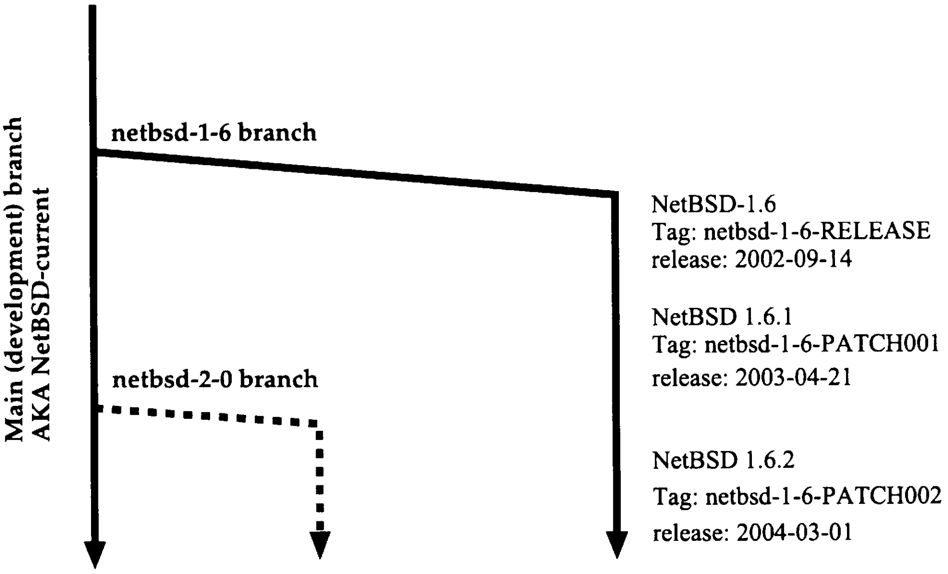


Figure 2: NetBSD development and release branches.

Formerly, major releases were numbered 1.x, with patch releases 1.x.y containing mostly bug fixes. The latest patch release of the old release branch, NetBSD 1.6.2, was released on March 1, 2004.

The next major release will be called 2.0. Patch releases will be called 2.1, 2.2 ..., and the major release after that will be called 3.0. The 2.0 release branch has reached its first release candidate at the time of writing this paper.

Unfortunately, a pkgsrc-like database for the base system ("syspkgs") is not yet activated, so tracking binary patches to the base system isn't yet possible. Thus, security advisories normally contain, per source tree branch affected, a source patch or a date the source repository was fixed, and instructions telling what part of the system to rebuild from sources.

3.2 Building The System

Building the system is done using a central script, `build.sh`, that controls building the toolchain — cross-compiler, cross-assembler, linker, auxiliary programs — into a seperate space, then building the system into a seperate directory structure, than creating the release archives. As of 2.0, this includes building of the X11 window system.

No elevated privileges are needed to do this, which allows for unattended operation. In fact, currently two project machines are bulding both release branches about twice a week (normally, the development branch and the latest releaase branch), for all target architectures, and a WWW page shows the last couple of lines of output for the last build and the last successful build for each architecture. New problems building the system are thus early noticed, even if specific to some architecture with slow (by today's standards) machines.

In 2.0, most CPU architectures will use the newest set of the GNU toolchain: gcc 3.3.x, gdb 5.3, binutils 2.13.2.

3.3 Device Support

Here is a list of newly supported devices:

- The AMD64 CPU. That's the fourth 64bit architecture supported by NetBSD (after Digital Alpha, 64 bit MIPS and 64 bit Sparc).
- IDE drivers were split into the different chipsets.
- Serial ATA
- A generic 802.11 software infrastructure is available, allowing to implement 802.11 access points.
- Wireless LAN cards building on the above.
- some RAID controllers
- Gigabit Ethernet cards
- A TCPA hardware driver is in development, but won't make it into the release 2.0.
- An HPPA (hp700) port is being developed.

3.4 Storage

- There is new filesystem code for Apple-UFS (as of MacOS-X) and Kirk McKusicks UFS2. The former is especially interesting for users of PowerPC- Macs who want to use both operating systems. The latter is necessary for people using very large file systes ($> 1TB$ when using 512 bytes block size).
- There is a SMB file system – it's possible to mount SMB shares.
- The volume manager "VINUM" was integrated.
- There is a new pseudo device "fss(4)" to take snapshots of a file system – useful for taking backups of an active system.

3.5 Kernel Internal Changes

- Multi CPU Support

  Support for multiple CPU machines was added for AMD64, i386, MacPPC, 32bit and 64bit Sparc. (Multi CPU support was already there for VAX and Alpha in 1.6).

- Multithreading

  Up until the 1.6 branch NetBSD didn't have kernel assisted multi threading. After that, an implementation of "Scheduler Activations" was brought into the kernel[15] and an N:M Posix thread library based on this was added.[14]

- Kqueue

  Kqueue(2) is a generic interface for the kernel to send notifications to user processes originally written by Jonathan Lemon for FreeBSD [8]. In its NetBSD integration, device and filesystem events are reported. Support for reporting USB and network events is planned.

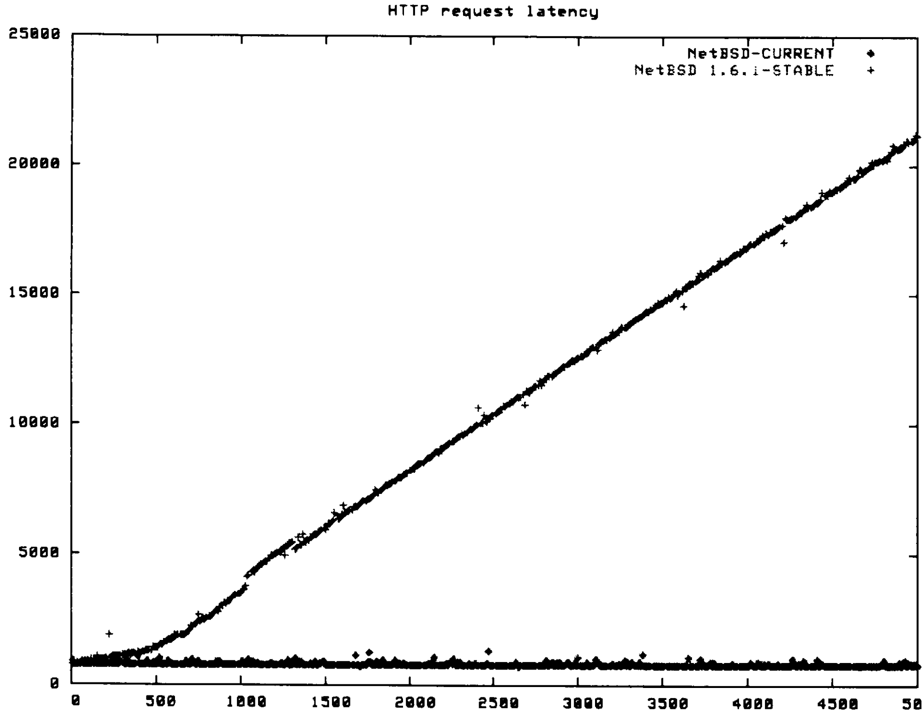


Figure 3: Scalability of a simple HTTP server (latency in μs vs. number of simultaneous requests, not counting the connect(2) time, using poll(2) for NetBSD-1.6.1 and kqueue(2) for NetBSD-current. This benchmark was done end of 2003 by Felix v. Leitner[7].

- Log-structured File System(LFS) with Unified Buffer Cache (UBC) support

  LFS uses the UBC[11] now, to allow for more file caching and synchronize mmap(2) with read(2)/write(2).

3.6 Performance Tuning

These are some of the performance-related NetBSD changes:

- pmc: a new API to access performance counters
- Hardware checksumming of some network interface cards is used by their drivers.
- The not so new Virtual Memory Subsystem UVM of NetBSD makes it possible to effectively loan out memory pages to other address spaces[1]. This was used to implement a transmitter side "Zero Copy TCP" (and UDP).

  To use this, an application has to mmap(2) (or directly create data in a local buffer), then write(2). The buffer may not be changed anymore by the application until the data are sent, else a copy on write fault will be created. [12]

3.7 Security

In this section some new features are enumerated, that help the system administrator to build secure systems:

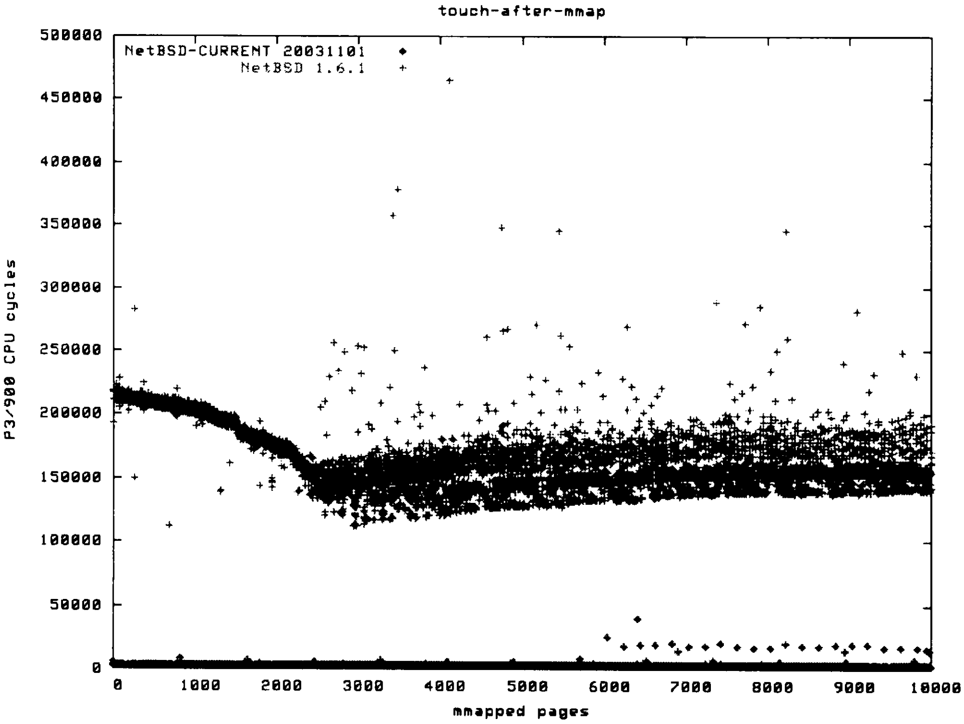


Figure 4: Touch after mmap benchmark, done end of 2003 by Felix v. Leitner[7].

- In the default installation no network services are activated (like on the 1.6 branch); they have to be activated by the system administrator in `/etc/inetd.conf` or `/etc/rc.conf`.
- On some architectures, that provide hardware support for this, stack and heap are not executable per default.
- If so configured, NetBSD only allows to execute programs that match their registered (in the kernel) checksum.
- Systrace[10] allows to control privileges of running programs at the system call + parameter level. E.g., binary programs can be restricted to only write in a specific directory, or not to do network calls, etc.
- There is a new pseudo device driver "cgd(4)" which is a encryption layer for disk partitions. This works with preconfigured keys (for data partitions) and with random keys (to secure swap).

Of course, known security fixes for the integrated OpenSSL, OpenSSH, BIND, Sendmail etc. are incorporated.

