AsiaBSDCon 2008
Proceedings

March 27-30, 2008

Tokyo, Japan

Published in Japan, March 2008

In Memory of Jun-ichiro “itojun” Hagino

Our friend and colleague Jun-ichiro “itojun” Hagino (a BSD hacker famous for IPv6 implementation, and CTO of ipv6samurais.com) was working as a member of our program committee, but he passed away on October 29th 2007. Itojun was a valued member of the BSD community both for his technical and personal contributions to various projects over his career. We who are working on AsiaBSDCon would like to express our condolences to Itojun’s family and friends and also to dedicate this year’s conference to his memory.

INDEX

FreeBSD on the Desktop

Who am I?

Matt Olander

Walnut Creek CD-ROM

BSDi

iXsystems

FreeBSD Marketing Team

PC-BSD Project Management

PC-BSD

Why?

PC-BSD

PC-BSD

What is PC-BSD?

It is FreeBSD (with hugs!)

*Graphical Installer
* Integrated KDE
* Xorg Config
*Wireless tool
* Other useful GUI tools

*PBI - Installation Method

Push Button Installer
Self-Contained Package Management
Freedom from Dependencies

Target Market

Windows Users

Linux Users

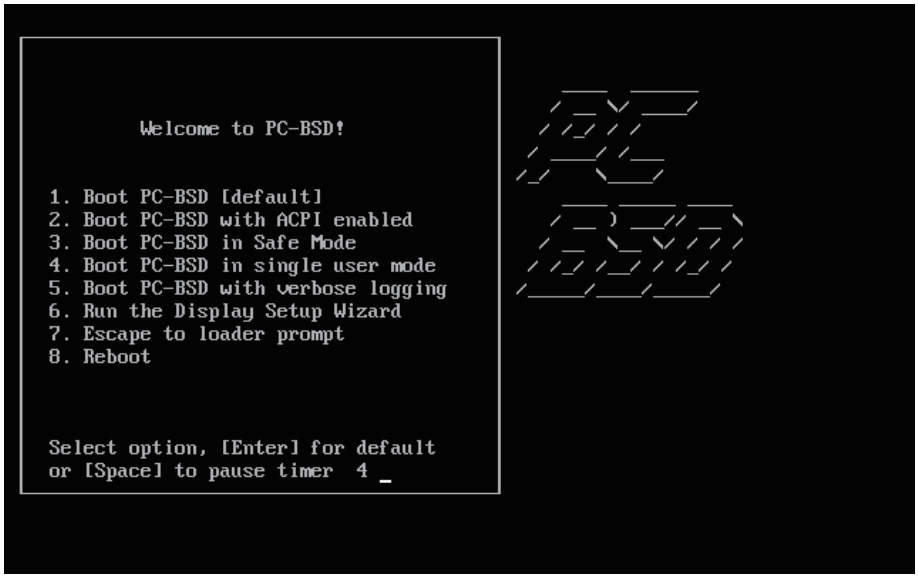
Lazy FreeBSD users :)

PC-BSD 1.5

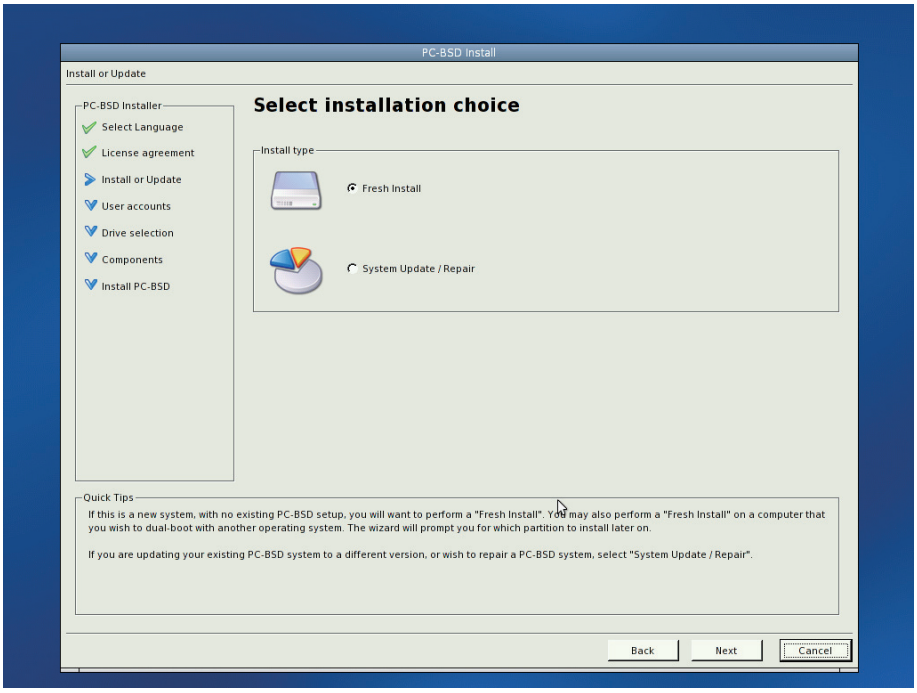
* Xorg 7.3
* KDE 3.5.8
* FreeBSD 6.3 Release
* NEW System Updater tool
* Improvements to WiFi tool
* Improvements to the PBI Removal tool
* NEW sound detection program! Uses XML backend to identify and load modules
* NEW amd64 build of 1.5