3.8 Miscellaneous

- Linux binary emulation was enhanced to better support Java and OpenOffice on i386 and PowerPC.
- On PowerPC machines a MacOS-X emulation is available.[4]

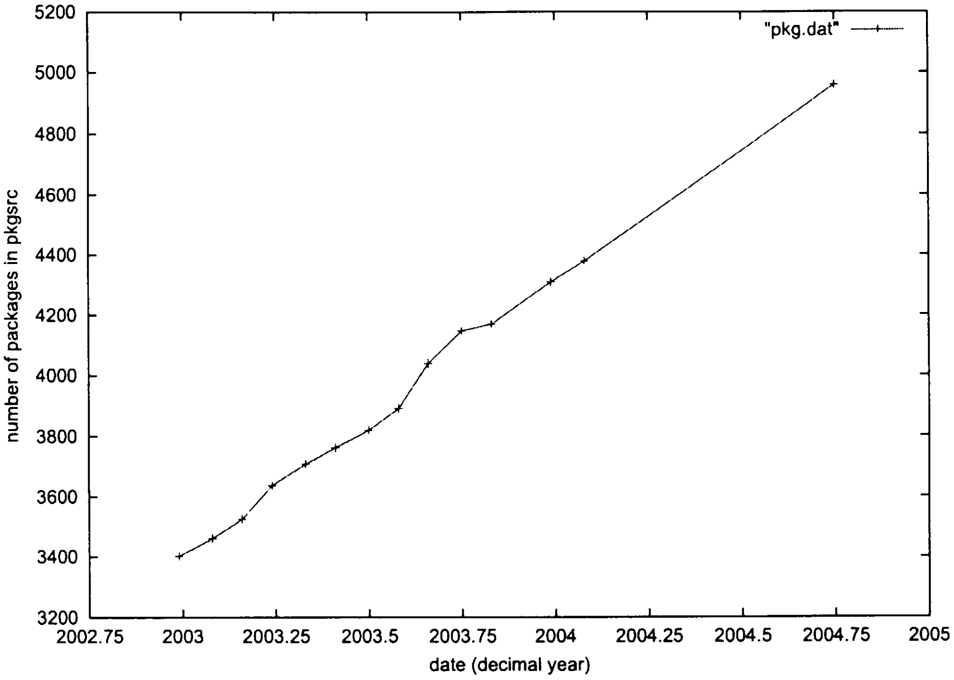


Figure 5: The growth of the `pkgsrc` collection between EuroBSDCon 2002 and EuroBSDCon 2004

- The sysctl interface is dynamically created: New kernel modules can enhance the sysctl tree without requiring a recompilation of `/bin/sysctl`.
- Long host names are supported in `utmpx`, `wtmpx`, `lastlogx`.
- All system binaries (including /bin and /sbin) are dynamically linked now. For emergency situations, a small `/rescue` directory with a few statically linked binaries is available.
- Lots of externally maintained but integrated software has been upgraded to their latest versions: pppd, tcpdump, file, named, gcc, binutils (as, ld, etc.), postfix, sendmail, cvs, routed, texinfo , diff, grep, amd, openssh, openssl, less ...

4 pkgsrc: The NetBSD 3rd Party Software Installation Infrastructure

Part of NetBSD is "pkgsrc", a system for building and installing third party software on NetBSD (and elsewhere[3]), and for tracking the version of installed packages, the affected files, and dependencies on other packages.

Pkgsrc originated with FreeBSD, but was heavily modified for NetBSD and beyond. Currently it contains nearly 5000 files.

Most of pkgsrc is not traditionally cross-compilable. However, some effort has been done to do batch builds, and to do cross-compilation by using a mixture of using a cross-toolchain automatically and running parts of the pkgsrc tools in an emulator.[13]

Security advisories for pkgsrc-installed packages are not issued in a human readable form. Instead a (text form) database is maintained, that connects package version numbers to security problem types and an URL describing the problem. The (pkgsrc) tool "audit-packages" can be used to automatically check the installed packages against the list, and to decide what packages to upgrade.

Pkgviews([2][6]) — a system that allows to install multiple versions of a package in parallel, thus allowing to upgrade packages and their dependencies on a live system without making it unusable for an extended time -- are integrated in the source code, but not yet supported by all packages, and thus not activated.

There has been a pkgsrc development conference in Vienna in the summer of 2004, and the next one is planned for next year.

5 (Not only) New Documentation

- NetBSD contains the usual Unix-style Manual Pages. The usual eight chapters known from other Unix-like systems are enhanced with a ninth chapter about kernel functions and interfaces.
- Frederico Lupi's "NetBSD Guide" is maintained by the NetBSD project now [9].
- A NetBSD Device Driver Guide is being written[5].
- Both documents, and many more design documents and HOWTOs, can be found on the NetBSD WWW servers in the *Documentation* subdirectory.
- The NetBSD WWW pages are available in multiple languages, selectable from the home page.

Important URLs

[netbsd]	NetBSD – homepage: http://www.NetBSD.org/
[changes]	NetBSD – changes: http://www.NetBSD.org/Changes/
[autobuild]	binary snapshots of NetBSD-current and the release branch from the autobuild machines: http://releng.netbsd.org/ab/

References

[1] Charles D. Cranor und Gurudatta M. Parulkar, *The UVM Virtual Memory System,* in: Proceedings of the USENIX Annual Technical Conference, Monterey, CA, USA 1999, http://www.usenix.org/events/usenix99/full_papers/-cranor/cranor.pdf

[2] Alistair G. Crooks, *PkgViews* — , 2nd European BSD Conference, Amsterdam 2002, http://www.NetBSD.org/Documentation/software/pkgviews.pdf

[3] Alistair G. Crooks, *A Portable Packaging System*, Proceedings of the European BSD Conference 2004, Karlsruhe, Germany.

[4] Emmanuel Dreyfus, *MacOS X binary compatibility on NetBSD*, Proceedings of the European BSD Conference 2004, Karlsruhe, Germany.

[5] Jochen Kunz, *NetBSD Device Driver Writing Guide*, http://www.unixag-kl.fh-kl.de/~jkunz/NetBSD/

[6] Johnny C. Lam, *User's Guide to PkgViews*, http://mail-index.netbsd.org/tech-pkg/2004/01/06/0004.html

[7] Felix v. Leitner, *Unix Scalability Benchmarks,* work in progress, http://bulk.fefe.de/scalability/

[8] Jonathan Lemon, *Kqueue: A generic and scalable event notification facility,* in: Proceedings of the FREENIX track, 2001 USENIX Technical Conference, Boston, MA, USA, http://www.usenix.org/events/usenix01/-freenix01/full_papers/lemon/lemon.pdf

[9] Federico Lupi et al., *The NetBSD Operating System: A Guide* (6 Languages), http://www.netbsd.org/Documentation/#netbsd-guide

[10] Niels Provos, *Improving Host Security with System Call Policies*, 12th USENIX Security Symposium, Washington, DC, USA, August 2003.

[11] Chuck Silvers, *UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD,* in: Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference, San Diego, CA, USA, http://www.usenix.org/publications/library/-proceedings/usenix2000/f reenix/full_papers/silvers/silvers.pdf

[12] Jason Thorpe, *Experimental zero-copy for TCP and UDP transmit-side,* http://mail-index.netbsd.org/current-users/2002/05/02/0016.html

[13] Krister Walfridsson, *Cross-Compiling Packages*, Proceedings of the European BSD Conference 2004, Karlsruhe, Germany.

[14] Alexandre Wennmacher, *Symmetrie in NetBSD,* freeX 1/2003–3/2003

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

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

Ignatios Souvatzis
University of Bonn, CS Dept., Chair V
<ignatios@cs.uni-bonn.de>

29th September 2004

1 Introduction

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

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

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

We are considering to offer SR and MPD for this, too. As the original language distributions are only targeted at a few commercial Unix systems, some porting effort is needed, outlined in the SR porting guide[4].

The integrated POSIX threads support of NetBSD-2.0 should allow us to use library primitives provided for NetBSD's phtread system to implement the primitives needed by the SR run-time system, thus implementing 13 target CPUs at once and automatically making use of SMP on VAX, Alpha, PowerPC, Sparc, 32-bit Intel and 64 bit AMD CPUs.

This paper describes work in progress.

2 Generic Porting Problems

Given the age of the software and the gradual development of the C language and the operating system environments available, some adaptation is to be expected. Fortunately, the latest distribution of SR (version 2.3.2) has already been portend to two relatively modern Unix-like environments (Solaris 2.2 and Linux), so the necessary changes turned out to be confined to a one area:

`gcc 3`, the system compiler of NetBSD-2.0, doesn't provide old `<varags.h>` variable argument functions anymore, so those had to be converted to `<stdarg.h>` syntax. Also, none of those functions had fixed arguments. Most of the functions had a first logical parameter `char *locn` which could be changed into a fixed parameter. A few functions had a first integer parameter (a count of the remaining parameters). In one

Implementation	Context switch times
i386 assembler	0.059 μs
SVR4 system calls	6.025 μs

Table 1: *Raw context switch times*

case (sr_cat), the loops extracting the parameters from the variable argument list had to be changed to be initialized with the newly introduced fixed parameter.

3 Verification methods

SR itself provides a basic and an extended verification suite for the whole system; also a small basic test for the context switching primitives.

The basic suite should be run to test an installation; the context switch tests and the extended suite are used to verify a new porting effort.[4]

3.1 Context Switch Primitives

The context switch primitives can be independently tested by running make in the subdirectory csw/ of the distribution; this builds and runs the cstest program, which implements a small multithreaded program and checks for detection of stack overflows, stack underflows, correct context switching etc.

3.2 Overall System

When the context switch primitives seem to work individually, they — and the building system used to build SR, and the sr compiler, linker, etc. need also to be tested.

A basic verification suite is in the vsuite/ subdirectory of the distribution; it can be extended with more tests from a seperate source archive vs.tar.Z. It is run by calling the driver script srv/srv, which provides normal and verbose modes, as well as using the installed vs. the freshly compiled SR system. The only test that is expected to fail is the srgrind source code formatter — it needs the vgrind program as a backend.

4 Performance evaluation

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

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

5 Establishing a baseline

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

Test description	i386 ASM	SVR4 s.c.
loop control overhead	0.01 μs	0.01 μs
local call, optimised	0.07 μs	0.07 μs
interresource call, no new process	1.45 μs	1.39 μs
interresource call, new process	2.95 μs	22.20 μs
process create/destroy	2.46 μs	26.14 μs
semaphore P only	0.07 μs	0.07 μs
semaphore V only	0.05 μs	0.05 μs
semaphore pair	0.11 μs	0.11 μs
semaphore requiring context switch	0.39 μs	9.09 μs
asynchronous send/receive	1.71 μs	1.63 μs
message passing requiring context switch	1.90 μs	14.50 μs
rendezvous	2.65 μs	27.05 μs

Table 2: *Run time system performance. The median times reported by the SR script* `vsuite/timings/report.sh` *are reported.*

The first tests were done by using the provided i386 assembler context switch routines. After verifying correctness and noting the times (see tables 5 and 5), the same was done using the SVR4 module instead of the assembler module.

All tests were done on a 500 MHz Pentium III machine with 16+16 kB of primary cache and 512 kB of secondary cache, and 128 MB of main memory, running NetBSD-2.0_BETA as of end of June 2004.

The table shows a factor-of-about-ten performance hit for the operations that require context switches; note, however, that the absolute values for all such operations are still smaller than 30 μs on 500 MHz machine and will likely not be noticable if a parallelized program is run on a LAN-coupled cluster: on the switched LAN connected to the test machine, the time for an ICMP echo request to return is about 250 μs.

6 Possible improvements using NetBSD library calls

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

However, we should be able to do better than the `svr4` module without writing our own assembler modules, as NetBSD 2.0 (and up) contains its own set of them for the benefit of its native Posix threads library (`libpthread`), which does lots of context switches within a kernel provided light weight process ([5]). The primitives provided to `libpthread` by its machine dependent part are the two functions `_getcontext_u` and `_setcontext_u` with similar signatures to `getcontext` and `setcontext`.

There are a few difficulties that arise while pursuing this.

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

Second, there is no context initializing function at the same level as `_setcontext_u` and `_getcontext_u`. `makecontext` looks like it would be good enough but this has to be analyzed further.

7 Work items left to do

7.1 Building a package for pkgsrc

To ease installation, a prototype package for the NetBSD package system has been built. It needs a bit of refinement, though, but will be available soon. (As the NetBSD package system is available for more operating systems than NetBSD, a bit more work is needed.)

7.2 Implementing and testing multithreaded SR

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

References

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

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

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

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

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

The A-tree - a Simpler, More Efficient B-tree

Alistair Crooks, atree@alistaircrooks.com

29th September 2004

Abstract

Traditionally, searching for text is accomplished by using a data structure fitted to the task, and database access methods have tended to standardise on B-tree multi-way trees, especially where proximity searching may be desired.