Installation

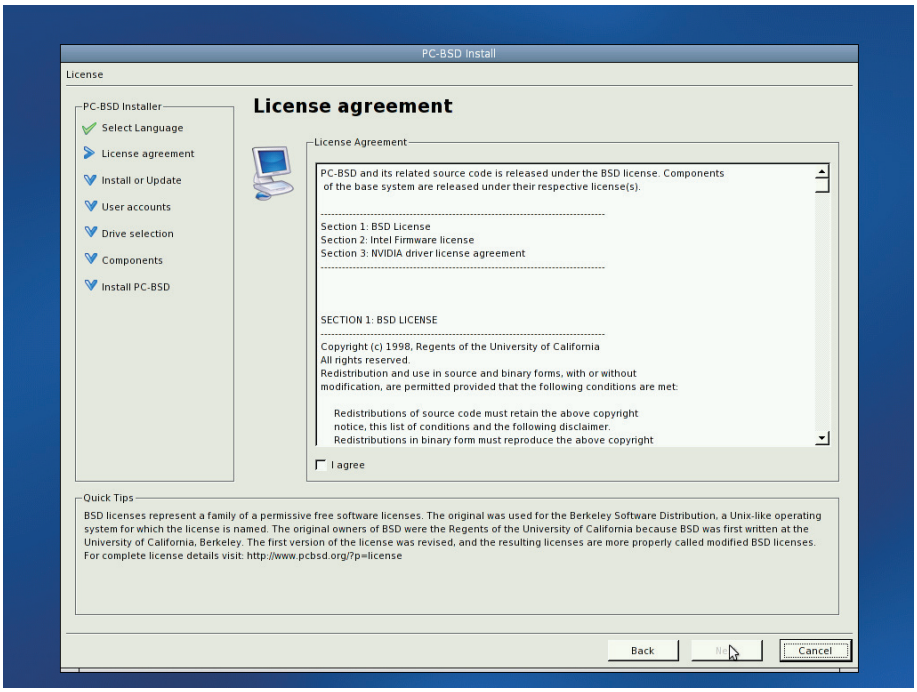
Booting CD



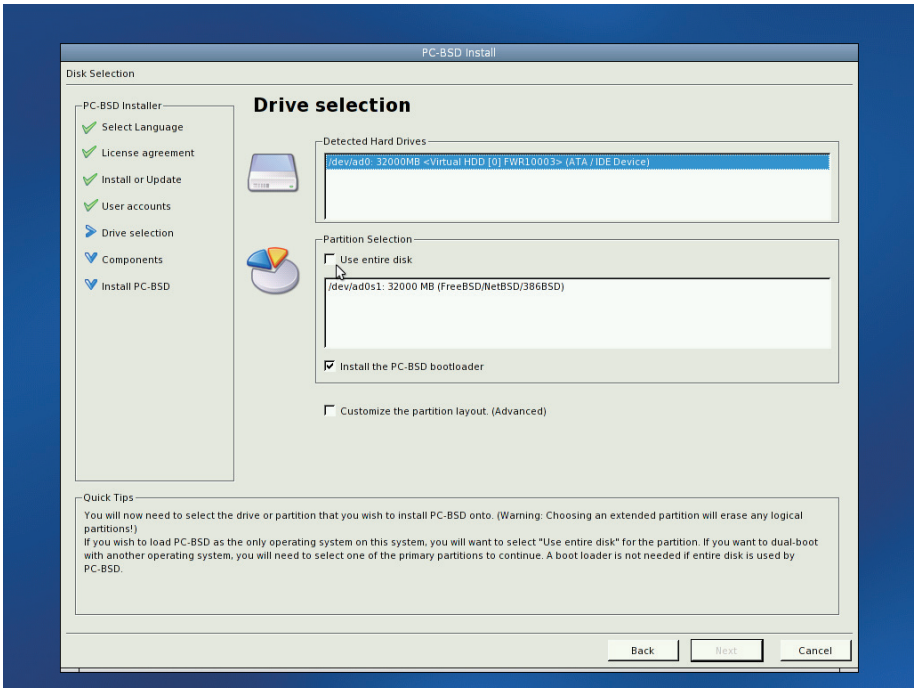
Install or Upgrade



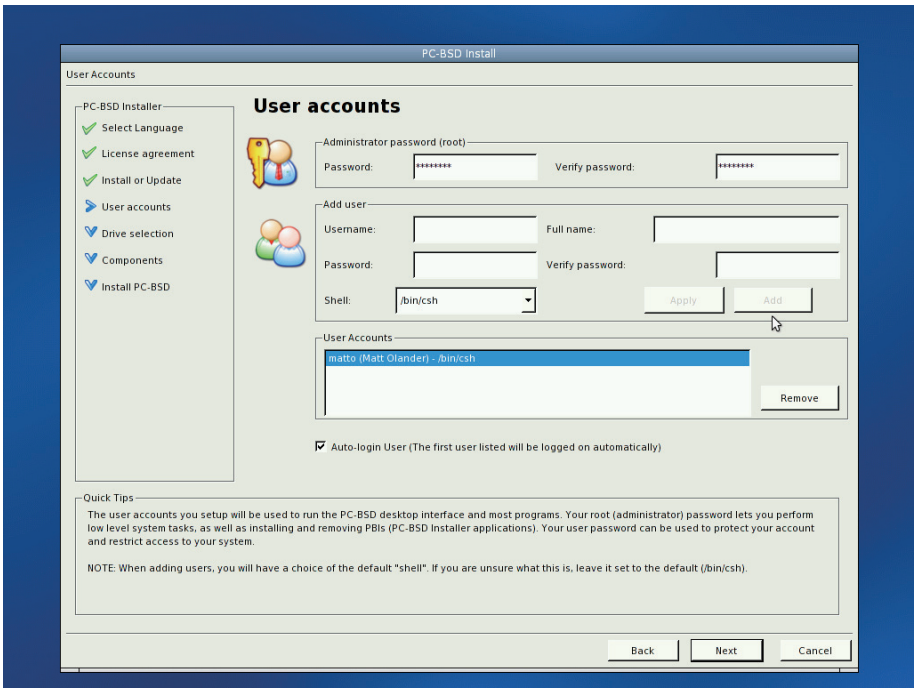
License



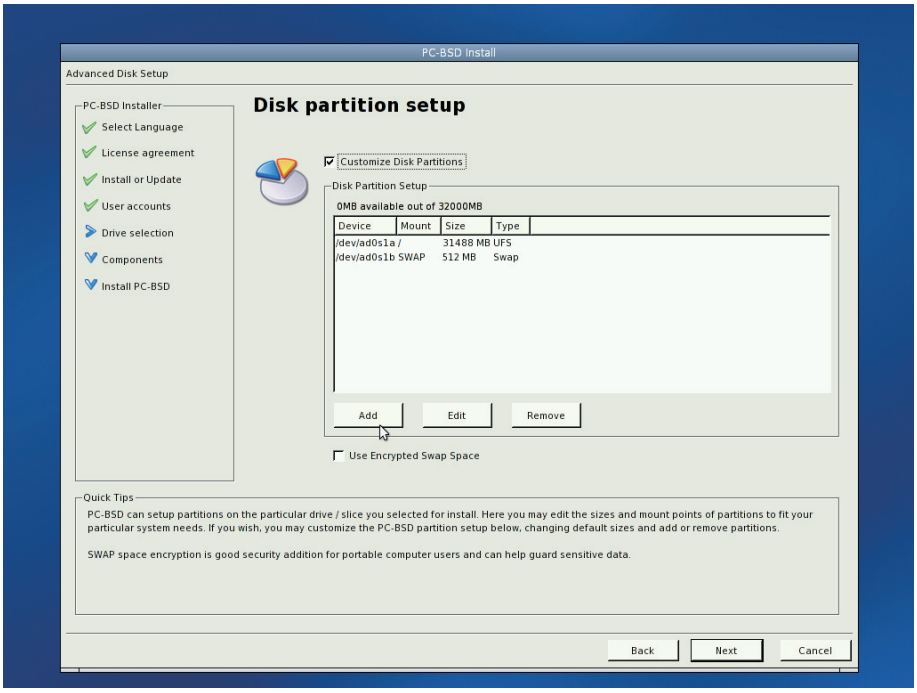
Drive Selection



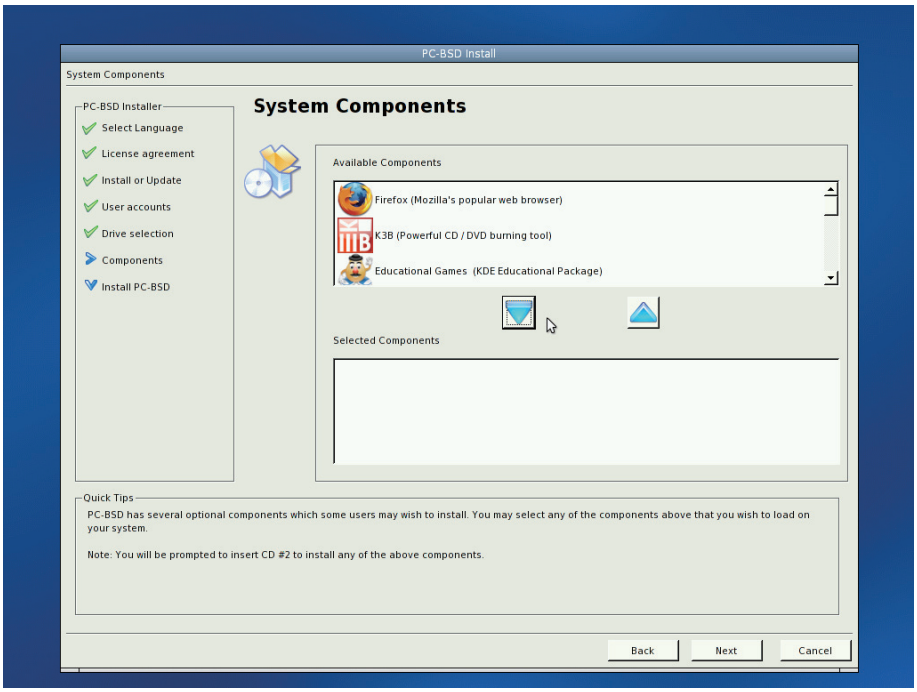
User Accounts



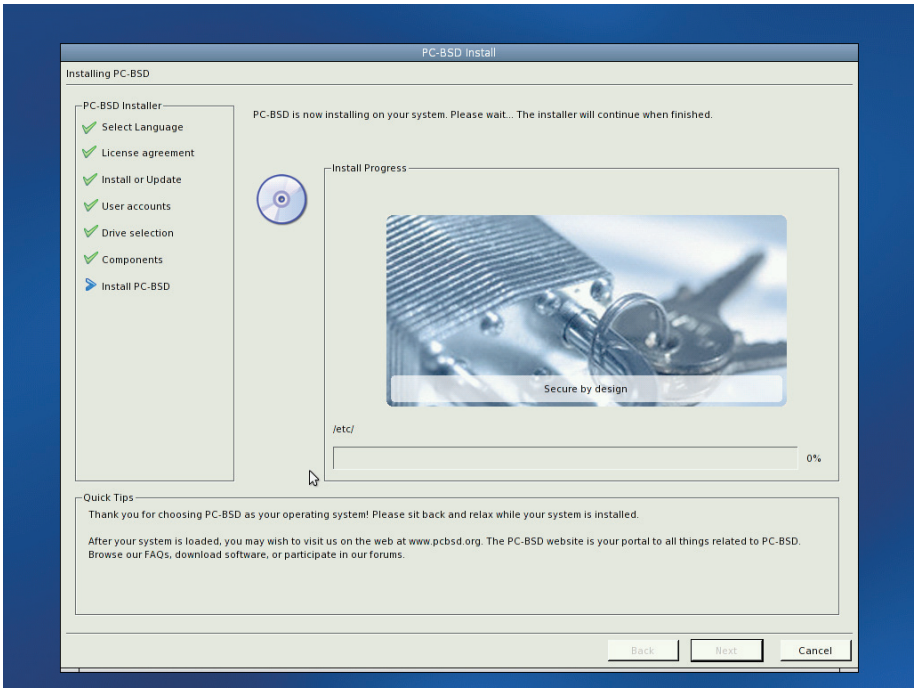
Partitioning



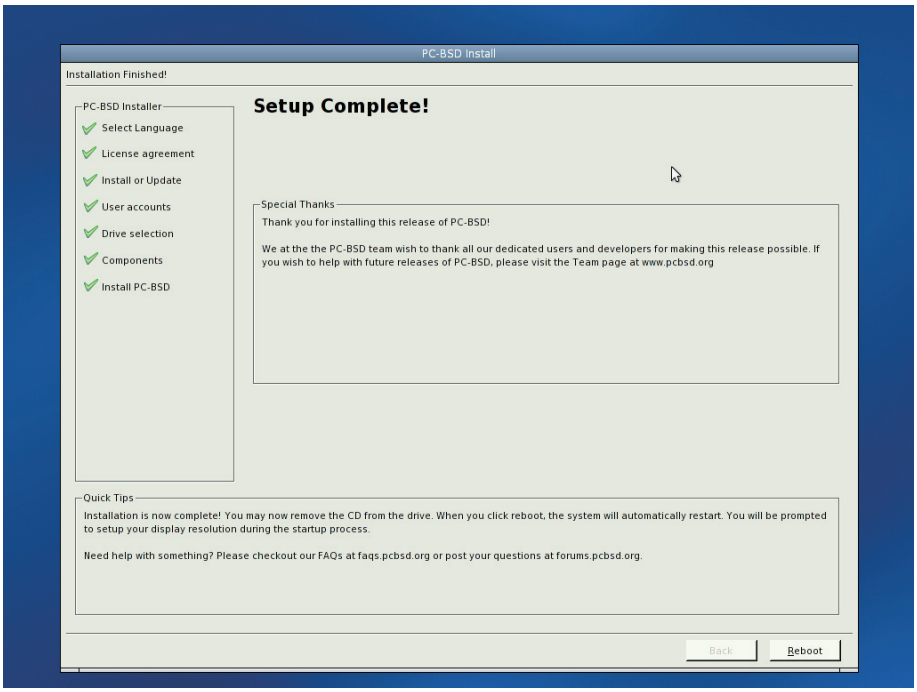
Components



Final Install

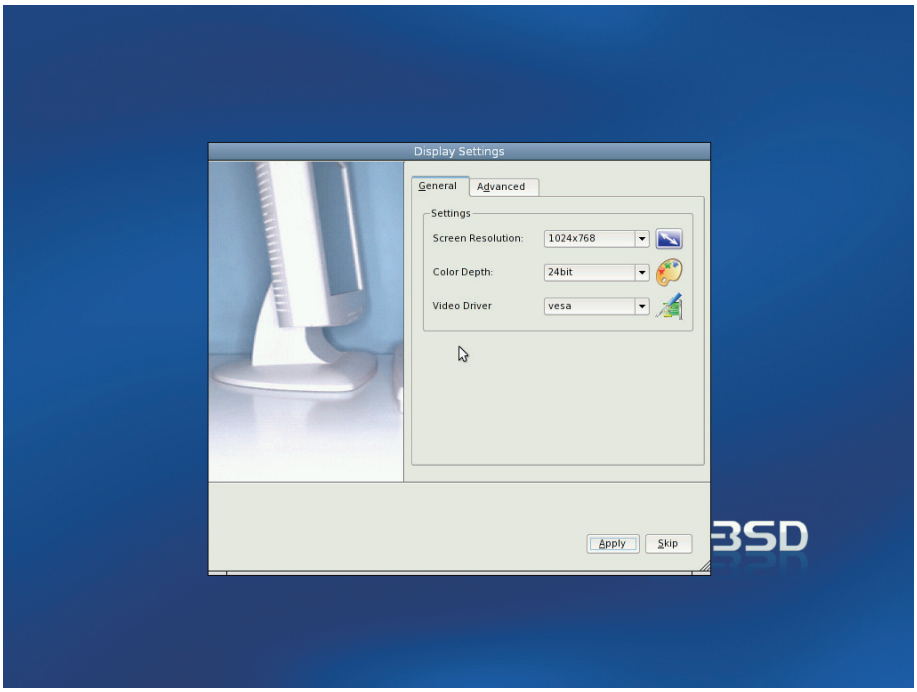


Completion

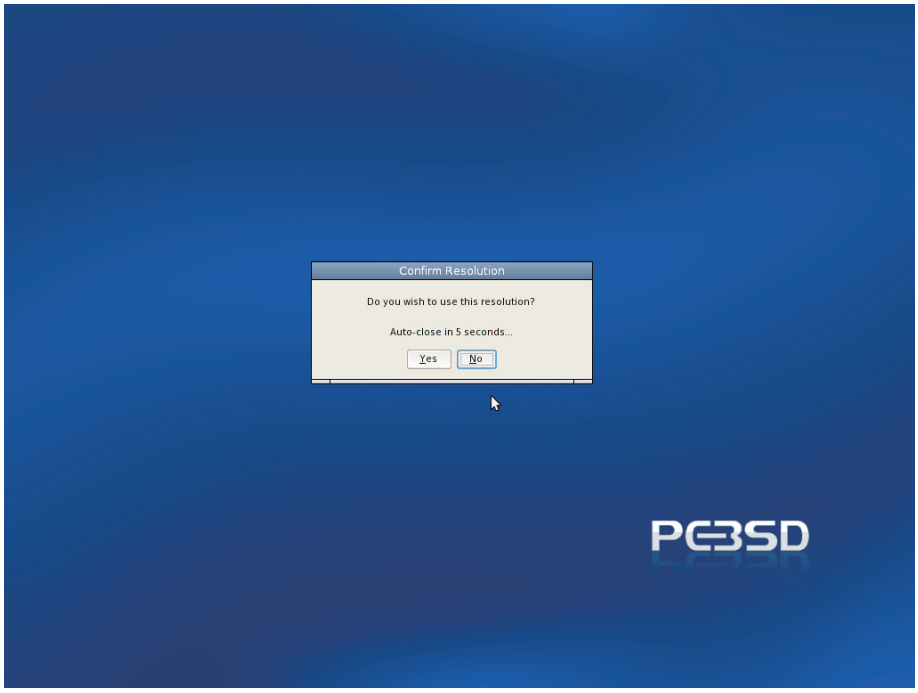


After Install

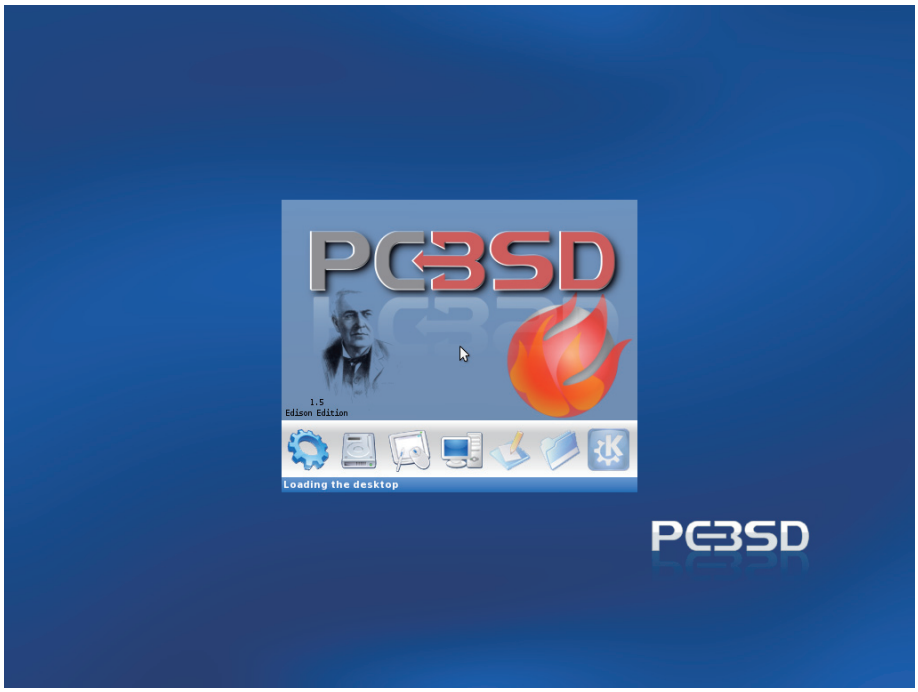
X Window Config



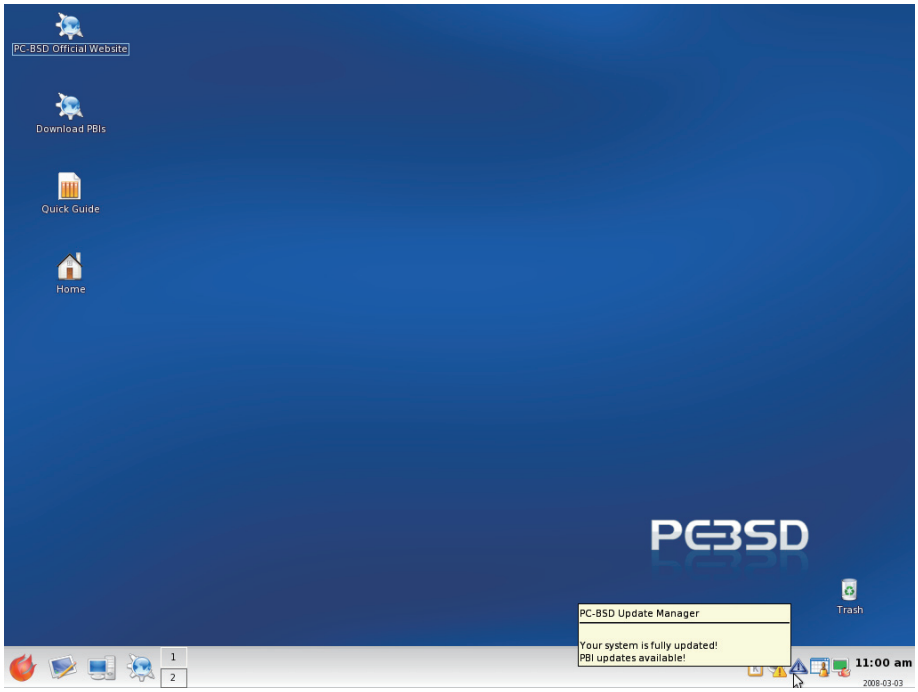
X Window Config



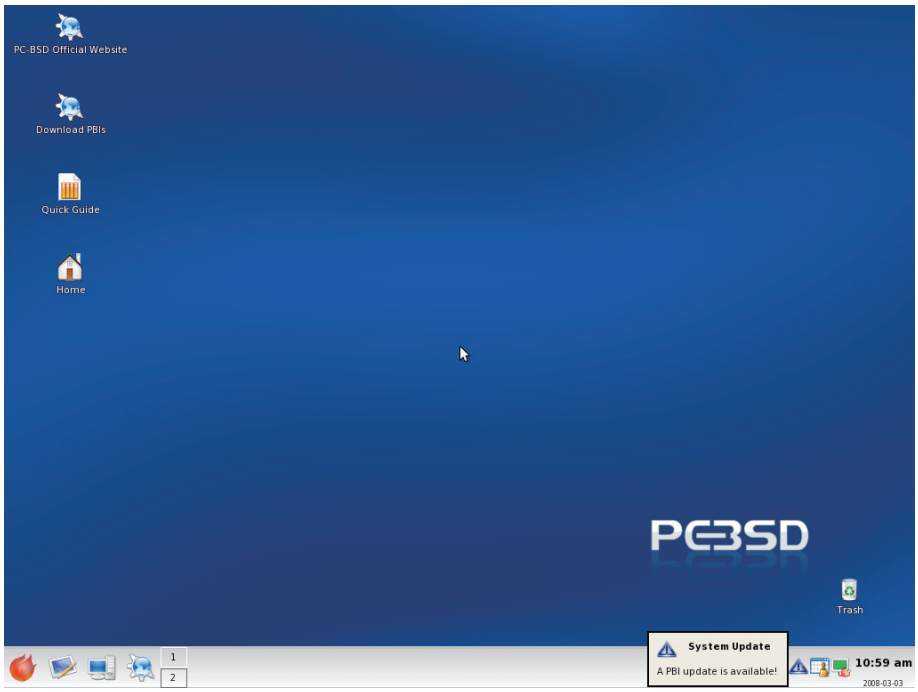
Splash Screen



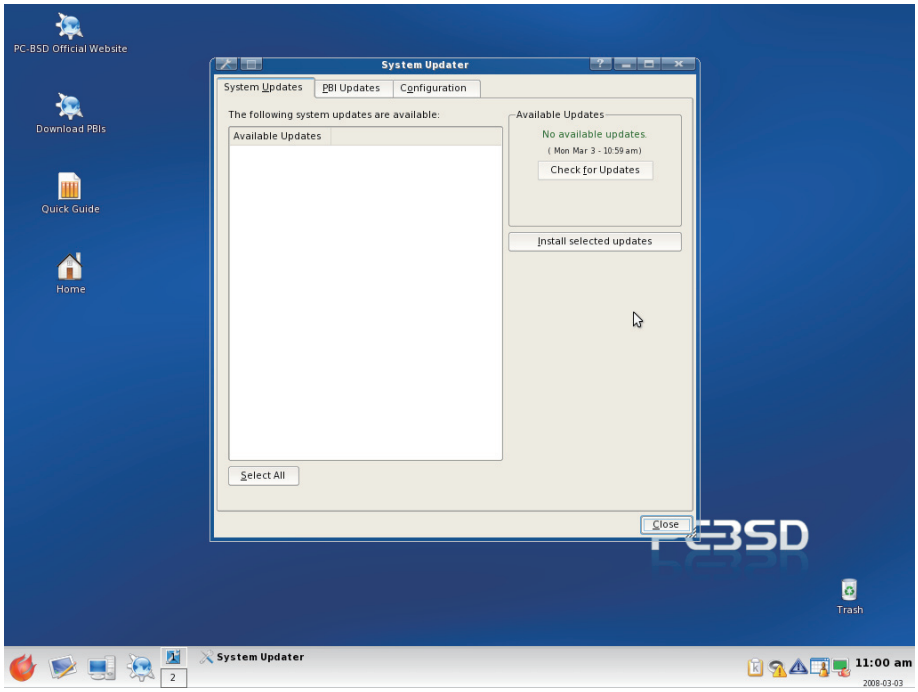
System Updates



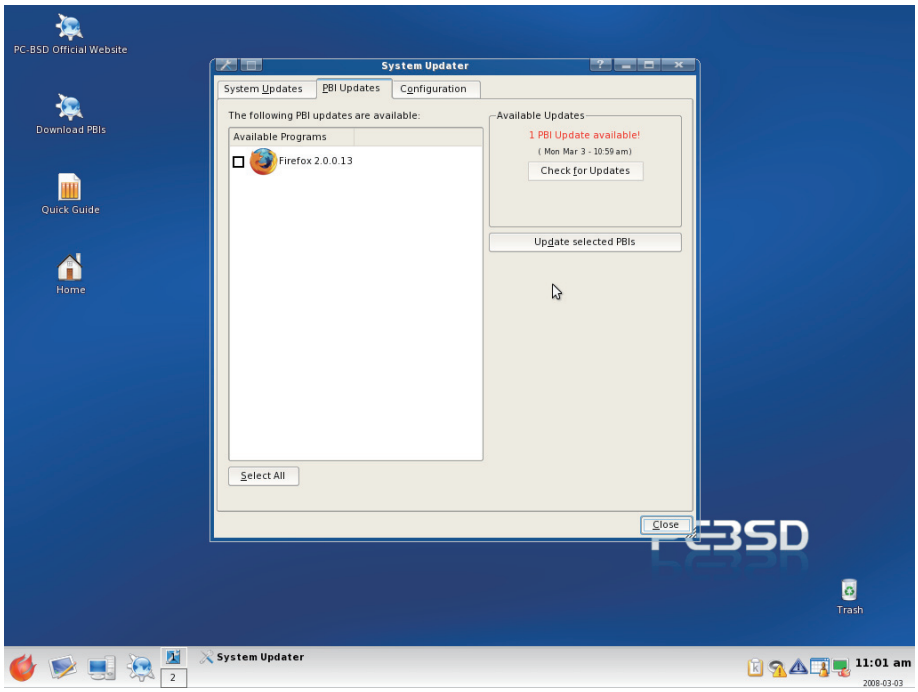
PBI Updates



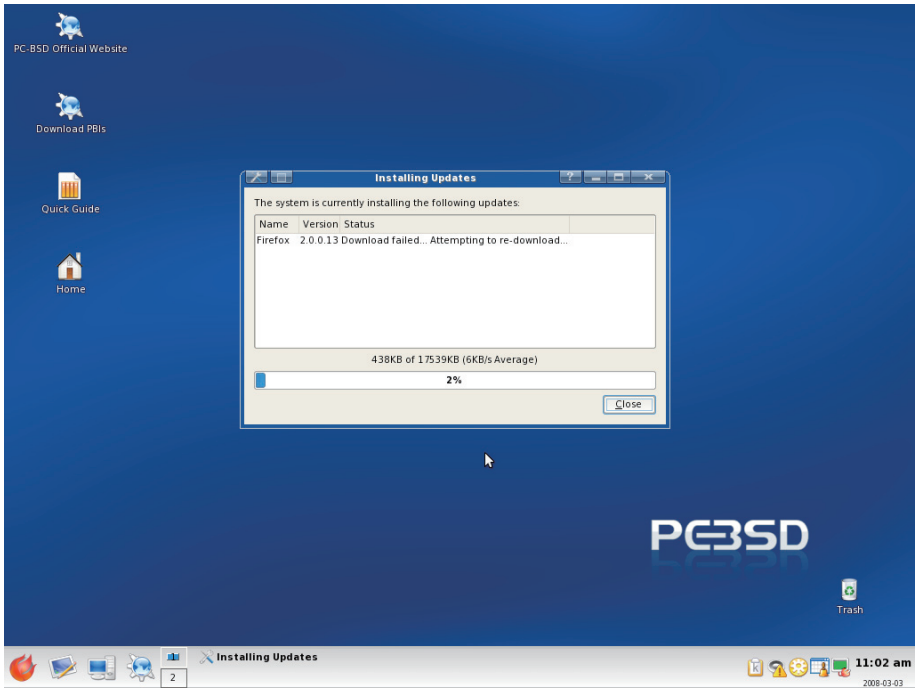
Update Manager



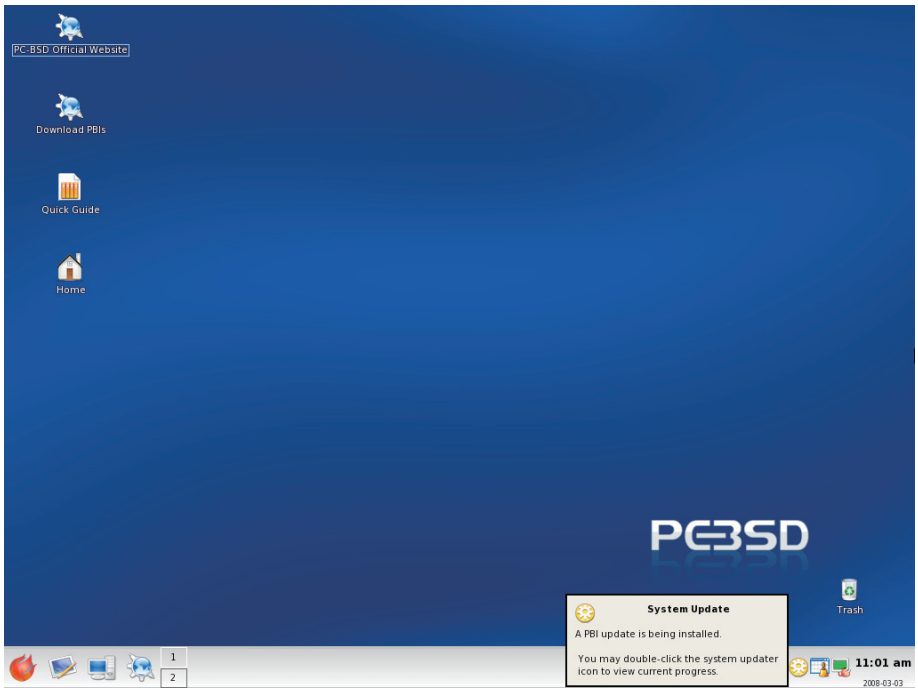
PBI Update



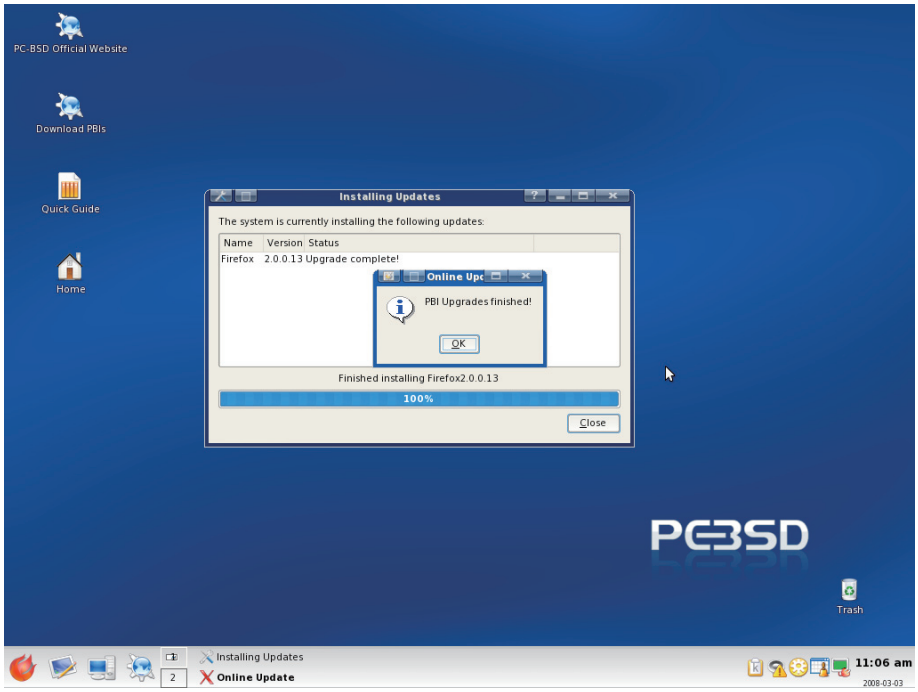
Upgrading PBI



Updating PBI

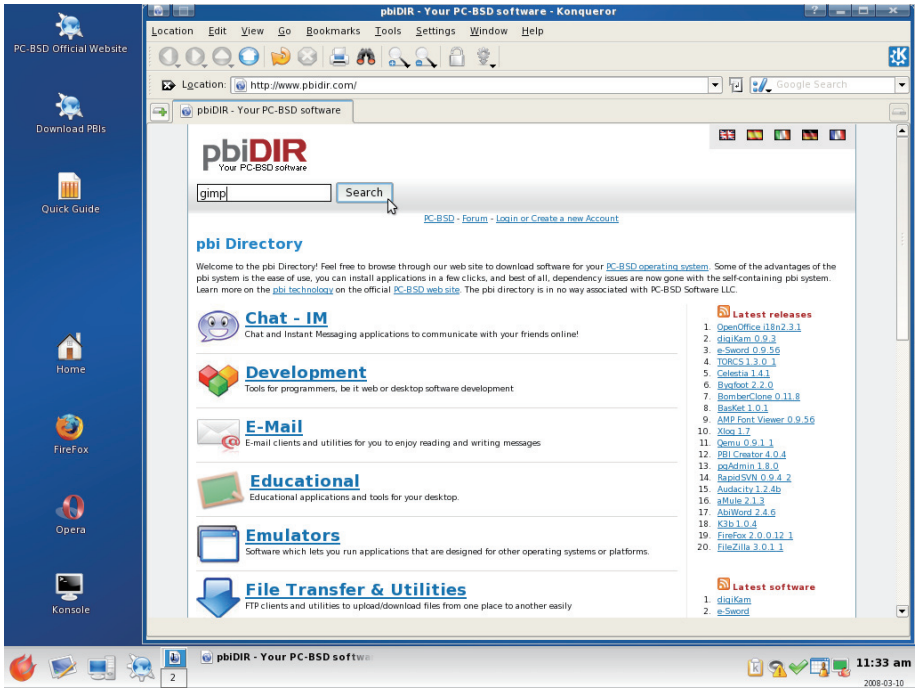


Finished Upgrade

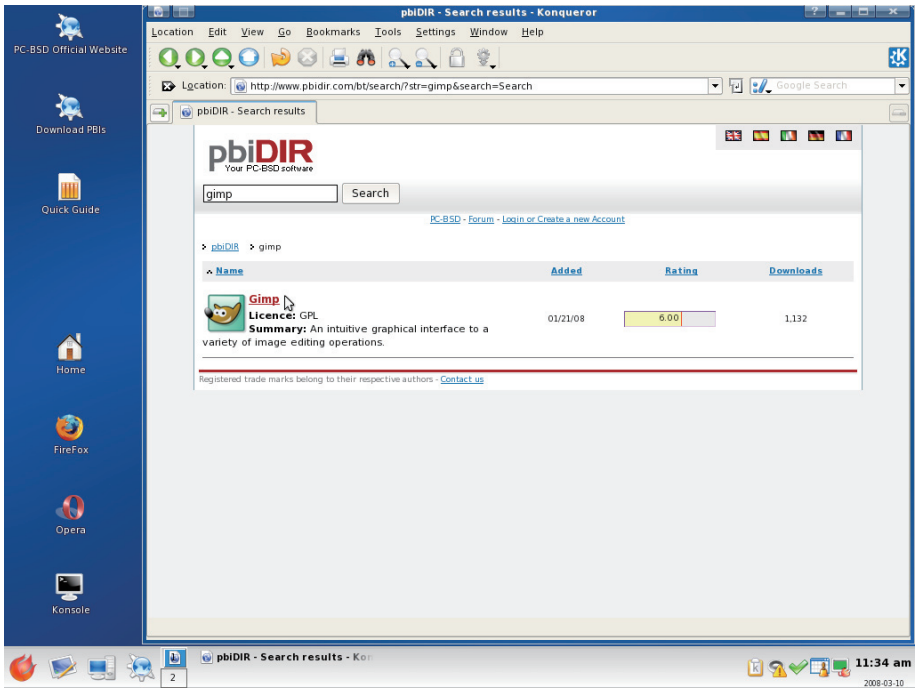


Installing a PBI

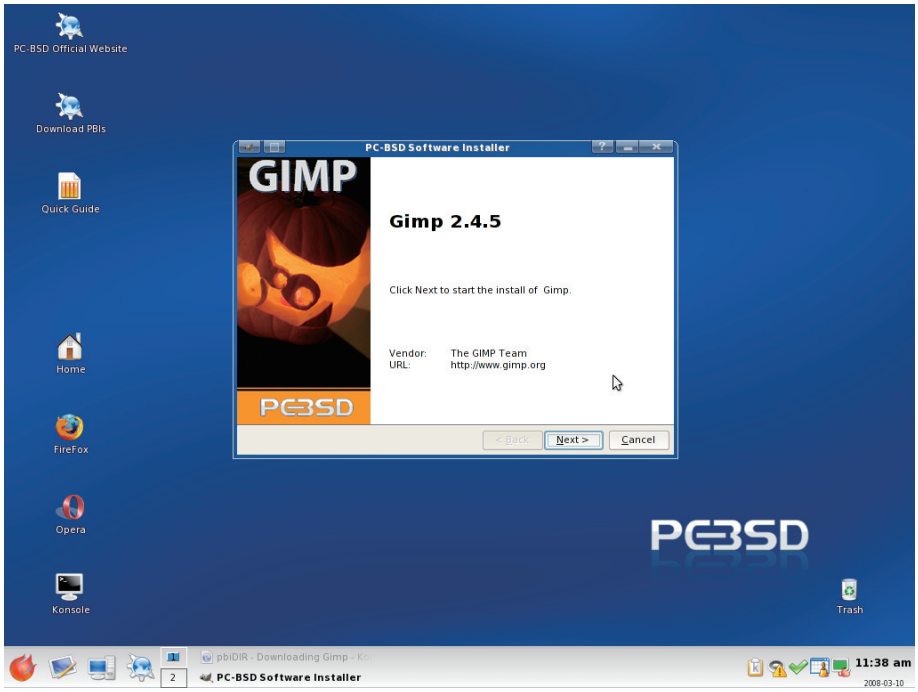
PBI Dir



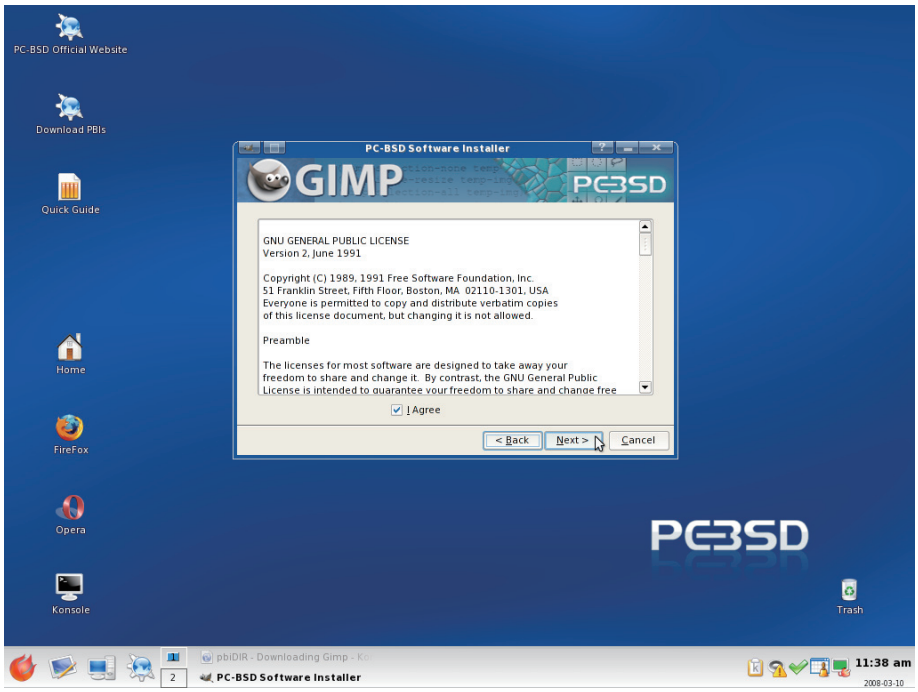
Searching PBIs



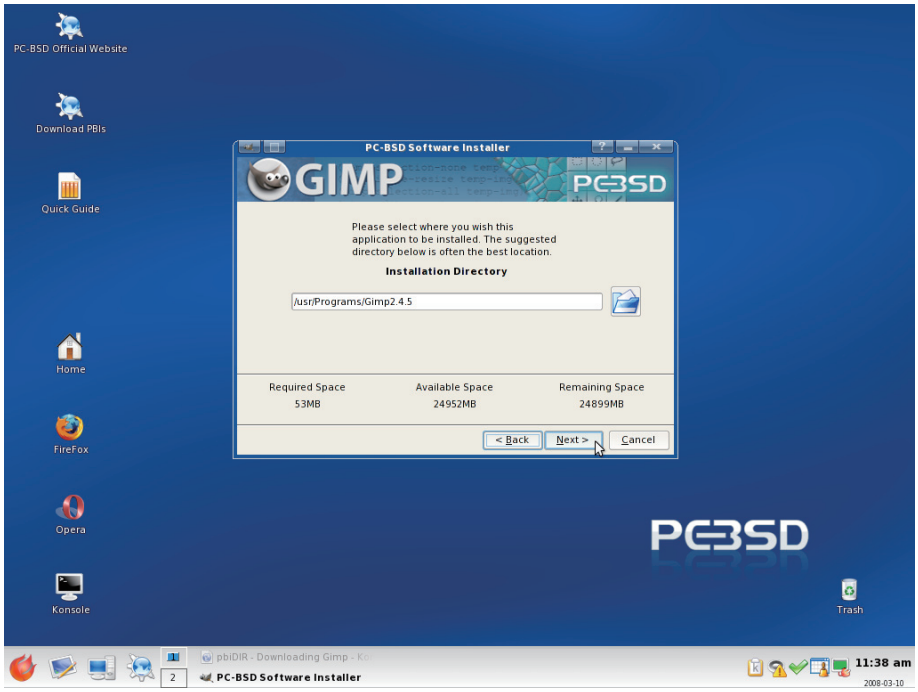
Installing GIMP



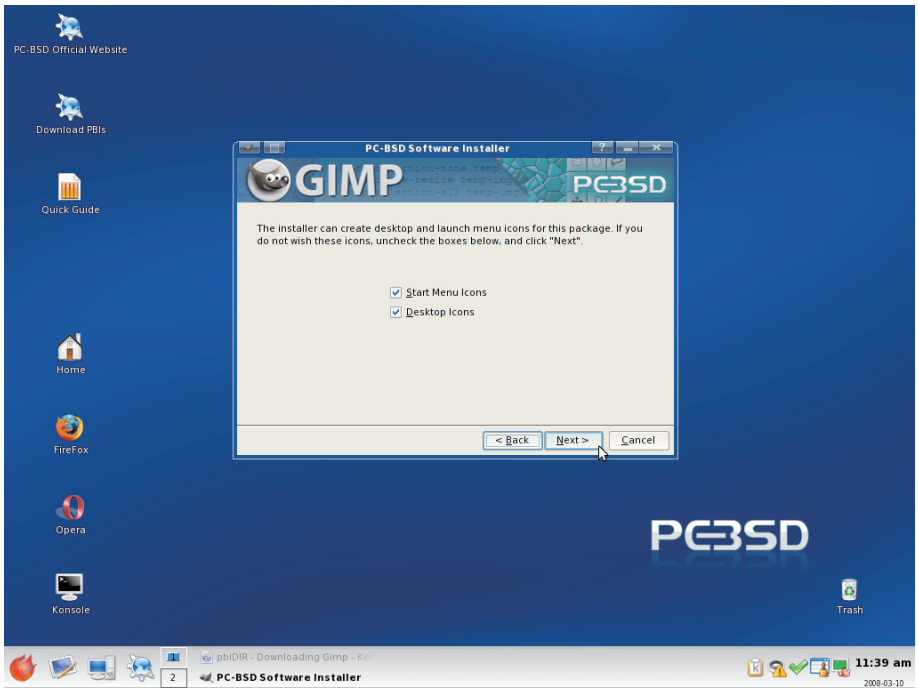
Installing GIMP



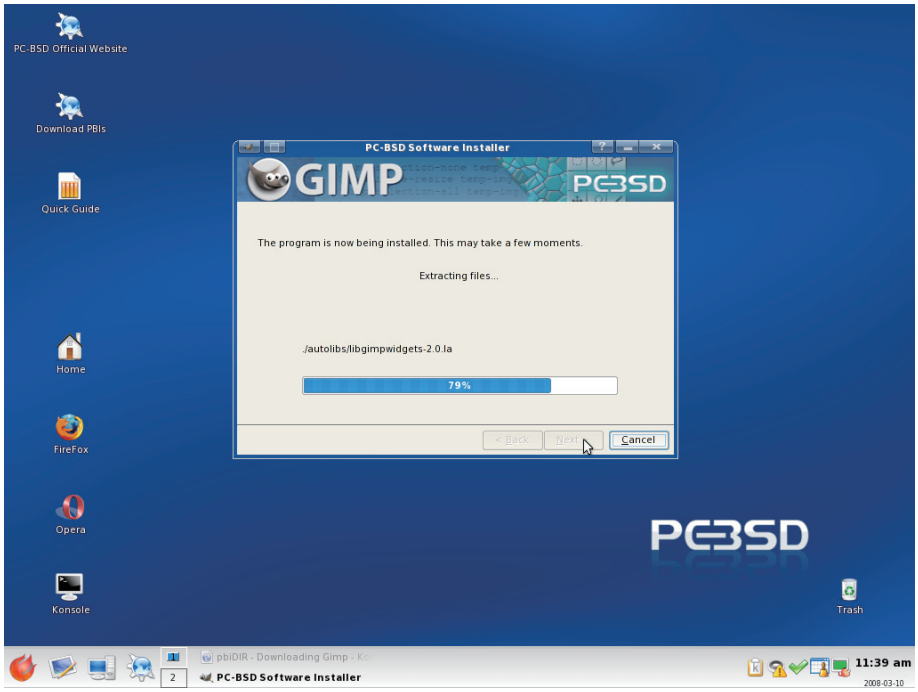
Installing GIMP



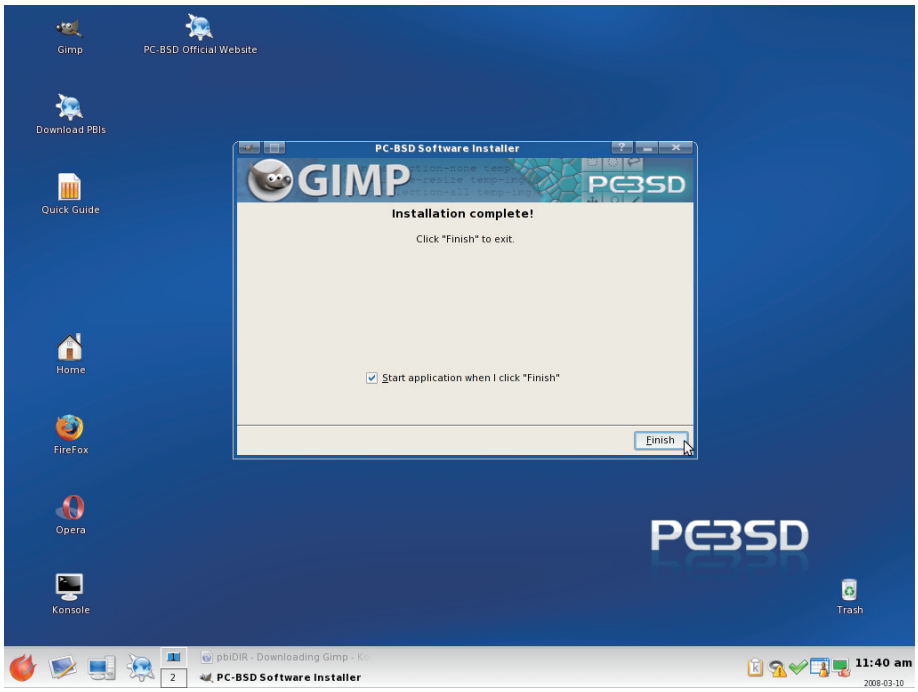
Installing GIMP



Installing GIMP

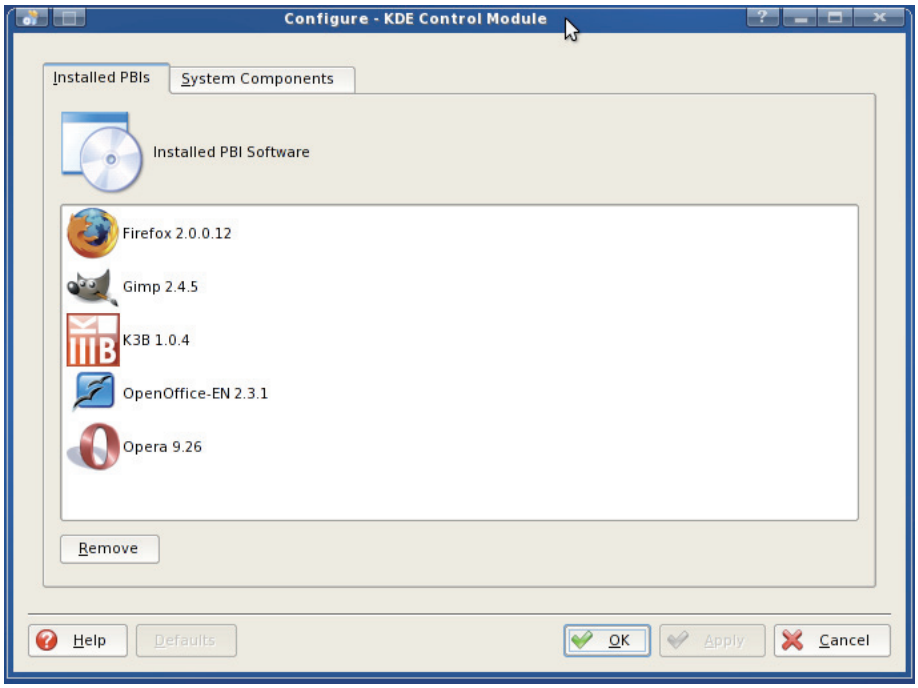


Installing GIMP

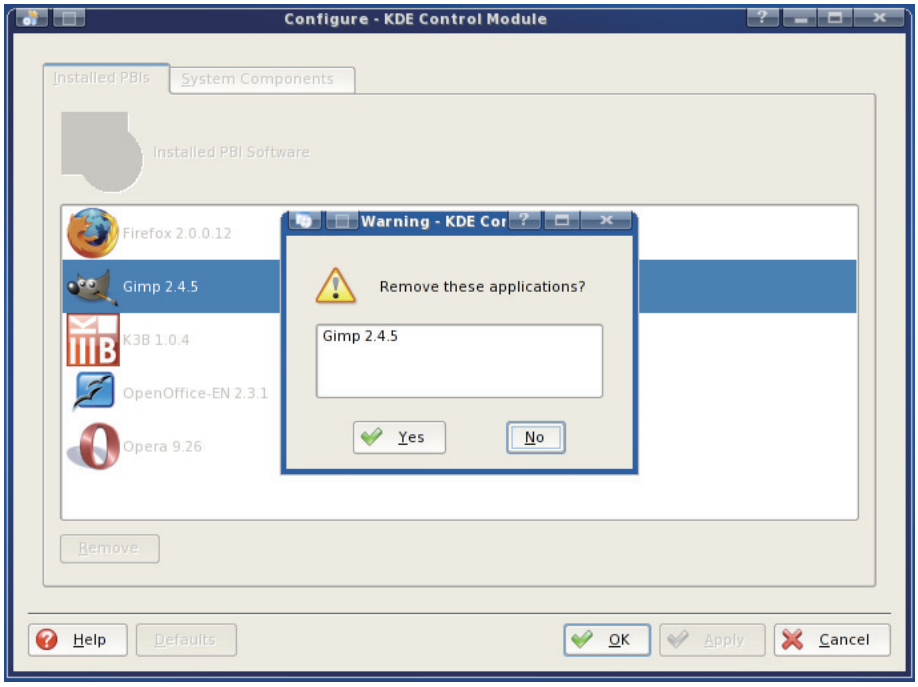


Removing a PBI

Remove PBIs



Confirm Removal



PC-BSD Useful Tools

PC-BSD Toolset

* X Windows Config GUI
* WiFi GUI Config Utility
* User manager
* Add/Remove PBIs
* Simple Firewall (pf) manager

How Can I Help?

* Download, test, and bug report!
* Documentation
* PBI Creation and maintenance
* C/Qt/Shell programming
* Evangelize!

PC-BSD

Questions?

Tracking FreeBSD in a Commercial Setting

M. Warner Losh
Cisco Systems
Broomfield, CO
imp@freebsd.org

Abstract

The FreeBSD project publishes two lines of source code: current and stable. All changes must first be committed to current and then are merged into stable. Commercial organizations wishing to use FreeBSD in their products must be aware of this policy. Four different strategies have developed for tracking FreeBSD over time. A company can choose to run only unmodified release versions of FreeBSD. A company may choose to import FreeBSD's sources once and then never merge newer versions. A company can choose to import each new stable branch as it is created, adding its own changes to that branch, as well as integrating new versions from FreeBSD from time to time. A company can track FreeBSD's current branch, adding to it their changes as well as newer FreeBSD changes. Which method a company chooses depends on the needs of the company. These methods are explored in detail, and their advantages and disadvantages are discussed. Tracking FreeBSD's ports and packages is not discussed.

1 Problem Statement

Companies building products based upon FreeBSD have many choices in how to use the projects sources and binaries. The choices range from using unmodified binaries from FreeBSD's releases, to tracking modify FreeBSD heavily and tracking FreeBSD's evolution in a merged tree. Some companies may only need to maintain a stable version of FreeBSD with more bug fixes or customizations than the FreeBSD project wishes to place in that branch. Some companies also wish to contribute some subset of their changes back to the FreeBSD project.

FreeBSD provides an excellent base technology with which to base products. It is a proven leader in performance, reliability and scalability. The technology also offers a very business friendly license that allows companies to pick and choose which changes they wish to contribute to the community rather than forcing all changes to be contributed back, or attaching other undesirable license conditions to the code.

However, the FreeBSD project does not focus on integration of its technology into customized commercial products. Instead, the project focuses on producing a good, reliable, fast and scalable operating system and associated packages. The project maintains two lines of development. A current branch, where the main development of the project takes place, and a stable branch which is managed for stability and reliability. While the project maintains documentation on the system, including its development model, relatively little guidance has been given to companies in how to integrate FreeBSD into their products with a minimum of trouble.

Developing a sensible strategy to deal with both these portions of FreeBSD requires careful planning and analysis. FreeBSD's lack of guidelines to companies leaves it up to them to develop a strategy. FreeBSD's development model differs from some of the other Free and Open Source projects. People familiar with those systems often discover that methods that were well suited to them may not work as well with FreeBSD's development model. These two issues cause many companies to make poor decisions without understanding the problems that lie in their future.

Very little formal guidance exists for companies wishing to integrate FreeBSD into their products. Some email threads can be located via a Google search that could help companies, but many of them are full of contradictory information, and it is very disorganized. While the information about the FreeBSD development process is in the FreeBSD handbook, the implications of that process for companies integrating FreeBSD into their products are not discussed.

2 FreeBSD Branching

The FreeBSD development model strikes a balance between the needs of the developers and the needs of its users. Developers prefer to have one set of sources that they can change arbitrarily and not have to worry about the consequences. Users prefer to have a stable system that is compatible with the prior systems. These two desires are incompatible and can cause friction between developers and users.

FreeBSD answers the need of both groups by providing two versions of its code. The project maintains a main line for its developers, called "current." This branch contains all the latest code, but all that code might not be ready for end users. All changes to FreeBSD are required to be first committed to the current branch. The quality of the current branch varies from extremely stable to almost unusable over time. The developers try to keep it towards the stable end of the spectrum, but mistakes happen.

To provide a stable system users can use, FreeBSD also maintains a stable version of the OS. Every few years the current version is branched and that branch becomes the new stable version. This branch is called either "stable" or "RELENG X" where X is the major version number for that branch. Stable branches are well tested before they are released. Once released, only well tested patches from the current branch are allowed to be merged into the branch. Once a stable branch is created, its ABI and API are never changed in an incompatible manner, which allows users to upgrade to newer releases that are made from the stable branch with relative ease. Stable branches tend to have a lifetime of about 2-6 years.

An even more stable version of FreeBSD is available than the stable branch. For each release made off a stable branch, a release branch is also created. The only changes that go into these release branches are security fixes and extremely important bug fixes. These are designed for users that wish to run a specific release, but still have high priority bugs fixed and available in a timely fashion. Since release branches are targeted only at end users and have so few changes, the rest of this paper will treat them as a release.

Figure 3 tries to show the relationships between the different branches over time. It shows what should have theoretically happened if FreeBSD had a major release every two years. The horizontal axis is time (in years). The vertical axis is the amount of change, in arbitrary units. Vertical arrows point to the theoretical release points (with the release number under the arrow when the name fits). After about three years, the branches stop being used in favor of newer releases.

Figure 4 shows data from the FreeBSD project since early 1997.[1] There are many features of this graph that differ from the idealized graph. The two that are most relevant are that major branches live beyond the three year idealized vision and that the timing of the release branches isn't completely regular. These points will be important later in deciding which method fits the company's needs the best.

The FreeBSD ports system (which is used to generate the packages that appear in FreeBSD's releases) is not branched at all. Instead, it supports both the current branch, as well as the active stable branches of the project. For each release, the tree is tagged so that it can be reproduced in the future if necessary. These policies are different than the main source tree. Tracking of the ports tree is not addressed further in this paper because its model is different and the author has fewer examples from which to draw advise and conclusions from.

3 Branching Choices

There are a wide range of companies using FreeBSD in their products today. On the simplest end, companies load FreeBSD onto boxes that they ship. On the most complex end, companies modify FreeBSD extensively to make it fit their needs. Over the years four different approaches to tracking FreeBSD have evolved.

The simplest method involves using the stock FreeBSD releases unmodified. Companies doing this grab FreeBSD at its release points and make no changes to the software and just configure th system and install the packages that their customers need. Typically no sources are tracked and only binary packages from FreeBSD's web pages are used.

The next simplest method involves grabbing a release of FreeBSD and using that as a basis for their product. FreeBSD is effectively forked at this point as the company makes whatever modifications are necessary for their product. No thought is given to upgrades or contributing bug fixes back into the community.

[1] The author used fairly simple scripts to extract this data from the commit logs, whose format changed in 1997. Some flaws exist in the data, but they do not affect the shape of the graph.

Companies often setup repositories of FreeBSD stable branches. In this model, the tip of a stable branch (or the latest release point) is imported into some SCM. The company will then make fixes and improvements to its private branch. The company will import newer versions of FreeBSD on this stable branch from time to time. Better run companies will try to contribute their fixes back into FreeBSD to simplify their upgrade path.

The most complicated method involves mirroring the FreeBSD development process. The company will import the latest version of the FreeBSD development branch. They will setup automated scripts for pulling in newer versions. They will make their changes to FreeBSD in this mainline of development. Rather than using FreeBSD's stable branches, the company will decide when and where to branch its version. Once branched, it will control what fixes are merged into its branch.

3.1 Stock FreeBSD

The most widespread use of FreeBSD involves this method. In this method, the company grabs the binaries from a FreeBSD web site or commercial vendor and uses them as built. They layer packages on top of FreeBSD, typically a mix of stock packages from the release and their own additional scripts or programs. The focus of these companies is to have a system that they can deploy and use for a particular purpose.

Customization of the system is typically tracked in some kind of source code management (SCM) system. These customizations include the `/etc/rc.conf` file (which controls most of the global settings for the system), as well as configuration files and other data used by the system. Some of these companies will also compile customized kernel configurations. These files can typically be tracked in any SCM as the demands on the SCM are modest.

These companies typically upgrade only when they need to do so. Once they find a stable version they stick with it until they need something from a newer version. This could be support for newer hardware (drivers or architectures), or application level features such as threading support. Often times they will track newer security releases with services such as FreeBSD update and/or portupgrade in package mode.

FreeBSD meets the needs of these companies fairly well. They don't require additional features or bug fixes not in the current releases. They don't need to optimize FreeBSD for any given platform beyond what the standard system tunables provide for them. The main advantage for these companies is that FreeBSD is a drop in solution. There's very little overhead necessary to get their machines and applications running and FreeBSD's standard install tools can be used to create images for their products (if they even need separate images at all). Some of these companies participate in the community and contribute to the community in many ways. Some of these companies do not. The choice is up to the individual company and its needs, sensitivities and desires.

3.2 Grab and Go

Another easy way to use FreeBSD sources is the grab and go method. In this method, the companies grab FreeBSD at some version and then never upgrade FreeBSD. No attempts to track FreeBSD or pull bug fixes in from FreeBSD are made. The company grabs the source and starts hacking. They layer in their own build and packaging system often times. Sometimes they port to a new architecture. FreeBSD typically is the base for a more extensive application of appliance which the company has total control over.

There are a few advantages to this method. The company can concentrate on making their product work without the distractions introduced when software versions are rolled. The company manages its risk by doing everything themselves. The company can keep any information about what they are doing from being inferred by competitors looking at their bug submissions to FreeBSD. The company's employees are not distracted by interactions with the FreeBSD community. Without these distractions, it is believed that this method allows a company to bring its product to market more quickly.

However, there are many disadvantages to this method. The biggest problem is that companies using this method often find it difficult to get support for the community. Most of the active members in the community have moved on to newer versions of the software, so are unable to help out with problems in older versions. Many of the bug fixes in newer versions of the software are difficult to back port because they depend on other changes to the software that aren't present in the older versions of the software. Often times, interaction with the community on problems for recent releases of the software can save tremendous amounts of time for the company's employees because they can leverage the knowledge of others who have had similar problems.

Companies often times think they are in total control of the hardware platform, but in reality this is a mistaken assumption. Hardware platforms are made of up chips that one buys from manufacturers. These chips go obsolete at an alarming rate sometimes, forcing changes to the underlying hardware to even be able to continue to build it. These new chips often times require new changes to the software. Just as often, others in the community have used the newer parts and have migrated the necessary changes into FreeBSD. So often times companies that go down this path are forced to redo work that has already been done in the community when their supplies tell them that they will no longer be able to give them a certain chip, and no replacements from other vendors exist.

Some companies have managed to start out with this method and later transition to one of the other methods described in this paper. One is even rumored to have recently completed the jump from FreeBSD 2.1.6 (released in 1996) to FreeBSD 6.2 and are now using the stable branch tracking method described below. Other times, the outcome isn't so good and the product is migrated to another system, or the product is killed.

3.3 Stable Branch Tracking

One nice feature of FreeBSD's stable branches is their stability. One can typically count on them to build and not have critical problems. The stable branch tracking strategy takes advantage of this feature.

The first major release on a branch is imported into a private SCM for the company to use. The sources are imported using the 'vendor branch' facility of the SCM. This facility allows one to keep a pristine copy of the sources from FreeBSD separate from the modified sources for the company. This separation allows developers to produce patches between the two. These patches can be used to determine which changes should be contributed back to the FreeBSD tree. In addition, by importing into a vendor branch and merging into the company's private branch, the company can upgrade versions of FreeBSD at any time. They can pull either a whole new FreeBSD tree, or individual files that have the fixes they need. The company can choose when to roll forward the basis of their tree, and the branching features of most SCMs make this procedure relatively easy. As new stable branches of FreeBSD become available, this process can be repeated for them in a separate module or directory in the SCM.

The big advantage to this approach is the underlying nature of the stable branch itself. The FreeBSD project has policies and practices that ensure that the branch will be stable, especially near releases off of that branch. The ability to "cherry pick" fixes from newer versions of FreeBSD without affecting the rest of the branch helps to mitigate risks associated with upgrading. In addition, by using the vendor branch feature, these changes will not interfere with future imports of a more complete system when it is appropriate to do so. Since the ABI and API are also frozen for the entire branch, one can grab fixes and changes from newer versions without worrying about breaking applications under development within the company. The isolation of major releases into separate modules in the SCM allows a company that has several products built on FreeBSD to selectively upgrade them to newer versions as market conditions warrant.

There are a few disadvantages for this approach. First, to fully leverage the FreeBSD community, it is desirable to push back bug fixes to the community in a timely fashion. When this isn't done, as is often the case when deadlines are tight, the chore up upgrading increases because one must bring forward all of the changes to the system. Second, if the company makes extensive changes that aren't merged back into FreeBSD and want to migrate to the next major version, they will need to redo their changes after the next major branch is created. If they are in an area of FreeBSD that has changed between the two branches, this can take quite a bit of time and effort.

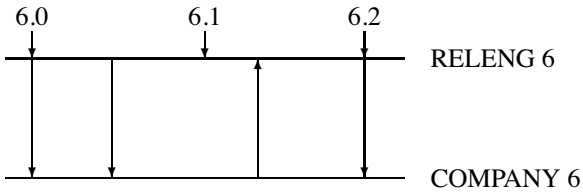


Figure 1: Code Flow between FreeBSD RELENG 6 and Company's Version

Figure 1 shows this graphically. This figure shows an idealized flow of patches into the company tree and back to FreeBSD. It also neglects to picture the required trip through FreeBSD current required for all patches to be committed to stable branches. The number of changes to the branches are also abstracted out, unlike Figure 1 and 2 presented above. The arrows pointing to the RELENG branch represent FreeBSD releases from that branch. The arrows from the RELENG branch to the COMPANY branch represent merges of code from FreeBSD into the company's repository. The arrows from COMPANY to RELENG represent patches that have successfully been contributed back into FreeBSD and have been merged into FreeBSD's RELENG tree.

3.4 Own Branching

One way to keep current in FreeBSD is to track FreeBSD's main development branched called "current." Many developers do this in the FreeBSD perforce tree and it works well for them. This method follows that practice, but also adds stable branches, akin to FreeBSD's stable branches in concept, but not tracking any specific FreeBSD release.

The company would import FreeBSD's current code as its starting point for its FreeBSD development efforts. They would start making changes to their current branch. In addition, source code pulls from FreeBSD's current branch would be frequent to keep the company's current branch close to FreeBSD's current branch. Just after these pulls, the company's current branch would be exactly FreeBSD's current branch with only the company's changes layered on. The company would then merge the relevant change from its current tree into FreeBSD's current tree by working with the FreeBSD community to produce acceptable patches.

The company would also emulate FreeBSD's branching practices. When the tree is in a good state to branch, possibly driven by delivery schedules for its end products, the company would branch its own stable branch from their current branch. They would merge bug fixes and new features from their current branch into this stable branch and build products from this stable branch.

The main advantage of this approach is that it is easier to keep current with FreeBSD than the stable branch tracking approach. To generate patches, a simple diff(3) between the FreeBSD sources and the company sources will generate the patches. As patches are merged with FreeBSD, the next pull will automatically include those changes and the delta between the company's sources and FreeBSD's will drop. By controlling the branching times, there's no need to wait for FreeBSD to create new a stable branch, so the company can drive released schedules more easily than companies tracking stable branches.

The main disadvantage of this approach is that the company loses the work done by the FreeBSD community to keep its stable branches stable and useful. Since there is no connection between the company's stable tree and FreeBSD's stable tree, improvements to FreeBSD's stable branch aren't automatically reflected in the company's stable branch. An engineer will need to watch changes going into either the current branch from FreeBSD, or into FreeBSD's stable tree and manually pull them into their own stable branch. Typically, there are on the order of 100-200 commits to a FreeBSD stable branch a month, so this load can be quite large. In addition, except around the time a new branch is cut in FreeBSD, FreeBSD's current branch may have periods of instability and it can be quite difficult to know when a good time to branch might be as many of the stability or quality problems that are in FreeBSD's current branch often lay undiscovered for months or years because it doesn't get the intensity of testing that a FreeBSD stable branch receives.