B-trees were discovered in 1970. Recent advances in processor speeds and memory speeds have resulted in a very high speed for memory to memory copying, and for a much larger set of information to be held in memory at any one time. Some of the restrictions and constraints which were in place when B-trees were discovered are no longer in place.

This paper describes a new method of data organisation for searching, the A-tree, and explains the background and reasons for its design and implementation. Performance characteristics of different multi-way trees are examined and discussed.

Introduction

Advances in Memory and Processor Speed

1970

In 1970, the multi-way B-tree was discovered by [Bayer1972], and independently at about the same time by M. Kaufman [unpublished]. In 1971, the Intel 4004 was released by [Intel1971], with a clock speed of 108 KHz, able to address 1 KB of program memory and up to 4 KB of data memory. [IBM1971]shipped its first 370 in 1970, and it typically shipped with 1 MB of core memory, and a roomful of disk drives, probably totalling 200 MB. In 1973, the big mainframe [IBM1970a]disk drive was model 3330-11: 400 MB for $111,600 or $279/MB.

2004

In 2004, only 35 years later, commodity microprocessors are omni-present in datacentres, on desktops and in embedded work. Disk sizes of 160 GB and greater are common and cheap, and we are seeing disk sizes of 400 GB appear as commodity items. Processor speeds, admittedly taking advantage of clock multiplication, are at 3.4 GHz, with faster CPUs expected, and memory sizes of more than 1 GB are normal.

[Economist2004]summed up the differences between the PDP-7 which Ken Thompson used to develop the early versions of Unix as:

> ...it is necessary, though difficult, to recall just how comparatively primitive the state of computing was 30 years ago. The first version of Unix was written by Dr Thompson for the PDP-7, a computer made by the Digital Equipment Corporation, which cost a mere $72,000, and came with eight kilobytes of memory, and a hard disk a bit smaller than a megabyte. By contrast, a desktop computer today typically costs a hundred times less, has roughly 64,000 times as much memory and a hard disk 40,000 times as big.

Implications of Progress

Between 1970 and 2004, processor speed, on-chip caches and main memory have all been speeded up in a manner which would have been inconceivable when Knuth wrote his seminal work on sorting and searching in 1973 [Knuth1973]. A memory to memory copy on a relatively slow processor such as the ARM is now regularly achieving speeds of well over 1 Gigabyte per second for both aligned and unaligned copies under the NetBSD operating system [NetBSD2004].

It's also necessary to put the era in perspective: in 1970, man had just landed on the moon, Unix was undergoing an internal rewrite within Bell Labs to add pipes, and to use a high-level language like C, neither Microsoft nor Apple Corporations had yet been founded, and the teletype was the usual interface to a multi-user computer. Unix was five years away from making its way into the educational community. Kurt Cobain was 3 years old, it would be 17 years until Joss Stone would be born, and Bill Gates was 15 - Windows 3 was still 20 years away. The troubles in Northern Ireland were only just starting, and the USA was still fighting a war in Vietnam. Usually, mainframe computer systems were large, multi-user systems, maybe with magnetic drum memory, and certainly in their own area in a data center.

To jump forward just 34 years, CPUs and memory are strikingly inexpensive, and disk space has grown to the state where we are no longer being asked by friendly system administrators to clear up unwanted files, and desktop and laptop computers (undreamt of in 1970) are now much faster than the largest supercomputer in 1970. LCD screens are usually the user's means of interaction with a computer, and mice and windowing environments are the standard. Virtual memory is an integral part of every chip but the smallest embedded device, and even most new handheld telephones have an embedded chip inside which has an integral MMU. Peripheral speed has advanced, DMA is the standard means of transferring data from a peripheral to memory, and we have moved to 64-bit technology on a number of operating systems and platforms.

1 Access Methods

When an access method is deployed in computer systems today, there are generally two possibilities

- the B-tree
- Hashing

This paper will now add a third method

- the A-tree

and then compare and contrast their uses, benefits and drawbacks.

2 The B-tree

2.1 Description

Since Bayer & McCreight's discovery of the B-tree, a number of people have improved and extended the original description. Most of these are documented in [Comer1979]'s work. To recap, a B-tree is an example of a multi-way tree, where:

1. Every page contains at most 2n items (where n is the order of the B-tree)
2. Every page, except the root page, contains at least n items
3. Every page is either a leaf page i.e. has no descendants; or it has m + 1 descendants, where m is the number of entries in the page
4. All leaf pages appear at the same level

(see [Wirth1976])

If we take the Berkeley DB implementation of B-trees as the reference implementation for just now (version 1 of the Sleepycat db implementation is standard on the *BSD distributions, mainly due to licensing issues with later versions), a B-tree is made up of leaf and internal pages. Each page holds a number of (key, data) pairs. A traversal of the leaf elements from first to last will produce an ordered linear list of the elements of a tree. A B-tree has an order property, where the order is the number of elements which may exist in a leaf page (a B-tree of order 2 will have at least 2 entries in each leaf page, and at most 4 entries). The sparse property of B-trees leads to a number of benefits - insertion into a tree is usually simple, since, by law of averages, there will be space to insert a (key, data) pair into a B-tree page. On the rare occasions that B-tree insertion overflows a leaf page, then the insertion will overflow into neighbouring pages, and internal pages will receive new indices themselves, denoting the leading entry in the page underneath the internal page. In a similar fashion, deletion from a B-tree may mean underflow in rare cases, and so the tree will shrink in an ordered manner, again perhaps involving re-calculation of indices in internal pages higher up the B-tree.

When updating a B-tree by adding or deleting entries, in an environment where multiple access by different processes is allowed, may require the whole internal page to be locked, as well as parent pages, in case of overflow and underflow.

2.2 Overview

In all, a B-tree provides a convenient structure for ordered searches (where predecessor and successor entries can be easily found, and the whole tree itself can be traversed in order). In normal use, a B-tree will organise itself into a flat structure, given a large enough order, and searching within pages of a B-tree is done by binary searching, which proves to be extremely efficient in practice.

2.3 Extensions

In order to avoid duplication of data, B-trees are usually implemented with entries in internal pages pointing directly to their associated values - there is no need to duplicate this entry in the correct position in the leaf page, since the searching process will hit the entry in the internal page first. This is the usual method of implementation, and is also referred to in literature as a B+-tree. One side-effect of this is that the data at the leaf pages, when read off from first to last, is still ordered, but incomplete, as the internal pages now contain some of the data.

A B*-tree is a standard B-tree with the following characteristics:

1. Every page except the root page has at most m children (where m is the order of the B*-tree)
2. Every page, except for the root and the leaves, has at least (2m - 1)/ 3 children (this means that we use at least two thirds of the space available in each page)
3. The root has at least 2 and at most (2 * floor((2m - 2)/3)) + 1 children
4. All leaves appear on the same level
5. A non-leaf page with k children contains (k - 1) keys

from [Knuth1973] pages 477-478.

The B-tree can be extended to provide multi-dimensional key-searching (see Guttmann's R-trees for an example of this), by keeping a relatively small order, and storing maxima and minima values for the extra index fields in each internal page. Internal pages of R-trees hold the maxima and minima values of key fields from subsequent pages. The query optimiser can then be used to make searching more efficient for multi-dimensional searches.

B-trees were originally conceived to hold their data internally to the page - this improves locality of reference. Most modern implementations usually hold pointers to (string) keys in separate pages which are themselves cached.

2.4 Usage

B-trees are typically the access method of choice for database administrators. Searching by B-tree is almost as quick as by using hashing methods. Typically costly overflow and underflow operations happen rarely, and proximity searches are possible. Scanning whole trees in order is possible.

2.5 Drawbacks

In light of these speeds quoted above, the whole design of a B-tree seems outmoded.

- a B-tree duplicates the virtual memory hardware in user-level software, with a resulting slowdown in performance
- typical operations on B-trees, even B-trees with large orders, result in a number of calls to move small areas of memory around
- hand-crafting iterative statements in a high-level language such as C seems to be missing the point - given the right addresses to copy from and to, the CPU is much more efficient when using hand-tuned assembler routines to copy memory from one location to another, rather than trying to optimise the amount of memory to be copied, and perhaps performing that copy in a loop.
- the original B-tree, as described by Knuth, expects keys to either be integers with keys which are solely integers - strings are much more common. Knuth also briefly mentions the B*-tree, which keeps variable length strings in the leaf pages, presumably for locality of reference. This has been overtaken by time, and now most implementations of B-trees (such as the Sleepycat db implementation) use a separate cache for variable-length keys. Implementing caches on top of the underlying hardwares own caches may provide suboptimal performance in real-world applications.
- if these strings are held in the B-tree page, to provide locality of reference, then when an entry moves from one page to another upon insertion or deletion, overflow or underflow, the data must be transferred to the new page, which is time-consuming and inefficient.

  Consider, for example, the example of a file system which uses a B-tree to order a directory. If the directory entry information is retained within the B-tree page, then locality of reference is assured, and directory traversal can be achieved by just using the information in the page itself. Conversely, when adding or deleting directory entries, directory entry information may need to be copied between B-tree pages, which may be time-consuming and inefficient. If the directory information is held in a separate string table, it is unclear how that table would be organised to allow efficient searching, insertion and deletion, even allowing for efficiencies provided by reference counting - obviously a segmented, ordered approach would be best, but a recursive B-tree is, unfortunately, out of the question.
- data can be added to a B-tree in such a way as to minimise performance and maximise tree constructions times, by causing pages to overflow in a pathological way.

2.6 Illustration

Figure ??, taken from [Wirth1976], illustrates the way a B-tree of order 2 will grow when data is added to the tree in the following order:

20
(a)

40 10 30 15
(b)

35 7 26 18 22
(c)

5
(d)

42 13 46 27 8 32
(e)

38 24 45 25
(f)

The data are specially chosen to exercise all aspects of the B-tree code, including overflow of leaf pages and internal pages.

3 Hashing

3.1 Description

By using a transformation function or "hashing function" on the key, a value is obtained - the "hash value" - and is used as a subscript to index into an array which holds pointers to the data. Hashing is a very simple access method, and only needs a hashing function and an array to be implemented. When the hash, or transformation function, of two distinct values results in the same value, we say that a collision has occurred. There are a number of methods of resolving collisions

- one way is to have the array value point to a linear list of values, external to the array. If there are a lot of insertions and deletions, this method can be costly, as linear lists have to be traversed in order
- another way is to step on by a number of slots in the array, but this approach can prove problematic if the original value is deleted (since the fact that the subscript is "busy" is used as a trigger to step on). If the array itself is full or near full, this approach can be costly

3.2 Overview

In real-world usage, the performance of hashing is proportional to the values produced by the hashing function - if these are a uniform spread across the array, then the chances are that the access method will perform well. If the array is itself too small, then performance problems will ensue. For good performance, an intelligent choice of hash function must be made.

In general use, there are a number of hashing algorithms in general use:

- the hashing algorithm used in Perl has been found to produce a good spread of values
- the simple hashing function from [Kernighan1976] is not the best, but is simple to understand
- the[FNV]hashing algorithm as found in various places - the FreeBSD 4.3 NFS code, Linux's NFSv4 code, etc, although the NetBSD project, after extensive benchmarking, concluded that there were better hashing algorithms than FNV
- the sdbm hashing code by Ozan Yigit performs well on various sample sets
- [Yigit2001]gives an excellent comparison of various hashing methods as applied to real-world uses, by using all the C identifiers in X11R4 sources, hashed onto an 8192-entry hash table and an 8209-entry hash table. More information is available at [Yigit2004] and in [Jenkins2004]

The holy grail of hashing algorithms is called the perfect hashing algorithm - one where, for every distinct input key, there is a distinct and unique hash value. Usually, these take a long time to compute, and can only be deterministic in the case of a known and pre-computed set of inputs, but there is software available which will compute the best hash function, given a known set of inputs - see [gperf2004].

3.3 Usage

For some reason, hashing is usually the access method of choice for programmers, and often is used in systems programming. FreeBSD have added hashed entries in directories (over a certain threshold) to speed up directory traversal.

4 The A-tree

4.1 Description

The A-tree can be thought of as an array of pointers to storage elements, grouped together logically into virtual pages. This array is sparse. The size (maximum number of storage elements) of a virtual page is called the *order* of the A-tree. An A-tree has the following characteristics:

- The array is sparse, so that there are at most *order* elements in a virtual page, and every virtual page except the first one must have at least one element. The empty A-tree has 0 elements in its first virtual page.
- If there is a positive number of elements in the A-tree, there will always be an element in the first element of a virtual page.
- Searching within an A-tree uses (conceptually) two binary searches; the first binary search is done using the first element in each virtual page, so that the correct virtual page is found, and the second binary search takes place within the virtual page, to test for the presence or absence of an element.

Our A-tree implementation uses two arrays:

- one array for the elements, which contains total tree size elements; each element is a pointer to the (*key, value*) tuple. The key and the value are both sized strings.
- one array of integers, which contains (*total tree size / order*) elements; each element of this array contains the size of the virtual page

Conceptually, if a B-tree had all its elements in leaf pages (i.e. a standard B-tree as described by Knuth), and its nodes were laid end-to-end in order, the resulting array would be similar to an A-tree. Pictorially, an A-tree can be envisioned like this:

5 7 10	15 18 20 22	26 30	35 40
Page 0	Page 1	Page 2	Page 3

```
Number of elements in each virtual page: 3, 4, 2, 2
```

With 11 elements and four virtual pages in the A-tree, a search for an element would start by finding the correct virtual page in which to search.

This is done by a binary search, using the first element of the virtual page as the key, and using two temporary bounds to find the virtual page.

For example, assuming C/C++ array subscript notation, let us search the A-tree illustrated above for the search key "10".

The binary search would start with the higher bound at (virtual page) 3, and the lower bound at (virtual page) 0, and so the mid-point is at (3 + 0) / 2 == 1. The first element of virtual page 1 is "15". If the search element is less than 15, then the higher bound will be set to (1 - 1) == 0, and the lower bound will stay at 0. When the two bounds are equal, or the lower bound is higher than the upper bound, the virtual page has been found.

The binary search then continues within the virtual page - the upper bound is set to (3 - 1) == 2, the lower bound to 0. Quickly the search finishes by finding the element "10" in subscript 2 of the virtual page.

4.2 Characteristics

Using the insights from the increased memory and CPU speeds as a base, and observing some of the characteristics of both B-trees and hashing, it was possible to come up with the basic characteristics for A-trees:

- a large array of pointers to key and value tuples is kept
- this array is sparse
- the array is conceptually split up into virtual pages - in reality, a separate array of "number of items in a virtual page" is kept
- each virtual page must have at least one element in the virtual page
- binary searching in two stages:

1. finding the correct virtual page by using the first element in virtual pages, and then
2. another binary search within the virtual page, used to locate elements

- fast memory to memory copying is used when inserting elements into the A-tree. If an element can be inserted into a virtual page, any elements which need to be moved up the virtual page will be moved. If there is no room in the page, an oscillating search is used to find the nearest virtual page which has room, and elements are moved up or down the A-tree accordingly, to make space for the element to be inserted.
- when deleting elements, the same fast memory-to-memory copying is used to move any elements in the virtual page down. If underflow occurs, the virtual pages are all moved down 1, again using the fast memory-to-memory copy. It does not matter if unused elements are copied, since this overhead is vastly outweighed by the speed in using memmove(3).

The A-tree itself is implemented by using a resizable sparse array of elements, and a smaller array of virtual page sizes is kept - the size of this array is the total size of the array of elements divided by the order of the A-tree.

4.3 Searching

When searching an A-tree, the following pseudo-code is used:

1. calculate low and high virtual page subscripts, and from these, the mid-point virtual page subscript
2. compare the first element of the mid-point virtual page to the search element, and modify low and high virtual page subscripts accordingly. Repeat steps 1 and 2 until the search converges on a single virtual page (in which the search element either resides, or would reside, if it was present in the tree).
3. within the virtual page, calculate the low and high indices for elements
4. perform a binary search within the page to find the desired element

Sample C code for searching for the virtual page and the position within it is shown below:

```
/* do a binary search in the virtual page */
static int
binsearch(atree_t *ap, const DBT *key,
          int *pg,int *lo, int *hi)
{
    *lo = 0;
    *pg = -1;
    if ((*hi = ap->virtpgsizec - 1) < *lo) {
            return 0;
        }
        for (;;) {
            do {
                int mid = (*lo + *hi) / 2;
                int cmp;
                cmp = (*ap->info.compare)
                        (key, &KEY(ap,
                               (PAGE_FOUND(*pg)) ?
                               mid :
                               SUBO(ap, mid)));
                if (cmp <= 0) {
                    *hi = mid - 1;
                }
                if (cmp >= 0) {
                    *lo = mid + 1;
                }
            } while (*lo <= *hi);
            if (PAGE_FOUND(*pg)) {
                return (*lo - *hi == 2);
            }
            /* normalise the page number */
            if ((*pg = (*lo - *hi == 2) ?
                 *hi + 1 : *hi) < 0) {
                *pg = 0;
            }
            *lo = SUBO(ap, *pg);
            if ((*hi = *lo + ap->virtpgsizev[*pg]
                 - 1) < *lo) {
                return 0;
            }
        }
    }
```

4.4 Insertion

When adding an element to the A-tree, the following steps take place:

1. if the tree is full, then resize (by doubling) the main array, and the array of virtual pages. After resizing, make the array more sparse, by moving half the elements out of each virtual page into the next virtual page

2. perform a binary search to find the correct page and position within the page to insert the new element. If the element is already in the tree, do not attempt to insert it.

3. mark the A-tree as "updated"

4. if there is no room in the virtual page, then perform an oscillating search to find the nearest page which has room. Assuming the current page is *i*, the order of the page search will be (*i* + 1), (*i* - 1), (*i* + 2), (*i* - 2), etc until room is found. When room has been found (and there must be room to insert an element, because of Insertion(1)), the elements are moved inter-page using memmove(3). No attempt is made to optimise the number of elements to move - every element in the desired range is copied.

5. the necessary elements in the page are moved up one to make room for the element to be inserted

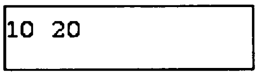
6. the element is inserted in the A-tree

Pictorially, a simple insertion of an element looks like:

Before:

```
Number of elements in virtual pages: 1


Adding element "10"
```

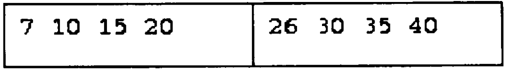
After:



```
Number of elements in virtual pages: 2
```

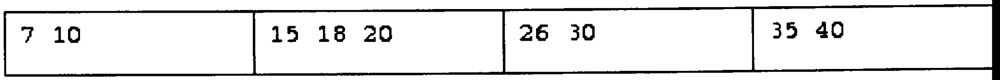
and an insertion which results in the growth of the atree looks like:

Before:



```
Number of elements in virtual pages: 4, 4


Insert element "18"
```

After:



```
Number of elements in virtual pages: 2, 3, 2, 2
```

4.5 Deletion

When deleting an element from the A-tree, the following steps take place:

1. the element to be deleted is located using a binary search

2. the storage allocated to the element is freed
3. if there is only one element in the virtual page, each virtual page is moved down one, and each element in the main array is moved down by "order" elements
4. if there is more than one elements in the virtual page, then move the elements in the virtual page down one

4.6 Illustration

Using the same data as provided in the B-tree illustration, and with an A-tree of order 4 to simulate the same growth characteristics as the A-tree, we obtain the following picture:

It is interesting to note some of the aspects of the growth of the A-tree when it is used in conjunction with the same data as used in the B-tree illustration.

- The number of virtual pages in the A-tree grows in powers of 2
- The data in the array, although sparse, is always in increasing order
- The array containing the number of elements in the virtual pages has not been shown, but is insignificant compared to the data stored - for an A-tree with 8 virtual pages, an array of 8 integers is needed
- The virtual page size can be tuned to the underlying page size of the virtual memory subsystem, to improve performance
- The oscillating search will find the nearest page with any room. This is a much cleaner way of finding space than the "merging leafs" method of B-tree space management.

5 Comparisons

5.1 A-tree and B-tree Comparison

If we contrast an A-tree with a B-tree, the following observations can be made:

- the B-tree provides a segmented, ordered list of pages in the same way that an A-tree does
- a B-tree (not a B+-tree or a B*-tree) will have a list of leaf elements along the "bottom" of the B-tree, and is similar to an A-tree
- binary searching is performed in much the same way that a B-tree accomplishes its searches

- A-tree code is much simpler than B-tree code - especially during B-tree deletion, when underflow of pages occurs, there are many corner cases to consider, and this usually results in large object codes. In comparison, the heart of A-tree insertion and deletion is a memmove(3) call, which will be hand-optimised assembly code on most modern machines and operating systems.

- with an A-tree, an ordered walk of the tree is possible, from head to tail, or from tail to head. Proximity searches can be performed with ease.

- all kinds of B-trees segment their dataspace into pages - this is done at the user-level, rather than at the hardware level, and the abstraction wastes time and space with modern computers

- A B-tree will typically store strings as data and keys. These strings need to be stored somewhere - for locality of reference, the strings are often stored in the page of the B-tree, which reduces the amount of space which can be allocated to elements, and which can make the number of elements in a B-tree page unpredictable. This is not a problem in B-trees, since the higher-order pages provide the indices through which the correct sub-page can be located, but it can be a problem if there are many insertions into a tree, and the strings need to be moved between pages.

- an A-tree has none of the possible "pathological insertion" problems which may occur with B-trees. An A-tree's size is governed by the number of elements in that tree.

5.2 A-tree and Hashing Comparison

When compared with a hashing, the following observations can be made:

- an A-tree will take longer to search for an element, since two binary searches are taking place using an A-tree, whilst hashing will involve calculating the hash value and then testing for its presence in the hash table, and any overflow chains, or subsequent location searches it may have to perform

- an A-tree will be as quick as hashing to insert elements into a table. In addition, hashing the key will lose information, and so no proximity searching can be performed using hashing

- both A-trees and hashing use sparse arrays to contain the elements

Future Work

There are a number of aspects of A-trees which would be good to investigate:

1. Some research must take place into the best method of growth for an A-tree - when an A-tree contains a small number of elements, doubling the size of the

A-tree is easy and efficient. When the number of elements in the A-tree grows large, the addition of a number of slots for elements in the A-tree is the best approach.

2. Extension to multi-dimensional searching, by way of an analogy of Guttman's R-trees would be a beneficial area of research. Guttman found that typical pages in an R-tree would only contain a small number of entries, and it would be interesting to contrast that with A-trees, where the information would also be held in the A-tree entry, but where fast memory copies and comparison functions could be used to good effect.
   To adapt R-tree methods of searching subsidiary keys to A-trees would involve solving the problem of "summary" information being held in higher-order pages further up the R-tree towards the root page.

Performance

The speed of searching in an A-tree is slower than when using a hashing scheme, but preserves order, which is useful in certain situations. The speed of searching an A-tree is roughly equivalent to that of searching a B-tree. Insertion and deletion from an A-tree are much simpler than the corresponding insertion and deletion from a B-tree.

Conclusions

The size of memory on today's machines, and the speed of the CPUs, busses and peripherals mean that we have come to rely on older data organisations which were conceived at the time that memory was scarce and CPUs were slow, and no virtual memory was available.

In 2004 and beyond, both disk and memory space are plentiful and inexpensive, and the constraints which existed when B-trees were first discovered are no longer in place.

In practice, the same characteristics of a B-tree apply to an A-tree - the segmented nature of its elements, which allow relatively easy insertion and deletion - are more efficiently encoded in an array, rather than as separate pages in a B-tree. Ordered searching and proximity searching are possible when using B-trees and A-trees, although not the B+-tree, which is the standard mechanism for B-tree implementation.

An A-tree will grow in size by doubling when inserting an element and the A-tree is full. The A-tree does not shrink - rather, its elements migrate towards the smaller end of the array.

A-trees and B-trees use the same two-step binary search mechanism.

If the 0-th element in a virtual page is unaccessed, it will be quite possible for the virtual memory subsystem within the operating system to reuse the memory allocated to the page, and very little overhead is associated with this.