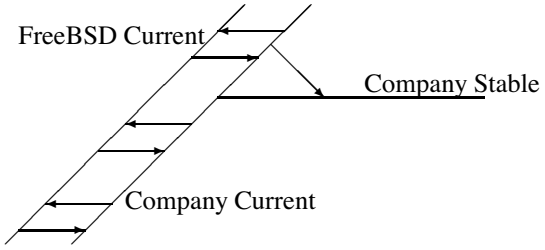


Figure 2: Relationship between FreeBSD current and company branches

Figure 2 shows this graphically. This figure shows an idealized flow of patches into the company tree and back to FreeBSD. The two parallel current branches are shown diagonally, with the company's custom stable branch shown horizontally, much like Figures 1 and 2 presented above. No FreeBSD release points are included, since they are largely irrelevant to the method. The exact delta between the two current branches is also abstracted out, as this will ebb and flow over time and needlessly complicates the graph. The arrows represent changes being merged from one branch to another, either between the two current branches, or from the company's current branch to its stable branch.

4 Acknowledgments

I would like to thank the crew at Timing Solutions: Ben Mesander, John Hein, Patrick Schweiger, Steve Passe, Marc Butler, Matthew Phillips, and Barb Dean for their insight and implementation of the 'Stable Branch Tracking' method described in this paper. We deployed it across 4 major versions of FreeBSD.

I would like to thank Julian Elischer for the many conversations that we have had about development method. He provided much of the input into the 'Own Branching' section.

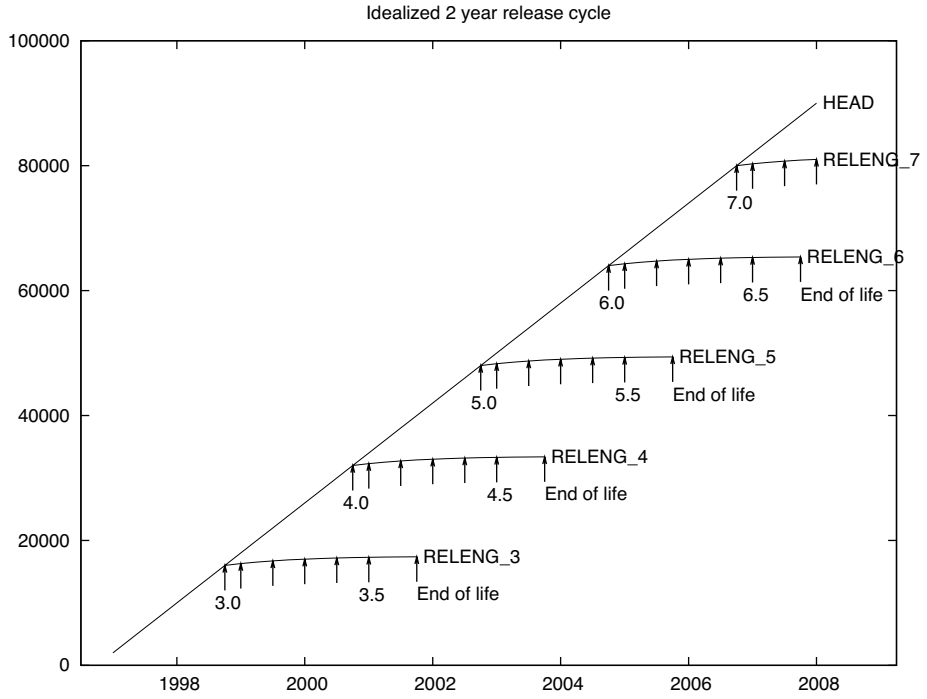


Figure 3: Idealized branching model

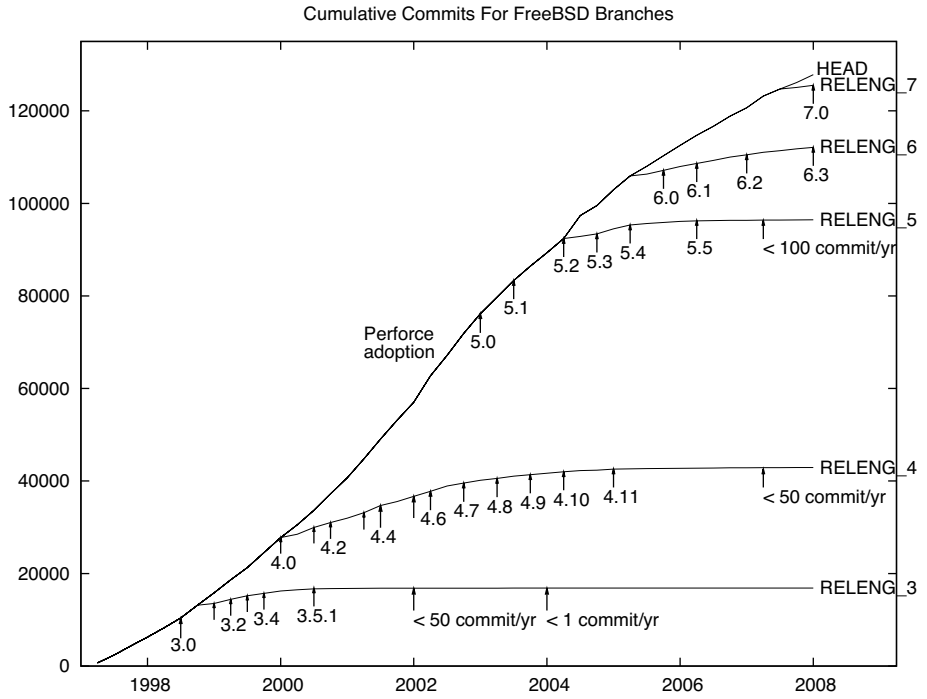


Figure 4: Actual FreeBSD branching history

Gaols

Implementing Jails Under the kauth Framework

Christoph Badura
The NetBSD Foundation
bad@netbsd.org

Abstract

FreeBSD's jail facility provides a light-weight form of virtualization by restricting access to the file system, to privileged system calls, to network resources and the ability to see and interfere with processes in other jails for the imprisoned processes.

NetBSD 4.0 introduced a clean-room implementation of Apple's kauth(9) framework. This framework replaces the open-coded checks for appropriate privilege with standardized calls for authorization of specific enumerated actions. Thus allowing fine-grained control over which actions are allowed and which are denied. The kauth(9) framework also provides a mechanisms to dynamically add and remove security modules that take part in the authorization process and can alter the default behaviour.

This paper explores the implementation of FreeBSD style jails under the NetBSD kauth(9) framework. Jail-like functionality was chosen because it is both relatively simple to implement and at the same time stresses the limits of the framework because of the need to not only return "allowed"/"not allowed" decisions but also to modify network addresses in the callers environment.

This paper presents work in progress.

1. Introduction

FreeBSD Jails

FreeBSD 4.0 introduced a light-weight virtualization facility called jails [KAWA2000]. The facility builds on the chroot concept and extends it by imposing additional restrictions on the *imprisoned* processes:

- File system access:
  Jailed processes run chrooted and have access to the files below the chroot point only.
- Network access:
  Process in a jail are allowed to bind sockets to a single specified Ipv4 address only.
- Restricted root rights:
  Privileged system calls that would affect the host system or other jails are forbidden.
- Restricted visibility of resources outside prison:
  Jailed processes cannot see processes and sockets that are in different jails.

Jails don't provide full virtualization because they do not restrict usage of resources like CPU, memory, I/O and network bandwidth.

Jails predate the FreeBSD MAC kernel framework which provides fine-grained control over and authorization for privileged actions. Interestingly, the jail facility was not re-implemented as a MAC module, although it uses some of the services of the MAC framework, when that is compiled into they system. Also, the MAC system has special knowledge of jail system.

The kauth Framework

The kauth(9) framework was introduced by Apple in Mac OS X 10.4 Tiger [TN2127]. Elad Efrat provided a clean-room implementation of the framework for NetBSD 4.0 [EFRAT2006].

On the caller side this framework replaces the open-coded calls to suser(9) etc. that check that the requester has appropriate privileges with calls to a generic authorization framework asking permission for specific, named actions. On the kauth(9) framework side the authorization actions were, for the first time, explicitly enumerated and run through a modular, pluggable, and extensible system.

The system can be extended by linking additional security models into the kernel that participate in the authorization processes and registering them with the authorization framework. Security models can also be added at runtime as loadable kernel modules.

The proponents of the kauth(9) framework claim the following benefits:

- Reducing coding errors at the calling places by replacing the maze of a twisty little open-coded calls, all slightly different, with easy to understand “boilerplate” code to check authorization for the requested action.
- Reducing coding errors in the authorization modules by factoring out common code into small, modular, and easy to understand methods.
- Introducing a modular and extensible framework with pluggable security models. The claim is that this allows new security models to be implemented that refine and extend the traditional Unix security model.
- The ability to completely replace the default Unix security model in the operation system with an alternate model.

Role based access control (RBAC) and mandatory access control (MAC) are the usual examples cited for candidate security models. However, they are complex and difficult to implement. To the author's knowledge, no implementation has been attempted to date for the kauth(9) framework in NetBSD.

The first two benefits have been proven almost immediately by implementing kauth(9) and converting the kernel to the new framework. However, until to date nobody has tried to implement a new security model that extends the existing Unix security model and does something new or otherwise unforeseen by the designers of the kauth(9) interface.

One of the important factors in the success of Unix and the fact that it is still extremely competitive in the market 40 years after its invention is that its architecture and interfaces are so artfully designed that has been possible to almost seamlessly integrate new functionality into the system that the original designers had not anticipated. E.g. demand paged virtual memory, sockets and networking, virtual file systems, etc.

When a new subsystem is introduced into the kernel it is therefore important to exercise its capabilities and find out if it adds real new capabilities to the system that its designers haven't anticipated or if it is merely a factoring out of existing code that doesn't open any new possibilities.

This paper explores the implementation of a FreeBSD jail(2)-like security model under the kauth(9) framework. The jail API is a particularly good test case for stressing the design of the interface because it imposes only a few restrictions in addition to the traditional Unix security model. However some of these restrictions require non-trivial side-actions that were probably not foreseen by the designers of the kauth(9) interface. For example mapping the INADDR_ANY address to an IP address assigned to the jail for certain socket operations.

2. Background

Kauth Basics

With the kauth(9) framework the open-coded calls in the kernel that check for sufficient privilege are replaced with calls to the function *kauth_authorize_action()* to authorize the requests.[1] The arguments to the function are:

- a named scope
- the credentials of the entity asking for authorization
- a numeric constant identifying the action in the scope to be authorized
- up to 4 context specific arguments giving further details about the request

kauth_authorize_action() dispatches the request to the listeners that the security models interested in the particular scope have registered with the framework. The listeners examine the request and return one of KAUTH_RESULT_ALLOW, KAUTH_RESULT_DENY, or KAUTH_RESULT_DEFER indicating that they approve the request, deny it, or have no opinion. If at least one listener denies the request, *kauth_authorize_action()* will deny the authorization.

Typical scopes are the system scope, the process scope, the network scope, and the generic scope.

The kauth(9) framework provides a standard way for security models to attach model-specific data to credentials. There is also a special credentials scope for managing this security model specific data when credentials are copied or freed.

Differences to FreeBSD MAC framework

In the kauth(9) framework additional context-dependent information is usually passed to the authorization modules to allow making decisions based on this information.

Also, the kauth(9) framework provides a standard way to attach additional security model specific data to in-kernel credential structures.

These two features are not found in the FreeBSD MAC framework.

3. Implementation

1 Actually scope specific wrapper functions for convenience.

Prison Management

The central data structure for jails is the prison structure that records the attributes of a jail. When a process is being jailed a reference to a system wide prison structure is attached to the process' credentials. And the presence of a reference to a prison structure in the process credentials indicates that a process is jailed. We use *kauth_set_data(9)* to associate a prison reference to the process credentials structure. This is transparent to all other users of credentials. As the credential structure itself hasn't changed no other source code of the system has to be recompiled. In FreeBSD the credentials structure was changed to add a field with a pointer to the prison structure.

In NetBSD the attributes of a prison are passed into the kernel in the form of a *proplib(3)* plist dictionary to the *gaol(2)* system call. [PROPLIB3] This was done to allow future extensibility. In particular, for allowing to associate multiple network addresses with a jail.

Prison structures need to be reference counted and disposed of when the last credential structure referencing a prison is no longer in use. Fortunately, the kauth(9) framework provides a nice infrastructure for just this purpose. The gaol security model registers a listener for the credentials scope and is henceforth notified whenever a credential structure is initialized, copied, freed, or the referencing process forks. For copies and forks the prison's reference count is increased, and for frees the reference count is decreased, automatically disposing of the prison structure when the reference count reaches zero.

Jail-specific System Calls

The system call interface consists of the two system calls *goal(2)* and *gaol_attach()* which are closely modeled after the FreeBSD system calls *jail(2)* and *jail_attach(2)*. The difference is that *gaol(2)* takes a *proplib(3)* plist dictionary instead of a jail structure as argument. We chose different names for the system calls to make it obvious which interface version is used.

The kernel side of the implementation is boringly similar between NetBSD and FreeBSD. There are minor differences in internal kernel interfaces that need to be taken into account.

The biggest difference is that for *gaol(2)* the *proplib(3)* dictionary containing the jail arguments needs to be transfered into the kernel. For some strange reason the *proplib(3)* interface provides standardized methods to pass *proplib(3)* objects in and out of the kernel with *ioctl(2)* system calls, but there is no support for passing *proplib(3)* objects to other, more general system calls. This is easily fixed with a bit of almost trivial refactoring.

Authorizing Requests - The Trivial Ones

As mentioned in the FreeBSD jails paper [KAWA2000] the authors had to carefully read the kernel source code and identify all the places where privileges where checked and modify those places to make them aware of jails.

With the introduction of the kauth(9) framework most of this was done already in the NetBSD kernel. There were only a handful of places left that need to be made jail-aware but had no corresponding kauth(9) checks. These are described later.

Authorization decisions are made by so-called listeners for specific scopes. The listeners for the goal

security model have a common code pattern:

- If the credentials do not have a reference to a prison, they punt the decision by returning KAUTH_RESULT_DEFER.
- Otherwise the default return value is set to KAUTH_RESULT_DENY, indicating that the requested action should be denied, as is appropriate for most of the actions.
- Then individual actions that are not always denied are examined in detail and the return value is set to KAUTH_RESULT_ALLOW, if appropriate.

```
int
secmodel_gaol_system_listener(kauth_cred_t cred, kauth_action_t action,
    void *cookie, void *arg0, void *arg1, void *arg2, void *arg3)
{
      int result;

      /* if not in jail, defer */
      if (gaoled(cred) == NULL)
            return (KAUTH_RESULT_DEFER);

      result = KAUTH_RESULT_DENY;

      switch (action) {
      case KAUTH_SYSTEM_ACCOUNTING:
      ...
      default:
            break;
      }

      return (result);
}
```

This covers the overwhelming majority of all requests because we deny most of them for imprisoned processes.

Authorizing Requests - The Easy Ones

The next class of authorization requests are the ones that are allowed unconditionally for imprisoned processes: e.g. *chroot(2)*. They are handled in the big switch statement of the listener function and set the result value to KAUTH_RESULT_DEFER.

```
      ...
      case KAUTH_SYSTEM_CHROOT:
            result = KAUTH_RESULT_DEFER;
            break;
      ...
```

There is also a class of privileged actions that are allowed inside jails depending on the setting of a sysctl variable: e.g. changing the system file flags. They are handled in the code of the listener functions similar to the unconditionally allowed ones by setting the result value to KAUTH_RESULT_DEFER when appropriate.

```
      ...
      KAUTH_SYSTEM_CHSYSFLAGS:
            if (gaol_chflags_allowed)
                  result = KAUTH_RESULT_DEFER;
            break;
```

```
...
```

The next class of authorization actions are the ones that require actual checks of prison attributes, e.g. the ones that ensure that processes from one prison can't see or interfere with processes in a different prison or ones that are not imprisoned.

```
...
case KAUTH_PROCESS_CANSEE:
      result = prison_check(cred, ((struct proc *)arg0)->p_cred);
      break;
...
```

The most complicated of these checks is the check done upon opening a socket that restricts new sockets to PF_LOCAL, PF_INET, and PF_ROUTE protocol families. However, all the required information was already passed to the scope listener and the implementation is straight forward.

Authorization Requests - The Interesting Ones

Thus far implementing the gaol security model was almost boringly easy. Now, we come to the more interesting parts that are not as easy to implement.

Implementing these checks requires externally visible changes to the kauth(9) framework to be made. New authorization actions and sub-requests need to be defined. Because of this the enums detailing the authorization actions and sub-requests in the affected scopes need to be extended with new codes for the new actions.

This changes the binary interface of the kauth(9) KBI. Therefore it needs to be centrally coordinated. All existing security models need to be carefully checked whether they need to be made aware of the new authorization actions. Typically, at least the default bsd44 security model needs to allow these requests.

Unfortunately the kauth(9) framework has currently no means to check that a given security model is up-to-date with all the defined authorization actions. Thus it is possible to e.g. load an incompatible security model as an LKM.

Two new actions needed to be defined in the system scope. One authorizes the use of System V IPC system calls. Whether requests are denied or deferred depends on a sysctl variable that controls the behaviour for all jails. The other authorizes the use of the quotactl(2) system call and is denied for imprisoned processes. This is different from requiring super user privilege for certain quota operations.

The processes scope was extended with an authorization action to determine whether the calling process can wait(2) for a named processes. The request is denied if the named process is outside the prison of the requesting processes.

The network scope was extended with a request to check whether a given credential holder is allowed to see a particular interface address. Imprisoned processes can only see interface addresses that are specified when the prison is created.

When an imprisoned process binds a socket to a local address and when using raw sockets, which is optionally allowed, the kernel needs to check that the local address specified or embedded in the packets is on the list of allowed local network addresses for the prison. This is implemented in a

straight forward manner with the new KAUTH_NETWORK_ADDR_CANSEE action in the network scope.

Because these new requests fit well with the kauth usage model the necessary changes are straight forward. On the calling sites standard code for the checks is added. The listeners in the system's default scope have to be modified to unconditionally allow these requests. All other security models have to be carefully checked if they need updating so that these requests are not erroneously denied.

What remains are a couple of special cases that at first glance don't seem to fit well into the kauth model. They are all related to replacing certain network addresses with an address from the list of local network addresses permitted in the prison.

The first case is in the routing socket code where interface addresses are returned to userland. FreeBSD returns only the prison's IP address for imprisoned processes. Implementing this with the kauth(9) framework requires to deviate from the purist model of only returning a go/no-go response from a request to authorize an action. Never letting ones sense of morals getting in the way of doing what is right, we extend the semantics of *kauth_authorize_action()* in the case of KAUTH_NETWORK_IFADDR so that “allow” means: “use the interface address as you normally would”. In the “deny” case, an alternate address that is to be used instead of the original interface address is passed back to the calling site through one of the optional parameters of *kauth_authorize_action()*.

Similarly, when an imprisoned process requests to bind a socket to the unspecified or loopback address or tries to connect to the loopback address, the jail code needs to fix up the address with an address from the list of local network addresses permitted in the jail. This is done through the KAUTH_NETWORK_LOCALADDR authorization action. The return value is ignored as it cannot fail. Instead the action simply fixes up the socket address that is passed in as one of the parameters.

The last case is sending packets through a raw socket. Again, the kernel needs to fix up the source address of outgoing packets with an address from the list of allowed local addresses for the prison. This is accomplished by another newly introduced authorization action in the network scope that returns a suitable local address through one of the parameters of *kauth_authorize_action()*.

4. Limitations

FreeBSD jails limit the visibility of sockets that belong to different jails. This has not yet been implemented. Emulating the FreeBSD *cr_canseesocket()* in the kauth(9) framework is just another trivial variant of the KAUTH_XXX_CANSEE actions. But it requires changes in the kernel infrastructure to attach full credentials to sockets. Due to time constraints this hasn't been looked at yet.

When this is implemented it could be reused by the curtain security model to control visibility of sockets with credentials holding different uids/gids.

Setting and testing a jail-specific securelevel has not been implemented.

Changing the jail-specific hostname from within the prison has not been implemented.

Shoehorning these two actions into the kauth(9) framework is challenging at best. The kauth(9) framework always invokes all listeners for a scope. However, the order in which they are invoked is not defined. One approach might be to adapt the techniques from the secmodel_overlay example.

In FreeBSD the ddb kernel debugger's *ps* command indicates indicates jailed processes with the flag letter 'J'. This has not been implemented.

FreeBSD's procfs indicates the jail's hostname in the per-process status file. This has not been implemented either.

5. Conclusions

Implementing jails within the kauth(9) framework as a proof-of-concept was surprisingly easy. However there remain a couple of points that are not entirely satisfactory. In particular the requests that need to chcange the network addresses at the calling sites makes assumptions that no other security model does the same. While this works out OK in practice, it is not safe under all theoretically possible scenarios. How this might be solved in a production-quality implementation requires further thought.

It is feasible to introduce new authorization scopes, because the scope identifiers are dynamically assigned. Adding new authorization actions or action-specific sub-requests, however, needs careful planning and central coordination, because of their implementation as enums. Also the lack of kauth(9) ABI versioning might lead to difficulties.

References

[KAWA2000] Poul-Henning Kamp and Robert Watson, Jails: Confining the omnipotent root. http://phk.freebsd.dk/pubs/sane2000-jail.pdf

[TN2127] Apple Computer Inc. Technical Note TN2127 Kernel Authorization http://developer.apple.com/technotes/tn2005/tn2127.html

[EFRAD2006] Elad Efrad, NetBSD Security Enhancements, EuroBSDCon 2006, http://www.netbsd.org/~elad/recent/recent06.pdf

[KAUTH9] kauth(9) man-page, kauth – kernel authorization framework http://netbsd.gw.com/cgi-bin/man-cgi?kauth++NetBSD-4.0

[PROPLIB3] proplib(3) man-page, proplib – property container object library http://netbsd.gw.com/cgi-bin/man-cgi?proplib++NetBSD-4.0

BSD implementations of XCAST6

Yuji IMAI, Takahiro KUROSAWA, Koichi SUZUKI, Eiichi MURAMOTO,

Katsuomi HAMAJIMA, Hajimu UMEMOTO, Nobuo KAWAGUTI

XCAST fan club, Japan.

Abstract: XCAST [RFC5058] is a complementary protocol of Multicast. In contrast with the group address of Multicast, XCAST specifies the destinations by the list of unicast addresses, explicitly. Multicast is very scalable in terms of the number of receivers because membership of the group-address destination is implicitly managed on the intermediate routers and the number of receivers is potentially infinite. On the other hand XCAST is very scalable with respect to the number of groups. It is necessary for bi-directional multi-party communication systems such as tele-presence to deal with the large number of groups. We implemented XCAST6, the IPv6 version of XCAST, to prove the concept. Using our implementation, we operate multi-party video conference systems both on experimental overlay network and on native IPv6 network. In this paper, we will describe detailed implementation of XCAST6 on FreeBSD and NetBSD. We will also discuss about simplicity not only of implementation but also of operation.

1. Introduction

XCAST (eXplicit Multi-Unicast) is a protocol to deliver one datagram for small number of destinations simultaneously. It is considered as a complementary mechanism of Multicast. While Multicast is suitable to send a datagram for very large number of receivers, XCAST is good for delivering a datagram to small number of receivers, but capable of dealing with the large number of groups.

We implement XCAST6, the IPv6 version of XCAST, on the various flavor of BSDs to validate usefulness of the protocol. XCAST can be implemented so simply because the protocol itself is designed simply based on existing unicast mechanism. Using our implementation, communities can easily operate multi-party conference systems both on experimental overlay network and on native IPv6 network.

2. XCAST: eXplicit Multi-Unicast

Ordinary, “Multicast” is a system defined by [STD 5]. An IP Addresses in the special IP address range, formerly called Class D of IPv4, is assigned as identifier of the group of hosts or interfaces connected to the Internet. Potentially, any host can join to and leave from the group any time. The embership of the groups are maintained on the routers with distributed manner. A datagram transmitted for the multicast address is relayed and forwarded to all member of the group, duplicated on the routers.

It is a buried history that in a very early discussion of the Internet community,Multicast was not for a virtual group address but the list of unicast addresses [AGUILAR]. In its design, destinations were explicitly embedded in the option header of the IP datagram. Group address extension was introduced as an improvement from Aguilar's. The main point of the improvement was a scalability of the number of receivers. Maximum number of the original Multicast was limited by the length of IP option header. The Internet community considered it must be too small. So, they choose to introduce the special addresses for the destinations. By keeping membership on routers, datagrams can be transmitted for the special addresses as same as for unicast destination. By this extension “Multicast” got very good scalability with respect to the number of members per group, potentially unlimited.

Based on this design choice, “Multicast” deployment was started and difficulties appeared. One of the points is how to maintain the Multicast routing information. [RFC2902] Because the membership of group-multicast is dynamic,

routing information changes so frequently and looks difficult to be aggregated. Some insist it is impossible [SOLA]. Recently, it becomes consensus that scalability of group-multicast with respect to number of groups is not so good.

For some types of multi-party communication, the lack of the scalability of the number of groups is critical. Multi-party video conference is typical example. The participants, the source and the destinations of multicast", are sparsely distributed over the Internet. The system needs the Multicast routes as many as the participants. From the viewpoint of network operators, routers must maintain the multicast routes as many as the transmitters. It is easy and natural to consider the potential multicast transmitters are over hundred-millions, according to the number of users of instant messages or softphones like Skype. Today full routes of the unicast of the Internet exceed 250,000 and IAB considers it is facing serious scaling problems. [RFC4984] That means we need other mechanism than Multicast to realize this type of communication.

XCAST is a re-invention of primitive work of Aguilar's [RFC5058]. One of the improvements from Aguilar's is the way to store the list of destinations. For IPv4, they are encoded in the newer defined option header and for IPv6, the routing header. The number of destinations is up to 124 for IPv6 version. For typical usage of human multi-party communication, this limit would not be problem because it would be difficult for people to make conversation with more than 100 people simultaneously.

As XCAST datagram has an explicit list of unicast addresses, routers can duplicate and forward them using existing unicast routing information without any other like those of Multicast.

One big problem to deploy XCAST in the existing Internet is how to make XCAST datagram pass over the non-XCAST routers. For this purpose, XCAST6 prepares the mechanism called semi-permeable tunnel. The raw XCAST6 datagram starts with IPv6 header with special destination address ALL_XCAST_NODE, the group-multicast address specially assigned. It indicates that the datagram is XCAST6 one and a list of destinations is following in the routing header. The semi-permeable tunnel is encapsulation trick like IP over IP. A semi-permeable XCAST datagram is covered with an additional IPv6 header and a hop-by-hop options header. In the outer IPv6 header, a temporal address is embedded in the destination field that is one of the list of XCAST destinations the datagram has not been reached yet. The hop-by-hop options header marks the need for XCAST routing process. With this preamble headers, a semi-permeable datagram looks like an ordinal IPv6 datagram with a unicast destination. So, XCAST6 datagram can travel through the IPv6 network even that include non-XCAST6 routers. Only when the datagram passes on the XCAST6 routers, it detects the hop-by-hop options header, then checks the list of destinations in the routing header and duplicates the datagram if needed.

3. Implementation

We implemented XCAST6 on NetBSD and FreeBSD kernels. The set of codes consists of the following components:

- interface for user processes
- routing header processing
- the xcst network interface

When a user process issues an XCAST6 packet with the sendmsg(2) system call, the kernel needs to handle the request and send it to the network by processing the XCAST6 routing header. The packet is forwarded by routers and then reaches a node with XCAST6 support. The node should parse the packet, process the routing header, and forward it to the destinations listed in the routing header. If the list of the destinations contains the address of the node, the packet should be passed to the upper layer (UDP or ICMPv6).

The XCAST6 packets are usually sent in the semi-permeable tunneling format. When the kernel sends them to the network, it should encapsulate them. Also, it should be ready to receive the encapsulated packets. The xcst network interface is used for handling encapsulation.

The following sections describe how those components act on processing XCAST6 packets.

3.1. Interface for user processes

User processes can send XCAST6 packets by sendmsg(2). With sendmsg(2), IPv6 extension headers are added to the packets using the msg_control field of the msghdr structure as described in [RFC3542]. On sending XCAST6 packets, ALL_XCAST6_NODES is specified as the destination address in msg_name (regardless of whether semi-permeable tunneling is required or not) and the XCAST6 routing header in msg_control. Here we implemented codes so that the routing header is not rejected as unknown but is stored and associated with the message for later processing; we don't process the routing header itself here but simply pass it to the packet output routine.

On receiving side, user processes can receive XCAST6 packets just the same way as unicast packets. There is no need to change receiver code of socket interface. Also, we don't need to change code for joining/leaving multicast groups since XCAST6 doesn't require keeping track of joining/leaving unlike group multicast or source specific multicast.

3.2. Sender side

The ip6_output() function builds IPv6 packets and outputs them to network interfaces. The XCAST6 routing header associated with the message is also placed in the packet by the function, but is not processed yet.

At the end of ip6_output(), the function outputs the packet to the link layer according to the routing table. By routing packets destined for ALL_XCAST6_NODES to the xcst network interface, we can pick up them there with keeping changes to the ip6_output() function as few as possible.

The output function of the xcst interface checks whether the packet has a XCAST6 routing header or not. Packets with XCAST6 routing headers are passed to the routine of XCAST6 routing header processing. Packets that don't have XCAST6 routing headers are dropped.

3.3. Routing header processing

The XCAST6 routing header contains a destination address list and a bitmap. The bit in the bitmap indicates whether the packet should be delivered to the corresponding address or not. The address can be classified from the viewpoint of reachability as follows:

- unreachable
- assigned to the local node
- directly reachable from the local node (ex. on the same link)
- reachable via routers

As for addresses reachable via routers, there may be addresses that next hop routers are the same. We should group addresses by the next hop and send a packet for each group in order to minimize the number of copies of the packet in the network. For this reason we need to look up the routing table first for all the addresses that the packet needs to be sent. Once the grouping of the addresses is done, the packet can be sent for each next hop, normally encapsulated in the semi-permeable tunneling format. Addresses directly reachable from the local node are handled as no other addresses shares the same next hop. If the address list in the routing header contains the address

assigned to the local node, the local node is also expected to receive the packet. Such a packet is passed to the upper protocol layer (UDP or ICMPv6) in addition to being forwarded to next hops.

3.4. Forwarding XCAST6 packets

The node that supports XCAST6 should forward the incoming XCAST6 packets sent by other nodes in addition to sending packets requested by user processes. This section describes how incoming XCAST6 packets are processed.

The XCAST6 packets are usually encapsulated in the semi-permeable tunneling format. Packets without encapsulation can easily be detected as the XCAST6 packets because the destination addresses are ALL_XCAST6_NODES. The destination address of encapsulated packets may not be the local node address, but the hop-by-hop options header in the packet indicates that each router on the path should inspect the header deeper. We needed to add the code to process hop-by-hop options header with the type of XCAST6 and to pass the packet to the extension headers routines.

Also, packets with encapsulation need to be decapsulated. We implemented the decapsulation in the xcst interface by using encap_* functions provided by netinet/ip_encap.c just like gif(4) or gre(4).

The extension headers of the incoming XCAST6 packets are then processed as normal IPv6 packets. We added the codes where the routing header is handled in order to pass the packet to the routine of XCAST6 routing header processing. The same routine that is described on sending the packets is also used here.

3.5. The xcst network interface

The xcst network interface is a pseudo device that is introduced for picking up XCAST6 packets. As described above, packets that are going to be sent by the local node are picked up at the output routine of the xcst interface with keeping changes to the ip6_output() function as few as possible. Also, the encapsulated packets is picked up by the xcst interface and then decapsulated. The xcst interface contributes for localizing the changes to the existing IPv6 implementation and simplify the XCAST6 implementation.

3.6. ICMPv6 support

It became apparent that diagnostic packets were necessary during the deployment of XCAST6. The path of an XCAST6 packet between a sender and one of receivers is usually not the same as the path of a unicast packet between the sender and the receiver. On IPv6 unicasts, the sender can check reachability to the receiver by the ICMPv6 Echo Request message. On XCAST6, the ICMPv6 Echo Request message is unusable since the destination address field in IPv6 header is a multicast address.

In order to check the reachability from the sender to one of the receivers, we experimentally introduced a sub-option in Hop-by-hop options header. The sub-option specifies which destination should respond with the ICMPv6 Echo Reply message. We have slightly modified the receiver routine of ICMPv6 to implement this functionality. The “ping6x” userland program has also been implemented.

4. Related work

In this section, we describe the related implementations on XCAST6 and the operational activities called “X6bone”.

4.1. Group management

The group management for XCAST6 is completely separated from the function of forwarding or routing. That enables application developers to utilize their own application-specific group management functions free from group as well as group multicast address schemes, There are several implementations that help XCAST

application developers to make the group formation and management.

4.1.1. Xcgroup

The application of XCAST6 requires the list of destinations. We created a CGI script for httpd and a client program called "xcgroup". The application user invoke “xcgroupsrv” with a URL of CGI for managing the information of group name and its membership (i.e., participating nodes). Client “xcgroup” periodically sends a query to the “xcgroupsrv” CGI script by http and the CGI script acquires the IPv6 source address of the http query and records it in a list. Then it retrieves all the IPv6 source addresses recorded in the list and provides the client program that information. As a result, consistent membership information among the participants can be provided. Xcgroup announces the list of acquired unicast addresses for Mbone tools via mbus [RFC3259] so that XCAST applications on the same host can share the address list.

4.1.2. Group Management using SIP

The method how to arrange the multiple IP address for XCAST session had been discussed in [SIPSDP]. It proposed the method to exchange IP addresses and port numbers between participants using SIP and the method to carry multiple port numbers in XCAST header.

4.2. Testbed and deployment activities

To prove the concept of XCAST, WIDE XCAST WG started operating the testbed called X6Bone.It consists of a virtual overlay IPv6 network over IPv4 network by tunnels connecting many SOHO routers to a HUB XCAST router via ADSL and FTTH. The NAT traversal tunnel is made by the implementation of DTCP: Dynamic Tunneling Configuration Protocol[DTCP] and L2TP[L2TP] which invoke DHCPv6 PD over PPP over L2TP over UDP over IPv4. The XCAST6 datagrams traveling on the X6Bone could be branched at the HUB routers, thus eliminating the problem of inefficient daisy-chained connections by semi-permeable tunnel trick.

4.2.1. XCAST6 enabled VIC and RAT

Well-known Mbone tools VIC and RAT have been modified and integrated with XCAST6. On the sender side, original VIC and RAT bind the transmission socket to the multicast group address. We modified the procedures to call library function, XCASTAddMember() to append unicast addresses of the participated node that is provided by group management mechanism. On the receiver side, no modification is necessary since the payload of an XCAST6 datagram can be acquired by calling ordinary recv() function. The total amount of additional code is less than 200 lines for vic and 150 lines for RAT.

We have utilized the modified VIC and RAT on the X6bone. Next section explains the deployment activities with them.

4.2.2. Deployment activities

We envision that typical target usage of XCAST as casual conversations to exchange emotional atmosphere among private community like families and friends. Based on this assumption, we made various trials on the X6bone including the open source developer BoFs, the wedding ceremony, conversations with person in transportation, moving bicycle[ROMAIN] and boarding on the airplane flying over north pole. By January of 2008, the number of groups joining the meeting has increased to 10 organizations including BSD User groups in Asia and France.

In order to make these kind of conversation easily and fruitful, many related tools were developed utilizing *BSD variants as follows.

- Fukidashi-kun: Tools to display words on his/her VIC screen just like mumbling by Yasushi Oshima of Nagoya BSD user club.

- The device driver modifications of Web cam for FreeBSD/NetBSD by Takafumi Mizuno of Nagoya BSD user club.

- Instant XCAST6 CD: Bootable *BSD environments with XCAST6 packages. "Ebifurya" based on NetBSD. "FreeSBIE" based were contributed by Daichi Goto.

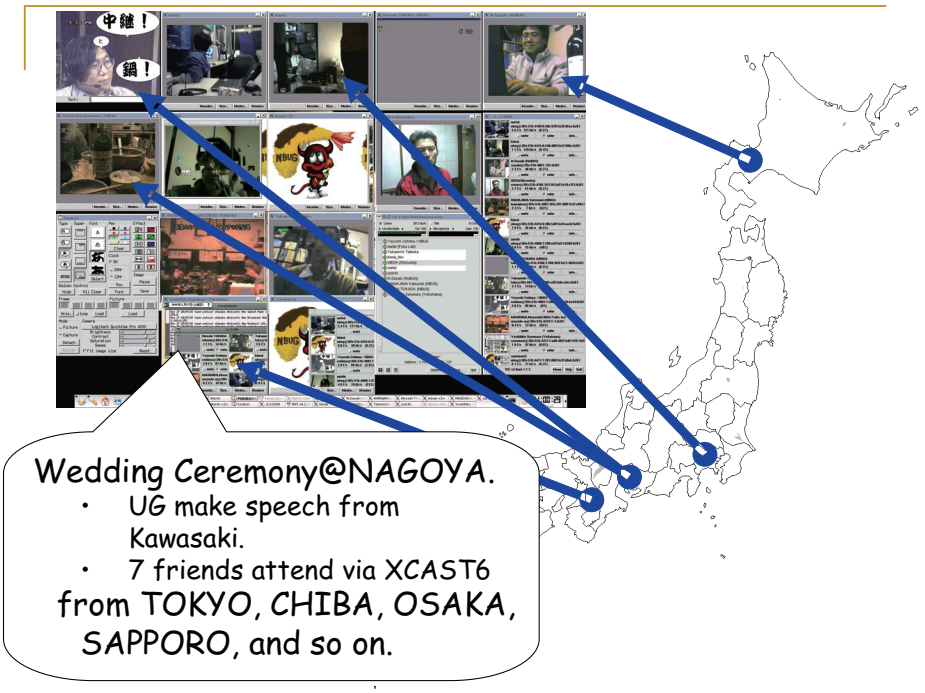


Figure 1: example screen shot of XCAST meeting on X6Bone in Japan.

5. Conclusion

We implemented XCAST6 protocol onto FreeBSD and NetBSD. In order to realize XCAST logic, we utilize existing unicast basis *BSD intrinsically equipped. As a transmission node, XCAST packet is just formulated and triggered to send out in the way of extended socket API, [RFC3542]. Kernel determines directions of the datagram to be forwarded looking up the next hop information of unicast routing basis and branches if needed. Tunneling pseudo device helps the datagram passing over the non-XCAST networks.

To deploy the XCAST6 network, we can use many tools to establish IPv6 connectivity without any modification. We operate X6Bone, the experimental XCAST6 network, using l2tp, dtcp, PPPoE and Ethernet emulation of Packetix. We can use such tools if they exchange ordinal IPv6 unicast datagrams.

Style of the interface for application keep similar with one of group multicast so that MBone tools can be easily modified to handle XCAST6.

We made the extended version of unicast reachability diagnosis tools such as ping6, traceroute6 with small modification with ICMP6 functions. Using these tools, it become very easy to check XCAST6 reachability compared with the group multicast specifics like mtrace.

With our X6Bone experience, we discussed with IETF community and convinced them

that there should be other class of multicast than group-address one. For the topic, SAM RG: Scalable Adaptive Multicast Research Group was made in IRTF. We keep reporting XCAST operational experiments including implementing status for various OS, Linux, Windows and *BSD of course.

Acknowledgment

Yoichi SHINODA of JAIST, Jun-ichiro "itojun" Hagino of KAME project and Hideaki YOSHIFUJI of Usagi project gave us great advice for our XCAST6 implementation. Takamichi TATEOKA, Sohgo TAKEUCHI and Tomo TATSUMI has been contributing the operation of X6bone. Rick Boivie of IBM and John Buford of Avaya kept encouraging us to make XCAST concept as an experimental RFC. And we express our greatest gratitude to WIDE Project, Asian Internet Interconnection Initiatives(AI3), Korean IPv6 community, ETRI and IRISA for the cooperation of our experiments.

Reference

[STD 5] S. Deering, "Host Extensions for IP Multicasting", STD 5, RFC 1112, September 1989.

[AGUILAR] L. Aguilar, "Datagram Routing for Internet Multicast- ing", Sigcomm84, March 1984.

[SOLA] M. Sola, M. Ohta, T. Maeno. “Scalability of Internet Multicast Protocols”,INET'98, http://www.isoc.org/inet98/proceedings/6d/6d_3.htm

[RFC2902] S. Deering, S. Hares, C. Perkins, and R. Perlman, "Overview of the 1998 IAB Routing Workshop", RFC 2902, August 2000.

[RFC3259] J. Ott, et al., "A Message Bus for Local Coordination", RFC3259, April 2002

[RFC3542] W. Stevens, M. Thomas, E. Nordmark, T. Jinmei. “Advanced Sockets Application Program Interface (API) for IPv6”. RFC3542, May 2003.

[RFC4984] D. Meyer, L. Zhang, K. Fall, “Report from the IAB Workshop on Routing and Addressing”, RFC4984, September 2007

[RFC5058] Boivie,R.,N. Feldman, Y. Imai,W. Livens,D. Ooms, “Explicit Multicast (Xcast) Concepts and Options”, RFC5058, November 2007.

[SAINT] Y. Imai, H. Kishimoto, M. Shin, Y. Kim, “XCAST6: eXplicit Multicast on IPv6”, IEEE Symposium on Applications and Internet Workshops, January 2003.

[DTCP] H.Umemoto, DTCP homepage, http://www.imasy.or.jp/~ume/published/dtcp/

[L2TP] H.Umemoto, L2TP document, http://www.imasy.or.jp/~ume/presentation/CBUG-20070421/l2tp-pd.odp

[ROMAIN] R. KUNTZ,”The E-Bicycle Demonstration Setup on Tour de France 2006”,http://member.wide.ad.jp/tr/wide-tr-nautilus6-ebicycle-tour-de-france-00.pdf

[SIPSDP] B. Van Doorselaer, “SIP for the establishment of xcast-based multiparty conferences”, http://www.tools.ietf.org/html/draft-van-doorselaer-sip-xcast-00, July 2000.

Using FreeBSD to Promote Open Source Development Methods

Brooks Davis, Michael AuYeung, Mark Thomas
The Aerospace Corporation
El Segundo, CA
{brooks,mauyeung,mathomas}@aero.org

Abstract

In this paper we present AeroSource, an initiative to bring open source software development methods to internal software developers at The Aerospace Corporation. Within AeroSource, FreeBSD is used in several key roles. First, we run most of our tools on top of FreeBSD. Second, the ports collection (both official ports and custom internal ones) eases our administrative burden. Third, and most importantly the FreeBSD project serves as an example and role model for the results that can be achieved by an open source software projects. We discuss the development infrastructure we have built for AeroSource based largely on BSD licensed software including FreeBSD, PostgreSQL, Apache, and Trac. We will also discuss our custom management tools including our system for managing our custom internal ports. Finally, we will cover our development successes and how we use projects like FreeBSD as exemplars of open source software development.

1 Introduction to Aerospace

The Aerospace Corporation operates a Federally Funded Research and Development Center for National Security Space. From the corporate web site[Aerospace]:

> Since 1960 The Aerospace Corporation has operated a federally funded research and development center in support of national-security, civil and commercial space programs. We're applying the leading technologies and the brightest minds in the industry to meet the challenges of space.

The company employs approximately 2400 engineers on a wide range of disciplines. In today's engineering climate, a large potion of these engineers write software, up to thousands of programs by some counts.

Due in part to the fact that these engineers are not trained software developers, the quality of software and software development methods varies widely. Since Aerospace helps oversee the development of massive software projects, we have a significant number of people who are trained to develop these types of software. They represent one of two historical groups of developers at Aerospace. They use big, heavy development processes which produce reliable software suitable for all sorts of applications, but require significant numbers of full-time developers and large paper trails.

The other camp takes a laissez-faire approach to software development. They tend to use little or no processes to the point that one of the more advanced groups was using a shared file system for development with a white board to lock files before the AeroSource team started working with them. As would be expected, this approach to development yields highly variable results. A number of pieces of software are very useful and some are even distributed outside the company, but even with those we've heard reports of problems like features disappearing between releases.

Past attempts to encourage developers of the more important pieces of software to adopt more rigorous development practices have met with limited success. One problem is that these developers quite reasonably fear the more heavy weight processes they see employed to build big systems. In addition to the process overhead of these methods developers worry about the cost of tools and the need to learn new tools. Other problems include inertia in the face of demanding schedules.

AeroSource is our current attempt to bring modern software development methods to the more ad-hoc development projects within Aerospace. We are promoting the idea that using tools and methods from open source software development provides a useful midpoint between big, expensive software methods and current practices. In addition to promoting open

source software and development methods, we are also promoting the open source development philosophy within the company. We call this internal open source, enterprise source software. Enterprise source software enshrines principles of open source, but is restricted to the enterprise. Users of enterprise source are free to read the source code, build and run it, make changes to it, and redistribute modified versions of it as long as they do so within the bounds of the company. External software distribution remains governed by existing processes.[1]

In the rest of this paper we discuss our experiences designing, developing and promoting AeroSource and the enterprise source concept. We discuss our use of FreeBSD throughout, both as the foundation of our infrastructure and as an example of both what can be achieved with open source methods and one set of highly effective methods. In the next section we discuss open source and enterprise source software. We then discuss our efforts to promote the enterprise source concept and the reactions we have encountered. Coming from the open source software world, we often find it hard to credit the issues people raise, but we have found it is critical to do so if we are going to convince people to support enterprise source. As part of this section we discuss our implementation of AeroSource, a resource for collaborative software development using FreeBSD and other open source technologies. We also talk about our successes and failures in recruiting projects to use it. Finally we conclude with a discussion of future directions for AeroSource.

2 Open Source and Enterprise Source Software

According to the Open Source Initiative "open source is a development method for software that harnesses the power of distributed peer review and transparency of process."[OSI] Many definitions of open source exist including the OSI *Open Source Definition*[OSD]. For our purposes we define an open source project as one that allows the four freedoms defined by the Free Software Foundation[Wikipedia] (the wording below is ours):

- The freedom to run the software, for any purpose
- The freedom to study how the software works, and adapt it to your needs
- The freedom to redistribute copies
- The freedom to improve the software, and release the improvements to anyone for the benefit of all

Advocates of the open source development mode argue it has numerous benefits. Those benefits include "better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in"[OSI]. These benefits derive directly from the four freedoms listed above. The quality and reliability claims derive from the idea that with more people working on the code, bugs are more likely to be discovered and fixed. There is a common idea in the open source community that "given enough eyeballs, all bugs are shallow"[CatB]. In our experience this is true for simpler bugs, but for very complex issues, there often are not enough people who understand the problem for this to work. One case where we do find quality to be better is adherence to unified code styles. In our experience and that of others we have talked to, large open source projects tend to have cleaner, more readable source code that internally developed code. Flexibility and protection from lock-in derive from the fact that users can modify the software themselves or hire someone to make the changes they want. As a result users can adapt to unforeseen software needs and add the functionality they want rather than things the developers' marketing department thinks they can sell. Lower cost is obvious since the software is free.

In addition to these benefits, open source development methods provide other advantages within the enterprise. Because open source developers often have another day job, they generally can not be bothered with excessively involved procedures. Thus, open source projects tend to use processes that are low friction. By adopting these processes, developers can build higher quality software without resorting to traditional, high overhead methods. Another benefit within the enterprise is that if people publicly share their code and others can find it, duplication of effort can be reduced. For example the world only needs so many tools to parse the same telemetry format.

Much of the software produced at Aerospace that would benefit from the wider exposure open source development brings is not possible or practical to release to the general public for a range of technical, legal, and political reasons. When promoting open source methods, we discovered we needed a term to describe the internal use of those methods since simply talking about open source or internal open source often lead people to think we would be posting their code to Source Forge or another public site. To capture this concept we coined the term *enterprise source software*. Enterprise source software is everything that open source software is, but restricted to an enterprise. All of the four freedoms hold for enterprise source soft-

[1] We have ambitions to encourage the release of more Aerospace code as open source. Promoting enterprise source the first of several steps in that direction.

ware, but with the added restriction that it must stay within the organization. At Aerospace this means that enterprise source source software may leave the company only though official software release channels.

We believe that the growth of enterprise source at Aerospace will improve the quality of the software we develop and increase the skill of our software developers.

3 Promoting Enterprise Source

In an effort to improve the development practices used by the less formal software projects at Aerospace we are working to promote the enterprise source concept for internal use. Our efforts center on encouraging the internal publication of source code and the use of open source tools and methods to develop that software. AeroSource, our internal collaborative software development environment, lies as the heart of our efforts. It allows users to "get their feet wet" without all the effort of maintaining their own systems. We discuss AeroSource in detail later in this section.

3.1 Promotion Efforts

Our promotional efforts are targeted in several different directions. We work to educate Aerospace employees on the benefits of open source software and development methods and encourage them to adopt them where practical. We also work to convince management of these benefits to support them from above in addition to our more grass roots efforts.

The most basic level of advocacy is using open source software or open source derivatives. Most people in our organization use BSD, Linux, or Mac OS exclusively and all our department servers are hosted on open source OSes. We also host a variety of semi-official corporate services including open source software mirrors, a list server, and a number of wikis.

The next level of advocacy is formal open source education. We have given a number of lectures on benefits of open source and open source development methods at internal forums. We also developed a tutorial on open source development methods which we presented at the Ground System Architecture Workshop in 2007[GSAW]. In these talks we promote the variety of great software available as open source both for it's own sake and to demonstrate that the non-traditional development efforts involved can and do produce top quality software.

FreeBSD is a key component of this promotion effort. We use it extensively in our infrastructure and because we are extremely familiar with it's development process, we can speak with authority on the processes involved. This is helpful in convincing people that we really do know what we are talking about with regard to open source development. The availability to resources such as the *FreeBSD Developers Handbook*[GSAW] and Robert Watson's *How the FreeBSD Project Works*[Watson] talk helps in this regard. Other projects we use as examples include Ganglia, K Desktop Environment (KDE), and Linux.

The most specific form of advocacy is AeroSource. With AeroSource we give developers the tools they need and help train them in the tools and best practices for using them. We help with things like repository layout, process, and usage. Eventually we hope to provide continuous integration tools like tinderboxes.

3.2 An Overview of AeroSource

AeroSource provides collaborative tools to software projects including tightly integrated version control, bug tracking, and a wiki. We also provide e-mail lists that can be integrated with the bug tracking and version control systems. This functionality is provided by Trac and GNU Mailman with version control provided by Subversion. Trac is one of several projects that aim to create a complete, web-based collaborative environment for open source development. Trac is open source (BSD licensed) and is built on top of a large stack of other open source software. In our installation we use Subversion for version control, PostgreSQL as the database, Apache for the web servers, and FreeBSD for the operating system.

When developing AeroSource we looked at several alternatives including GForge, SourceForge, and building our own system. GForge was rejected due to prior experience: it worked, but upgrades were time consuming and difficult. SourceForge was not an option because we wanted to keep software internal and we were not prepared to purchase the commercial version. After finding Trac we concluded that any benefits from building our own system were likely to be minimal compared to starting with an already working system. Systems we did not consider at the time but would consider today include CollabNet and Retrospectiva.

Today, AeroSource contains over 50 projects ranging from small repositories of scripts to large established projects. Our most prominent win is is the Satellite Orbital Analysis Program (SOAP), a cross platform (MacOS, UN*X, and Windows) 3D orbit visualization

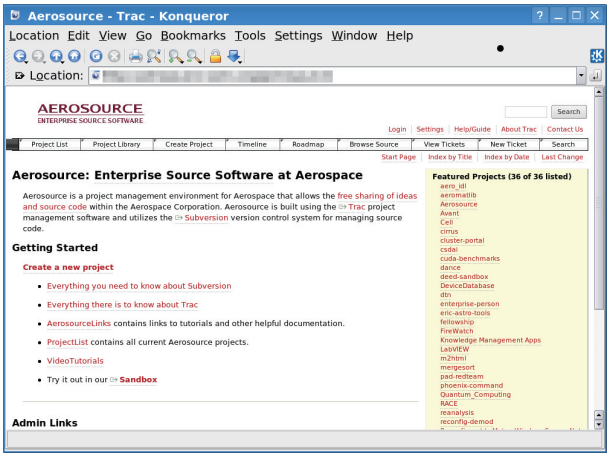


Figure 1: AeroSource.aero.org

and analysis program. SOAP has been under development for more than a decade in numerous forms and is one of Aerospace's crown jewels, so winning the development team over was a major milestone for AeroSource. Other projects include collections of Perl, IDL, and Matlab scripts and configuration files for a number of internal systems including AeroSource it self.

Users seem generally pleased with Trac and Subversion, but we have encountered a few problems. The most severe one is that the wiki implementation has no support for simultaneous edits. In it's current form, if two users edit the same page, the second user loses all their work when they submit. This is arguably the worst of all possible behaviors. Otherwise, Trac is working fairly well for us. The only other significant issue we have found is that some parts of Trac are more easily customized that others. There is a solid plugin framework, but if what you want cannot be accomplished through it maintaining modifications can be complex.

3.3 Maintaining AeroSource

With AeroSource, we do our best to "eat our own dog food" and use Trac and Subversion as much as possible to aid in maintenance. We store configuration, custom Trac modules, and administration scripts in AeroSource. The project homepage is a Trac instance and we use the ticket system to track most maintenance operations. The AeroSource front page can be seen in Figure 1.

The maintenance operations of AeroSource are fairly normal, but a few things stand out. We use `freebsd-update` to keep the base system up to date and install most of our software using the FreeBSD ports collection.

One one very useful customization we have developed is a set of local ports stored in an AeroSource hosted subversion repository. We call this collection AeroPorts. We check our ports out under `/usr/ports/aero` and ports live in `<category>/<port>` subdirectories. Figure 2 shows the top-level `Makefile` and Figures 3 and 4 show an example of the make files for each `<category>` sub directory. With this setup, we can easily maintain local ports of things that are not useful to the general public, or custom modifications of existing ports to perform non-standard tasks. One example of this is a custom version of the `security/pam_ldap` port that authenticates based on LDAP queries on userid's instead of usernames. Another is the `misc/aero-bootstrap` port which is a meta-port we use to install basic administrative tools on our FreeBSD machines. This method of incorporating local ports in the ports tree is based on a suggestion by Scot Hetzel on the freebsd-ports mailing list[Hetzel].

To simplify management of these local ports we have a wrapper for the `portsnap` and `svn` commands call `apt` (Aerospace Ports Tool). The `apt` command performs an `svn update` and `portsnap update` using the "`-l descfile`" option to refresh the ports tree and build combined `ports/INDEX*` files as needed. This yields functionality virtually identical to that of `portsnap`, but with full integration of our local ports.

3.4 Results

Thus far, our efforts have met with a number of successes, but we still have some work to do. As we mentioned in Section 3.2 we have recruited over 50 projects to AeroSource thus far. We have also had some projects that were not able to become enterprise source software express interest in the tools.

The import of the Satellite Orbital Analysis Program (SOAP) to AeroSource represents a major win and were in fact funded to make the transition. The first release has not yet been cut, but development is well under way and the developers have made significant progress in using the tools.

Some other projects have resisted the idea for a variety of reasons. Some want absolute control over the code they perceive ownership of. Reasons for wanting that control range from not wanting others to see their code to wanting to ensure that no one releases a modified version lest they be blamed for bugs introduced by someone else. We have had some success with the first case and a bit with the second, but we have not won all

```
COMMENT=        Ports specific to Aerospace Corp

SUBDIR+=        archivers
SUBDIR+=        astro
SUBDIR+=        misc
SUBDIR+=        net
SUBDIR+=        science
SUBDIR+=        shells
SUBDIR+=        sysutils

descfile:
        @cd ${.CURDIR}; ${MAKE} describe | grep -v '^===>' > descfile

.include <bsd.port.subdir.mk>
```

Figure 2: aero/Makefile

```
COMMENT=        Local Aerospace system utilities

SUBDIR+=        apt
SUBDIR+=        diskprep-aero
SUBDIR+=        macports
SUBDIR+=        powerctl

.include <bsd.port.subdir.mk>
```

Figure 3: aero/sysutils/Makefile

```
# This file needs to be copied into every aero/*/ subdirectory to set
# common variables.

# Used to set the origin of the local port
PKGORIGIN=      aero/${PKGCATEGORY}/${PORTDIRNAME}

# Used in the local ports tree to set dependencies on other local ports.
AEROPORTSDIR=  ${PORTSDIR}/aero

#Uncomment if you want your local packages to have a "-aero" suffix.
#PGKNAMESUFFIX?= -aero
```

Figure 4: aero/sysutils/Makefile.inc

the arguments. In once case we have even heard that developers have threatened to quit if forced to open their code. A variant of the argument that only the current developers know enough to modify the code is that only the developers can use the code properly. We agree this can happen, but think that is not in and of itself a good reason not to open the code. These were all arguments we expected to some extent based on past experiences. We also ran into a couple we were not expecting. In one case some people felt other developers should implement a certain algorithm as a right of passage. We weren't sure how we felt about that one. In another case, developers were concerned that people might like their code so much they should improve it and then the developers would have to incorporate the improvements. We thought seemed like a good thing rather than a problem.

4 Future Directions & Conclusions

We are generally pleased with our progress in introducing open source software development methods to Aerospace. We a large organization that is largely staffed by engineers with decades of experience, we do not expect to convert everyone over night. We feel many pieces of software within Aerospace could also benefit from full, open source release, but for now we are content with modernizing internal development efforts.

AeroSource itself is functioning very well. We hope to continue to incremental improve the management processes to make project setup easier and to enhance the ability of our users to perform their own project maintenance. We will also continue to monitor Trac development and the development of competing systems to provide our users with the best environment we can. Other future work includes more tutorial materials and more streamlined startup processes for new projects.

References

[Aerospace] The Aerospace corporate web site. October 11, 2007. `http://www.aero.org/`

[CatB] Eric S. Raymond. *The Cathedral and the Bazaar*. September 11, 2000. `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`

[GSAW] The FreeBSD Project. *The FreeBSD Developers' Handbook*. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/index.html`

[GSAW] Brooks Davis, Sam Gasster, Jorge Seidel, Mark Thomas. *Open Source Software Methods in Ground Systems*.

[Hetzel] E-mail to the `freebsd-ports@freebsd.org` mailing list. November 14, 2006. `http://docs.freebsd.org/cgi/mid.cgi?db=irt&id=790a9fff0611141011q4bd9ee97h9357e6d959f95abb@mail.gmail.com`

[OSD] The Open Source Initative's Open Source Definition. July 7, 2006. `http://www.opensource.org/docs/osd`

[OSI] The Open Source Initative web site. January 23, 2008. `http://www.opensource.org/`

[Watson] Robert N. M. Watson. *How the FreeBSD Project Works*. In Proceedings, 2006 EuroBSDCon, Milan, Italy.

[Wikipedia] Wikipedia article on free software. January 23, 2008. `http://en.wikipedia.org/wiki/Free_software`

Send and Receive of File System Protocols: Userspace Approach With *puffs*

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

ABSTRACT

A file system is a protocol translator: it interprets incoming requests and transforms them into a form suitable to store and retrieve data. In other words, a file system has the knowledge of how to convert abstract requests to concrete ones. The differences between how this request translation is handled for local and distributed file systems are multiple, yet both must present the same semantics to a user.

This paper discusses implementing distributed file system drivers as virtual file system clients in userspace using the Pass-to-Userspace Framework File System, *puffs*. The details of distributed file systems when compared to local file systems are identified, and implementation strategies for them are outlined along with discussion on where and how to optimize for maximal performance.

The design and implementation of an abstract framework for implementing distributed file systems on top of *puffs* is presented. Two distributed file system implementations are presented and evaluated: psshfs, which uses the ssh sftp protocol, and 9puffs, which uses the Plan9 9P resource sharing protocol. Additionally, the 4.4BSD portal file system and *puffs* user-kernel communication are implemented on top of the framework. The performance of userspace distributed file systems are evaluated against the in-kernel NFS client and they are measured to outperform NFS in some situations.

Keywords: distributed file systems, userspace file systems, software architecture

1. Introduction

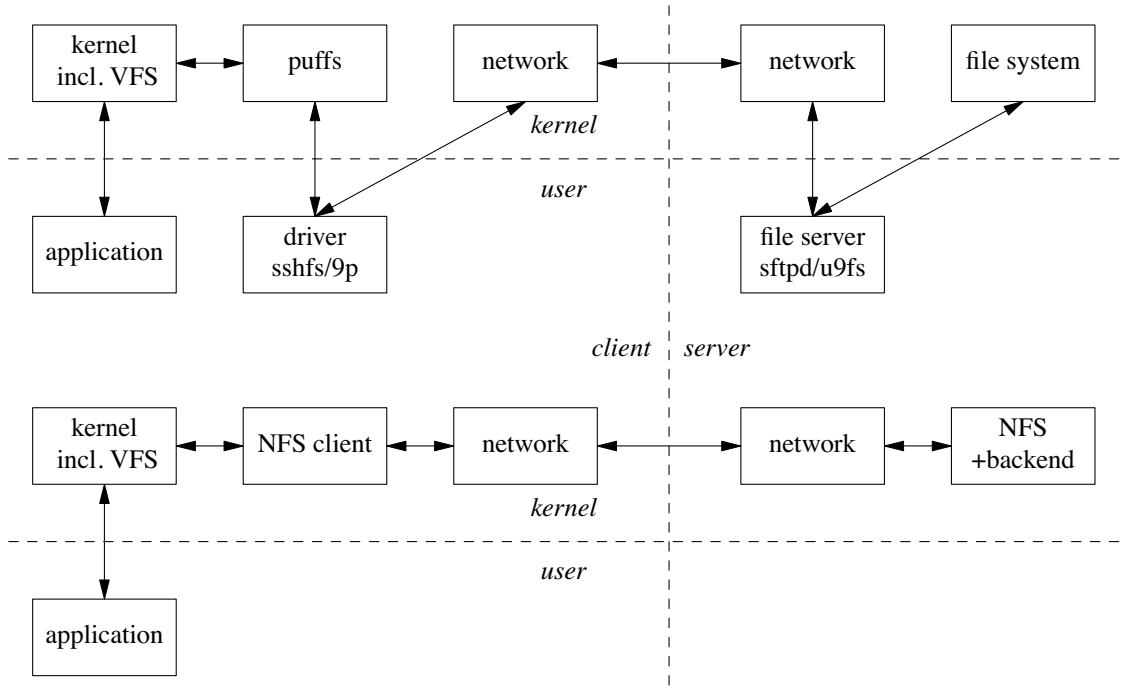
One taxonomy for file systems is based on where they serve data from:

- Fictional file systems serve a file system namespace and file data which is generated by the file server. Examples are *procfs* and *devfs*.
- Local file systems serve data which is located on the local machine on various types of media. Examples are *FFS*, *cdfs* and *tmpfs* for hard drive, CD and memory storage, respectively.
- Distributed file systems serve non-local data, typically accessed over a network. Examples are *NFS* [1] and *CIFS* [2].

A typical distributed file system will serve its data off of a local file system, but it is also free to serve it from a fictional file system, its own database or even another distributed file system.

Distributed file systems can be subdivided into two categories. In client-server type file systems all served data is retained on dedicated servers. The examples *NFS* and *CIFS* given earlier are examples of this kind of a file system. Peer-to-peer file systems treat all participants equally and all clients may also serve the file system's contents. Examples of peer-to-peer file systems are *ivy* [3] and *pastis* [4]. We concentrate on client-server systems, although all discussion is believed to apply to peer-to-peer systems as well.

Figure 1:
Structural comparison of userspace and kernel distributed file system drivers



Core operating system services such as file systems are historically implemented in the kernel for performance reasons. With ever-growing machine power, more and more services are being pushed out of the kernel into separate execution domains. This provides both improved reliability and an easier programming environment. The idea of abandoning a monolithic kernel itself is not new and has been around in systems research for a long time with operating systems such as Mach [5]. The idea has, however, recently gained interest especially in file systems because of the FUSE [6] userspace file system framework.

It is, however, incorrect to assume that a userspace file system implementation will solve all problems by itself. In fact, it is nothing more than pushing the problems of implementing a file system from one domain to another.

This paper explores implementing distributed file systems in userspace on NetBSD [7]. While details are about NetBSD, the ideas are believed to have wider usability. The attachment for file systems is provided by *puffs* [8]. The file systems interface already exported to userspace is not replaced for distributed file systems [9], but rather extended by building a framework upon it.

The following contributions are made:

- Explaining file system concepts relevant to implementing distributed file systems in userspace.
- Presenting the design and implementation of a framework for creating distributed file systems in userspace.

Two file systems have been implemented:

- **psshfs**: a version of the ssh file system written specifically to use the features of *puffs* to its maximum. As its backend, psshfs uses the ssh sftp [10] sub-protocol.
- **9puffs**: a file system client implementing the Plan9 9P [11] resource sharing protocol.

Unforeseen uses include:

- **portalfs**: the 4.4BSD portal file system
- **puffs**: by treating *puffs* itself as a peer-to-peer file system, the framework can be applied for transmitting requests from and to the kernel.

The remainder of this paper is organized as follows. Chapter 2 gives a very short overview of the concepts of *puffs* relevant to this paper. Chapter 3 presents an overview of what a file system is and points out key differences between local and

distributed file systems from an implementor's point of view. Chapter 4 presents a framework for implementing distributed file systems. Chapter 5 contains experimental results for the implementations presented in this paper. Chapter 6 provides conclusions and outlines future work.

2. Short Introduction to *puffs*

This section provides readers unfamiliar with *puffs* the necessary overview to be able to follow the paper. A more complete description of *puffs* can be found elsewhere [8,12].

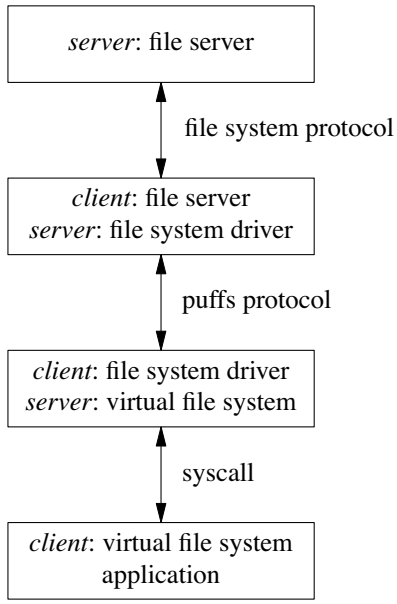
puffs is a framework for building file system drivers in userspace. It provides an interface similar to the kernel virtual file system interface, VFS [13], to a user process. *puffs* attaches itself to the kernel VFS layer. It passes requests it receives from the VFS interface in the kernel to userspace, waits for a result and provides the VFS caller with the result. Applications and the rest of the kernel outside of the VFS module cannot distinguish a file system implemented on top of *puffs* from a file system implemented purely in the kernel. Addtionally, the kernel part of *puffs* implements the necessary safeguards to make sure a malfunctioning or mischievous userspace component cannot affect the kernel adversely.

For the implementation of the file system in userspace a library, libpuffs, is provided. libpuffs not only supplies a programming interface to implement the file system on, but also includes convenience routines commonly required for implementing file systems. An example of such convenience functionality is the distributed file system framework described in this paper.

A file system driver registers a number of callbacks with libpuffs and requests the kernel to mount the file system. The operation of a file system driver is driven by its event loop, in which the file system receives requests, processes them and sends back a response. Typically, a file system driver will want to hand control over to `puffs_mainloop`() after initialization and have it take care of operation. For example, the file system drivers described in this paper hand control over to the mainloop. Nevertheless, it is possible for the file system also to retain control with itself if it so desires and dispatch incoming requests using routines provided by libpuffs.

Since distributed file system operations cannot usually be completed without waiting for a response from a server, it is beneficial to be able to have multiple outstanding operations. In most programs this is accomplished by threads or an event loop with explicitly stored state. *puffs* takes a different route: it provides cooperative multitasking as part of the framework and allows file system builders to schedule execution when needed. This, as opposed to using threads, means that the file system driver is never scheduled unexpectedly from its own point of view. Each execution context has its own stack and machine context, so yielding and continuing can be done with minimal programming involvement and without explicitly storing register and stack state.

Figure 2:
Multiple Levels of Client and Server



Every file system callback dispatched by the library has an associated execution context cookie, *puffs_cc*. This is used to yield execution by calling `puffs_cc_yield`(). Execution is resumed by calling `puffs_cc_continue`() on the same context cookie. The cookie may be passed around like any variable. It is invalidated once the request has been handled.

3. Structure of a Distributed File System

Distributed file system architecture along with this paper's terminology is presented in *Figure 2*. The term *file server* is used to describe the entity servicing the file system namespace and file contents using the file system protocol. The *file system driver* translates requests from the *kernel virtual file system* to the file server.

Distributed file systems operating over the network send out queries to the server to satisfy requests. Queries include an identification tag, which is used to pair responses from the server to issued requests. Of the file system protocols discussed in this paper, the Remote Procedure Call [14] mechanism used by NFS contains a transaction identifier, XID, ssh sftp [10] uses a 32bit request identifier and 9P [11] uses a 16bit tag.

The format of a message inside query frames is dependent on the file system protocol. However, at least the following operations, in some form or another, are common:

- open files to create file handles (and close file handles)
- read and write given file handle
- read the entries in a directory
- get and set the attributes of a file
- create and remove files, directories and special files

The discussion in the rest of this chapter applies to both the 9P and sftp protocols, although some of the mentioned features have been so far implemented only for psshfs. 4.4BSD NFS [15,16] is used for comparison in select places.

3.1. Network vs. Local Media

File system protocols commonly use TCP[1] for transport. A TCP connection is effectively a FIFO queue with latency and bandwidth characteristics. Once data is placed into the network socket, it will be transmitted in-order. This means that on a slow link with a large amount of data already in the buffer, it can take several seconds before anything inserted into the buffer will reach the peer. To take a concrete example, consider one thread doing a bulk read of a file and another thread doing `ls`. If several hundreds of kilobytes of incoming data has been requested and already queued into the socket by the server, it will take several seconds for the response to the directory read to reach the requesting end. Additionally, since reading a directory typically requires an EOF confirmation, it will take a minimum of two of these several second round trips. It is important to notice that after we send a request which causes the server to queue up large amounts of data, we cannot "unrequest" it any longer even though we might need the bandwidth for something more urgent in interactive use.

A local file system's media access is not as limited. Requests are queried and can be answered out-of-order depending on how the multiple layers from the disk scheduler to the driver and device itself see best. While large bulk transfers will slow down smaller requests such as a directory read, they will not necessarily completely stall them.

There are two approaches to dealing with this in distributed file systems:

Request throttling: do not allow one thread to issue requests saturating the pipe for several seconds. This is notable especially when the virtual memory system does read-ahead requests for large amounts of data. Since the purpose of read-ahead is make sure data is already locally cached when an application demands it, disabling read-ahead would cause application request latency. Ideally, the amount of read-ahead should be based on latency, available bandwidth and the total number of outstanding requests in the file system driver. As the heuristics to optimize this get complex fast, a much more pragmatic approach was taken: a command line option to limit the number of read-ahead requests per node. While far from perfect, this takes care of massive bursts and mitigates the problem for the most part.

Two separate channels: one for bulk data and one for metadata. Even though both connections share the same bandwidth, they will operate in parallel, and bulk transfers will not completely stall other requests. However, opening two TCP connections brings additional complications. First, we must authenticate twice. Second, all operations which create state must be duplicated for both channels, e.g. we must *walk* the file hierarchy for both connections with 9P. While in theory this option will provide better benefit, due to these complexities, it was not implemented – it is better to wait for the adaption SCTP [17] to solve the difficulties of multistreaming for us.

3.2. Distributed vs. Local File Systems

Some virtual file system operations are biased to the file system driver having direct access to the storage medium. This is not an issue for local file systems and also for distributed file systems specifically designed to inter-operate well with the virtual file system layer (e.g. NFS). However, all file system protocols (e.g. sftp) do not support the necessary functionality and must resort to alternative methods.

[1] NFS is transport-independent and has support for e.g. UDP transport, but as that is not applicable for remote sites, it is not discussed here.

This section discusses differences between distributed and local file systems and points out what is important to keep in mind when implementing a distributed file system driver in userspace. It also includes tips on increasing the performance of distributed file systems.

Permissions

Access control is not done in operations themselves, but rather using the *access* method. This presents problems for distributed file systems in several places: the typical I/O system calls (read, etc.) are not expected to return `EACCES`.

Some file system protocols do not present an opportunity to make access checks without making calls themselves. For example, with sftp we have no definitive idea in the file system driver of how our credentials map at the other end and therefore cannot do access checks purely by looking at the permission bits. The options are either to ignore the proper *access* method all together or execute shadow operations to check for access.

Luckily, in most of the cases applications deal well with returning `EACCES` from an I/O call, especially *read*/*write*. However, *readdir* is an exception and without implementing the *access* method properly, applications will only see an empty directory without any error message even if *readdir* returns an error. This is because `readdir`() is implemented in the system library and ignores permission errors from the `getdents`() system call. However, most file system protocols allow and require a directory to be opened for reading before fetching the contents. If the file system driver returns a permission error already when opening the directory for reading, the error is displayed properly in userspace.

Lookup

Lookup is the operation by which a file system converts a pathname component into an in-memory data structure to be used in future references to that file. This means that the file system should create an internal node for the file if found. In addition to a structural pointer, *puffs* requires three other pieces of information on the file:

- file type (regular file, directory, ...)
- file size (if a regular file)
- device number (if a device)

Typically the best strategy for implementing *lookup* in distributed file systems is doing *readdir* for the directory the *lookup* is done from and scanning the results locally.

Permissions also present an extra step for *lookup*. Lookup should return success for an entry which is inside an unreadable directory. To circumvent this, *lookup* can first attempt to read the directory, and if that fails, issue the equivalent of the protocol's *getattr* operation to check if the node exists.

It is possible to implement the *lookup* operation directly as a *getattr* operation, but it must be kept in mind that this will introduce an *n*latency* network penalty for looking up all the components in a directory, while doing a directory entry read once, caching the results and just scanning the locally cached copy introduces a much smaller cost.

While some file system protocols provide attributes for the files directly in the *readdir* return response, others might require extra effort such as real *getattr* operation. Next we discuss some optimizations possible in those cases.

The Unix long ls listing, `ls -l` is a fairly typical operation, which lists directory contents along with their attributes. Unless done right, this operation will also reduce performance down to *n*latency* because of waits for the *getattr* operations to complete and essentially doing nothing meanwhile.

While both 9P and sftp already supply attribute information as part of the *readdir* operation, an experimental version of psshfs was done to simulate a situation where it does not. This involved opportunistically firing off *getattr* queries for each of the directory entries found already during *readdir* and using cached values when *getattr* was issued to the file system driver. Two issues affecting performance were discovered and are listed here as potential pitfalls.