In all, the A-tree is a much simpler alternative to a B-tree, has the same benefits, and is more performant.

References

[Bayer1972] [Acta Informatica (1972), 345 - 349]

[Intel1971] http://www.intel4004.com/

[IBM1971] http://www-1.ibm.com/ibm/history/history/year_1970.html

[IBM1970a] http://www.beagle-ears.com/lars/engineer/comphist/ibm360.htm

[Economist2004] http://economist.com/science/tq/displayStory.cfm?story_id=2724348

[Knuth1973] Sorting and Searching, Addison-Wesley, 1973

[NetBSD2004] http://www.netbsd.org/

[Comer1979] "The Ubiquitous Btree", ACM Computing Surveys, Vol 11, pp 121-137, June 1979.

[Wirth1976] Algorithms + Data Structures = Programs, Prentice-Hall, 1976

[Kernighan1976] The C Programming Language, Prentice-Hall

[FNV] http://www.isthe.com/chongo/tech/comp/fnv/

[Yigit2001] http://mail-index.netbsd.org/tech-kern/2001/11/28/0034.html

[Yigit2004] http://www.cs.yorku.ca/~oz/hash.html

[Jenkins2004] http://burtleburtle.net/bob/hash/

[gperf2004] http://www.gnu.org/software/gperf/gperf.html

[Guttman1984] "R-trees: a Dynamic Index Structure for Spacial Searching" in Proceedings of ACM SIGMOD, 1984

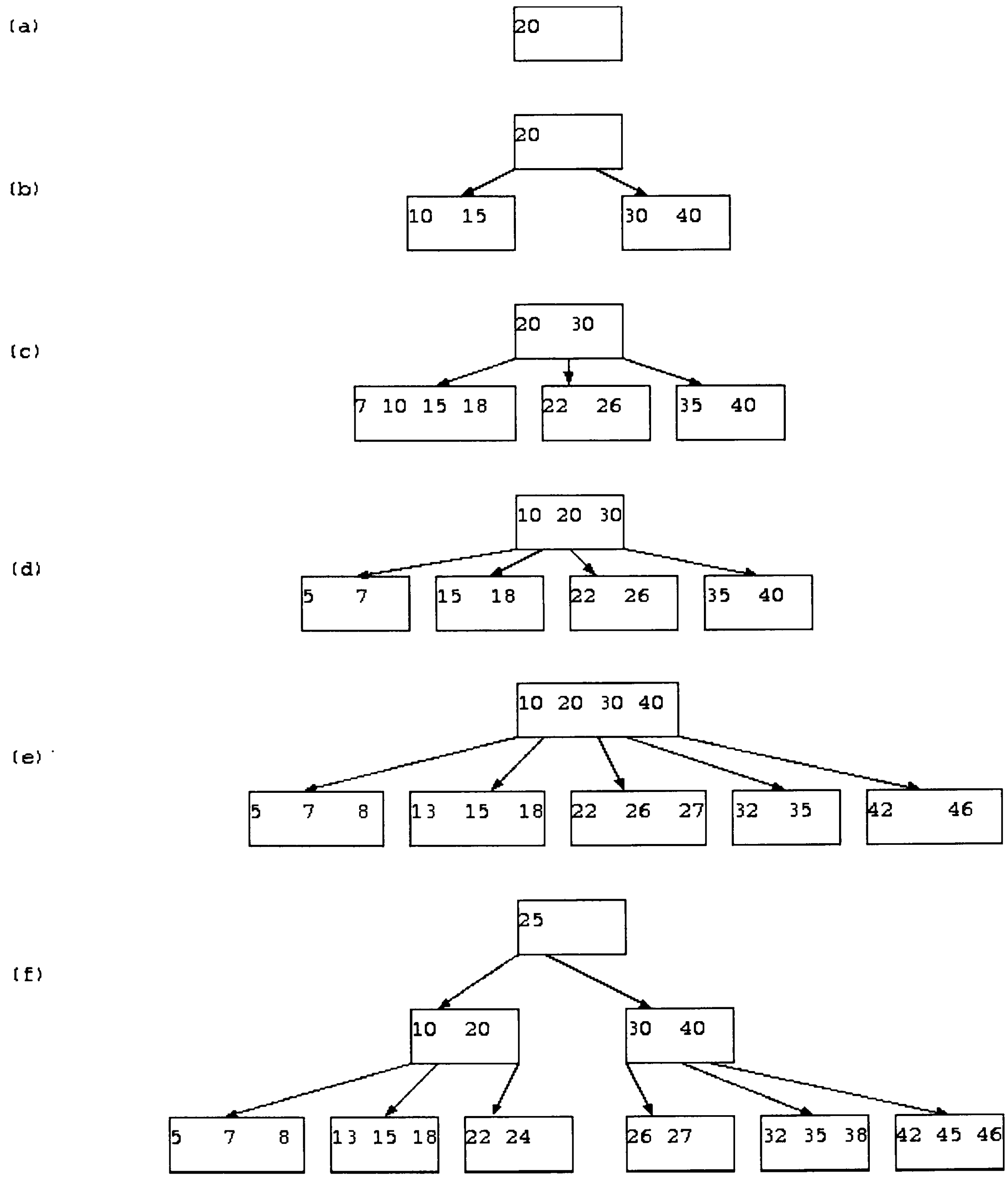


Figure 1: Addition to B-tree

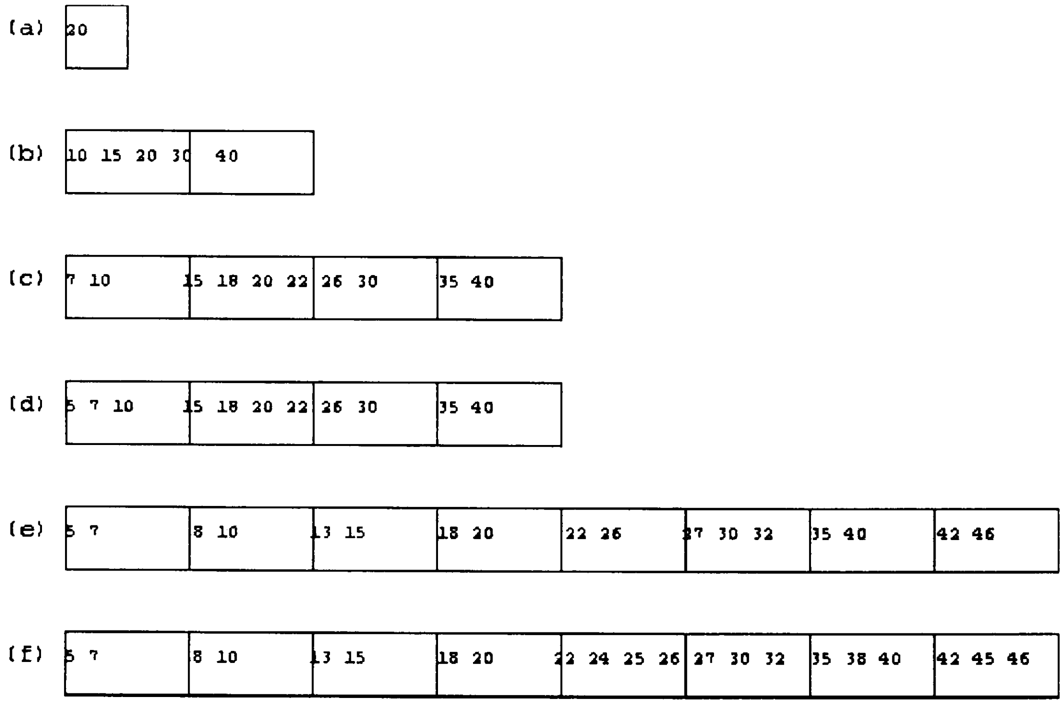


Figure 2: addition to A-tree

Track B Sunday

Notes:

Integrating Monitoring Data

...or how to get Management to approve your next gadget

By Christoph Sold

Since computers were invented, they were monitored: as a scarce resource computing power has always been a valuable thing. Even today, there is not nearly enough computing power available to solve all the numerical problems computers are easily applied to – let alone all those non- numerical problems you can imagine.

This paper deals with the requirements to further the development of monitoring and the effects this will have on software development and deployment. The effects of a concerted plan for monitoring will be shown, as well as how monitoring will help you to better predict your businesses needs.

Monitoring History

In the earliest days, arrays of *blinkenlights* were the tool of the day. Since then, many additional tools were invented . Some of them have survived the tides, others were swept into oblivion.

Although computer science has made a good deal of progress since then, the basic monitoring tools have not yet made as much progress. Many UNIX administrators still rely on basic tools like *ps*, *top*, or *netstat* as their primary tools of the trade. Some commercial UNIX systems have built upon those basic tools, e.g. Solaris *sar*, which allows for historical analysis, or AIX *Smitty*, which integrates configuration and monitoring. Commercial entities have developed various cross- platform monitoring tools such as *Borderware Patrol*, *Big Brother* (now from Quest Company), Lund *MetaView* or *Tivoli* (today a part of IBM). There are Open Source monitoring tools available, too, such as *Big Sister*, *Nagios*, or *NetSaint*.

Standards have been developed to help integrate monitoring data formats. The Simple Network Monitoring Protocol *(SNMP)* has been around for some time. The Application Response time Measurement *(ARM)* standard describes how monitoring data has to be logged and transmitted. Unfortunately, besides SNMP, there is no widespread standard to integrate various monitoring tools with each other. To make our life more miserable, even within SNMP there is no standard besides the numerical tree.

Why Monitor?

Like cars, defects can be fatal for your system. Thus, ***event (fault) monitoring*** needs to signal problems as timely as possible to minimize their effect. A well-known example of fault monitoring is the brake warning light in your car. If this thing lights up, you want to come slowly to a controlled halt to prevent dire consequences.

Performance monitoring continuously records system performance for deferred analyzation. This helps to predict system capacity needs. The fuel gauge in your car is a performace monitoring tool of sorts: it monitors your fuel efficiency.

Finally, ***service level monitoring*** watches for service level violations, giving a picture of your customers system perception to you. In modern cars, a built-in fault memory stores all faults. The car shop should analyze all the stored faults the next time you turn the car in.

Monitoring Tools Users

As in cars, the monitoring tools in the computing business have different customers.

Help desk workers as well as on-duty system administrators need ***event monitoring*** as the most important tool. Detecting problems before your customer calls is the killer application for event monitoring. A properly designed event monitoring system will help you to minimize the impact of your problem to the customer.

Performance monitoring helps your technical architects to predict your customers needs. Depending on system usage, it can be helpful for the system administration team to predict system utilization to some degree. Your sales people will also benefit, because knowing what your customer needs before he knows is always a good selling tool.

Service level monitoring is the tool of the trade for both your customer as well as for your sales and managerial people. Knowing the service level of your system enables your sales people to set the expectations of your customers into the right frame. In case of differences between you and your customers it is easy to check recorded service level data against your service level agreements.

Basic Monitoring Tools

Like any exact science, monitoring is based on repeatable, simple tools. Mathematics applied to this simple tools yields additional informa tion. To make your private crystal ball work, you have to pay extreme caution when implementing the basic building blocks.

Counting

Regardless of what you plan to measure, it basically comes down to counting. Be it the size of the average file in your hard disk, the packets down a pipe during a given period, or the seconds between two events, the basic process which has to be implemented is to count the basic unit of interest. This can be as simple as a variable incremented every time an event to measure occurs, or a simple routine counting the number of entities in a queue. There is one point you have to watch out for: Counts have to be *atomar*. If the number of entities changes while you count them, the count will be rendered invalid.

In addition, counts should be implemented with minimal impact. It is wise to invent a method to selectively disable counts not needed, such as a kernel variable, or to selectively enable only counts when needed such as in ipfw count rules.

Counts can be differentiated into *sum counts,* which integrate the number of units by adding them continuously until reset. Implementation is usually as simple as incrementing the counter each time the entity is detected.