1. *readdir* operations generally require at least two round-trips for any file system protocol: one to deliver the results and a second one to deliver EOF. If *getattr* queries are queued or sent before parts 2-n of the *readdir* operation, the *getattr* requests are processed before the *readdir* operation completes. The file system driver will be waiting for the file server to process a lot of *getattr* operations to which the results are not needed yet. Therefore, the *getattr* operations should be fired off only after the *readdir* operation is completely done.
2. The attributes of the first file in the directory are requested from the file system driver after

readdir finishes; almost always before the results for the opportunistic *getattr* arrives from the file system server. If the file system driver discovers there is no cached result waiting and just fires off another query without checking if there is an outstanding request that should be waited for, all of the *getattr* requests targeted at later directory entries will be processed before the one we are currently after. Therefore, if an outstanding request is already active, it should be waited for instead of firing a new one.

Inactive

The *inactive* method for a file system node is called every time the kernel releases its last reference to a node. The purpose of *inactive* is to inform the file system that the node is no longer referenced by anything in the kernel and the file system may now free resources associated with the node. As, for example, executing the common command `ls -l` will issue an *inactive* for most of the files in the directory (all the ones without other references), *inactive* is an extremely common operation. However, typically a file system requires a call to *inactive* only in special cases, such as when a file is removed from the file system. Calling the *inactive* method in the kernel just costs a pointer indirection through the VFS layer and a function call, so it is cheap. When calling a userspace method the cost is much higher and should be avoided if possible.

The currently implemented solution to the cost problem is giving a file system the option for *inactive* to be called only when specifically requested. This is done with a *setback* operation. When the file system driver discovers the operation it is currently performing requires *inactive* to be called eventually, it issues an inactive *setback*. The means that a flag is piggy-backed on the request response to the kernel and set for the node structure in the kernel. In addition to incurring next to zero cost, the *setback* also solves problems with locking the kernel node – deadlocks could occur if we simply added a kernel call to flag this condition as we would be making it from the context of the file system driver. In case the *inactive* flag is not set for a node when the *inactive* kernel method is called, the request is simply short-circuited within the kernel and not transported to the userspace file system driver. For example, the *open* method may request inactive to be called for reasons explained in the next section.

Open Files and Stateful File Handles

Local file systems operate on local mass media and access file contents by directly accessing the media. Actual *read* and *write* operations, including their memory-mapped counterparts, do not perform access control. Access control is done earlier when a file descriptor is associated with the vnode. This means that as long as the file descriptor is kept open, the file can be accessed even though its permissions might change[2]. Local file systems do not open any file system level handles, as they can access the local disk at any time a request from above mandates they do so. The same applies to stateless versions of the NFS protocol.

However, most distributed file systems behave differently. For example, 9P and sftp require an explicit protocol level file handle for reading and writing files. These file handles must be opened and closed at the right times for the file system to operate correctly. For instance, assume that our file system driver opens a file with read/write access. Now our local system is guaranteed to be able to write to the file. Even if some other client accessing the file system changes the permissions of the file to read-only, our local system is still able to write to the file because of the open file handle.

Two different file handles are required for each file: one for reading and one for writing. If a file is opened read/write, it is possible to open only one handle. However, individual read and write handles must be opened separately, as the file's permissions might not allow both.

Opening handles for reading and writing is done when file opening is signaled to the file system by the *open* operation. It should be noted that this operation can be called when the node is already open. The file server should prefer to open only one handle if possible. It is possible to open a node only once for all users due to the credentials of the file handle being irrelevant; recall, access should be checked earlier. As some file servers and file system protocols might limit the amount of open file handles, the file handles should be closed once there are no users for the file on the local system.

On any modern operating system, file contents are accessed in two ways: either with

[2] The BSD kernel provides a routine called *revoke*(), which can be used to revoke open file handles. However, it is not typically used for regular files.

explicit read/write operations or through the virtual memory subsystem using memory mapped I/O. Regular I/O requires an open file descriptor associated with the file. However, memory mapped I/O can be performed without the file descriptor being open, as long as the mapping itself was done with the descriptor open. Even if the file is closed, it is still attached to the virtual memory subsystem in the kernel and therefore has a reference. Assuming there are no other references, once the virtual memory subsystem releases the file (due to `munmap`() or similar), *inactive* will be called. Therefore *inactive* is the right place to close file handles instead of the *close* method.

In the course of this work closing file handles in *close* was attempted. It included keeping count of how many times a file was opened in read-mode and how many times in write-mode. Also, the *mmap* method was changed to provide information about what type of mapping, read/write/execute, was being done so that the file system driver could keep track of it. However, the rules for determining if it was legal to close a file handle in *close* proved to be very convoluted. For instance, a file might have been closed less or more times than it was opened depending on the special circumstances. Also, as already noted, the only way a file system is notified of the virtual memory subsystem no longer using a file is *inactive*. The conclusion was to avoid *close* and prefer *inactive* unless there is a pressing reason to attempt to do otherwise.

Caching

Local file systems have exclusive access to the data on the file server. This means that every change goes through the file system driver. The same does not hold for distributed file systems and the contents can change on the file server through other file system drivers as well. This presents challenges in keeping the cache coherent, i.e. how to make sure we see the same contents as all other parties accessing the file server.

Currently, the *puffs* kernel virtual file system caches file contents (page cache) and name-to-vnode lookup information (name cache). The userspace file system driver, if it chooses to, caches the rest, such directory contents and file attributes. While caching in the userspace file system driver is less efficient, in practice the difference is minimal: compare the cost of network access to a peer with the cost of a local query to userspace. The important point is that caching in userspace allows a policy decision in the file system driver. Based on knowledge of the file system protocol, this can be made correct.

Neither sftp nor 9P support leases and therefore it is not possible to implement fully coherent caching. However, it is possible to add one based on timestamps and timeouts. Every time a file's attributes or a directory's contents are requested by the virtual file system, the current timestamp is compared against a stored one[3]. If the difference is smaller than the timeout value, the cached data is returned. Otherwise the file server is consulted and if a mismatch is found, the kernel virtual file system is requested to invalidate its cache: page cache for regular files and name cache for directories.

In addition to being able to specify a timeout value in seconds, it is also possible to make the cache always valid and never valid. It is important to note that an always invalid cache is not the same thing as no kernel caching at all. The system's ability to do memory mapped I/O and therefore execute files is based on the page cache. If we completely disable caching for a file system (mount with the *puffs* option `nocache`), we are no longer able to execute files off of it. By always invalidating the cache at our checkpoints, the cache is still frequently invalidated but MMIO is functional. However, for typical use a timeout of a few seconds is better even if data is frequently modified from under the file system driver. Finally, a user-assisted method of invalidating caches is provided: sending `SIGHUP` to the server invalidates all server and kernel caches.

As was mentioned above, the kernel caches file contents in the page cache. The page cache works in two ways: first, file content can be satisfied from the cache when read, and second, writes can be coalesced in memory and written to stable storage later to avoid lots of small I/O requests. The second case sometimes poses a problem for distributed file systems. For example, if copying a file to the mail server from where it is to be sent as an attachment, one expects it to be fully transferred after `cp` finishes. Instead the file might still reside completely in the local page cache. To avoid these kinds of situations, the *write through* cache mode for the puffs virtual file system is used: all writes are flushed immediately after they are done. Notably though, this does not cause modifications via memory mapped I/O to be flushed immediately. This can be solved by

[3] NFS checks timestamps also during file read.

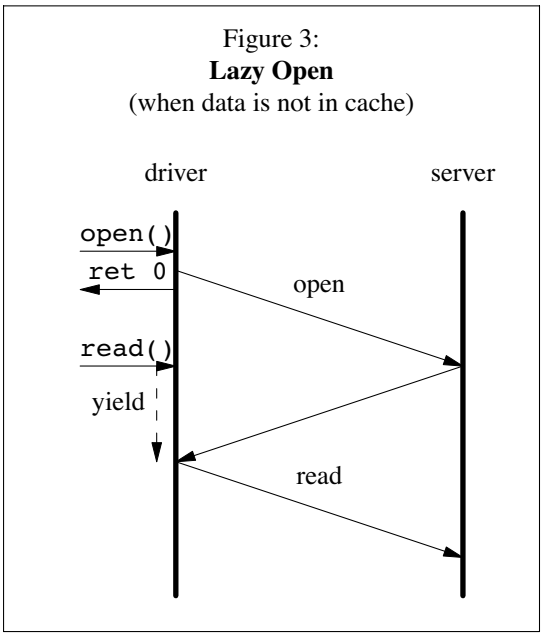


Figure 3:
Lazy Open
(when data is not in cache)

periodically issuing a flush request from the file system driver, but in practice there have not been any problems, so this has not been implemented.

Lazy Open

A typical operation sequence to read a file is *lookup*, *open*, *read*, *close*. The results for lookup and read can be cached at least to some degree as they are idempotent and we can make (user-assisted) assumptions about the stability of the data on the server. Open and close are different, as they change state on the file server and therefore cannot be cached. If the file content is cached locally, waiting for the opened file handle from the server is unnecessary, as it will not be used for serving data from the local cache. This can be a problem especially over slow links with hundreds of kilobytes of outstanding requests – it will take several seconds for the response from the server to be received.

This can be solved by lazily waiting for the file handle. The driver's open method sends a request to open a file handle, but returns immediately. Only if a read or write is actually issued, the file handle is waited for. This way data can be immediately served from the local cache if it is available. When implementing this scheme, care must be taken to handle open and close properly. The file might be closed (and reopened) before the original open request from the server returns, so state must be maintained to decide if a response to an open request should prompt closing the handle immediately.

Unix Open File Removal

An example user of *inactive* in the kernel is the Unix file removal semantics, which state that even after all links to a file are removed from the file system directory namespace, the file will continue to be valid as long as there are open references to it. A removed file will actually be removed only when *inactive* is called.

NFS client implementations on Unix systems feature the *silly rename* scheme, whereupon if a file is removed from a client host while it is still in use, the NFS client renames the file to a temporary name instead of deleting it. For example, 4.4BSD uses the name *.nfsAxxxx4.4* [16]. When the open file is finally closed, the *inactive* routine is called and the renamed file is removed. This scheme is due to the statelessness of the NFS protocol and has four problems.

1. If the client crashes between rename and the call to *inactive*, the renamed file is left dangling [1].
2. The file is still accessible through the file system namespace, although by a different name.
3. If another client removes the file, this scheme does not work.
4. Empty directories with silly renamed files are unremovable until the files have been closed.

A file handle's usefulness in dealing with the Unix open file semantics depends on file system protocol. In NFS, file handles are stateless; they are not explicitly opened and closed making it clear they cannot support this kind of behavior. The ssh sftp protocol uses file handles which are opened or closed, but the protocol specification [10] says that stateless or stateful operation is up to the server implementation. However, upon examination at least the OpenSSH sftpd supports the semantics we desire. The 9P protocol specification [11] leaves it open to the implementation and states that Plan 9 itself will not allow to access a removed file while implementations such as the Unix server *u9fs* will allow it.

While a file system driver should transparently support the semantics local to the system it runs on, such effort has not yet been made with *puffs* and the distributed file system drivers described in this document. They will work correctly with some servers and fail with others. As distributed and local semantics can never truly fully match, we do not consider this a big problem. If it is considered a problem, a *silly rename* scheme can be implemented.

4. Framework

Next, an abstract framework for implementing distributed file systems [18] is presented. The following properties of the framework are discussed:

- A buffering scheme for allocating memory for protocol data units (PDUs) and matching incoming buffers as responses to sent requests.
- Routines for cooperating multitasking, which handle scheduling automatically for file systems using the framework.
- An I/O descriptor subsystem, which allows to supply the framework with file descriptors used for data transfers.
- An event loop which reads incoming data from the I/O descriptors and the kernel, dispatches requests and writes outgoing data.

To use the framework, the file system driver must register callbacks which handle the driver semantics. An overview is presented here and each callback is later discussed in more detail.

`readframe`
: Read a complete frame from the network into the buffer provided by the framework.

`writeframe`
: Write a complete frame. A buffer given by the framework is used as the source for data.

`framecmp`
: Compare two frames to see if the one is the response to another.

`gotframe`
: Called for incoming frames which are determined to not be responses to outstanding requests.

`fdnotify`
: Notify the file system driver of changes the framework detected in I/O descriptor state.

4.1. Buffering

Sending and receiving traffic over the network requires buffers which host the contents of the protocol data units (PDUs). While the contents of a PDU are specific to the file system, the necessity of allocating and freeing memory for this purpose is generic.

For the purpose of memory management, *puffs* provides routines to store data in automatically resizing buffer: the puffs framebuffers [18], *puffs_framebuf*. In addition to automatic memory allocation, the buffering routines provide a read/write cursor, seeking ability, maximum written data offset and remaining size. When writing to a buffer it is possible to write as much data as there is available memory, but reading from the buffer will fail for locations beyond the maximum written data offset.

Additionally, the buffer supports opening a direct memory window to it. This is useful especially when reading or writing the buffer to or from the I/O file descriptor, because it avoids having to copy the data to a temporary buffer. As the framework does not know if data is being read or written in the window, the maximum size is also increased to the maximum mapped offset. Therefore, readers of the buffer should only map the buffer size's worth.

For processing the buffer contents, file systems typically want to add another layer which understands the contents of the buffer. For example, `fs_buf_write4`() would write 4 bytes of data into the buffer using *puffs_framebuf* routines after adjusting the byte order if necessary. Similarly, `fs_buf_readstr`() would read a string from the buffer using the protocol to determine the length of a string at the current cursor position. For example, for a protocol with "Pascal style" strings, the routine would first read n bytes to determine the string length and after that read the actual string data.

4.2. Multitasking

As mentioned in the *puffs* introduction earlier in Chapter 2, *puffs* implements its own multitasking mechanism without relying on platform thread scheduling. This means that in addition to not requiring any data structure synchronization calls in the file system driver[4], resource sharing can be better implemented and taken into account by the framework.

Commonly, threaded programs rely on implicit scheduling and contain local state in the stack. If a threaded program executes a blocking call, another thread is scheduled by the thread scheduler. The blocked thread is released when the blocking call completes. However, for distributed file systems the resource upon which blocking calls are made is the shared network connection, and therefore pure implicit state management will not do: received data must be mapped to the caller and additional state management is required. The *puffs framebuf* framework takes care of this state management and automatically

[4] Unless the driver chooses to create threads on its own, of course.

schedules execution where required, therefore making the task of file system implementation easier.

The points to suspend execution of a request are when a request is queued for network transmission. The framework automatically resumes the suspended request when the response has been read from the network. This functionality is discussed more later in the chapter "I/O Interface".

4.3. I/O File Descriptor Management

By default the framework is interested in the file descriptor which communicates *puffs* operations between the kernel and userspace. If the file system driver wishes the framework to listen to other descriptors, it must register descriptors using the `puffs_framev_addfd`() call. This can happen either when the file system driver is started or at any point during runtime. The prior is a likely scenario for client-server file systems after having contacted the file system server, while the latter applies with peer-to-peer file systems as new peers are discovered. Conversely, descriptors can be removed at any point during execution. This releases buffers associated with them, incoming and outgoing, and returns an error to blocked operations allowing them to run to completion.

I/O file descriptors have two modes: enabled and disabled. A disabled file descriptor will not produce any read or write events and therefore the callbacks will not get executed. The difference between disabling a descriptor and removing it is that disabling leaves the buffers associated with the I/O descriptor, incoming and outgoing, intact. This is useful for example in cases where the protocol has a separate data channel and the file system driver wishes to read data from it only when a VFS read request has been issued (see Chapter 4.6).

In addition to descriptor removal by the file system driver, the the framework must deal with abruptly closed connections. This means that it must provide the file system driver a notification when it detects an error condition with a descriptor. The `fdnotify`() callback is used for this purpose. As it is legal to half-close a file descriptor and still use the other side [19], the framework must track and notify the file system driver separately of the closing of either side. Similarly to file system driver initiated descriptor removal, the framework automatically releases all blocked waits and flags them with an error also in this case.

Once a descriptor is closed, certain conditions are imposed by the framework. It is not possible to write to a descriptor with the write side closed and attempting to do so immediately returns an error. However, if only the read side is closed, it is still possible to write to a file descriptor but waiting for the result is not allowed. The file system driver can further decide if this is a sensible condition in the `fdnotify`() callback. It also has the option of just giving up completely on a file descriptor when it receives the notification of either direction closing.

4.4. I/O Interface

Each file descriptor has its own send queue. A PDU can be queued for sending using four different routines:

- *enqueue_cc*: yield the current execution context until a response is received after which continue execution.
- *enqueue_cb*: do not yield. instead, a callback function, a pointer to which is given as a parameter, will be called from the eventloop context when the response is received.
- *enqueue_justsend*: just enqueue and do not yield. A parameter controls whether or not a response is expected. This is required to differentiate between a response and a request from the file server. However, the contents of the possible response are discarded.
- *enqueue_directsend*: yield until the buffer has been sent. Does not assume a response.

As the framework is completely protocol agnostic, it delegates the job of reading and writing frames to and from the descriptor to the file system driver via the `readframe`() and `writeframe`() callbacks. Readframe is called for incoming data while writeframe is used to transmit buffers in the send queues. As these routines are in the file system driver and can examine the buffer contents, they also know when a complete PDU was received or written. They signal this information back to the framework.

Of the above enqueueing routines, the first three require the ability to match an incoming response to a request sent earlier. The `framecmp`() callback provided by the file system driver is used for this. Once a complete frame has been read from the network, all the outstanding requests for the descriptor the frame was read

from are iterated over. As requests typically arrive in-order and even for a very busy file system the maximum number of outstanding requests is typically tens, the linear scan is cheap. Once the original request for the newly arrived response is located, execution is resumed.

If no matching request for the frame is found, the `gotframe`() callback is called. If the callback does not exist, the frame is dropped.

In case the comparison routine can determine from the incoming frame under examination that it is not a response at all, it can set a flag to short-circuit the iteration. This avoids going through all outstanding requests in cases where it is evident that the incoming frame is a request from the server and not a response to any of the file system driver's requests.

4.5. Event loop

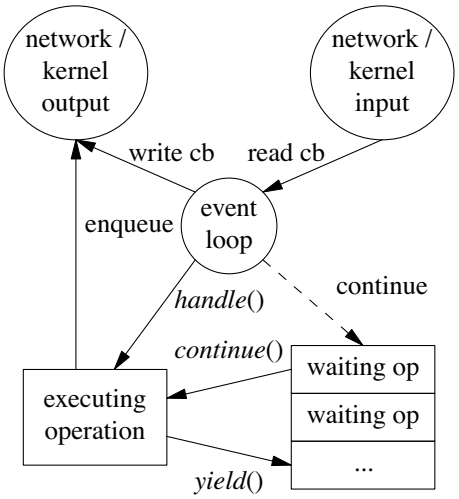
Finally, the event loop is discussed. It is the driving force behind file system driver operation and dispatches handlers for requests and responses as they come in.

The event loop, `puffs_mainloop`(), provided by libpuffs is a generalized version of the event loop first used directly in the psshfs file system [8]. Initially psshfs and 9puffs had their own event loops due to the standard libpuffs event loop lacking the features to support a distributed file system. However, as the framework was created the event loop was enhanced so that distributed file systems can use it. This new version is presented as a diagram in *Figure 4* and as pseudocode in *Figure 5*.

Each file system driver can specify a "loop function". This is a simple callback which is called once every loop. Single-threaded file servers can use it for tasks which need to be executed periodically. If the loop function needs periodical execution, maximum blocking time for the async I/O multiplexor in the event loop can be set using `puffs_ml_settimeout`(). While not realtime quality, this timeout is fairly accurate for correctly implemented file system drivers until very high loads.

For enabled descriptors, read polling is always active. Write polling is enabled only selectively, as otherwise the write event would always trigger. Its enable and disable in the event loop depend on the previous status and if there is data in the queue. Also, since the common case is that all enqueued data can be written immediately, the event loop attempts to write enqueued data

Figure 4:
Event Loop



before enabling write polling for a certain descriptor. Only if all data cannot be written, write polling is enabled.

The I/O multiplexor `kevent`() system call uses the kqueue [20] event notification mechanism. It operates similarly to `poll`() and `select`(), but is a stateful interface which does not require the full set of descriptors under surveillance to be communicated to the kernel in each call. However, changes can be made simultaneously to event query, and the event loop uses this to change the status of write polling when necessary.

4.6. Other Uses: The Portal File System

Distributed file systems are not the only application for the *puffs* buffering and event framework. Another example for the use of such a framework was found in the reimplementation of the portal file system [21] using *puffs*.

The portal file system is a 4.4BSD file system which provides some support for userspace file systems. It does not, strictly speaking, implement a file system, but relies on a provider to open a file descriptor, which is then passed to the calling process. What happens is that a process opening the file `/p/a/file` will receive a file descriptor as the result of the open operation and is in most cases not able to distinguish between an actual file system backing the file descriptor. A configuration file specifies which provider the portal daemon executes for which path.

Figure 5:
Event Loop Pseudocode

```
while (mounted) {
   fs->loopfunc();

   foreach (fd_set) {
      if (has_output)
         write();
   }

   foreach (fd_need_writechange) {
      if (needs_write && !in_set)
         add_writeset();
      if (!needs_write && in_set)
         rm_writeset();
   }

   kevent();

   foreach (kevent_result) {
      if (read)
         input();
      if (write)
         output();
   }
}
```

To facilitate this type of action, the original portal file system passes a pathname to the userspace portal daemon as part of the open method. Upon receiving a request, the daemon *fork*()s, lets the child take care of servicing the request, and listens to more input from the kernel. After the child has opened the file descriptor, it communicates the result back to the kernel. The kernel then transfers this descriptor to the calling process. The child process exits, but depending on the type of provider it might have spawned some handlers e.g. using *popen*(). Other types, such as TCP sockets, require no backing process.

The *puffs* portal file system driver behaves toward applications exactly like the old portal file system and even reuses most of the code of the original portald userspace implementation. However, *puffs* portalfs operates like a real file system in the sense that the file system driver interprets all the requests instead of a file descriptor being passed to a caller.

The problem in using original portald code is that the portal providers can execute arbitrary blocking sequences, and allowing one to execute in the context of the file server blocks the access to other files. This can be avoided by either multiple processes or multiple threads. The original portald code we use relies on processes for cleanup in some cases, such as cleaning up after *popen*(), so processes were chosen.

Operation of the new portal file system driver is as follows. When the file system *open* method is called, the file system driver opens a socketpair and forks off a child process. The driver then yields after enabling the child socket descriptor as a valid I/O descriptor. Meanwhile, the child proceeds to open a file descriptor and sends it to file server using descriptor passing[5]. After receiving the descriptor the file system driver returns success to the process calling *open*.

Read and write calls require asynchronous I/O for the file system driver to support concurrent access properly. As opposed to psshfs and 9puffs, the descriptors produced by the portal daemon are enabled for reading only when an incoming read request arrives from the kernel. This way data is consumed from the descriptor only when there is a read request active.

The read request uses the framework's *directreceive* routine for receiving data in which the file system driver supplies the buffer to receive data to without having to get it via `gotframe`(). By threading the number of bytes the kernel wishes to read from the file to the *readframe* routine using the buffer, the driver can also avoid reading too much data. Reading too much data would result in having to store it for the next read call.

The reimplementation performs better in some cases. Since *puffs* calls are interruptible, the calling processes can interrupt operations. The original portalfs implementation had a problem that if the open call on portalfs blocked, the calling process could not be interrupted. An example is an unreachable but not rejected network connection, which will stall until the *connect*() system call of the portald provider child times out. As a downside for this implementation, calls now need to traverse the user-kernel boundary three times instead of operating directly on the file descriptor in the calling process. However, as portalfs is rarely, if ever, used in speed-critical scenarios, this does not constitute a problem.

[5] Another option would be to issue a version of *fork*() which shares the descriptor table between the parent and the child, but some form of wakeup from the child to the parent is required in any case.

4.7. Other Uses: Kernel VFS Communication

If we return to Figures 1 and 2, we notice that the situation between the file server and and kernel virtual file system is symmetric: both are used by the file system driver through a communication protocol. After writing the framework, kernel communication could be adopted to use it instead of requiring special-purpose code. All incoming requests from the kernel are treated as `gotframe` and are dispatched by the library to the correct driver method. Using the framework for kernel communication also enables forwarding the *puffs* protocol to remote sites just by adding logic to route PDUs.

Not all communication in the file system is originated by the kernel. For example, the file system driver can request the kernel to flush or invalidate its caches. In this case a request is send to the kernel. As the response to the request is not immediate, it must be waited for. By treating the kernel virtual file system just as another file server, the framework readily handles yielding the caller, processing other file system I/O meanwhile, and rescheduling the caller back when the response arrives. This scheme also works independent of if the kernel virtual file system is on the local machine or a remote site.

5. Comparisons

This section presents three comparative studies for the framework. The first one compares implementation code size before and after the framework was introduced. The second measures performance between userspace file system drivers and the in-kernel NFS. Finally, a feature and usage comparison between psshfs and NFS is presented.

5.1. Code Size Comparison

Originally, both psshfs and 9puffs were implemented with their own specific buffering routines and event loop. These routines were first written when developing psshfs and were adapted to 9puffs with some changes.

Figure 6 (NFS included for comparison) shows that two thirds of the code used for networking and buffering could be removed with the introduction of the framework; what was left is the portion dealing with the file system protocol. As a purely non-measurable observation, the code abstracted into the framework is the most difficult and error-prone code in the file system driver.

Figure 6:
Code Size Comparison

total lines of code

	before	after	save (%)
psshfs	2885	2503	11%
9puffs	2601	2140	18%
NFS	24286	n/a	n/a

code involved in events / networking / buffering

	before	after	save (%)
psshfs	355	119	66%
9puffs	411	150	64%

Building packets is done by linear construction code while parsing is the reverse operation. On the other hand, the network scheduling code and event loop depends on timings and the order in which events happen. Moreover, the data structures required to hook this into the *puffs* multitasking framework are not obvious. With the networking framework the file system driver author does not need to worry about these details and can concentrate on the essential part: how to do protocol translation to make the kernel virtual file system protocol and the file server talk to each other.

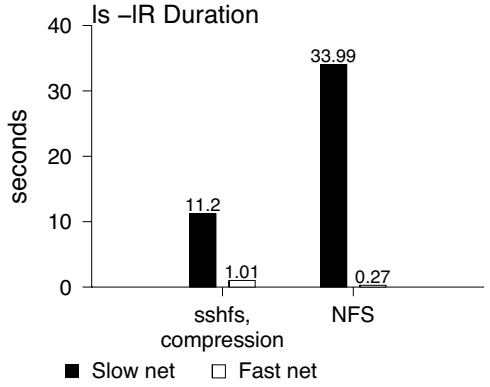
5.2. Directory Traversal

In this section we explore the performance of a file system driver and issues with the file system protocol when executing the commonplace Unix long listing: `ls -l`.

The command `ls -l` is characterized by three different VFS operations. First, the directory is read using *readdir*. Second, the node for each directory entry is located using the *lookup* operation. Finally, the node attributes for the `ls` long listing are fetched using the *getattr* operation. These operations map a bit differently depending on the file system protocol. For example, on NFSv3 the *readdir* operation causes an *NFS_READDIR* RPC to be issued. For each *lookup*, *NFS_LOOKUP* procedures are issued. Since the *NFS_LOOKUP* operation response also contains the node attributes, they are cached in the file system when *getattr* is called and no network I/O will be required for satisfying the request.

As discussed already in *Chapter 3.2*, the bottleneck in the above is the serial nature of the process: one operation must complete before the

Figure 7:



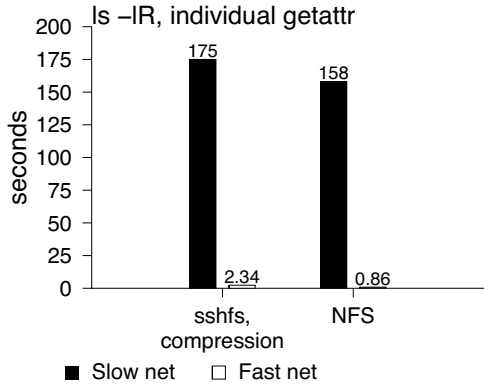
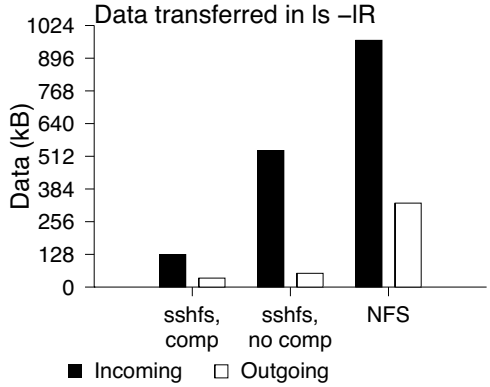
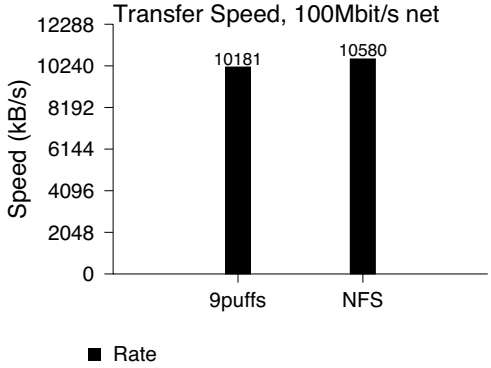


Figure 8:





next is issued. NFS solved this by introducing the *NFS_READDIRPLUS* procedure in protocol version 3. It returns the attributes for all the nodes in the *readdir* response. That way the file system driver will already have the information for *lookup*/*getattr* cached when it is requested. However, this information might well be wasted, since *readdir* is acting only opportunistically. Even further, as the BSD NFS implementation creates a new vnode for each previously non-existing one (to cache the attribute information in), listing a directory might prompt valid vnodes to be recycled. This is why the use of *NFS_READDIRPLUS* is disabled by default and recommended only for high latency mounts.

The sftp (and 9P) protocol always includes the attribute information in *readdir* responses. Our implementation differs from the kernel version in such a fashion that we cache the attributes and the directory read results in the directory structure; not new nodes. Therefore this solution does not force kernel vnodes to be recycled – although it could not do so even if it wanted to, as vnode life cycles are completely controlled by the kernel in *puffs*.

The `ls -lR` measurements are for the time it takes to traverse a directory hierarchy with around 4000 files. The initial measurements were done over a 11Mbps wireless link with a 3ms RTT. These results results showed the psshfs was faster than NFS – a result quite unexpected. The cause was discovered to be the bandwidth use, so another measurement was done with the ssh compression option being on (default compression level). This improved results even further. Technically it is possible to compress the NFS traffic also using the IP Payload Compression Protocol (IPComp), but doing so is multiple times more difficult than using the ssh compression option and the effects of doing so were not investigated.

The measurements presented in *Figure 7* contain both the duration for the operations on a high-latency, low bandwidth link and a low-latency high bandwidth local area network.

Performance is measured both for coalesced *getattr*s and individual ones. NFS is mounted using a TCP mount. Apart from compression, psshfs is used with OpenSSH default options.

For the preloaded attributes case, it is easy to see that psshfs wins on the slower network because it requires much less data to be transferred. However, on the high speed network the performance penalty inherent in multiple context switches per operation is evident. Even though latency is canceled for the link by using attribute preloading, the psshfs file system server must still *getattr* each file using individual system calls when the NFS server can simply perform these operations inside the kernel without context switch penalty. Additionally, psshfs must encrypt and decrypt the data. Finally, we do not control the sftp server and cannot optimize it.

Without preloading attributes ("individual getattr") NFS dominates because the operation becomes driven by latency, and NFS as a kernel file system has a smaller latency.

5.3. Data transfer

To measure raw data transfer speeds, large files were read sequentially over a local area network using both NFS and 9puffs[6]. The results are hardly surprising, as reading large files uses read-ahead heuristics. Data requested by the application has already been read into the page cache by the read-ahead code and can be delivered to the application instantly without consulting the file system server. It should be noted, though, that the userspace model uses more CPU and on fast networks such as 10GigE, the performance of the userspace model may be CPU-bound.

5.4. psshfs vs. NFS

As NFS and psshfs are roughly equivalent in performance, it is valid to question which one should be preferred in use. The following section lists reasons NOT to use the protocol in question:

psshfs:

- No support for hard links. Two hard-linked directory entries will be treated as two files.
- No support for devices, sockets or fifos.
- No support for user credentials in the protocol: one mount is always one set of credentials at the server end.
- No support for an async I/O model: there is no certainty if written data is committed to disk.

NFS:

- Setup is usually a heavyweight operation meaning the protocol cannot be used without considerable admin effort.
- It is difficult, although entirely possible, to make the protocol operate from a remote location through IPSec tunnels.
- There is no real security model in the currently dominant NFSv3 version.

[6] psshfs was attempted first, but the CPU requirements for the encryption capped out the CPU of the server machine.

6. Conclusions and Future Work

This paper explored implementing distributed file system drivers in userspace on top of the *puffs* Pass-to-Userspace Framework File System. It explained concepts relevant to implementing distributed file systems and pointed out unexpected pitfalls.

A framework for implementing distributed file systems was presented. The I/O file descriptors, callbacks, continuations and memory buffers were discussed and interfacing with them from the file system driver was explained.

The performance characteristics of userspace file system drivers and userspace file systems was lightly measured and analyzed. The conclusion was that even though in-kernel file systems usually perform better, userspace file systems can shrink the gap by the possibilities in a more flexible programming environment.

Future work includes implementing a peer-to-peer file system on top of the framework. Notably though, the portal file system implementation briefly mentioned in this paper already shares some similar characteristics to peer-to-peer file systems in that it communicates using multiple I/O descriptors concurrently.

As distributed file systems have a high price to pay for reloading information from the server, the information should be cached as much as possible. File data is cached effectively in the kernel page cache, although a file system driver wanting a persistent cache will have to implement it by itself. However, metadata caching is currently completely up to the file system driver. This could be improved in the future by providing a method for caching metadata.

Related to metadata caching is the observation that the optimal way to perform directory reading and *lookup* is a similar procedure in both

of the distributed file systems we went over in this paper. The procedure should be generalized for any file system driver. This includes attribute caching for directory entries along with optionally preloading the attributes even though the file system protocol does not directly support it.

Availability

All of the code discussed in this paper is available for download and use in the development branch of the NetBSD [7] operating system. This development branch will eventually become the NetBSD 5.0 release.

For information on how to download the code in source form or as a binary release, please see *http://www.NetBSD.org/*. Documentation for enabling and using the code is available at *http://www.NetBSD.org/docs/puffs/*

Acknowledgments

This work was funded by the Finnish Cultural Foundation and the Research Foundation of Helsinki University of Technology. Karl Jenkinson provided helpful comments.

References

1. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, *Design and Implementation of the Sun Network Filesystem,* pp. 119-130, Summer 1985 USENIX Conference (1985).
2. Christopher Hertel, *Implementing CIFS: The Common Internet File System,* Prentice Hall (2003). ISBN: 978-0-13-047116-1.
3. Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen, *Ivy: A Read/Write Peer-to-peer File System,* Fifth Symposium on Operating Systems Design and Implementation (December 2002).
4. J-M. Busca, F. Picconi, and P. Sens, *Pastis: A Highly-Scalable Multi-User Peer-to-Peer File System,* Euro-Par 2005 (2005).
5. Michael Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, *Mach: A New Kernel Foundation for UNIX Development,* pp. 93-113, Summer USENIX Conference (1986).
6. Miklos Szeredi, *Filesystem in USErspace, http://fuse.sourceforge.net/* (referenced January 2008).
7. The NetBSD Project, *The NetBSD Operating System*. http://www.NetBSD.org/.
8. Antti Kantee, *puffs - Pass-to-Userspace Framework File System,* pp. 29-42, AsiaBSDCon 2007 (March 2007).
9. Yousef A. Khalidi, Vlada Matena, and Ken Shirriff, "Solaris MC File System Framework," TR-96-57, Sun Microsystems Laboratories (1996).
10. T. Ylönen and S. Lehtinen, *SSH File Transfer Protocol draft 02,* Internet-Draft (October 2001).
11. Bell Labs, "Plan 9 File Protocol, 9P," *Plan 9 Manual*.
12. *puffs -- Pass-to-Userspace Framework File System development interface* (January 2008). NetBSD Library Functions Manual.
13. S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX,* pp. 238-247, Summer Usenix Conference, Atlanta, GA (1986).
14. Sun Microsystems, Inc., *RPC: Remote Procedure Call Protocol Specification Version 2,* RFC 1057 (June 1988).
15. Rick Macklem, *The 4.4BSD NFS Implementation,* The 4.4BSD System Manager's Manual (1993).
16. Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley (1996).
17. Randall Stewart and Chris Metz, "SCTP: New Transport Protocol for TCP/IP," *IEEE Internet Computing,* Volume 5, Issue 6, pp. 64-69 (2001).
18. *puffs_framebuf -- buffering and event handling for networked file systems* (January 2008). NetBSD Library Functions Manual.
19. W. Richard Stevens, *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI,* Prentice Hall (1998). ISBN 0-13-490012-X.
20. Jonathan Lemon, *Kqueue: A Generic and Scalable Event Notification Facility,* pp. 141-154, USENIX 2001 Annual Technical Conference, FREENIX Track (June 2001).
21. W. Richard Stevens and Jan-Simon Pendry, *Portals in 4.4BSD,* USENIX Technical Conference (1995).