Resetting the sum periodically at fixed intervals yields an *integrated count*, which helps to measure something over time. If you intend to record these, pay attention to record the count before the next reset occurs. Many scripts do exactly that to report on the traffic flow through *ipfw* packet filter. Mac OS X as well as Solaris provides *sar*, which records sys tem usage statistics in five- minute- increments.

A third type of a count is the *event record*, which records the time an event occurred, eventually along with details what exactly happened then. *Syslogd* does this type of event recording. ipfw can be instructed to generate log events for syslogd, as well as Apache generates this type of log, which allows to calculate floating averages. On the down side more system resources are needed to store event records. ipfw implements limits to prevent against DOS attacks for this reason.

Queue length counts the number of items in a line of items to be processed. Since queues usually grow as your systems lag behind, queue length usually is a very good indicator for system overload. Well known queue length counts include network protocol I/O queues.

If only the sum of any entity is relevant, *total counts* are the tool of the choice. To help analyze entities which would overwhelm a total count, sampling into the data pool helps to provide insight. *Sampled counts* have to be implemented carefully: either have all of your samples synchronized, or the values will not sum up properly.

Throughput

Adding the time dimension to basic building blocks yields in throughput values. As mentioned above, *integrated counts* simply reset after a fixed amount of time. Count the number of events during a sliding period within an event record to get a *sliding integrated count.* Queue length over time yields *average queue length* counts. For queues, it is interesting to watch *queue length difference* over a fixed period of time.

Statistical Tools

Basic counting in place and properly unified, it is simple to apply standard statistical indicators such as variance, arithmetic or geometric mean values. Classical statistics are best adapted to jobs on historical data. In addition, they tend to be optimized for print media: Display a lot of static information in a small place with high resolution. They tend to print number forests, which makes interpretation on the fly difficult. Graphics such as box plots, scatter graphs or quantile- quantile- plots help to visualize these approaches.

NIST has developed a new approach to analyze statistical data, along with a tool to plot the lot.[1] The fresh approach they used to apply to engineering statistics is rather helpful when slicing and dicing through monitoring data woods.

[1] *NIST/SEMATECH e-Handbook of Statistical Methods*, `http://www.itl.nist.gov/div898/handbook/`, 2004.
The tool is named *dataplot,* available at
`http://www.itl.nist.gov/div898/software/dataplot/homepage.htm`.

Basic Analysis Tools

Most monitoring tools are suited to provide real-time data to system administrators. They allow to glimpse into various aspects of your systems in real time. Others allow to analyze historically recorded data of one specific aspect of your systems.

The simplest tool is recorded basic monitoring data. To be of any help, all monitoring data should be recorded and archived. Mac OS X Server as well as Solaris monitor some essential performance data sets unconditionally during normal operation. For a short period of time (usually a few days), performance is sampled every five minutes and stored into the *sar* database. The database files are rotated daily, after a few days, they get overwritten. The *sar* utility allows to single out specific information as long as the database files are still intact.

Almost all descendants of *UNIX* still provide some file system quota mechanism along with the tools needed to monitor disk usage. While not recording anything when you‘re not over quota, they'll get verbose when you cross soft or hard quota limits. It‘s up to you to check disk quota logs against your policy.[2]

Network interfaces usually get recorded, too. netstat -i reports recorded network interface statistics. It‘s up to the kernel to record the packet counts. The interface netstat offers into your TCP/IP stack statistics resets the statistics every time you view them. Once viewed, the statistics are gone.

The tools mentioned above have some things in common: a back end records the data, while a front end lets you get at that data in human-readable form. Using back ends to record the data helps to minimize monitoring impact on your system. Calculate front end values only when they are needed.

A front end application like netstat, iostat or vmstat lets you access the recorded data when you wish to get at it. Back when network access was trusted in the days of the r-tools, secondary front ends to machine load were commonplace: xdm lets you display the load of remote servers will-

[2] Quotas can be a real PITA because often applications try to allocate files unknown to average Joe User. Ever tried to open a nicely compressed JPEG photo fresh from your digital camera with *GIMP*?

ing to let your X server connect. ruptime shows load data like the data displayed by w, ps, or top. All of those tools tap into the in-kernel load variables.

Unfortunately, even simple monitors like top differ in how they record monitoring information: on a machine with four CPUs, does a load of 1.14 mean there is one CPU working 100 %, another one is nearly idle, while two others are doing nothing at all, or are there all CPUs happily crunching numbers, while a little excess load waits for any CPU to become free? Both points of view are valid, and, unfortunately, both have been implemented in different operating systems. Remember to make sure things named the same are measured using the same algorithm before comparing the measured values.

Historically, minimal data was recorded to keep the impact on system performance as well as disk cost as low as possible. Today, there is no such thing than a history of machine loads. Recording Veritas Volume Manager managed file system performance for all your SAN volumes would be nice to analyze performance historically. Guess if it is available out of the box.

Monitor Displays

There is no such thing as a single monitor display fits all. Depending on the situation, you'll need different displays for different users in different situations. Having a ssh-to-SMS-Gateway may be a nice feature, but have you ever tried to recognize systat -vm output on a 40x4 mobile display?

In addition, different users of monitoring data want to view the data in completely different detail. Upper management is usually only interested if everything works within predefined parameters, while your system administrators need a lot more information to get the job done. Customers tend to ignore all monitoring data but end to end cycle times. Project managers are interested often only in the most damaging limit, which may change from minute to minute.

Defining Display Consumers

Before any monitor is designed, the users of the display have to be stated clearly. Each display has one primary consume as well as any number of secondary consumer groups. *Primary consumers* will need to examine the data in greater detail and at a higher frequency than *secondary consumers*.

Interactive consumers should be able to change the way data is displayed, while *passive consumers* need only to access a default data display. Sometimes this will include raw data exports to satisfy the need for interactive customers.

Defining the consumer groups helps to make clear who can ask for features in a new display, and -often more important who will have to live with what primary consumers defined. Make sure there is only one primary consumer to avoid conflicting interests.

Analyze Monitoring Baseline

After defining all your monitor user groups , define the baseline for all monitors. To do so, all monitoring data needs to be classified. For each monitor, the primary user requests resolution and retain period of monitored data. Thus, it will define how much storage this monitor needs while running. In most situations, an experienced systems administrator has to assist the customer during the definition phase.

The configuration of both the monitor as well as monitored object has to be recorded, too. Analyzing CPU loads while assuming the wrong hardware can produce results way out of the ballpark.

Data Compression Algorithms

Depending on your monitors needs, it may be possible to implement techniques to reduce space requirements of historical data.

Archiving trades access time for space. Database files, while fast, use notorious amounts of space to gain best access speed. Exporting and archiving older data saves space. A frog view reduces time resolution for older data. One simple algorithm to do this is to calculate mean values over a few measurement cycles and store them instead of the original data. Resolution Reduction transforms high resolution measurement values into low resolution, e.g. from float to single byte integer values.

Even if space is no concern, it may be wise to export historical data to keep access speeds of your live monitoring tools up to the job.

There is no way to reconstruct monitoring data once it has been deleted. Working with interpolated data will give wrong results, so be careful what you throw away.

Monitoring Data Interfaces

There is no such thing than universal software. Software, as well as hardware, will be replaced sooner or later, not teccessarily with something know to you today. Defining and documenting a data format as well as the interface between acquisition, back end, and front end will help you to retain valuable historical information when any piece in your monitoring systems chain changes.

Agree on Warning and Alert Levels

Monitors needing neither warnings nor alerts are rare things. Before starting to implement, collect all alert and warning level definitions from the customers. There may be monitors doing nothing else than warnings or alerts. Both system level and application level alerts need to be defined consistently .

Define an Priorization and Alarm Plan

Clearly define which level of alert needs what level of attention. Your alert plan should clearly state who has to be informed by whom. In case of disaster, there is no time to discuss who can help, too. Ideally, each monitor will be linked to its alarm plan.

For less dire situations, it is helpful for newcomers to have events already prioritized. Track all events, regardless how minor they seem. Having a record of lots of small problems with any device helps to convince management to replace it with something with lower maintainance cost.

Choose Presentation Formats

Depending on your data as well as your customers needs, choose presentation formats. Depending on your customers needs, this means there will be multiple monitors displaying the same data in different views in differing detail. Make sure each primary customer gets exactly the view she asked for, no matter how silly this may seem to you.

Implement the Data Acquisition Layer

After collecting all neccessary information shown above, coding can be started. Depending which tool you have choosen, this may be as simple as hacking your definitions into Tivoli through its console, or as complicated as patching away against packet filter code to collect raw data.

At this time, choose the monitoring method which gives you just the information you have been asked for at the cheapest cost.[3] Store the monitored data into the data store of your choice, and forget about it.

Remember to monitor the monitoring functions, too. Many a monitor has been grounded for too much false alarms, or for not reporting dire problems in time. Monitoring irregular cycles is especially difficult. Watch out for unusually long capture periods. If at all popssible, use an artificial heartbeat to make sure your monitor still works.

Integrate Information

There is no such thing than a luser-free SAN or an applicationless database. All of those things have their real-world use. Unfortunately, employees within departments tend to cling together, and easily view other departments as enemies. This often leads to internalization of information. Having a DBA Team in a full strength search for a performance leak caused by the Unix team moving the index table space from fast local hard disks to bigger, but slower NAS filers will lead to both: performance problems for your customer, as well as hostilities between teams.

Team people within various departments using the same hardware or software to prevent this kind of problems. Make sure both administrators as well as users of each subsystem know each other, as well as they should now about each others needs. To solve rising problems as soon as possible, shorten the line of communication between complimentary teams. To make sure both team know what the other speaks about both have to be able to look at each others monitoring displays.

Doing the Whiz: Predicting Future

Aside from the tools mentioned above there are a lot of tools available to predict the future of your systems: magic eight-balls, crystal balls, gut feeling, and throwing dices, to name a few. Comparing the cost of the former to the cost of well-implemented monitoring as described above makes those cheap gizmos even more attractive.

Applying modern statistical analyzation techniques to properly recorded monitoring data will lead to a prediction hit rate much higher than the average 50 percent by throwing coins. Choosing appropriate data collec-

[3] Depending on your needs, cost can be either minimal implementation cost or minimal impact on the monitored resource. Your call.

tion mechanisms will allow to apply the monitoring techniques in real-time.

This will put you into a much better picture to decide how your systems should be changed. Identified resources in excess of someones need may be better allocated to another use. Bottlenecks discovered may be solved before their impact punches through to the customer. Even better, analyzation of recorded incidents will show parts of your system with lots of small problems before they become big problems – and you'll have the paper to back your requests for new tools to widen the bottleneck.

Literature

NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/, 2004.

Toutenburg, Helge: Deskripvtive Statistik, Springer Berlin Heidelberg 2004

FreeSBIE - A code walkthrough and a case study

Matteo Riondato <rionda@gufi.org>, Massimiliano Stucchi <max@gufi.org>

October 11, 2004

1 Introduction

FreeSBIE is a LiveCD based on the FreeBSD operating system developed by an Italian group of people, and supported mainly by the Italian FreeBSD Users Group, known as GUFI. The first release (1.0) was released on April 15th, 2004, and has already been downloaded almost 20.000 times. But FreeSBIE is not only an ISO file you can download from one of our mirrors, it is also a way to create your very own, fully customized, LiveCD. In this paper, we're going to depict how the FreeSBIE provided by the team can accomplish the task of letting everybody become a Release Engineer on his own.

2 The GUFI

The Italian FreeBSD Users Group is a small group of people whose aim is to spread the word about FreeBSD in Italy. It is composed of 15 people making up the staff, working together to keep the mailing lists and the IRC channel in good state. The main event for the group is GUFICon, having place every year around the end of September. This time it was in Milan on October 2 and 3.

3 The beginning

The project was started by Davide "dave" D'amico, in order to fulfill the need for a LiveCD based on FreeBSD. There already was a solution around, created by a Brazilian group, but it wasn't as flexible as dave needed. He brought the project on and found help form Dario "SaturNero" Freni initially, and from the whole GUFI, in a couple of months.

4 The team

The development team is now composed by five people, with many obviously cooperating on the official mailing list or on the IRC channel. We're looking towards finding more people to work on some areas that need reoworking, or simply things that need to be created from scratch. By the way, the development is constantly evolving, following the changes in the original FreeBSD source tree, and adapting FreeSBIE to work with it. We're still missing some key features that will be later depicted, and maybe will do it in the next release.

5 The architecture

The power of FreeSBIE is given by its simple nature. The concept behind the downloadable ISO is that everyone can create his own LiveCD, including all the packages he or she wants, obviously according to the space available on the media being used. That's why FreeSBIE is normally considered as a set of scripts rather than an operating system by itself.

6 Behind the magic

There is no great magic behind how FreeSBIE works. In fact, it makes use of simple facilities provided by all the BSD systems, such as buildworld and installworld. What makes the core of the system is a simple line in the kernel configuration file, which sets the system root to a device other than the normal hard disk. Support to handle usage on non-master devices such as a slave CD-ROM has been recently added and will be present in FreeSBIE-1.1, along with other important fixes to known problems. The creation of a new distribution is done starting from a buildworld and installworld procedure, using a destination directory other than the default one. After building a complete environment, including a custom kernel, the only thing remaining left is to decide the packages to be included on the media being built.

7 As easy as 1,2,3

There have been lots of concerns about the difficulty of creating a LiveCD from scratch, but with FreeSBIE it is enough that anybody can try. It is only a matter of downloading the sources from CVS maybe this is sometimes the harder part and running *./freesbie* inside the downloaded directory. A set of dialogues will guide the user through the creation procedure. At the moment the interface is in english only, but there are plans to bring it to be i18n compatible.

8 Performance

The first test releases suffered from a media problem, which consisted in great slowlyness accross all the system. This was due to the fact that data was roughly stored on the CD, without any compression, and it had to be read according to this. With FreeSBIE-1.0 a Cloop filesystem was introduced to reduce the size of the */usr, /var* and */root* partitions, so that more data could be stored this meant more application could be bundled and read times could be improved. This cloop filesystem relies on the GEOM framework, so this introduction meant that FreeSBIE could not work anymore, as it was, on 4.X systems, where GEOM is not available, and so is the module written by Max Kohn.

9 Bundled applications

FreeSBIE is being distributed mainly as an ISO for people wanting to try it out, or just to give FreeBSD a try. There is a great deal of applications included in the official distribution, ranging from games, to browsers, to media players. Davide D'Amico, who released the first official version, tried to create a general purpose environment by using XFCE as desktop manager a good mean between GNOME and twm and putting in browsers such as Firefox and Dillo along with more expert level software such as ethereal or ntop. OpenOffice is not included, since it requires special work to be adapted to a FreeSBIE environment, and it needed so much space that it may have created serious problems. Anyway, the FreeSBIE scripts can already handle its addiction, but the part of code related to the issue is normally commented, and can simply be activated.

10 Childs

One of the projects that derive from FreeSBIE is called MiniBSD. It was developed by Gianmarco Giovannelli to adapt FreeSBIE to be run on embedded machines by installing it on flash cards. MiniBSD can fit on 16mb flash cards, and can bring all the options that FreeSBIE has into smaller systems. It is also included in a subdirectory in the FreeSBIE CVS repository.

There have also been rumors in the past about the need to have a frontend for IPFW or may even be IPF or, now PF included in FreeSBIE, for easier firewall configuration, but nobody ever started to effectively work on it.

11 Errors

Like any other project, FreeSBIE is no error-prone. The first release was shipped with an Italian heart. This meant that the localization of great part of the software is hardcoded to be in Italian, and can hardly be changed. This will be fixed in the next release.

We are also aware that the code that identifies video cards and sets up an XFree86Config file has problems identifying many cards, often falling to the use of the VESA driver. This can hardly be fixed, but the arrival of the XOrg system, and its autoconfiguration capabilities may improve version 1.1 for what concerns this aspect.

12 Future

As said before, FreeSBIE now only has the */usr, /var and /root* partitions on a Cloop-compressed filesystem, thus we're planning on compressing more of them to increase speed and available space for applications.

There are also other aspects we are going to delve into, such as creating official ISO's for architectures other than I386 - mainly AMD64 and SPARC64 -.

The part we all would like to focus on now is the most requested feature of all, an installer. Many linux LiveCD's already have the hability to be installed from the CD to a hard disk drive, but FreeSBIE is still missing it. This is a long standing issue that will be the primary focus in order to release FreeSBIE-2.0, but work has just started on this. Edson Brandi recently took over it, announcing that he already as some parts of it done. We are all looking forward to be able to install a FreeBSD system starting from a FreeSBIE CD.

Using Aspect-Oriented Programming to Run the NetBSD Kernel as a Server Personality on Top of the L4 Microkernel

Michael Engel

Dept. of Mathematics and Computer Science,
University of Marburg,
Hans-Meerwein-Str., D-35032 Marburg, Germany
engel@informatik.uni-marburg.de

October 31st, 2004

ABSTRACT

Compared to monolithic operating system kernels like the ones used in standard BSD-derived systems, second-generation microkernels like the Pistachio and Fiasco variants of L4 [1] provide a minimal operating environment running in privileged mode and delegating all other functionality to servers running as tasks in user mode on top of the microkernel. Running an adapted version of a standard Unix kernel on top of them can provide many advantages, such as checkpointing entire systems, running multiple personalities in parallel on one machine, controlling resource allocations or taking control of network connections using virtual devices.

First-generation microkernels were already used to provide Unix functionality on top of the microkernel. Projects using the Mach microkernel [2] as a basis include Mach+Lites [3], providing a 4.4BSD personality on top of Mach 3 and MkLinux [4], a Linux 2.2 kernel personality running on OSF/Mach. Since the microkernel takes control of the hardware, the kernel source code has to be adapted to run on top of the microkernel. Access to interrupts, page tables, task and thread creation have to be relegated to the microkernel and replaced by interprocess communication calls between the kernel personality and the microkernel.

In this paper, we present our approach to adapting NetBSD to run on top of L4. L4 already hosts a variety of operating systems, e.g. L4Linux[5] is a modification of a Linux 2.4 kernel to run on top of L4. This adaptation required many manual changes in the Linux kernel source tree. The basic idea of our approach for adapting NetBSD is to simplify parts of the adaptation process by using aspect-oriented programming technologies to replace components of the kernel as well as create the necessary IPC protocols between the microkernel and the NetBSD personality. The achieved performance is compared to L4Linux running on top of L4 as well as Lites running on top of the Mach microkernel.

References

[1] J. Liedtke. Toward Real ?*micro*-kernels, Communications of the ACM, 39(9), pp. 70-77, September 1996

[2] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones. Mach: A System Software Kernel, roceedings of the 34th Computer Society International Conference COMPCON 89, February 1989.

[3] J. Helander. Unix under Mach – The LITES Server. Master Thesis, Helsinki University of Technology, 1994.

[4] F. Barbou des Places. Linux on the OSF Mach3 Microkernel, First Conference on Freely Redistributable Software, Cambridge, MA, 1996.

[5] H. Haertig, M. Hohmuth, J. Wolter. Taming Linux, Proceedings of PART '98.

Handling FreeBSD's latest firewall semantics and frameworks

by ADRIAN PENIŞOARĂ*

Abstract

Despite having a powerful firewall service in its base system since early versions, known as the IPFW facility, FreeBSD has imported over time another popular packet filtering framework, the IPF system, and now has just imported the new kid on the packet filtering block, namely the open PF framework raised on the OpenBSD grounds.

Having three packet filtering frameworks at your feet might sound delightful, but actually choosing one of them or making them cooperate might give you the shivers. Each framework has its strengths and weaknesses and often has it's own different idea on how to handle certain aspects.

Note: an extended version of this article will appear on the conference's website.

1 Introduction

1.1 The players

For quite some time the FreeBSD users have been stuck with the traditional IPFW firewall service which had its strengths and weaknesses. The last years though have brought the project two new packet filtering services: Darren Reed's IPFilter and OpenBSD's Packet Filter.

The IPFW framework appeared in FreeBSD in the early 2.0 releases and has been extended and reworked over time. The `dummynet(4)` traffic shaping module has been introduced in FreeBSD 2.2.8 and statefull extensions have been committed around FreeBSD 4.0. In summer 2002 the IPFW engine has seen a major overhaul under the "ipfw2" codename.

The IPFilter framework is the work of Darren Reed, a packet filtering service that was designed to be portable across several Unix distributions. It offers advanced packet filtering and NAT services out of the box; some of its strengths are the packet authentication, statefull packet tracking and hierarchical rulesets.

*You can contact the author at *ady@freebsd.ady.ro* or see his webpage on *www.ady.ro*

The Packet Filter framework is an offspring of the IPFilter package; it was written by the OpenBSD folks as an alternative to Darren Reed's IPFilter when they learnt that the licensing terms were not open enough to permit uncontrolled derivative works. It has all that IPFilter offered and even a lot more; its only drawback may currently be that it's hierarchical rulesets feature doesn't allow nesting. Lately it has been adapted to work with the ALTQ traffic shaping framework with which combined they offer network QoS[1] services. Until recently the PF framework was available only as a port for FreeBSD 5.x, but this summer the framework has been imported in the FreeBSD-CURRENT branch.

1.2 Common issues

One important aspect that you always need to take into account is that the forwarded packets enter the kernel twice, first when they are received inbound and second when they leave the system outbound. This means that these packets will travel through the firewall's rules twice and you will most probably want to process them once. One workaround for this issue is using the statefull extensions; make sure though that you have enough resources to scale up to your traffic flow needs.

Firewalls have a default policy regarding packet handling when no user configured rule has been matched. This policy may be either "open", which means the packets will be permitted to travel the firewall by default, or "closed", in which case the packets will be dropped by default. This policy can be adjusted either from the kernel configuration file at compile time or at run time through a sysctl or a firewall rule. Most users will probably leave their firewall "open" for usability reasons but paranoid users will always choose to "close" it.

Be very careful when reconfiguring your firewall remotely! Do not forget that your remote session is basically a network transmission and any mistake may result in immediate and definitive isolation from the system you were reconfiguring. Remember that any command output is also sent through the network and if you loose the network connection then the command chain you launched may not be entirely processed; try redirecting the output or make the utilities run "quietly". For these reasons you should always try to do your firewall reconfiguration from the systems' console and not through a networked session.

And last, but not least, make sure you have enough network buffers – otherwise you may experience packet drop problems, usually detectable locally on the machine from the "no buffer space" errors. Tune up your NMBCLUSTERS paying attention to the `tuning(7)` manual page.

1.3 "Hello world !"

Before going deeper into the details let's have a taste of what we are offered with an example firewall configuration for each of the three frameworks.

[1] Quality of Service – a framework which permits guaranteed service provisioning

Let's suppose we have a machine with the IP 80.0.0.1 and we want to make sure that only two systems with IPs 190.0.0.1 and 190.0.0.2 may connect through SSH to our machine. We will assume that the SSH service runs at the default port 22 and no specific default firewall policy.

```
add 1000 allow tcp from 190.0.0.1,190.0.0.2 to 80.0.0.1 22 in
add 1100 allow tcp from 80.0.0.1 22 to 190.0.0.1,190.0.0.2 out
add 2000 deny tcp from any to 80.0.0.1 22
```

Figure 1: IPFW ruleset example

In figure 1 we see a sample IPFW ruleset that either needs to be loaded with "`ipfw -f <file>`" or you can load it one rule at a time by running `ipfw` with each rule as the argument. Observe that we specified the "in" and "out" keywords to exactly match the incoming and outgoing packets and that we need to catch both the incoming and outgoing packets.

```
pass  in  quick proto tcp from 190.0.0.1 to 80.0.0.1 port = 22
pass  in  quick proto tcp from 190.0.0.2 to 80.0.0.1 port = 22
pass  out quick proto tcp from 80.0.0.1 port = 22 to 190.0.0.1
pass  out quick proto tcp from 80.0.0.1 port = 22 to 190.0.0.2
block in  quick proto tcp from any to 80.0.0.1 port = 22
```

Figure 2: IPFilter ruleset example

The same configuration for the IPFilter framework is presented in figure 2 and it can be loaded with "`ipf -f <file>`". Observe that the rules are unnumbered and that the packet does not exit the firewall when matching a rule except when told so through the "quick" keyword. While the "in" and "out" keywords were optional for IPFW, they are required in the IPFilter rules. Unfortunately one cannot specify multiple distinct addresses on the same rule except when they can be aggregated into one network subnet.

```
me = "80.0.0.1"
my_hosts = "{ 190.0.0.1, 190.0.0.2 }"

pass  in  quick proto tcp from $my_hosts to $me port 22
pass  out quick proto tcp from $me port 22 to $my_hosts
block in  quick proto tcp from any to $me port 22
```

Figure 3: PF ruleset example

Looking at figure 3 you probably started wandering if are a Perl class by now, but don't worry, we are still on track with the PF framework; while the other firewall frameworks may use macros in combination with an external pre-processor, the PF framework has internal support for them. If you replace the macro definitions you will easily recognise the same rule format from the IPFilter; however, PF holds many more surprises for us. These rules should be loaded with "`pfctl -f <file>`".