Logical Resource Isolation in the NetBSD Kernel

Kristaps Džonsons
Swedish Royal Institute of Technology, Centre for Parallel Computing
kristaps@kth.se

Abstract

Resource isolation is a strategy of multiplicity, the state of many isolated contexts within a parent context. Isolated resource contexts have functionally non-isomorphic resource mappings: contexts with equivalent domain identities map to non-intersecting ranges in the resource co-domain. Thus, in practise, if processes a and b return different values to an equivalent identity (say, for the hostname), then the callee context, for this identity, demonstrates resource non-isomorphism. Although isolation is by no means a new study in operating systems, the BSD family offers few implementations, at this time limited to FreeBSD's Jail and, potentially, the `kauth(9)` subsystem in NetBSD. These systems provide a framework with which one may construct isolated environments by cross-checking and switching over credentials at the kernel's boundary. In this document, we consider a radically different approach to resource isolation: instead of isolating at the kernel boundary, we consider a strategy of collecting entire kernel sub-systems into contexts, effecting bottom-up resource isolation. This document describes a *work-in-progress*, although a considerable implementation exists[1].

1 Introduction

Resource isolation may strictly be defined as a non-isomorphic mapping between unique resource identities (the domain) and mapped entities (co-domain): multiple contexts, with the same domain identity, mapping to non-conflicting range entities. Instead of a single, global task context, where all tasks have a common mapping, resource isolation implies a set of contexts, with set members defined by the commonality of their mapping function. Our use of the term resource, in this regard, refers to *mutable* entities.

In practise, resource isolation provides Unix processes ("tasks") a different view of their environment depending upon the context. For example, a particular process a, if isolated, may only view processes in its calling context A, while another process, b, can only see processes in its context B. The contexts A and B are non-overlapping; in other words, no process in A can see into B and vice-versa. Conventional Unix environments, contrarily, are non-isolated (or isolated to a single context).

In this document, we use set notation to describe tasks and resources. We define a mapping function f to accept a resource identity x and produce an entity y. The set of resources available to a task is $f(x_0) \dots f(x_n)$, or, equivalently, $y_0 \dots y_n$ mapped from $x_0 \dots x_n$. Resource isolation implies a set $F = \{f_0 \dots f_k\}$, where the ranges of any two f_i are non-overlapping. Thus, in a traditional Unix system with a single, global resource context, $F = \{f_0\}$. This single context f_0 has an equivalent range and co-domain.

We consider each f to be a black-box within the kernel, and F defines the kernel proper. In practise, this decomposes into each $f \in F$ having a unique identifying credential; two resource requests, for example, for the hostname, correspond to a single f_i in the case of a single-context environment, and f_i and f_j in a multiplicity environment. We consider "resources" in the broad sense as system calls or in the fine sense as system calls combined with particular arguments.

The complexity of our example isolation scenario

[1] See http://mult.bsd.lv

is considerable: there are many elements entangled in a process. In order to be isolated, the set of contexts must be injective; in other words, a co-domain entity may be mapped from only one domain identity. A process abstracts considerable complexity: a process is composed of memory maps, file descriptors, resource limits, and so on. To isolate one process from another, all non-injective conditions must be removed. For example, although process a may not be able to signal process b, it may try to affect it by changing resource limits, or manipulating the controlling user. If any of these resources conflict, then isolation is not maintained. We call these conditions resource conflicts.

A resource conflict may be effected both directly and indirectly. In the former case, a breach could occur if a were able to signal b using a non-standard signalling interface (through, e.g., emulated system calls that bypass the isolation mechanism). In the latter case, improperly isolated user checks could allow a to affect b by changing user resource limits. Since $f(x_j) = y_c$ and $f\prime(x_j) = y_c$, the system is no longer isolated.

In this document, we'll propose a system that implements resource isolation in order to provide efficient multiplicity without sacrificing elegance. Before discussing the methods of this system, we'll consider the reason for its implementation; in other words, why we chose to implement a new design instead of re-using existing systems. In order to properly discuss these systems, we'll introduce an informal taxonomy for multiplicity systems.

2 Terminology

In this section, we refine our introductory notation. Since multiplicity has many different forms, it's difficult to introduce a generalised notation that covers all scenarios. In the previous section, we used the term context to describe the mapping $f \in F$. We now introduce the term "operating instance".

Definition If x is a resource identity (e.g., system call), and y_0 is the entity mapped by f_0, with f_1 producing y_1 where $f \in F$ and $y \in Y$, then we define each $f \in F$ as an operating instance in a multiplicity system if and only if there are no two $f_i \in F$ returning the same y for a identity x.

In practise, multiple operating instances in a multiplicity system may only exist if there are no conflict points where $f_i(x) = f_j(x) = y$. In the introductory section, we used the term resource isolation to define this property; in this section, we introduce the notion of operating instances as those entities with isolated resources.

The existence of an operating instance doesn't necessarily imply instance multiplicity: in a standard Unix system, there always exists one operating instance. The property of multiplicity arises when the co-domain Y is completely partitioned into ranges of $f \in F$, where no $y \in Y$ conflict.

Definition A system may claim *operating instance multiplicity* when there are multiple non-overlapping ranges in the resource co-domain, mapped from different $f \in F$.

In this document, we consider only operating instance multiplicity. There are other targets of multiplicity, like operating system multiplicity, which will be addressed only in passing.

3 Scenario

There are many scenarios involving multiplicity: service consolidation, testing, containment, redundancy, and so forth. We choose three common multiplicity scenarios that, as we'll see, span the diapason of multiplicity strategies. It's important to stress that this analysis is of operating instance multiplicity: we disregard, for the moment, scenarios calling for whole operating system multiplicity. This isn't the focus of this paper, as many well-known operating system multiplicity systems exist for the BSD family.

We'll primarily focus on a cluster computing environment. In this environment, we must allow for many distinct process trees with isolated resource contexts, preferably rooted at `init(8)`, all of which are completely contained from one another. There must be negligible over-head in the isolation mechanism; more importantly, these systems must start

and stop extremely quickly, so as to offer fine-grained scheduling of virtual environments on the processor. There must be thousands of potential instances, all, possibly, running simultaneously on one processor. In an alternate scenario, some instances may have priority over others; some, even further, may be entirely suspended, then restarted later. This document focusses largely on the in-kernel isolation strategy for such a system.

We also consider a virtual hosting environment. Like in cluster computing, we account for the possibility of many instances competing for resources. Unlike in cluster computing, we consider that the run-time profile for these systems is roughly similar; thus, sharing of code-pages is essential. Further, while cluster computing stresses the speedy starting and stopping of instances, virtual hosting emphasis fine-grained control of resources which are likely to remain constantly operational to a greater or lesser extent.

In both of these scenarios, each process must have a conventional Unix environment at its disposal. We decompose the term "resource" into soft and hard resources: a *soft* resource is serviced by the kernel's top-half (processes, memory, resource limits, etc.), while a *hard* resource is serviced by a physical device, like a disc or network card. Our scenario doesn't explicitly call for isolation between hard resources (we consider this a possibility for future work); an administrator may decide which devices to expose by carefully constructing `dev` nodes.

4 Related Work

At this time, of the BSD operating system family, only FreeBSD Jail[3] offers a complete isolation mechanism. Jail attaches structures to user credentials that define guest contexts within a host context. Guests have a resource range that is a strict subset of the host; all guests have non-intersecting ranges while the host's range is equivalent to the co-domain (thus, is allowed to conflict with any guest). FreeBSD Jail structures isolate primarily in terms of soft resources: the only isolated hard resources are the network, console, and pseudo-terminal interfaces. The Jail system first appeared in FreeBSD 4.0.

NetBSD 4.0 includes `kauth(9)`[1], which orchestrates the `secmodel(9)` security framework. This system allows kernel *scopes* (resource identity categories) and their associated actions (resource requests) to be examined by a pool of *listeners*. Scopes and actions are correlated with calling credentials and relevant actions are produced. This doesn't provide isolation *per se*, but we can, assuming other changes to the infrastructure and an implementing kernel module, envision a sytem interfacing with `kauth(9)` to provide in-kernel isolation.

We specifically disregard NetBSD-Xen (and other full-system virtualisers, including the nascent DragonFlyBSD `vkernel(7)`) from our study, as the memory overhead of maintaining multiple guest images is considerable and violates our stipulation for many concurrent contexts. Both of these systems full into the category of operating system multiplicity systems: instead of resource isolation, these virtualise the hardware context invoked by the generalised resources of an operating system. The overhead of this virtualisation is considerable. In general, we disregard operating system multiplicity systems (such as QEMU, Xen, and so forth) due to the high practical overhead of hosting virtualised images or manipulating them.

Furthermore, we also discard the `ptrace(2)` and `systrace(2)` mechanisms and inheriting isolation systems. The latter is being deprecated from the BSD family, while the former (which may suffer from the same error as the latter) requires considerable execution overhead to orchestrate. The `kauth(8)` mechanism, which will be discussed, may be considered an in-kernel generalisation of these systems.

Lastly, this document does not consider non-BSD kernel isolation mechanisms, of which there are many. Most of these systems are implemented using strategies equivalent to FreeBSD Jail, enacting functional cross-checks, or as `kauth(9)` through security policies. Linux has several implemented systems, such as OpenVZ and VServer, and Solaris has Zones.

5 Issues

There are a number of issues with the available systems. The most significant issue with both available systems is the strategy of isolation: checkpoints within the flow of execution, instead of logically isolating resources in the kernel itself. The former strategy we call *functional* resource isolation, which is an isolation during the execution of conflict points.

In FreeBSD Jail, prisons are generally enforced by cross-checks at the system call boundary. For instance, a request to `kill(2)` from process $a \in A$ to $b \in B$ (as in our above scenario) is intercepted and eventually routed to `prison_check(9)` or similar function, which checks if the prison contexts are appropriate. In the Jail system, a host may affect guests, but guests may not affect each other or the host. Each potential conflict between process resources must be carefully isolated:

```
int
prison_check(struct ucred *c1, struct ucred *c2)
{
    if (jailed(c1)) {
        if (!jailed(c2))
            return (ESRCH);
        if (c2->cr_prison != c1->cr_prison)
            return (ESRCH);
    }
    return (0);
}
```

This function, or similar routine, must wrap each conflict point in order to protect the isolation invariant of the instances. In order for this methodology to work, each conflict point must be identified and neutralised. Clearly, as the kernel changes and new conflict points are added, each must be individually addressed.

The `kauth(9)` system enforces a similar logic. When kernel execution reaches a fixed arbitration point, zero or more listeners are notified with a variable-sized tuple minimally containing the the caller's credential and the requested operation (additional scope-specific data may also be passed to the listener). Thus, a signal from $a \in A$ to $b \in B$ may be intercepted with operation `KAUTH_PROCESS_CANSIGNAL`, and the listener may appropriately allow or deny based on the involved credentials.

The Jail system, regarding resource isolation, has a considerable edge over `kauth(9)`: the `kauth(9)` framework does not provide any sort of internal identification of contexts. A practical example follows: if one wishes to provide multiple Unix environments, there's no way to differentiate between multiple "root" users. `kauth(9)` listeners receive only notice of user and group credentials, and has no logic to pool credentials into a authentication group. This considerably limits the effectiveness of the subsystem; however, it's not an insurmountable challenge to modify the implementation to recognise context, although instance process trees would not be rootable under `init(8)`.

Although FreeBSD Jail does have an in-kernel partition of resources, the implementation falls short of full partitioning. Each credential, if jailed, is associated with a `struct prison` with a unique identifier. When a request arrives to the kernel on behalf of an imprisoned process, the task credentials have the standard Unix credentials and also an associated prison identifier. This allows conflicting user identifiers to be collected into prisons.

Our issue with both systems is the strategy by which isolation is enforced: functional isolation, where every point of conflict must be individually resolved. This is flawed security model, where in order to guarantee isolation, one must demonstrate the integrity of every single conflict point. Since some conflict points are indirect, this is often very tricky, or outright impossible. A cursory investigation of the `sysctl(3)` interface reveals that the `hostid` of the host system may be set by guests. Although this may be deliberate, the onus is on the developer to ensure that all conflicts are resolved; or if they are not, to provide an explanation.

6 Proposal

We consider an alternative approach to isolation that provides *a priori* isolation of kernel resources: logical isolation. Instead, for example, of cross-checking the credentials of process $a \in A$ signalling $b \in B$, we guarantee $A \cap B = \emptyset$ by collecting resource pools into the resource context structures themselves. In other words, the system resources themselves are collected within contexts, instead

of requiring functional arbitration. Although this strategy is considerably more complicated to initially develop, the onus of meticulous entry-point checking is lifted from the kernel.

First, our method calls for a means of context identification. Like with Jail, we associate each credential with a context; in this document, we refer to this structure as the *instance* structure. Instance structures have one allocation per context and are maintained by reference counters.

In order to isolate at the broadest level, we comb through the kernel to find global structures. These we must either keep global or collect into the instance framework. Some resources, like the hostname and domainname, are trivial to collect. Others, like process tables, are considerably more difficult. Still others, like network stack entities (routing tables, etc.) are even more difficult. However, once these have been appropriately collected, we're guaranteed isolation without requiring complex border checks.

Further, we propose a forest of instance trees: instances may either be rooted at `init(8)` to create simultaneous, isolated system instances, or instead branch from existing instances, effectively creating a host/guest scenario. Child instances introduce some complexity; however, instead of building a selective non-injection into our original isolation model (as in FreeBSD Jail), we manage child instances through a management interface, instead of the violating our logical model. In practical terms, instead of issuing `kill(1)` to a child instance's process, a parent must operate through a management tool (described in "Implementation") with default authority over child operation.

Since the topic of hard resources (devices) is orthogonal to isolation as per our described scenario, we relegate this topic to the "Future Work" section of this document. The same applies for the proposed management interface.

7 Implementation

We focus our implementation on NetBSD 3.1. Our choice for this basis system was one of cleanliness, simplicity, and speed. FreeBSD proved to be too complex and already encumbered by the existing prison mechanism. OpenBSD, while simple and very well documented, can't compete with NetBSD (or FreeBSD) in terms of speed. NetBSD has proved to have very well-documented with concise code. The speed of the system is acceptable and the number of available drivers is adequate. Our choice of basis kernel is still open to change; the alterations, although extensive, are relatively portable among similar systems. From NetBSD we inherit a considerable set of supported architectures. Since this proposal doesn't affect the kernel's bottom-half, our project inherits this functionality.

The existing implementation, which is freely downloadable and inherits the license of its parent, NetBSD, carries the unofficial name "mult". The remainder of this document focusses on the design, implementation, and future work of the "mult" system.

7.1 Structure

The system is currently implemented by collecting resources into `struct inst` structures, which represent instances (similar to `struct prison` in FreeBSD's jail). Each instance has a single `struct inst` object. There are two member classifications to the instance structure: public and private. Public members are scoped to the instance structure itself; private members are anonymous pointer templates scoped to the implementing source file. What follows an abbreviated view of this top-most structure:

```
struct inst {
    uint              i_uuid;
    uint              i_refcnt;
    struct simplelock i_lock;
    int               i_state;
    LIST_ENTRY(inst)  i_list;

    char              i_host[MAXHOSTNAMELEN];
    char              i_domain[MAXHOSTNAMELEN];

    inst_acct_t       i_acct;
    inst_proc_t       i_proc;
    ...
};
```

In this listing, `i_host` and `i_domain` are public members: their contents may be manipulated at any scope. The `i_acct` and `i_proc` members are private; their types are defined as follows:

```
typedef struct        inst_acct *inst_acct_t;
typedef struct        inst_proc *inst_proc_t;
```

The definitions for `inst_acct` and `inst_proc` are locally scoped to source files `kern_acct.c` and `kern_proc.c`, respectively. The `inst_proc` structure is locally scoped with the following members:

```
struct inst_proc {
    uint              pid_alloc_lim;
    uint              pid_alloc_cnt;
    struct pid_table *pid_table;
    ...
};
```

This structure consists of elements once with global static scope to the respective source file. The following is an excerpt from the pre-appropriated members in the stock NetBSD 3.1 `kern_proc.c` source file:

```
static struct pid_table *pid_table;
static uint pid_tbl_mask = INITIAL_PID_TABLE_SIZE - 1;
static uint pid_alloc_lim;
static uint pid_alloc_cnt;
```

Other private members share similar structure. Deciding which non-`static` members to make private, versus instance-public, is largely one of impact on dependent callers throughout the kernel.

At this time, there are a considerable number of appropriated subsystems with one or both public and private members: processes, accounting, pipes, kevents, ktraces, System V interprocess communication, exit/exec hooks, and several other minor systems. The procfs pseudo-file-system has been fully appropriated with significant work on ptyfs as well. Subsystems are generally brought into instances on-demand, that is, when larger, more significant systems must be appropriated.

7.2 Globals

There also exists a global instance context for routines called from outside a specific calling scope, for example, scheduling routines called by the system clock. These may need to iterate over all instances. The global list of instances may be accessed from a single list structure `allinst`, much like the previous `allproc` for process scheduling. Instances may also be queried by identifier, which is assumed to be unique. This convention is under reconsideration for the sake of scalability and the possibility of conflicting identifiers with high-speed cycling through the namespace of available identifiers.

7.3 Locking

Locking is done differently for different services. Private members often have their own native locking scheme inherited from the original implementation. Some public members also inherit the original locking, most notably the process subsystem, which has several public members as well as a private definition. General locking to instance members, those without a subsystem-defined locking mechanism, occurs with the `i_lock` member of `struct inst`. The global instance list has it's own lock, appropriate for an interrupt context.

7.4 Life-cycle

Instances are created in a special version of `fork1(9)` called `forkinst1(9)`, which spawns an instance's basis process from the memory of `proc0`. At this time, the instance structure itself is created. The private members are passed to an appropriate allocation routine defined for each member; usually, the memory referenced by these pointers is dynamically allocated.

The life-time of an instance is defined by its reference count, `i_refcnt` in `struct inst`. When this value reaches zero, the instance is cleaned up, with each private member being passed to a corresponding release routing, and returned to the instance memory pool. These allocation and deallocation routines are similar for each instance:

```
int
inst_proc_alloc(inst_proc_t *, int);

void
inst_proc_free(struct inst *);
```

The release of an instance's final process (usually the `instinit(8)` or `init(8)` process) must be specially handled. In normal systems, processes must have their resources reclaimed by the parent or

`init(8)` under the assumption that `init(8)` never exits (or the system usually panics). This case is no longer true in our system, thus, a special kernel thread, `instdaemon(9)`, frees the resources of these special processes. Since kernel threads traditionally reparent under the `init(8)` process, and this process is no longer singular, kernel threads also are reclaimed by `instdaemoe(9)`.

7.5 Administration

There are several tools available with which one may interact and administer the instance framework. These interact with the kernel through either the `sysctl(3)` interface or a new system call, `instctl(2)`. The former is primarily to access information on the instance framework, while the latter manipulates it. The `instctl(2)` function operates on a single argument, which contains parameters for controlling the instance infrastructure:

```
int
instctl(const struct instctl *);
```

The `struct instctl` structure allows several modes of control: debugging information, starting instances, and stopping instances. We anticipate this structure to grow significantly to account for other means of control.

There are several implementations of these functions. The `instinfo(8)` utility lists information about instances and the instance framework, `instps(1)` is an instance-aware version of `ps(1)`, and `instctl(8)` which directly interacts with the `instctl(2)` system call.

An alternate `init(8)` implementation, `instinit(8)`, is currently used for non-default instances. This is a temporary measure while the semantics behind terminal sharing are formalised. The `instinit(8)` process acts like `init(8)` except that it doesn't start the multi-user virtual terminal system. It's a required process for all starting instances, i.e., it must exist within the root file-system in order for the instance to "boot".

The system for administering instances is still under consideration, and may change. At this time, we chose simplicity, in this regard, over elegance and scalability; our focus is on the system's stability and design continuity. Administration is a topic that we wish to re-consider when a significant degree of scalability has been achieved, and the administration of thousands of instances becomes necessary.

8 Future Work

The "mult" system has a fairly well-defined short-term future, with some interesting possibilities for long-term development.

In the short-term, we anticipate fully appropriating pseudo-devices into instances. Furthermore, we envision a system for delegating physical devices to instances in a clean, elegant manner. Although not strictly-speaking a short- or mid-term goal, optional mid-term work, once the framework interfaces has been finalised, is to optimise the boot and shutdown sequence of instances. Lastly, inter-instance communication and manipulation should also be added to the control framework divvying physical resources between instances.

Pseudo-devices require fairly significant consideration. The most important is the network, followed closely by the terminal. We intend on following a similar path as the cloneable network stack for FreeBSD[4], where the entire network stack infrastructure is fully brought into instances. Bridges will allow instances to claim an entire network device, whose data may be routed to another instance controlling the physical network interface.

Terminals may be appropriated by dividing between terminal instances and non-terminal instances, where the former is connected to a real terminal and the latter runs "head-less". At this point, all instances run head-less, i.e., they have no controlling terminal during boot. This necessitated re-writing `init(8)` as `instinit(8)` to skip multi-user virtual terminal configuration. Each terminal instance would connect to one or more virtual terminals. Obviously, since virtual terminals are a scarce resource, this will rarely be the case; however, connecting instances to terminals allows multiple sets of monitor, keyboard and mouse connecting to the same computer and interacting with

different instances.

There are a significant number of potential optimisations to the instance infrastructure. It's absolutely necessary that the scheduler and memory manager account for instance limits, allowing administrators to make real-time adjustments to the scheduling priority of individual instances. The Linux VServer uses a two-tiered scheduler for its instance implementation (which uses functional isolation): we envision a similar scenario but with an emphasis on real-time scheduling. Furthermore, we plan on introducing a per-subsystem configuration structure. At this time, each subsystem (processes, pipes, and so forth) is configured with the system defaults. By allowing these defaults to be changed, or disabled entirely, instances may be customised to their run-time environments.

With a completed isolation environment, we can begin to consider extending our model to support hardware. At this time, our model depends on an administrator to appropriately partition resources for each instance, much like with FreeBSD Jail. We envision extending isolation to the device level; instead of changing device drivers themselves, we consider an additional layer of logic at shared areas of abstraction code. Our approach is divided into two phases. First, we must have a means of reliably identifying devices; second, we must have a means to intercept data flow to and from these devices. Lastly, we must correlate device identity, operation, and credentials to arbitrate access.

We plan on drawing from the Linux kernel's `udev`[2] strategy to map devices to identifiers. Since the administrator will be creating the access module rules, there must be a means to flexibly label devices as they're recognised by the system. This strategy will involve our considerations of device pseudo-file-systems, which must be coherent with the notion of instances. This will govern the exposure of devices to instances, and considerably narrow the window of exposure.

Second, we must define arbitration points. These will most likely occur when requests are brokered to individual drivers. We plan on drawing on scope parameters from `kauth(9)` to find a generalised granularity. Realistically, most device isolation may be solved by an intelligent `dev` file-system. Both of these concepts, the device file-system and arbitration, must work together with the notion of instances. We propose an access module that arbitrates requests for hard resources, and limited interaction between other instance contexts. Since hard resources may not be logically isolated as may soft resources (as there's no natural correlation between a hard resource and a particular instance), the onus falls on the administrator to properly map instances onto devices.

9 Conclusion

In this document, we introduced the theory of isolation and discussed various existing strategies. After considering the limitations of these strategies, we proposed a new one. Our new strategy is not without its caveats: since the necessary altercations span the kernel diapason, keeping these sources synchronised with the origin kernel is a difficult task. However, since our changes primarily affect the kernel top-half, we believe that our sacrifice is warranted; we can still inherit additional drivers and sub-system changes from the main-line.

10 Acknowledgements

We'd like to thank the University of Latvia's Institute of Mathematics and Computer Science, where the system began development. Special thanks to the BSD.lv Project for its initial funding of the work, and a special thanks to Maikls Deksters for his support and contributions. Lastly, thanks to the Swedish Royal Institute of Technology's Centre for Parallel Computing for their continued support of this project.

References

[1] EuroBSDCon. *NetBSD Security Enhancemenents*, 2006.

[2] Greg Kroah-Hartman. udev: A userspace implementation of devfs. In *Proceedings of the Linux Symposium*, July 2003.

[3] SANE 2nd International Conference. *Jail: Confining the Omnipotent Root*, 2000.

[4] Marco Zec. Implementing a clonable network stack in the freebsd kernel. In *USENIX Annual Technical Conference, FREENIX Track*, 2003.

GEOM
in infrastructure we trust

Paweł Jakub Dawidek
<pjd@FreeBSD.org>

AsiaBSDCon
Tokyo, 2008

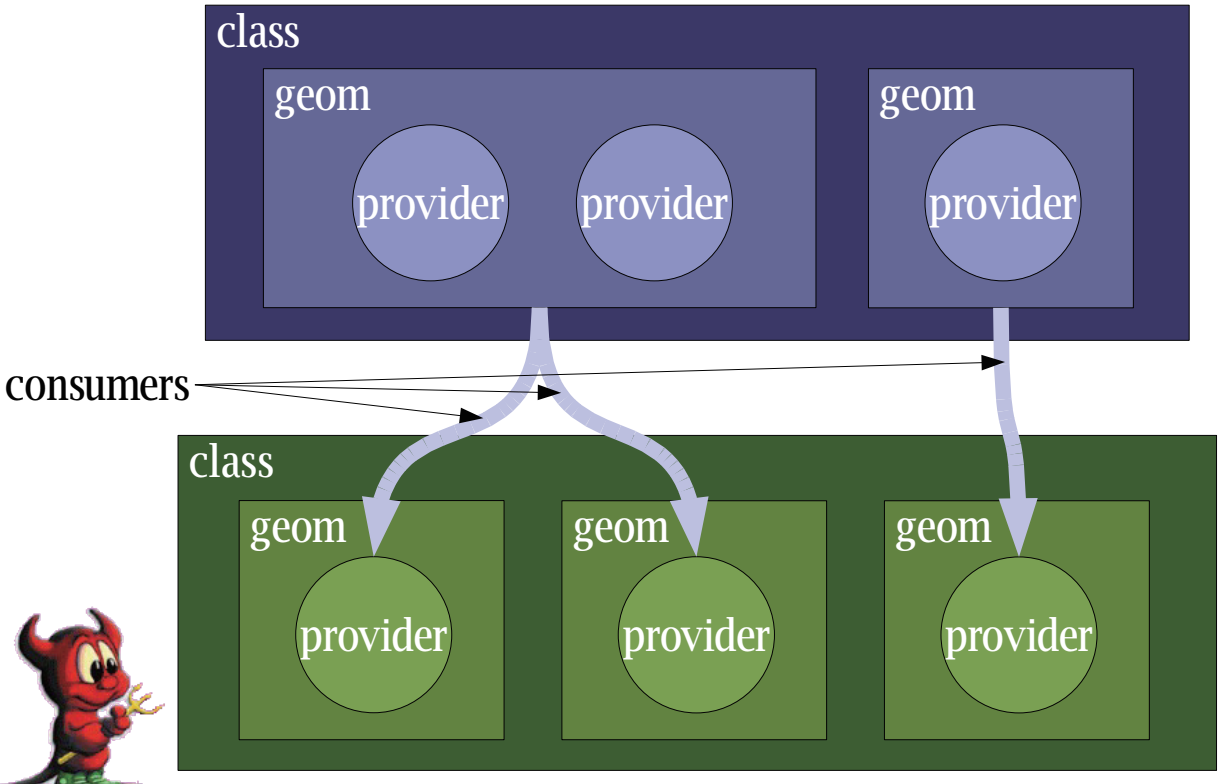
History

- the GEOM framework first appeared in FreeBSD 5.0
- implemented by phk@
- sponsored by DARPA
- first commit in March 11, 2002

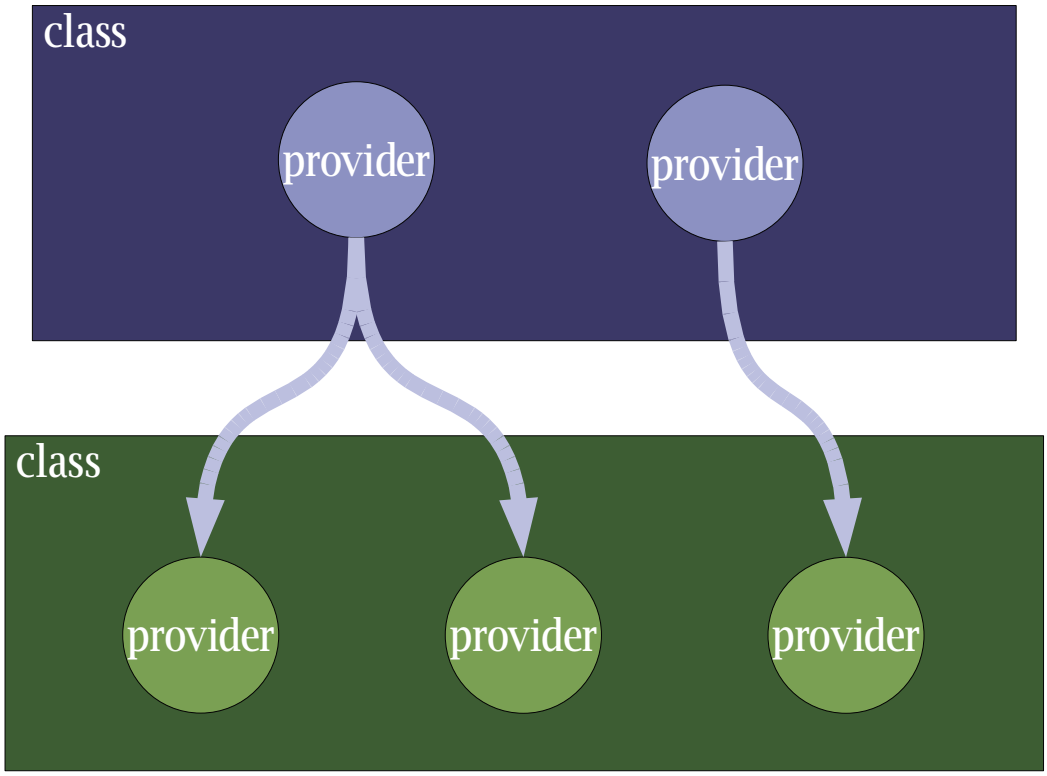
Nomenclature

- **class** – a kind of I/O transformation (eg. mirror, stripe)
- **geom** – an instance of a class
- **provider** – provides storage (eg. /dev/da0)
- **consumer** – connection between geom and provider

Nomenclature



What is worth to remember



Access

- **read**
- **write** – can't be already open exclusively
- **exclusive** – can't be already open for write

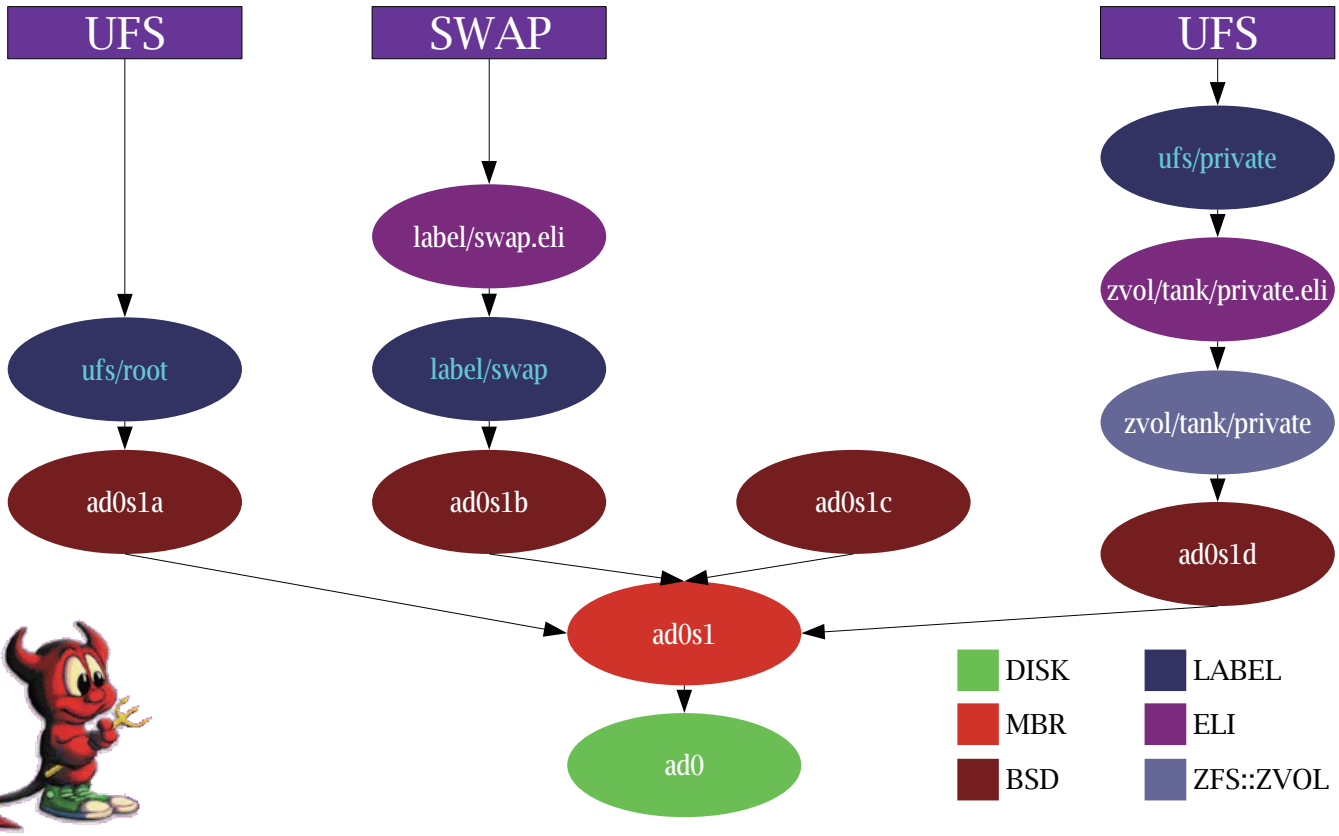
```
<provider id="0x84af5d00">
    <geom ref="0x84af5d80"/>
    <mode>r3w3e7</mode>
    <name>ad0</name>
    <mediasize>160041885696</mediasize>
    <sectorsize>512</sectorsize>
    <config>
        <fwheads>16</fwheads>
        <fwsectors>63</fwsectors>
    </config>
</provider>
```

I/O requests

- **BIO_READ** – read data
- **BIO_WRITE** – write data

- **BIO_DELETE** – destroy/free data
- **BIO_FLUSH** – flush cache, put data onto stable storage
- **BIO_GETATTR** – ask about properties

GEOM on my laptop



gconcat(8)

- simple provider concatenation
- appears in /dev/concat/name
- usage:

```
# gconcat label name da0 da1s1 da2s2d
```

da0	da1s1	da2s2d
0	4	7
1	5	8
2	6	9
3		10
		11

gstripe(8)

- RAID0
- appears in /dev/stripe/name
- usage:

```
# gstripe label name da0 da1s1 da2s2d
```

da0	da1s1	da2s2d
0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

gmirror(8)

- RAID1
- appears in /dev/mirror/name
- autosynchronization
- usage:

```
# gmirror label name da0 da1s1 da2s2d
```

da0	da1s1	da2s2d
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4

graid3(8)

- RAID3
- appears in /dev/raid3/name
- bigger sector size
- 2^a+1 providers
- usage:

```
# graid3 label name da0 da1s1 da2s2d
```

da0	da1s1	da2s2d
1/0	2/0	1^2/0
1/1	2/1	1^2/1
1/2	2/2	1^2/2
1/3	2/3	1^2/3
1/4	2/4	1^2/4

gjournal(8)

- block-level journaling (not file system level journaling)
- file system independent
- can be used for file system journaling with minimal knowledge on FS side
- currently can be used for UFS journaling
- usage:

```
# gjournal label da0
# newfs -J /dev/da0.journal
# mount -o async /dev/da0.journal /mnt
```

ggatec(8), ggated(8)

- exports storage over the network
- server usage:

```
# echo „10.0.0.0/24 RO /dev/acd0" > /etc/gg.exports
# echo „10.0.0.8/32 RW /dev/da1” >> /dev/gg.exports
# ggated
```

- client usage:

```
# ggatec create -o server /dev/acd0
ggate0
# mount_cd9660 /dev/ggate0 /mnt/cdrom
# ggatec create server /dev/da1
ggate1
# newfs -J /dev/ggate1
# mount /dev/ggate1 /mnt/data
```

gshsec(8)

- implements shared secret functionality
- all providers are needed to read the data
- appears in /dev/shsec/name
- usage:

```
# gshsec label name da0 da1s1 da2s2d
```

da0	da1s1	da2s2d
1/0	2/0	3/0
1/1	2/1	3/1
1/2	2/2	3/2
1/3	2/3	3/3
1/4	2/4	3/4

geli(8) 1/2

- provides encryption and integrity verification
- utilizes crypto(9) framework – uses crypto hardware automatically
- supports various encryption algorithms (AES, camellia, blowfish, 3DES)
- supports various authentication algorithms (HMAC/md5, sha1, ripemd160, sha256, sha384, sha512)
- key can be split over serval components (passphrase, random bits from a file, etc.)

geli(8) 2/2

- allows to encrypt even root provider
- passphrase strengthened with PKCS#5v2
- two independent keys can be used
- starts as many worked threads as many CPU cores the system has

Comming soon (currently in perforce only):

- suspend/resume support

ZFS

- FreeBSD port implements two GEOM classes:
  - ZFS::VDEV – consumers-only class used to access GEOM providers
  - ZFS::ZVOL – providers-only class used for ZVOLs

geom(8) 1/2

- control utility for most GEOM classes
- few standard commands that work with all classes (list, status, load, unload)
- usage:

```
# geom disk list
# geom bsd status
```

geom(8) 2/2

- class-specific functions implemented via libraries (/lib/geom/)
- for the above, one can use g<class> command
- classes aware of geom(8):
  - cache
  - concat
  - eli
  - journal
  - label
  - mirror
  - multipath
  - nop
  - part
  - raid3
  - shsec
  - stripe
  - virstor

Questions?

A Portable iSCSI Initiator

Alistair Crooks
The NetBSD Foundation
agc@NetBSD.org

Abstract

iSCSI is a method of accessing block storage across a network; it enables the SCSI protocol to work remotely, using a TCP/IP transport mechanism. The iSCSI infrastructure has two components - the target, which is the device; and the initiator, which is the end which communicates with the target. The initiator is normally situated in the operating system, sitting underneath the file system code, and making the remote storage appear as a normal, local SCSI device.

The NetBSD iSCSI target is implemented as a userspace iSCSI target. This paper covers the other half of the iSCSI world; the initiator, its interaction with the iSCSI target, and the desirable attributes for an ideal initiator. The portable iSCSI initiator is then described, and in particular describes the unique approach that was taken in the design and implementation of the portable iSCSI initiator. There follows some analysis and discussion of the overall iSCSI performance in real world benchmarks. The initiator's portability between operating systems is also discussed, along with experience gained from its deployment on other BSD systems, and further afield on Linux and Solaris. Some interesting uses of the initiator are described. The paper continues by examining remote storage as a whole, and outlines some places where remote storage could be used to considerable effect. The paper concludes by examining some areas in which iSCSI can bring benefits, and also some other areas in which iSCSI could be extended.

1. Introduction

Until recently, storage access across a network has been accomplished at two levels - at the block level via Storage Area Network (SAN) implementations, or at the file level using file systems such as NFS and Samba, often referred to as Network Attached Storage (NAS). More recently, the emphasis has been placed on block-storage SAN networks. Traditionally these networks have been implemented over custom networks of dedicated fibre, and using Fibre Channel (FC) to communicate. Recently, IP SANs have been deployed more commonly, taking advantage of already existing and cheaper TCP/IP networks, administration and management. Although the price of FC devices and networks have recently dropped, the perception still remains that they are the higher-priced option.

The virtualized world, with mobile and agile virtual machines which can be migrated from physical server to physical server, demands block storage which is geographical location independent. Although the virtual machines currently need to stay on the same VLAN when migrating, the storage may be completely remote. The provision of mobile IPv6 may address this restriction.

The market and opportunities for iSCSI storage exist, and until now, the BSD projects have been able to take advantage of this storage protocol, but only in part, by acting as a server, using the NetBSD iSCSI target. The NetBSD iSCSI target [Crooks2006] was added to the NetBSD src

repository in February 2006, and was subsequently made available for use on other POSIX-based operating systems, by packaging it up, complete with GNU autoconf scripts and templates, and added to pkgsrc. It was quickly added to the FreeBSD ports collection. It is portable to any POSIX operating system, and has even been compiled on Windows XP. Initiators, or client side of the protocol, despite a few examples, have not been forthcoming. There are a number of reasons for this, foremost amongst them being the necessity to embed the initiator within the kernel, which therefore limits their portability. iSCSI initiators thus are usually made available for only one operating system.

However, the iSCSI target is only half of the equation - the initiator is the harder part. It listens for operating system requests from a file system to manipulate storage (read, write, mkdir, mknod etc), formats the request and sends them to the target, and waits for the response, in turn waiting for the response to appear from the target, and sending the response back to the file system which requested the operation. The similarity of this operation to that of a userspace filesystem, as commonly found in the FUSE and ReFUSE examples, is striking.

2. Background

2.1 iSCSI Overview

The IETF proposed RFC 3720 in 2003, specifying the iSCSI protocol. RFC 3720 identifies the iSCSI protocol - in brief, two SCSI devices will now communicate by using a TCP connection as a transport mechanism, rather than a dedicated cable. The geographical independence that this provides gives rise to a number of dramatic benefits, but also raises a number of issues.

An iSCSI target can be thought of as the SCSI server - it sits waiting for requests to arrive from an initiator, performs work according to the nature of the request, and returns results to the initiator.

Immediately, a number of conclusions can be seen:

- geographical independence places a much greater load on TCP/IP communications networks
- geographical independence makes iSCSI a much more attractive solution for Disaster Recovery and Business Continuity
- existing SANs, typically high-end, expensive, Fibre Channel-based have a low-cost, pervasive, easily-managed competitor
- the security aspects are much more prevalent - MiTM attacks, spoofing, data traffic analysis, commercial espionage are now a reality (typically, existing SANs will be built on a custom Fibre network, separate from all other networks)

2.2 iSCSI Targets

iSCSI targets are the block servers for the iSCSI protocol. They are available from many vendors, although, typically, vendors have chosen to embed these in the operating system kernel. This is presumably to take advantage of speed benefits - there is no need for a network packet, destined for the iSCSI target, to move to userspace; instead, the SCSI request can be resolved within the kernel, with existing SCSI driver functionality if necessary, and the results communicated back to the initiator immediately. Whilst this is certainly true, the advent of zero-copy TCP functionality does nullify the performance benefits (to a certain extent). In addition, some things are better performed

in userspace - authorisation, and perhaps link aggregation, for example. Userspace programs are also much easier to develop and maintain, and have less impact in the unlikely situation that bugs exist.

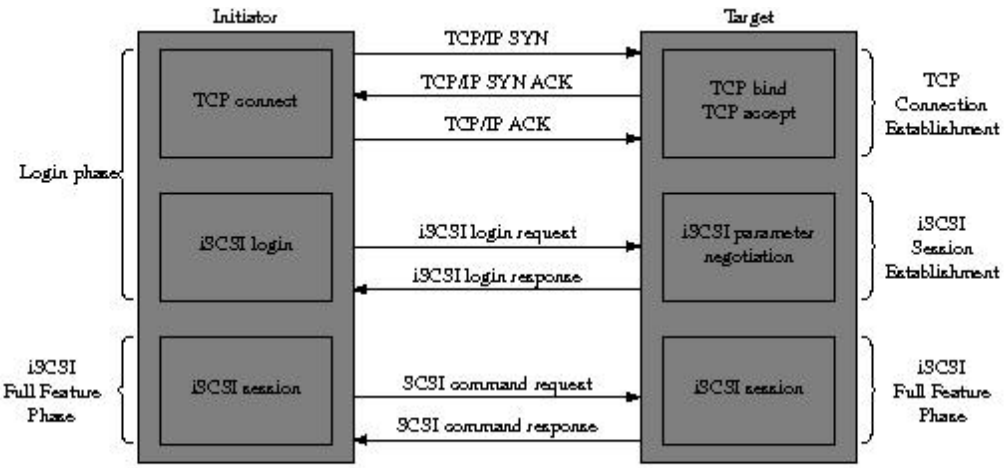
The NetBSD iSCSI target presents LUNs to initiators which connect to it. It can do a primitive form of LUN masking, by specifying a CIDR for requesting initiators, and not responding to requests from initiators which do not conform to this CIDR. The initiator also needs to "log in", although, since CHAP is used to do this process, this primitive form of authentication should not be considered to be any form of security.

2.3 iSCSI Initiators

To communicate with an iSCSI target, an initiator is used. Because of the nature of the initiator (by issuing SCSI requests, and interpreting responses, it mirrors a disk driver) initiators have typically been implemented as part of the operating system . This is analogous to traditional SCSI drivers within the operating system - issuing SCSI requests to block devices, and sending responses back to file system components, also within the operating system.

iSCSI initiators can take various forms: hardware initiators are available in the form of Host Bus Adapters (HBAs), and software initiators come with many operating systems. It is expected that some ethernet NICs will get iSCSI offload capabilities over the next few years. The rest of this paper will deal with software iSCSI initiators.

To illustrate the information flow in an iSCSI session, the establishment and operation of an iSCSI session is shown in the iSCSI Session Establishment Diagram below:



iSCSI Session Establishment Diagram

There are a number of software initiators already available for FreeBSD - the Lucent initiator from 2003, which is for an older version of FreeBSD and does not appear to have been actively maintained, and the kernel-based initiator which again seems to have less maintenance applied than

is desirable. For Linux, the most popular iSCSI solution is the UNH initiator and target, which is actively maintained, and regularly updated. For Mac OS X, Studio Networks provide an initiator, and the Microsoft Initiator is freely available for Windows XP. Solaris 10 has had an iSCSI initiator since Solaris 10 Update 2, and an iSCSI target since Solaris 10 update 5.

Despite the fact that iSCSI has been around as a protocol for over five years now, it has yet to make its mark in the BSD world. The NetBSD iSCSI target has been available for two years, and it has been incoporated into NetBSD's pkgsrc, the FreeBSD ports system, and is part of the FreeNAS distribution of storage-related products. Most of the use of the iSCSI target has been in presenting LUNs to initiators running on different operating systems – the target gets most use, measured through questions received, and “thank you” emails, from Microsoft and Linux initiators. The unenthusiastic take-up of iSCSI in the BSD community must be laid at the door of the initiator – it is difficult to use a product if the means of conversing and interacting with that product is not there.

For the BSD operating systems to take advantage of all the benefits of iSCSI storage, a useful, robust iSCSI initiator was needed. It must be portable across all the variants of the BSD world, though, since they use different SCSI and IDE subsystems, to say nothing of SATA and SAS devices. To achieve these differing goals, some lateral thinking would be necessary.

2.4 iSCSI Test Harness

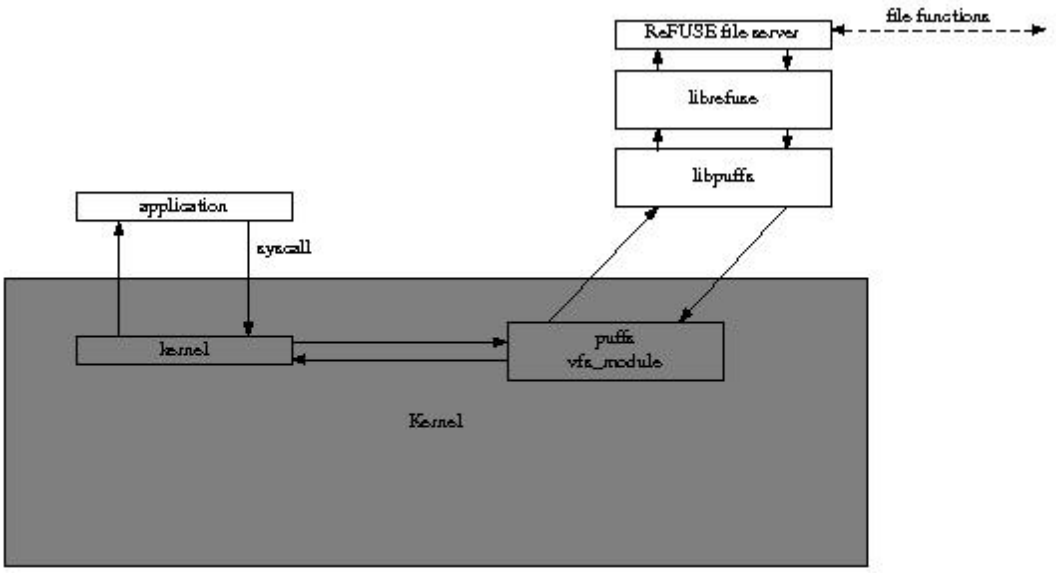
The iSCSI code in the NetBSD repository has initiator functionality, mainly for use in its test harness. All of the protocol encapsulation and decapsulation is available, working and tested. The test harness for the iSCSI target is a userspace program which issues iSCSI requests to the target, and verifies the responses it receives back.

The iSCSI target in the NetBSD src repository is split into a *libiscsi* library component, and a driver program for the target itself. The library will do the encapsulation and decapsulation of the iSCSI protocol itself, along with dealing with setting up the iSCSI connection, and tearing it down, and dealing with all the parameter negotiation and authentication that is necessary for a full-phase iSCSI login to take place.

2.5 puffs and ReFUSE

Recent work by Antti Kantee [Kantee2007] to provide "pass to userspace file system" framework, and by the author to re-implement the FUSE interface on top of puffs [KanteeCrooks2007], has resulted in a framework, within NetBSD, whereby user-implemented file systems can be written to take file system input requests from the kernel, perform work, and to return the reply to the kernel, thereby emulating a kernel-based file system.

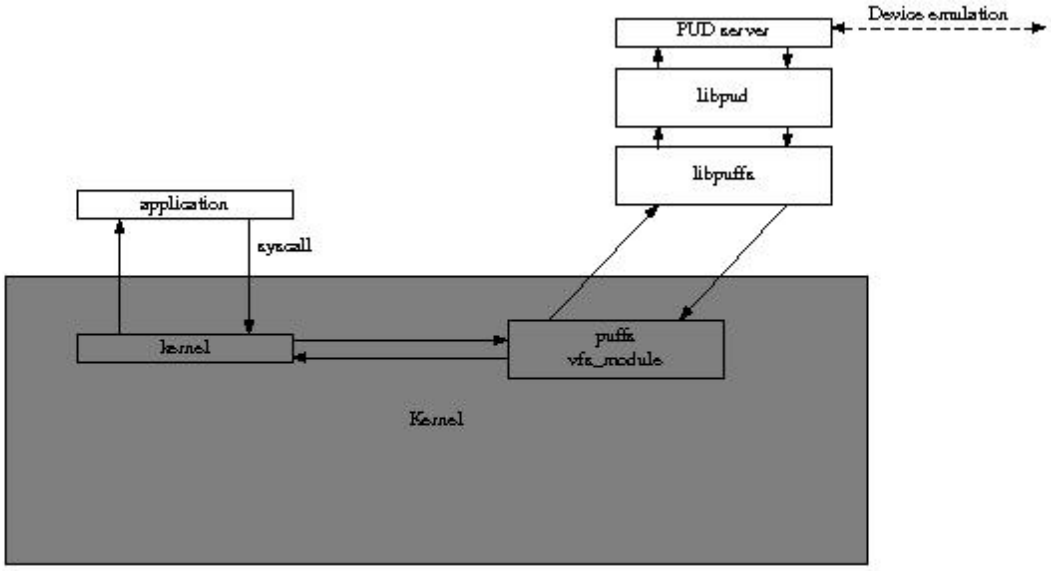
The FUSE interface is used within NetBSD to provide a standardised means of writing userspace-based file systems, the few puffs example file systems notwithstanding. This allows the puffs interface itself to mutate and change. Since FreeBSD presents a FUSE interface as its userspace-based file system, any userspace-based file system written to run via FUSE will run on NetBSD, FreeBSD, Mac OS X, Linux and Solaris.



ReFUSE infrastructure

2.6 Pass-to-Userspace Device Functionality

The same functionality, when used to satisfy block-device requests, would be tremendously beneficial to implement a userspace version of the iSCSI initiator.



Proposed PUD Infrastructure

A system call would be made from the application to the kernel, from where it would be routed through the kernel to the PUD server in userspace, which would issue the requests, gather the responses, and return that response through the kernel to the calling application.

However, puffs is perfectly capable of routing a request to a virtual device through its own protocol to a userspace file system, which can then route this request further. This means that there is no

reason to create new "pass to userspace device" functionality - the existing puffs userspace library and kernel functionality can be reused to satisfy device requests.

3. The Portable iSCSI initiator

3.1 Initiator Infrastructure

At a conceptual level, the iSCSI initiator itself takes requests for blocks (read, write, get size) etc from a higher level, encapsulates these requests in the iSCSI porotocol, sends them to the iSCSI target, and receives the response, which is decapsulated, and sent back to the caller.

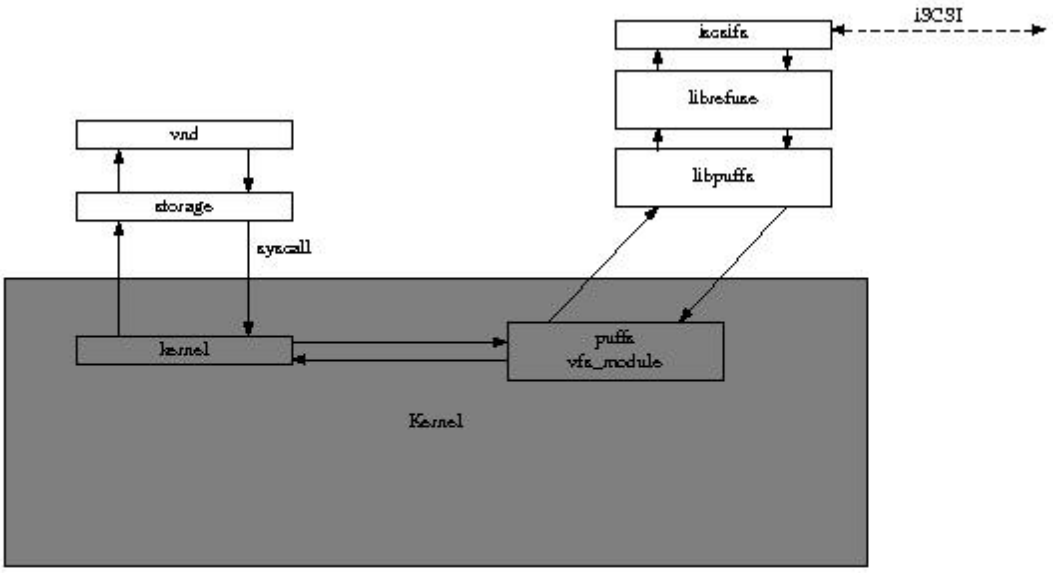
The initial development plan for the portable initiator - given the presence and availability of a number of building blocks such as the iSCSI test harness, and the NetBSD iSCSI target as a userland program, with a POSIX interface, and quite portable to other POSIX systems - was to build a device analogue of FUSE/ReFUSE/puffs functionality; where FUSE works on file system requests, this layer would work on device driver requests. The BSD-licensed FUSD [Nelson2003], a Linux Framework for User-Space Devices, was investigated - it is a kernel facility for Linux which provides user-space device support. Our new device functionality would interpret read/write/ioctl requests in the same manner, sending them to the userland for the component to re-route the response back through the kernel to the caller. This involved a significant amount of code duplication between the device level and puffs/ReFUSE.

During the implementation of the PUD functionality, a number of features were observed:

- substantial re-implementation of puffs functionality was being attempted
- the puffs transport was a multiplexing transport layer, and to re-implement that would not be a good use of resources
- a huge amount of code re-use should therefore be possible betwen puffs and pud
- the ability to use the userspace iSCSI test harness as a pud server became very attractive

After much discussion, during which one observation was made - that devices could work perfectly well on top of puffs, since a /dev instance on top of puffs as part of Antti Kantee's initial puffs testing, the author arrived at the current design - to use the current ReFUSE functionality to receive the requests via the standard FUSE or puffs/ReFUSE capabilities, rather than to implement pud functionality.

The portable iSCSI initiator was then built on top of the userspace puffs and ReFUSE; it uses initiator functionality available in the existing libiscsi to encapsulate requests and decapsulate responses from the target. An immediate advantage of the user-space implementation was the speed of development.



Portable iSCSI Initiator

In brief, the following example shows the portable iSCSI initiator in operation.

```
rxvt
[11:11:12] agc@inspiron1300 ~ 12 > priv iscsifs -u agc -h inspiron1300-wired /mnt &
[1] 17928
[11:11:19] agc@inspiron1300 ~ 13 > ls -al /mnt/inspiron1300-wired/target0/
total 2304
drwxr-xr-x  2 agc  agc       6656 Feb 10 10:55 .
drwxr-xr-x  2 agc  agc       6656 Feb 10 10:55 ..
lrw-r--r--  1 agc  agc         18 Feb 10 10:55 hostname -> inspiron1300-wired
lrw-r--r--  1 agc  agc          9 Feb 10 10:55 ip -> 10.4.0.42
lrw-r--r--  1 agc  agc         16 Feb 10 10:55 product -> NetBSD iSCSI
-rw-r--r--  1 agc  agc  104857600 Feb 10 10:55 storage
lrw-r--r--  1 agc  agc         43 Feb 10 10:55 targetname -> iqn.1994-04.org.netbsd.iscsi-target:target0
lrw-r--r--  1 agc  agc          8 Feb 10 10:55 vendor -> NetBSD
lrw-r--r--  1 agc  agc          4 Feb 10 10:55 version -> 0
[11:11:26] agc@inspiron1300 ~ 14 >
```

3.2 Benefits

A number of benefits are gained through the use of userspace components

- The first, and main, benefit of using a userspace initiator, is ease of development. Using the existing puffs functionality allows us to route file system requests through the initiator functionality in libiscsi with no new kernel components.
- The size of the iSCSI initiator, in terms of source code, is 17 kilobytes of commented C. Although this does not include the virtual directory routines which are used to keep track of directory entries in virtual file systems, these are common to all of the NetBSD-initiated

ReFUSE-based file systems in the *src/share/examples/refuse* directory.

- Because it is implemented in userspace, the iSCSI initiator is better suited to handling authentication, and to using existing userspace tools to take advantage of VPN, IPsec and other means of securing the communications mechanism between initiator and target.
- Portability is obviously a keen benefit of a userspace implementation, since the same initiator can be used, without modification, on a variety of different operating systems. Primarily, any POSIX operating system which supports the FUSE interface, is the only pre-requisite of the iSCSI initiator. Since not even the different BSD variants use the same kernel code for disk abstractions and drivers, this is a substantial win. This also increases hugely the number of testers, and the amount of testing, which the initiator receives. This is not constrained to BSD operating systems - Linux can use the portable initiator as well.
- Simplicity is an underrated benefit. If there is a lot to go wrong, Murphys law posits that it will go wrong. Conversely, if there is little which is capable of malfunction, then little or nothing will malfunction.
- Security is greatly increased with the above benefit, that of simplicity. Especially for such a piece of code, which is used to manipulate basic data, any improvement in the security is to be welcomed.
- A userspace implementation, especially on NetBSD, means that it is easily possible to stack other devices on top of the iSCSI initiator, such as encrypting block devices, or to place the initiator in a RAID implementation, to mirror simultaneously updates to the local file system on a remote site. This will be expanded in more detail later in this paper.

3.3 Drawbacks

There is a feeling in some BSD circles that using a userspace component for iSCSI initiation is somehow "cheating". Whilst it is certainly much easier to implement an iSCSI initiator in user-space, the ease of development, debugging and improvement in development time should not be underestimated.

Concerns have also been expressed about the performance of an iSCSI initiator with a sizeable userspace component. This will be addressed in Section 3.5 of this paper, Benchmarks and Real-World Experience; altogether, it is not considered an issue for the portable initiator.

By presenting the remote storage as a regular file in the local file system, there is a need for a mapping block device (vnd) to be used to give the regular file block-device characteristics, so that device management and manipulation will function properly. This device may not be available to all users of the computer. To avoid use of the vnd device, a mknod could be attempted by the initiator, or the equivalent of the umass USB attachment for mass storage devices could be made. This is an area of future work, although the use of devfs instances does mean that we lose some of the inherent portability by presenting the storage via a block device directly.

Security could be another issue related to performing iSCSI initiation in userspace. Since all data is travelling over communications lines, and block level devices have none of the file and directory permissions that pertain to networked file systems, this is always going to be a problem, and is not limited to iSCSI alone. There are a number of areas which have to be protected from a number of attacks, including replay, MITM, snooping and spoofing. Once again, this is the case for all SAN and IP SAN instances.

3.4 Availability

The portable iSCSI initiator is available under a 3-clause BSD licence, and should be portable to any system with a FUSE implementation, including FreeBSD, Mac OS X, Solaris and Linux. The NetBSD iSCSI target is available under a 3-clause BSD licence, and should be portable to any POSIX-like system.

3.5 Benchmarks and Real-world Experience

The first test that was run on the portable initiator (communicating with the iSCSI target on the same machine) was to do a build.sh run, which builds the complete NetBSD userspace and kernels. The run completed within 0.5% of the time taken to complete the build.sh run using standard disk-based storage. The machine used was an amd64 running NetBSD 4.99.34, with 8 GiB ram. Because of the memory size on that machine, memory caching will be a factor in the figures.

It is interesting to note that this was the first test carried out on the initiator. Usually, with a kernel component, there would be a small “Hello World” type of testing carried out, and gradually ramp up the resources to provide more and more functionality. In this case, since the *libiscsi* library had already been used and found to work reliably during the development of the iSCSI target with the iSCSI test harness, the code was known to work well.

Subsequent runs have shown that it is the usage pattern of data on the iSCSI volume which is the main factor in determining the speed of operations.

3.6 Portability

The portable iSCSI initiator, due to its implementation on top of ReFUSE, should work on any implementation of FUSE. This includes FreeBSD, Mac OS X, and extends to Linux and Solaris. If a traditional Berkeley Fast File System is used, there is a requirement for a special device to be able to be crafted on top of the storage virtual file described earlier, since FFS uses the vnode internally to detect repeated attempts to mount a file system on the same underlying storage. FreeBSD and Mac OS X have the *vnd* device, which can accomplish this, Linux has the *loopback* device, and Solaris has *lofi*, the loopback file device.

3.7 Features

Some of the features of the Portable iSCSI initiator are discussed below.

3.7.1 LUN Masking

In the same way that zones need to be able to specified in a better manner, we also need to be able to mask individual LUNs from discovery by certain hosts. To a certain extent, this is mandated by the desire of one commercial operating system to acquire whichever LUNs that it discovers. Accordingly, the LUNs presented by the target can be dependent upon a CIDR which is given in the /etc/iscsi/targets file. Although LUN masking can be perfomed at the present time, its ability is limited, and this part of the iSCSI solution will be enhanced in future.

3.7.2 LUN RAIDing

To a certain extent, the NetBSD iSCSI target allows for LUN RAID already - the */etc/iscsi/targets* configuration file is made up of two types of definition - the basic unit of storage is the extent which specifies the file or device, the starting address and its length. LUNs sit on top of extents or other LUNs, and can be arranged in a concatenated manner (raid0), or a mirrored manner (raid1). The areas for future enhancement here are

- the provision of dynamic configuration of additional RAID1 LUNs, to allow online backup to be made seamlessly
- the dynamic growing and shrinking of LUNs
- the possible addition of RAID5 (which may add too much complexity for too little gain in today's era of disk sizes of 1 Terabyte and more). File systems which are able to take advantage of these enhancements are necessary, too. NetBSD has long used RAIDframe as its component for providing resilience within the disk storage subsystem. RAIDframe can use an iSCSI "disk" as one of its components.

3.7.3 Block Device Encryption and Security

The iSCSI protocol does not mean to define or include any means of security; rather, it allows other forms of security to be used to authenticate and control access to the storage. It allows CHAP, SRP, Kerberos (not GSSAPI) or none authentication mechanisms to be used. In addition, to maintain data integrity, different digest algorithms can be used to verify that the storage has reached its destination without being corrupted (undetected, despite TCP's checksumming ability, which is considered weak by some).

Security of communication should be done by using some form of VPN to prevent the communications being understood, even if they are captured.

Security of the data at rest can be assured by using the appropriate block-level encrypted device, such as NetBSD's *cgd*, FreeBSD's *GBDE*, or OpenBSD's *svnd*. Once remote storage becomes a possibility, security aspects start to take on a larger role. Encrypted block devices have already been mentioned, but encrypting the storage is no good if all communications to and from the storage take place in the clear. For that reason, communication of iSCSI traffic should take place over IPsec wherever possible, or some other form of VPN. However, for domains which exist in a co-located facility, or which are not under one's complete physical control at all times (such as on a laptop or ultra-portable machine), some form of encrypted block device can be used, stacked on top of the iSCSI volume, to provide protection from people who can gain access to the physical disk storage backing the iSCSI volume. Instructions for using an encrypted block device in conjunction with the portable iSCSI initiator are provided elsewhere [Crooks2008] .

4. Further Work

This section discusses some possible enhancements that could be made to the NetBSD iSCSI system, and any further work that is envisaged.

4.1 Kernel-based initiators

It is quite possible to "drop" the small bits of functionality implemented for the iSCSI initiator into the kernel. In doing that, we would lose the extreme portability that we have now, but performance and efficiency may be gained.

4.2 iSCSI Booting

To be able to boot from an iSCSI device, network drivers would have to be modified. This is similar to modifying the libsa drivers to be able to etherboot. Ideally, to enable operating systems to boot from an iSCSI volume, iSCSI boot functionality needs to be available. This means that either the operating system, or some specialised hardware like an offload engine, or iSCSI Host Bus Adaptor, will issue the request to the target to read the bootblock from the iSCSI volume. This, however, is not a necessity - typically even the smallest device today has its own memory in the form of FLASH or ROM or memory card. This means that it can be used when a full network interface is unavailable. There is another way that iSCSI booting could be achieved without using custom hardware. A NetBSD kernel, with a ramdisk attached, could be downloaded from a tftp server, or other network-boot enabled appliance. As part of the boot sequence, the ramdisk will be set up, including the puffs device and refuse library, and then a userlevel initiator can be used to connect to a given iSCSI target, and to mount a file system on top of the iSCSI volumes presented by the target.

4.3 Multipathing

Being able to use resiliency features such as multipath is extremely attractive, especially in the use of iSCSI for Disaster Recovery and Business Continuity Planning.

4.4 Zoning

SAN zoning can be implemented at two levels: in hardware, and in software. Because disks are no longer attached to the buses on the computers, there needs to be a way to discover, from the initiator, which targets are available. Traditionally, FC SANs have used a Simple Name Server, or SNS, to provide this facility. iSCSI has a similar facility, called iSNS. iSNS can be thought of as similar to DNS - iSNS will return the TargetName of the target in response to a query from the initiator. As part of this facility, certain targets can be configured to be available to certain hosts. This creates zones of storage which are available to different hosts. For example, production zone storage could be presented to production zone servers. At the present time, no iSNS has been provided, although this may change in the future.

4.5 VM integration and Application Migration

iSCSI boot is essential for Virtual Machine mobility and migration. Only when the storage is accessible on the complete network can a virtual machine be moved between physical hosts. It is envisaged that Virtual Machine Service Providers could use this functionality to manage their client virtual machines much more effectively.

4.6 Clustered File systems

Remote storage adds some complexity to file systems, since it is now possible to have mltiple file systems attempting to use the same storage at the same time. Clustered file systems are used for this purpose, allowing multiple access simultaneously to different hosts, and propagating changes to other users of the same storage.

Linux has long been using the GFS file system, from Red Hat; a BSD port of this will be forthcoming. This is a standard clustered file system.

High availability and resilience is another area which can be used to go efct with remote block-level storage. Chironfs is a high availability file system, meant to provide resilience in the case where one piece of storage is unavailable. RAIF is another file system which is used for high availability.

4.7 Dynamic management of iSCSI Target LUNs

At the present time, the NetBSD iSCSI target will read a configuration file, and present LUNs depending on the contents of the configuration file. There is no means of modifying the configuration without stopping the iSCSI target, and re-starting it, this time using the modified configuration from the configuration file.

4.8 iSCSI Bridging

There is the possibility to bridge the iSCSI protocol - to have an appliance which takes the iSCSI protocol, and decapsulates requests, and changes them into requests for a different protocol. This could be anything within reason.

4.9 iFCP and FCIP

In the Fibre Channel SAN world, two RFCs have been provided by the IETF, which provide remote block Fibre Channel storage via ethernet. These are RFC 4172, which defines "A Protocol for Internet Fibre Channel Storage Networking; iFCP", and RFC 3821 on "Fibre Channel Over TCP/IP (FCIP)". These two RFCs provide a means of using Fibre Channel protocols by communicating over TCP/IP. At the present time, there is some interest in these two RFCs, although it is uncertain as yet whether either is useful enough to overcome the huge long-term existing investments in Metropolitan-Area dark fibre, system administration, and hardware for existing Fibre-Channel implementations. There could be a market for extending the range of existing SANs by using some form of iSCSI or FCIP.

5 Benefits and Opportunities

There are a number of areas where the BSD operating systems can take advantage of a full iSCSI solution – both initiator and target:

- virtualization support - with virtualization becoming such a big topic in computing these days, remote block-level storage is an increasingly important part of the virtualization roadmap. In order to migrate guest or domU domains between physical hosts (in order to maintain service levels, and accomplish maintenance with high availability uptimes), the guest domains can be migrated to different physical hosts. This can only accomplished when storage is held remotely from the host or dom0 domain. iSCSI is a natural fit for this.
- General Remote Block-level Storage. Remote storage provision may be useful for computing equipment which is in places which are difficult to access - for example, in hazardous places, or for appliances like IP security cameras, whereby there is enough power to grab images, but not enough to power storage. NetBSD has already been deployed in the International Space Station - iSCSI can also be used to great effect in space, by using storage over communications lines to upgrade and downgrade software easily. Bearing in mind the increased capacities of portable storage (the standard storage on a USB key is currently 8 or 16 GiB), and with the increased bandwidth available from telcos, and despite the labelling of any transfer speed greater than 64 Kb/s as "broadband", remote storage is becoming a much more interesting area, especially when the storage appears to be directly attached:

    - PDAs - the ability to have a central block of storage which can be accessed for all things, such as photographs, calendars, contact books and general storage is not to be underestimated. Using iSCSI for this storage is the obvious choice, since the provision of TCP has already been done. The benefits also include never having to "sync" a portable device ever again.
    - Mobile phones - today, the lines between the typical PDA and the mobile phone are getting blurred. The standard phone nowadays will include a 4+ Megapixel camera, video cameras, and the ability to listen to MP3 files. Most people are standardizing on one device and carrying it - the advantage of this to them is the ease of use. iSCSI can benefit this hugely, since MP3 files can take up a large amount of space.
    - PVR (Personal Video Recorders) - this set of appliances need even more space than the previous set, and this is certainly an area where iSCSI can be of benefit. The relative lack of screen size on these devices means that viewing on a conventional-sized television of the same video streams, using the same source, would be a benefit. That can be achieved by a single central store, accessed by both home network and PVR.
    - Security cameras are often placed in areas which are difficult, or hazardous, to maintain. Having to access the camera to change a tape has not been done for a number of years. However, CCTV can produce a large amount of video imagery, which needs to be stored somewhere, and using iSCSI storage for this would be a plus point
    - Media distribution - for some new films and movies, the standard way of making the movie available is to copy it, and to transport these copies to the individual cinemas, where they are displayed. Rather than using this intermediate copy, providing an encrypted iSCSI device, and letting initiators attach to it, with zoning and LUN masking in place, together with strict IP filtering, would appear to be a safer and less wasteful method
    - Software distribution - instead of providing ISOs which are downloaded, software

distributors can present the image to the net as an iSCSI volume. The initiator can then mount the image as a cd9660, in read-only mode, and the image will appear as if it is a local CD or DVD. No media needs to be used in this process, it is a simple network transfer. It is hoped that NetBSD 5.0 will be made available in this manner (as well as existing methods such as bittorrent, ISO image and ftp), and it will be interesting to see how much download traffic is used in this way. It is interesting to note that the Microsoft Initiator expects any LUN presented to it to be a writable piece of storage, and so it cannot mount ISO images across the internet, for example. Windows XP needs to initialize the LUN presented to it in the Disk Administration menu.

- There has been an upsurge in the number of companies offering remote storage provision. This can be used as off-site backups for Business Continuity Planning and Disaster Recovery purposes. The problem with using these facilities is usually that the interface demands a Windows installation. There are some concerns about these sites, especially in the security arena - who is looking at my backups, and how can I protect them without inconveniencing them myself? - but the Windows interface is the reason that the whole debate is rendered moot for BSD users. If remote storage providers were to provide an iSCSI interface, then the use of encrypted block devices on top of the iSCSI initiator would answer the previous criticism.

- Other opportunities - the iSCSI target and initiator together have a complete SCSI subsystem in software. There are opportunities to make appliances based on this software. Still other opportunities present themselves:

    - an appliance can be made which keeps a record, with timestamps, of all the modifications to a piece of storage, in precise time order. Using this with existing snapshots of devices, a complete copy of changes ensures that the precise state of any storage device can be determined at any given time. This can be useful for statutory or audit requirements.
    - an appliance can be made which will do precise online backups, using a copy-on-write model, with no snapshot mechanism necessary.
    - instead of making snapshots of production systems, online and continuous backups can be done to a secure remote site by the methods outlined previously. In the event of recovery being needed from remote data, read-only copies are available with no re-configuration necessary (this avoids logistical problems of tapes being copied and transported back to the original site)
    - storage service providers could use iSCSI rather than proprietary protocols
    - distribution of software by exporting data from read-only iSCSI targets could become a mechanism of choice. Not only can packet filters be used to govern access to the iSCSI host and port, but encryption can also be used to make sure that, even if unauthorised access is gained to the image being distributed, this access is rendered useless, since the means to decrypt the image is not known.
    - "anonymous" iSCSI could be used to allow read-only access to an image. This would mean, for example, that ISO images could be made available on the Internet, people could mount the images remotely, and upgrades and software installation could be accomplished without having to transfer the whole ISO image (with iSCSI, only the blocks which were required would be read, and would be saved locally in the page cache). This could also remove the intermediate step of writing a CD or DVD containing the ISO image.
    - the provision of remote storage for PDAs and personal electronic equipment has already been mooted. With iSCSI booting, it is possible to enable remote devices attached to the Internet to boot from a remote iSCSI target.

6. Conclusions

This paper has talked about the portable iSCSI target, and the unique design of the portable initiator. Both the target and initiator are designed to be portable, so that the benefits of iSCSI may be obtained on all of the BSD operating systems, and also further afield. The iSCSI protocol was specified a number of years ago, and the initial promise of iSCSI has yet to materialise - in some ways it has been held back in the BSD world due to the lack of a viable, robust iSCSI initiator. This paper has described such a portable, yet performant and robust initiator, and has suggested some opportunities for the deployment of iSCSI, and also some extensions to the iSCSI model, applications and implementation.