2 Statefull firewalling

2.1 Why do we need it

You saw in the previous section that building even a simple ruleset can be troublesome. One of the main problems is that we need to keep track of both the incoming and the outgoing packets. There is also a problem of scalability: when the rulesets grow very large and complex and the traffic flowing through the machine is big the time it takes for one packet to traverse the entire ruleset becomes a sensitive issue.

The statefull extension has started from a simple idea: one needs to check only the first packet of a data connection to determine whether the entire transmission will be allowed or not. If this packet is to be passed on then this connection will be marked in a dynamic firewall table where it will stay until this connection will be terminated or it will timeout.

We can already discern that this statefull extension is applicable for those protocols that can have a state recorded in time; it is the case of TCP mainly, but also UDP and ICMP with some approximations. The firewall will recognise the TCP connection states from the flags header, usually the SYN, FIN, RST and ACK flags. Some firewalls will even allow you to specify which flags may insert a new connection state.

2.2 How can it be done

Each of the three firewall frameworks has statefull firewalling extensions. We will rewrite the previous examples using these statefull extensions.

```
add 1000 check-state
add 1100 allow tcp from 190.0.0.1,190.0.0.2 to 80.0.0.1 22 setup \
                   keep-state in
add 2000 deny tcp from any to 80.0.0.1 22
```

Figure 4: IPFW statefull example

As you can see, IPFW needs a special rule to make him check the dynamic states table; the "setup" keyword will catch only packets that initiate a TCP connection. A similar mechanism exists in both IPFilter and PF but is more refined: the "flags" keyword permits specifying which flag bits need to be checked from a predefined set in order to match the rule. Commonly used flags are S(YN), A(CK) and R(ST).

```
pass  in  quick proto tcp from 190.0.0.1 to 80.0.0.1 port = 22 \
                flags S/SA keep state
pass  in  quick proto tcp from 190.0.0.2 to 80.0.0.1 port = 22 \
                flags S/SA keep state
block in  quick proto tcp from any to 80.0.0.1 port = 22
```

Figure 5: IPFilter statefull example

```
me = "80.0.0.1"
my_hosts = "{ 190.0.0.1, 190.0.0.2 }"

pass  in  quick proto tcp from $my_hosts to $me port 22 \
                flags S/SA keep state
block in  quick proto tcp from any to $me port 22
```

Figure 6: PF statefull example

Querying the dynamic tables is possible for all frameworks with commands such as "`ipfw pipe show`", "`ipfstat -sl`" and "`pfctl -s state`".

One big advantage of the statefull extensions is that the search time for packets is dramatically reduced. For example, in the PF case, the search time is reduced from an $O(n)$ search in the entire ruleset to an $O(\log_2 n)$ binary tree search in the states table.

2.3 The drawbacks

Not everything is perfect; this is true for the statefull technique too. Although it helps a lot optimizing and simplifying the firewall rulesets, it comes with a warning: watch out for resource consumption. The tables used for dynamic state tracking have a finite limit and DoS attack may be possible by flooding the firewall with connection attempts (usually SYN packets).

This is why one needs to periodically check the dynamic tables status and adjust various sysctls according to the kind of traffic that is passing through the firewall.

3 Traffic Shaping

3.1 Foreword

Please note that traffic shaping is not equivalent with QoS: the traffic shapers in general only limit or enforce network parameters, while the QoS frameworks must also guarantee these network parameters in a shared environment. For example, a traffic shaper will usually only limit bandwidth, but the QoS framework must also guarantee a minimal contracted bandwidth.

3.2 IPFW with Dummynet

The Dummynet module offers two objects which can help the administrator to shape the network parameters: the pipe and the queue. The module is not compiled in the kernel by default so you will need to recompile it with the following options:

```
options IPFIREWALL
options DUMMYNET
options HZ=1000        # not required but strongly recommended
```

Basically a pipe emulates a network link with a given bandwidth, propagation delay, queue size and packet loss rate. The queue is used in conjunction with the WF2Q+[2] policy which permits associating weights to network flows sharing the same bandwidth.

```
pipe 1 config bw 256Kbit/s
add 60000 pipe 1 ip from any to 192.168.0.0/24 out
```

Figure 7: Simple LAN clients limitation with IPFW pipes

Let's review in figure 7 a simple case in which we limit to 256Kbps the download of the LAN clients on the 192.168.0.0/24 network segment behind our firewall. As you can see we need to separately configure the pipe's emulated link characteristics and a firewall rule which will catch packets and place them in the pipe's network queue. Notice the "out" keyword which specifies that only outgoing packets will be caught; if we would have forgotten this keyword then the packets would have been processed twice and the clients would would get only half of the specified bandwidth limit.

Contrary to the default IPFW rules behaviour, once a packet matches a pipe rule it will exit the firewall; that's why these rules should be numbered close to the end of the firewall (65535 is the last "default policy" rule). This behaviour can be controlled through the *net.inet.ip.fw.one_pass* sysctl.

The Dummynet module offers another useful feature: dynamic pipes. Let's rewrite the above example considering that that each client from the 192.168.0.0/24 class should have its download limited to 64Kbps.

```
pipe 1 config bw 64Kbit/s buckets 256 mask dst-ip 0x000000ff
add 60000 pipe 1 ip from any to 192.168.0.0/24 out
```

Figure 8: Limiting per client with IPFW dynamic pipes

As you see in figure 8, each packet caught by the pipe rule is checked against the destination IP mask so that for each separate IP from the 192.168.0.X segment will have a separate dynamic rule cloned from the generic specified pipe. The "buckets" keyword

[2] Worst-case Fair Weighted Fair Queueing, an efficient variant of the WFQ policy

```
pipe 1 config bw 256Kbit/s
queue 10 config pipe 1 weight 10 mask dst-ip 0x000000ff
add 60000 queue 10 ip from any to 192.168.0.0/24 out
```

Figure 9: Weighted bandwidth sharing with IPFW queues

increases the number of hash table entries to meet the maximum possible number of separate dynamic rules.

Even further refinements can be done: the queue object helps us "connect" a set of flows to the same pipe in order to share its bandwidth proportionally to their weights. Please notice that these weights are not priorities so even a lower weight flow is guaranteed to get its bandwidth share even if other higher weighted flows have backlogged packets.

Figure 9 presents a configuration that uses dynamic queues cloned from a generic queue which are all connected to the same pipe so they share the same 256Kbps bandwidth. Because the weights are all the same each client will be given an equal share from the total bandwidth.

The configuration can be refined even further using the RED[3] queue management algorithm which will change the default "drop from the tail" policy and improves the TCP decongestion response time.

3.3 PF with ALTQ

Although harder to install and configure than Dummynet, the PF/ALTQ combo offers a superior class of service. The ALTQ[4] module offers a QoS traffic management framework with various packet scheduling algorithms (currently only CBQ[5], HFSC[6] and PRIQ[7] are supported).

FreeBSD support for the the PF and ALTQ frameworks is available only on the 5.x/6.x platforms. These frameworks have already been imported in the -CURRENT branch; if your machine is not running -CURRENT then you may use the `security/pf` port.

You will need in the kernel configuration file the options exemplified in figure 10 (the "`device pf*`" lines are not not needed if PF is loaded as a module, e.g. as installed from the `security/pf port`):

When it comes to configuring the firewall rules, the above mentioned packet schedulers need to be attached to an interface from which an hierarchical tree of classes will

[3] Random Early Detection – packets are randomly dropped when the queue is about to become full

[4] Alternate Queueing – an extensible BSD QoS framework, see `http://www.csl.sony.co.jp/person/kjc/kjc/software.html`

[5] Class Based Queueing

[6] Hierarchical Fair Service Curve

[7] Priority Queueing

```
# PF support
options PFIL_HOOKS
device pf
device pflog
device pfsync

# ALTQ support
options ALTQ
options ALTQ_CBQ
options ALTQ_HFSC
options ALTQ_PRIQ
options ALTQ_RED
#options ALTQ_NOPCC # only for SMP kernels
options HZ=1000     # not required but strongly recommended
```

Figure 10: Kernel configuration file for PF and ALTQ support

be organised. Packets will be sorted out in these classes based on normal PF rules bearing the "queue" keyword.

This mean you need to specify on which interface(s) these schedulers will be activated; traffic shaping can only be done for packets leaving the system through these interfaces. This is not always convenient but this is a basic requirement of the scheduling algorithms.

Figure 11 presents a configuration file excerpt for a typical situation: an office with an 1Mbps network connection (through the fxp0 interface) who uses web, mail and SSH services. The "default" class is used to "capture" all traffic that does not fall in other classes. The bandwidth will be shared among these four classes but each class will have a guaranteed share (70% for the web traffic, 10% for the mail traffic, 15% for SSH connections and 5% for what remains). Observe that the sum of the child class percentages need to make 100%. As a bonus, the CBQ scheduler permits that a class who has the "borrow" keyword in its definition to borrow bandwidth from the parent class if there is any available unused bandwidth left.

Each of the main classes may have other subsequent classes; it is the case of the "web" and "ssh" classes. Notice that the "acct" class may only use up to 40% the web bandwidth, while the "sales" class may use more than its 70% guaranteed bandwidth share, but only if the "acct" class is not entirely using his 40% share.

The "ssh" class is made up of two "interactive" and bulk" subclasses; they do not have a percentage share specified but priorities only which will make the scheduler to always choose to serve the interactive class first.

In the second section of the file we match the packets going out through the fxp0 interface (remember, the QoS schedulers can only shape outgoing traffic) to their respective class. Packet matching is done on a "last matched rule" base, putting by

```
sales_dept = "10.0.1.0/24"
acct_dept = "10.0.2.0/24"

altq on fxp0 cbq bandwidth 1Mb queue { default, web, mail, ssh }
queue default bandwidth 5% cbq(default)
queue web bandwidth 70% priority 5 cbq(borrow red) { sales, acct }
queue  sales bandwidth 60% cbq(borrow)
queue  acct bandwidth 40%
queue mail bandwidth 10% priority 0 cbq(borrow ecn)
queue ssh bandwidth 15% priority 3 { ssh_interactive, ssh_bulk }
queue  ssh_interactive priority 7
queue  ssh_bulk priority 0

pass out on fxp0 all queue default
pass out on fxp0 proto tcp from $sales_dept to any port 80 \
                 keep state queue sales
pass out on fxp0 proto tcp from $acct_dept to any port 80 \
                 keep state queue acct
pass out on fxp0 proto tcp from any to any port 22 \
                 keep state queue(ssh_bulk, ssh_interactive)
pass out on fxp0 proto tcp from any to any port 25 \
                 keep state queue sales
```

Figure 11: PF/ALTQ QoS example

default all packets in the "default" class.

For the SSH classes we used a tuple for the "queue" keyword where normally we should have specified a single class. The second class will be used for the matched packets which have a "low-delay" TOS[8] or they are TCP ACK packets with no data payload.

[8] Type Of Service

The conference team gives a big "Thank You" to

- all companies and organisations that supported the production of conference materials by buying ads or sponsoring the conference directly
- all speakers that helped with their talks to create an interesting conference program
- all employees at punkt.de GmbH involved in preparations for the conference
- all the numerous people that helped with preparations and realisation of the conference and worked for making the conference a success.