References

[Kantee2007] http://2007.asiabsdcon.org/papers/P04-slides.pdf

[KanteeCrooks2007] http://2007.eurobsdcon.org/presentations/Antti_Kantee/refuse.pdf

[Crooks2006] http://www.alistaircrooks.co.uk/agc/papers/2006-11-eurobsdcon/iSCSI_beyond_the_hype.pdf

[Crooks2008] http://mail-index.netbsd.org/netbsd-users/2008/01/05/0001.html

[RFC3720] http://www.ietf.org/rfc3720.txt

[Nelson2003] http://www.circlemud.org/~jnelson/software/fusd/

OpenBSD

network stack internals

by Claudio Jeker

The OpenBSD network stack is under constant development mainly to implement features that are more and more used in todays core networks. Various changes were made over the last few years to allow features like flow tagging, route labels, multipath routing and multiple routing tables. New features like VRF (virtual routing and forwarding), MPLS and L2TP are worked on or planned. This paper tries to cover the design decisions and implementation of these features.

Introduction

The OpenBSD network stack is based on the original BSD4.4 design that is described in TCP/IP Illustrated, Volume 2[1]. Even though many changes were made since then [1] still gives the best in depth introduction to the BSD based network stacks. It is the number one reference for most part of the network stack. Additionally IPv6 Core Protocols Implementation[2] covers the KAME derived IPv6 network stack implementation and should be considered the reference for anything IPv6 related.

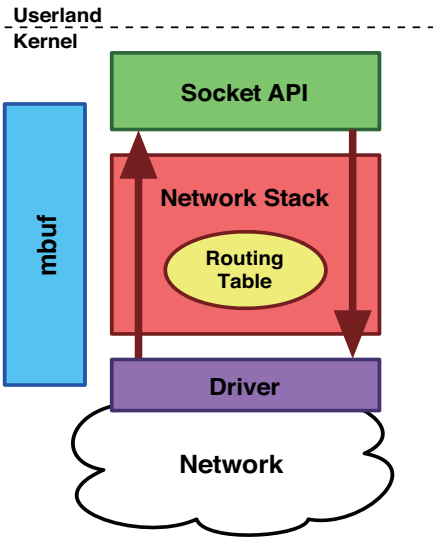


Figure 1: Building blocks of the network stack

The networking code can be split in four major building blocks: the network stack, the mbuf API, the routing table and the socket API. The various protocol support are hooked into the network stack on various defined borders. Protocols provide methods which are registered in a protosw structure. By looking up these structures instead of calling the functions directly a more dynamic and easily extensible stack can be built. There is an input and an output path that starts in the drivers and ends in the socket API and vice versa. Every packet passing through the network stack is stored in mbufs and mbuf clusters. It is the primary way to allocate packet memory in the network stack but mbufs are also used to store non packet data. Everything routing related is covered by the protocol independent routing table. The routing table covers not only the layer 3 forwarding information but is also used for layer 2 look ups -- e.g. arp or IPv6 network discovery. The calls to the routing table code are scattered all over the network stack. This entanglement of the routing code is probably one of the most problematic parts when making the network stack MP safe. Last but not least the socket API. It is the interface to the userland and a real success story. The BSD socket API is the most common way for applications to interact between remote systems. Almost any operating system implements this API for userland programs. I will not cover the socket API because non of the new features modified the socket API in detail.

mbufs

Packets are stored in mbufs. mbufs can be chained to build larger consecutive data via `m_next` or build a chain of independent packets by using the `m_nextpkt` header. `m_data` points to the first valid data byte in the mbuf which has the amount of `m_len` bytes stored. There are different mbuf types defined to indicate what the mbuf is used for. The `m_flags` are used in various ways. Some flags indicate the structure of the mbuf itself (`M_EXT`, `M_PKTHDR`, `M_CLUSTER`) and some indicate the way the packet was received (`M_BCAST`, `M_MCAST`, `M_ANYCAST6`). If `M_PKTHDR` is set an additional structure `m_pkthdr` is included in the mbuf. The first mbuf of a packet includes this `m_pkthdr` to store all important per packet meta data used by the network stack. The complete length of the mbuf chain and the interface a packet was received on are the most important ones.

Code Fragment 1: mbuf structures

```
struct m_hdr {
        struct  mbuf *mh_next;
        struct  mbuf *mh_nextpkt;
        caddr_t mh_data;
        u_int   mh_len;
        short   mh_type;
        u_short mh_flags;
};

struct  pkthdr {
        struct  ifnet *rcvif;
        SLIST_HEAD(packet_tags, m_tag) tags;
        int     len;
        int     csum_flags;
        struct  pkthdr_pf;
};

struct m_tag {
        SLIST_ENTRY(m_tag)      m_tag_link;
        u_int16_t               m_tag_id;
        u_int16_t               m_tag_len;
};
```

Mbuf tags are generic packet attributes that can be added to any packet. Mbuf tags are mostly used by the IPsec code and to prevent loops in the network stack when tunnelling interfaces are used. Up until OpenBSD 4.2 pf used the mbuf tags to store internal state information (`pkthdr_pf`). Every packet needs this state information if pf is enabled. Moving this structure from mbuf tags directly into the `m_pkthdr` almost doubled performance. The main reason of this speed up is that the allocation of mbuf tags is skipped. Mtag allocation is slow because `malloc(9)` needs to be used to allocate the dynamic length elements. Information that has to be added to every packet should probably be directly included in the packet header.

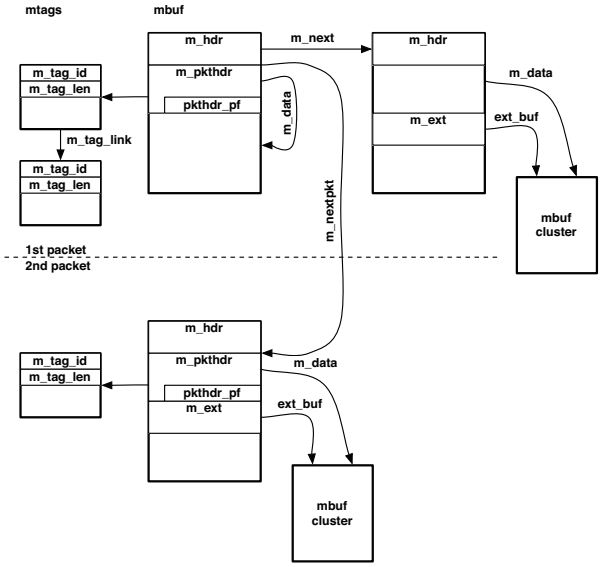


Figure 2: mbuf structures of two packets, 1st packet is built by an mbuf chain of two mbufs (first mbuf with internal data second with an external mbuf cluster).

Network Stack

Packets enter the network stack from userland through the socket API or by the network driver receive interrupt function. Packets received by the network card enter one of the layer 2 input functions -- `ether_input()` is the most commonly used one. This function decodes/pops off the layer 2 header and figures out the proper payload type of the data. In the Ethernet case the `ether_type` is inspected but first it is checked if it is a multicast or broadcast packet and the corresponding mbuf flags are set. Depending on the payload type an input queue is selected, the packet is enqueued and a softnet software interrupt is raised. This interrupt is delayed because `IPL_SOFTNET` has a lower precedence then `IPL_NET` used by the driver interrupt routine. So the driver can finish his work and when lowering the system priority level the softnet interrupt handler is called. The softnet handler checks `netisr` for any set bits and calls the corresponding protocol *interrupt* handler. Most of the time this is `ipintr()` or `ip6intr()` but the bridge, ppp and pppoe code use the softnet handler as well. So the `splnet()`/`splsoftnet()` dance has nothing to do with the layer 2/layer 3 border.

`ipintr()` dequeues all packets in the protocol input queue and passes them one after the other to `ipv4_input()`. `ipv4_input()` checks the IP header then calls `pf_test()` to do the input firewalling. Now the destination is checked and if the packet is not for this host it may be forwarded. Local packets are passed to the transport layer. Instead of hardcoding the corresponding handlers into `ipv4_input()` a more object oriented approach is used by calling the `pr_input()` function of a `protosw` structure. The `inetsw[]` array contains the various protosw structures indexed by protocol family.

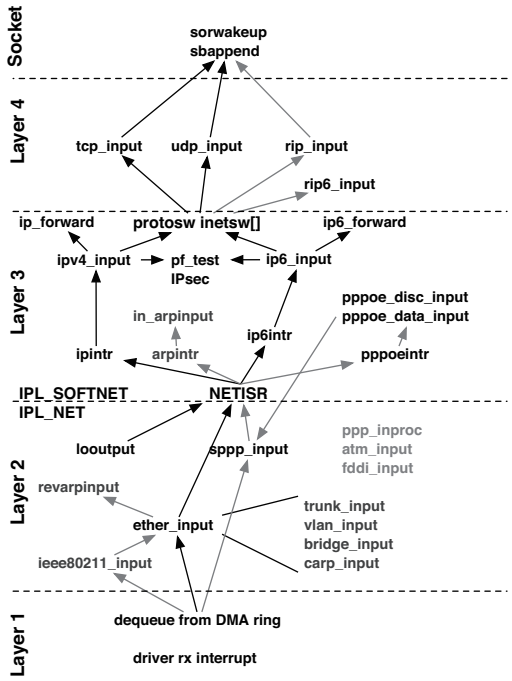


Figure 3: Network stack input flow

Common input functions are `tcp_input()`, `udp_input()` and `rip_input()` -- rip stands for raw IP and has nothing in common with the RIP routing protocol. These input functions check the payload again and do a pcb lookup. The pcb or protocol control block is the lower half of a socket. If all is fine the packet is appended to the socket receive buffer and any process waiting for data is woken up. Here the processing of a network interrupt ends. A process will later on call the `soreceive()` function to read out the data in the receive buffer.
In the forward path a route lookup happens and the packet is passed on to the output function.

Sending out data normally starts in userland by a `write()` call which ends up in `sosend()`. The socket code then uses the protosw `pr_usrreq()` function for every operation defined on a socket. In the `sosend()` case `pr_usrreq()` is called with `PRU_SEND` which will more or less directly call the output function e.g. `tcp_output()` or `udp_output()`. These functions encapsulate the data and pass them down to `ip_output()`. On the way down the output function is called directly (not like on the way up where between the layer 3 and 4 the protosw structure was used). In `ip_output()` the IP header is prepended and the route decision is done unless the upper layer passed a cached entry. Additionally the outbound firewalling is done by calling `pf_test()`. The layer 3 functions invoke the layer 2 output function via the `ifp->if_output()` function. For the Ethernet case, `ether_output()` will be called. `ether_output()` prepends the Ethernet header, raises the spl to `IPL_NET`, puts the packet on the interface

output queue, and then calls the `ifp->if_start()` function. The driver will then put the packet onto the transmit DMA ring where it is sent out to the network.

This is just a short fly through the network stack, in the real world the network stack is much more complex due to the complex nature of the protocols and added features. All the control logic of the network stack is left out even though it is probably the most obscure part of it.

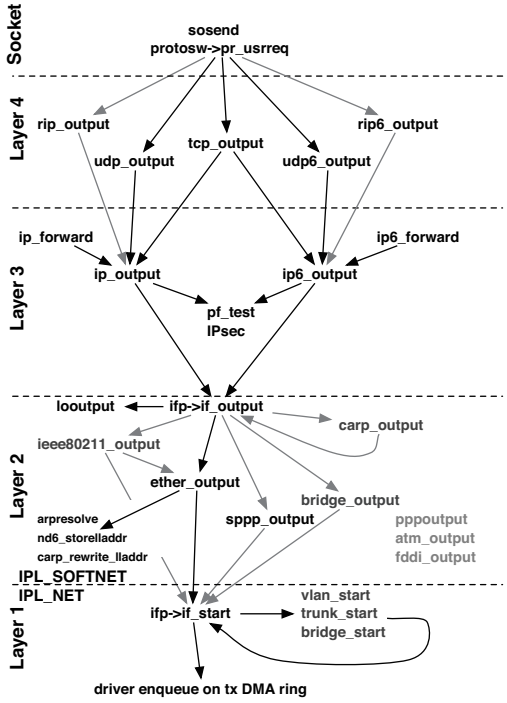


Figure 4: Network stack output flow

Routing Table

The routing table uses the same patricia trie as described in [1] even though minor extensions were done or are planned. The routing table stores not only layer 3 forwarding information but includes layer 2 information too. Additionally the patricia trie is also used by pf tables. There is a lot of magic in the routing table code and even minor changes may result in unexpected side-effects that often end up in panics. The interaction between layer 3 and layer 2 is a bit obscure and special cases like the arp-proxy are easily forgotten. The result is that routing changes take longer then expected and need to move slowly. Until now *only* routing labels, multiple routing tables, and multipath routing were implemented plus a major bump of the routing messages was done. The routing messages structure was changed to allow a clean integration of these features. Mainly the routing table ID had to be included into each routing header. With a few tricks it was even possible to have some minimal backward compatibility so that new kernels work with older binaries.

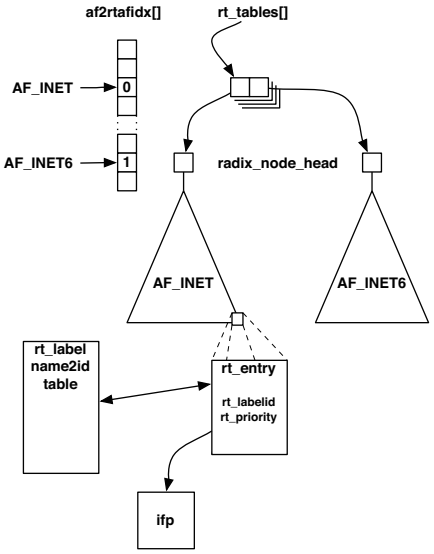


Figure 5: Overview of the routing tables

Routing Labels

Routing labels were the first OpenBSD specific extension. A routing label is passed to the kernel as an additional sockaddr structure in the routing message and so the impact was quite minimal. But instead of storing a string with the label on every node a per label unique ID is stored in the routing entry. The mapping is done by a name to ID lookup table. Labels can be set by userland and pf can make decisions based on these labels.

Multipath Routing

The implementation of multipath routing was initially from KAME but additional changes were necessary to make them usable. Especially the correct behaviour of the routing socket was a challenge. Some userland applications still have issues with correctly tracking multipath routes because the old fact of one route one nexthop is no longer true.

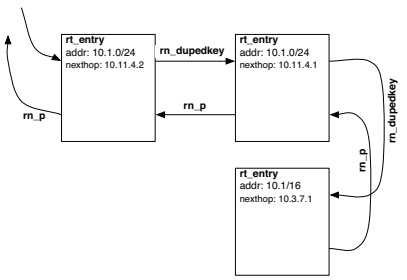


Figure 6: rn_dupedkey and multipath routes

Multipath routing is abusing the possibility of storing the same key multiple times in the routing table. This is allowed because of possible more specific routes with the same network address -- e.g. 10.0.0/24 and 10/8. Such identical keys

are stored in the `rn_dupedkey` list. This list is ordered by prefix length -- most specific first -- so all multipath routes are consecutive.

In the network stack some route look ups had to be exchanged with a multipath capable version. `rtalloc_mpath()` is the multipath aware route lookup which has an extra attribute -- the source address of the packet. The source and destination address are used to select one of the multiple paths. `rtalloc_mpath()` uses a hash-threshold mechanism[3] to select one of the equal routes and routes are inserted in the middle of the list of paths. This more complex mechanism to select and insert routes was chosen to keep the impact of route changes small.

Special sysctl buttons were added to enable and disable the multipath routing:

```
sysctl net.inet.ip.multipath=1
```

and/or:

```
sysctl net.inet6.ip6.multipath=1
```

Without these sysctl values set multipath routing is turned off even if multiple routes are available.

Multiple Routing Tables

Multiple routing tables are a prerequisite for VRF. The first step in supporting virtual routing and forwarding is to be able to select an alternate routing table in the forwarding path.

Every address family has a separate routing table and all routing table heads are stored in an array. With regard to VRF it was considered the best to always create a full set of routing tables instead of creating per address family specific routing tables. If a new routing table is created the `radix_node_heads` for all address families are created at once, see Figure 5. `pf(4)` is used to classify the traffic and to select the corresponding forwarding table. At the moment it is only possible to change the default routing table in the IP and IPv6 forwarding path. For link local addressing -- e.g. arp -- the default table is used.

VRF

The idea behind virtual routing and forwarding is the capability to divide a router into various domains that are independent. It is possible to use the same network in multiple domains without causing a conflict.

To support such a setup it is necessary to be able to bind interfaces to a specific routing table or actually building a routing domain out of a routing table and all interfaces which belong together. On packet reception the mbuf is marked with the ID of the receiving interface. Changes to the layer 2 code allow the use of alternate routing tables not only for IP forwarding but for arp look ups as well. With this, it is possible to have the same network address configured multiple times but completely independent of each other.

To create a system with virtualized routing many changes are needed. This starts with making the link local discovery protocols (arp, rarp, nd, ...) aware of the multiple domains. The ICMP code and all other code that replies or tunnels packets needs to ensure that the new packet is processed in the same domain. A special local tunnel interface is needed to pass traffic between domains and pf may need some modifications as well. Finally the socket layer needs a possibility to attach a socket to a specific routing domain. The easiest way to do this is via a `getsockopt()` call.

Unlike the vimage[4] approach for FreeBSD not a full virtualization is done. The main reason behind this different approach is in my opinion primarily the originator and his background. In FreeBSD, vimage was developed by network researchers the idea is to be able to simulate multiple full featured routers on a single box. Therefore vimage goes further then the OpenBSD implementation by having multiple routing sockets, protosw arrays and independent interface lists. In OpenBSD the development is pushed by networking people with an ISP background. The end result is similar but the userland hooks are different. In OpenBSD, userland will have access to all domains at once through one routing socket so that one routing daemon is able to modify multiple tables at once.

Routing Priorities

With the inclusion of bgpd, ospfd, and ripd the need for userland to easily manage various routing source in the kernel became more apparent. In case of conflicts it is necessary to have a deterministic mediator between the different daemons. E.g. prefixes learned via OSPF should have a higher preference than external BGP routes. Routing suites like xorp and quagga/zebra normally use a separate daemon to merge these various routing sources. This is a single point of failure and unnecessary because the kernel can do this just fine. Similar to commercial routers this can be done by the kernel by adding a priority to each route and giving each routing source one specific priority level. With the introduction of multipath routing it is possible to store a route multiple times in the kernel. Having a cost for each route and sorting the list of equal routes by the cost gets the desired behaviour. Only the routes with the lowest cost are used for routing -- in other words, this is now equal cost multipath routing.

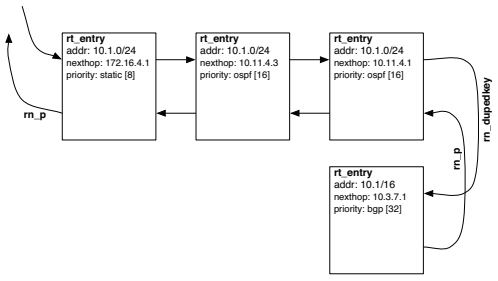


Figure 7: Using rn_dupedkey to implement priorities

Implementing this simple concept on the other hand released some evil dragons in the current routing code. Until now, the multipath code only had to handle inserts after the head element which is no longer true. The initial result were corrupted routing tables, crashes, a lot of head scratching, and even more debugging. The routing code and especially the radix tree implementation itself is complex, obscure, and a very sensitive part of the networking code.

References

[1] TCP/IP Illustrated, Volume 2
by Gary R. Wright, W. Richard Stevens

[2] IPv6 Core Protocols Implementation
by Qing Li, Tatuya Jinmei, Keiichi Shima

[3] Analysis of an Equal-Cost Multi-Path Algorithm, RFC 2992, November 2000

[4] The FreeBSD Network Stack Virtualization
http://www.tel.fer.hr/zec/vimage/
by Marko Zec

Reducing Lock Contention in a Multi-core system

by: Randall Stewart

Why do we lock?

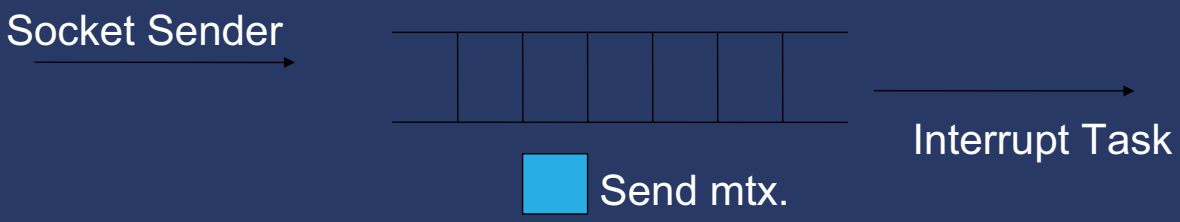
- With the advent of today's multi-core CPU's more and more operating systems are moving to an Symmetric Multi-Processor (SMP) environment!
- Each operating system must thus find ways to ensure sane and coherent operation when multiple CPU's access the same data structures simultaneously.
- One of the most common coherency mechanisms is the mutex.

How do we lock in the FreeBSD kernel?

- Mutex's provide a simple gate that allows one CPU to access a data structure while another waits its turn.
- In user land pthead_mutex's are available for this purpose, but not so in the kernel.
- In the FreeBSD kernel we have the “mtx” structure.
- Like pthread_mutex's these structures, once initialized, can be locked and unlocked for exclusive access to a data structure.

An example used by the SCTP stack in FreeBSD and MAC OS/X.

- In the SCTP implementation we have quite effectively used locking to allow a sender of data (the socket api user) and the transmitter of data (the interface interrupt task) to share a data structure via a mutex.



With a simple scheme

- With this simple scheme we solve the locking problem and our two or more CPU's do not have a problem.
- However we gain one down-side, lock-contention.
- Lock contention is how often one thread holds the lock while the other has it.
- The less lock contention, the more parallel our process will be and thus we will better utilize the multi-core systems we have available.

So how can we measure lock contention?

- FreeBSD comes with a kernel level tool-kit for this very purpose.
- If we build our kernel with the "LOCK_PROFILING" option we can measure our lock contention. (man LOCK_PROFILING)
- Our config file looks like:

....
options LOCK_PROFILING
...

- Build the kernel in the usual way and reboot

Getting a “lock profile” run

- Lock profiling on your new kernel is NOT enabled by default.
- You can turn on/off/examine lock profiling by:

```
 sysctl -a | grep lock | grep prof | grep debug
debug.lock.prof.stats: No locking recorded
debug.lock.prof.collisions: 0
debug.lock.prof.hashsize: 4096
debug.lock.prof.rejected: 0
debug.lock.prof.maxrecords: 4096
debug.lock.prof.records: 0
debug.lock.prof.acquisitions: 0
debug.lock.prof.enable: 0
```

Getting a “lock profile” run

- Change the enable flag to 1.

sysctl -w “debug.lock.prof.enable=1”

- Now run any tests that you want to profile.
- After you are done change the sysctl back to '0'
- Now do a sysctl -a > my_file.txt
- Vi/emacs your file and scan down until you find the symbol debug.lock.stats
- You should see a header/numbers that look like

```
max   total wait_total avg  wait_avg cnt_hold cnt_lock name
```

- And lots of numbers.

Mutex Profiling results

- Max – the max time this point waited in microseconds.
- Total – the total hold time in microseconds.
- Wait_total – the total accumulated wait time.
- Count – the number of times this lock was at this point **
- Avg – The average hold time in microseconds.
- Wait_Avg – the average wait time in microseconds.
- Cnt_hold – The number of times this lock was held when someone else wanted it. **
- Cnt_lock = The number of times someone else held the lock at this point **
- Lock name – the lock name and file and line number.**

An Example

Socket Sender

Count	Count hold	Count lock	Lock Name
12	0	0	sctp_output.c:5140
12240	5571	551	sctp_output.c:6454
12240	11	52	sctp_output.c:11088
59394	503	5631	sctp_output.c:11170
12240	5	4	sctp_output.c:11375

Socket sender
Interrupt transmitter

An Example

Count	Count hold	Count lock	Lock Name
12	0	0	sctp_output.c:5140
12240	5571	551	sctp_output.c:6454
12240	11	52	sctp_output.c:11088
59394	503	5631	sctp_output.c:11170
12240	5	4	sctp_output.c:11375

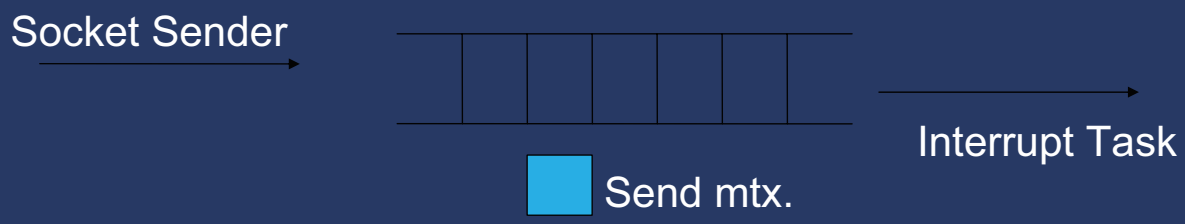
45.5% of the time its held here
4.5% of the time someone wants the lock when held

An Example

Count	Count hold	Count lock	Lock Name
12	0	0	sctp_output.c:5140
12240	5571	551	sctp_output.c:6454
12240	11	52	sctp_output.c:11088
59394	503	5631	sctp_output.c:11170
12240	5	4	sctp_output.c:11375

0.8% of the time its held here
9.4% of the time someone wants the lock when held

So how can we reduce contention and preserve sanity?



Is really

```
struct name {
        struct type *tqh_first; /* first element */
        struct type **tqh_last;        /* addr of last next element */
};
```

Where

```
struct name {
        struct type *tqh_first; /* first element */
        struct type **tqh_last;        /* addr of last next element */
};
```

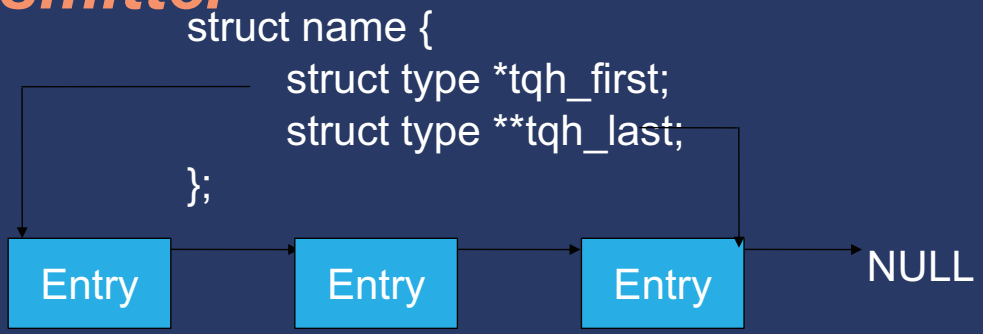
- The socket sender always appends to the tail
- The interrupt transmitter always pulls from the head and may not always pull the whole message off (considering that a msg can be larger than the PMTU).
- So can we reduce locking?

We can observer about the sender

```
struct name {
        struct type *tqh_first; /* first element */
        struct type **tqh_last;        /* addr of last next element */
};
```

- The sender never knows if the transmitter is active and is never sure if the list is empty or has entry's on it (tqh_last points to the head when empty).
- When adding data, we only use tqh_last.
- But without foreknowledge the socket sender MUST always lock the structure.

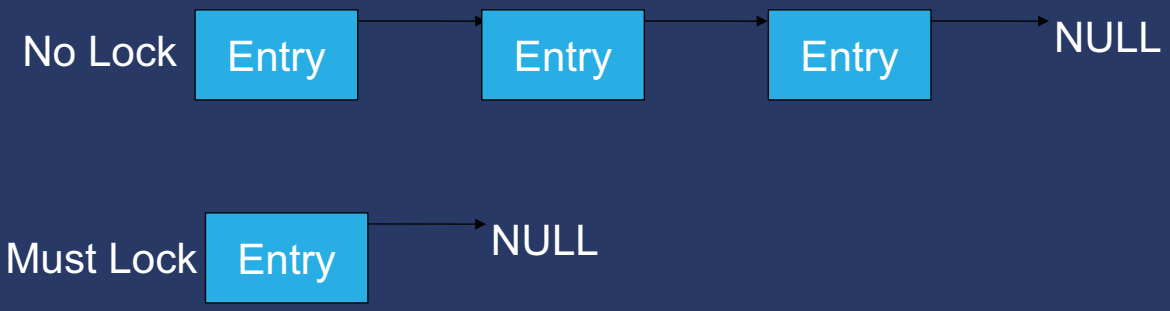
We can observer about the transmitter



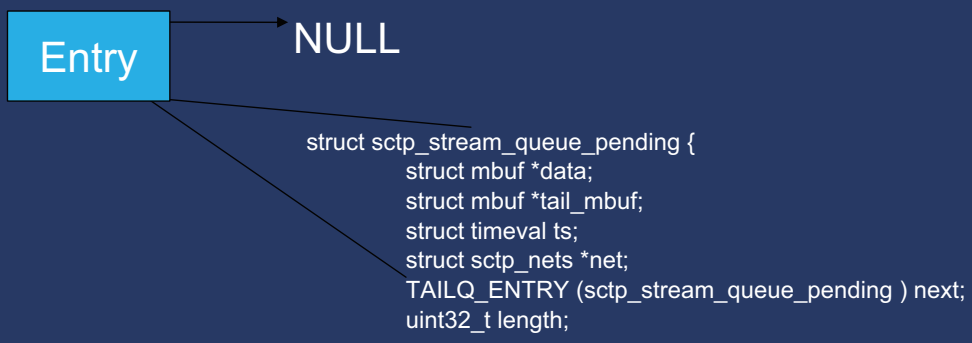
- But what about the transmitter?
- It does know:
  - If it is going to pull the entry off.
  - It can tell if there is already a next entry on the queue.

We can observer about the transmitter when Pulling an entry.

- So from this knowledge could we:
  - Only lock if we will pull the entry from the queue?
  - Don't lock if there is a next item in the queue (since the other thread is inserting there)?



So what about contention on the same entry, when more data is added?



- The data being added by the socket sender is a chain of mbufs.

So what about contention on the same entry, when more is added?

```
struct sctp_stream_queue_pending {
        struct mbuf *data;
        struct mbuf *tail_mbuf;
        struct timeval ts;
        struct sctp_nets *net;
        TAILQ_ENTRY (sctp_stream_queue_pending ) next;
        uint32_t length;
```

- If we always update the size last, after appending, then the transmitter will always see either the correct size, or a reduced size.
- Since we use mcopym() with the size, this limits us so when contending for one being added the most that can happen is we will take less than we could have.
- Note that we use atomic_add_int() to assure a barrier and that the compiler does not give us a surprise.

So what two things is the transmitter doing?

- When it goes to add data, don't get a lock unless the size of the copy will exhaust the mbuf completely. This allows continued addition to a message without the transmitter locking.
- When the transmitter decides to remove an entry it will only lock if the “next” pointer is NULL.

Results after our modification

Count	Count hold	Count lock	Lock Name
288	64	4	sctp_output.c:5141
344	157	16	sctp_output.c:6517
33549	19	232	sctp_output.c:11151
45787	1	0	sctp_output.c:11437

Note that the redesigned algorithms have one less lock.

Socket sender
Interrupt transmitter

The results show

- A huge drop in the percentage that the socket sender contends with the transmitter from 45% to .02%
- Very rarely does the transmitter even get a lock, its still a high percentage of contention but with a drop from 12,252 lock requests to 632.

Conclusion

- When adding a shared resource to a SMP O/S one needs to:
  - Carefully consider your data structures that the various locks protect.
  - Examine the level of lock contention.
  - Try to craft mechanisms that allow one of the threads/cpus to NOT lock when possible.
- No two problems are the same but the base concept presented here can be applied to both kernel and user level code.

Other things that can be done in Kernel land (use with caution)

- When wanting to cache resources for quick re-use per CPU lists can be created.
- One can use the critical_enter/critical_exit call to prevent being scheduled.
- The code would look something like:

```
free_item(entry_t *item) {
  critical_enter();
   cpu = curcpu;
   LIST_INSERT_HEAD(&cache[cpu].list,
                                item, next);
  critical_exit(); }
```

Sleeping Beauty – NetBSD on modern laptops

Jörg Sonnenberger <joerg@NetBSD.org>
Jared D. McNeill <jmcneill@NetBSD.org>

February 3, 2008

Abstract

This paper discusses the NetBSD Power Management Framework (PMF) and related changes to the kernel. The outlined changes allow NetBSD to support essential functions like suspend-to-RAM on most post-Y2K X86 machines. They are also the fundation for intelligent handling of device activity by enabling devices on-demand.

This work is still progressing. Many of the features will be available in the up-coming NetBSD 5.0 release.

1 Introduction

The NetBSD kernel is widely regarded to be one of the cleanest and most portable Operating System kernels available. For various reasons it is also assumed that NetBSD only runs well on older hardware. In the summer of 2006 Charles Hannum, one of the founders of NetBSD, left with a long mail mentioning as important issues the lack of proper power management and suspend-to-RAM support. One year later, Jared D. McNeill posted a plan for attacking this issue based on ideas derived from the Windows Driver Model. This plan would evolve into the new NetBSD Power Management Framework (PMF for short).

Major design goals were:

- The ability to suspend a running system, possibly including X11, and to restore the same state later.
- The ability to disable devices and restore the full state. This includes e.g. volume and current playback position for audio devices etc.
- Automatically sense which devices are not in use and place them into reduced power states.
- Provide a messaging framework for inter-driver communication for event notifications, e.g. hotkeys pressed by the user.

This paper will provide an overview of the core changes to the NetBSD kernel, the goals that have been achieved, and the challenges that have been faced along the way.

The first section will cover the PMF itself as it is the component with the greatest impact on the overall kernel. Afterwards the AMD64 and i386 specific changes to the ACPI infrastructure will be discussed. Last but not least is the special handling needed for video cards needed on many systems.

2 The NetBSD Power Management Framework

2.1 Overview

The NetBSD Power Management Framework (PMF) is a multi-layer device power management framework with inter driver asynchronous messaging support. It was originally inspired by the Windows Driver Model power management framework, but has since evolved into a model better fitting for the NetBSD kernel. The implementation is contained in:

- sys/kern/kern_pmf.c,
- sys/sys/pmf.h, and
- sys/kern/subr_autoconf.c.

2.2 Device interface

Device power management is implemented in layers:

- Device,
- Bus,
- Class (ie network, input, etc).

The basic entry points for a device driver to implement the PMF are pmf_device_register(9) and pmf_device_deregister(9). Both of these functions accept a device_t as their first argument, and in the registration case it also accepts optional driver-specific suspend and resume callbacks. These functions return true on success, and false on failure.

Device bus power management support is inherited from the parent device in the autoconfiguration tree. A device driver that attaches to the pci(4) automatically inherits PCI Power Management support. The PCI bus handlers take care of saving and restoring the common part of the PCI configuration part. They are also responsible for removing power from the device and restoring it.

Device class power management support cannot be derived as easily, so a device driver that requires class-level power management support will call the appropriate pmf_class_<type>_register and deregister functions when registering with the PMF. The currently implemented power management class types are 'network', 'input', and 'display'. Depending on the device class the register function takes additional arguments, i.e. the "struct ifnet" address for a network device.

Using this layered approach, the amount of duplicated code between device drivers is reduced and the per-driver code minimized. One example of where this is the wm(4) network driver. Since the PCI bus layer captures and restores common PCI configuration registers and the network class layer is responsible for stopping and starting the interface, no additional device specific power management code is required. Other device drivers such as bce(4) simply need a single function call in their resume callback (with no suspend callback) to restore the device to a fully operational state.

Due to its integration with the autoconfiguration subsystem, a device_t is required to register with the PMF. This differs from the former powerhook(9)

framework in previous NetBSD releases, which implemented global system suspend/resume by executing callbacks in order of registration with an opaque cookie for an argument. This interface made it impossible to control the order in which devices are suspended or resumed. With the PMF, global system suspend/resume is implemented by traversing the autoconfiguration device tree, ensuring that a device's parent is powered up before it is initialized and that a child device is suspended before its parent bus.

The most basic interfaces from kernel code to control device power are pmf_device_suspend(9) and pmf_device_resume(9). A power management device driver will typically want to suspend and resume the entire autoconfiguration tree or a subtree. In the case of a global power state transition, the power management device driver would use the pmf_system_suspend(9), pmf_system_resume(9), and pmf_system_shutdown(9) APIs. The "suspend" and "shutdown" functions are nearly the same with the exception of two points; a suspend will be aborted directly if any device in the autoconfiguration tree does not implement the PMF, and a shutdown does not invoke the bus power management layer support. Some additional support functions are available for power management drivers; pmf_device_recursive_resume(9), pmf_device_recursive_suspend(9), and pmf_device_resume_subtree(9). The first function can be used to resume a specific devices and all its parents. The other functions suspend or resume a subtree of the autoconfiguration tree.

Suspending drivers on shutdown using pmf_system_shutdown(9) avoids the problem of active devices trashing the system after a reboot e.g. with DMA. A number of drivers did this with ad hoc shutdown hooks before and doing it in the PMF provides it consistantly for all devices. This part the PMF will be extended at some point in the future to provide a separate optional hook for finer control.

2.3 Event interface

The PMF introduces a message passing framework for inter-device driver communication. Events can be directed (targetted to) a specific device, but in the most typical case anonymous events are used. An anonymous event is essentially a broadcast notification to all device drivers registered for that event.

One issue with making laptops "just work" was that there was no mechanism to associate a hotkey event with an appropriate action. Making decisions on how to handle events entirely in a userland script was considered too much of a kludge, so it was decided that an inter-driver event framework was necessary.

Consider brightness control hotkeys on a laptop. Every hardware vendor implements this in a different way, and the device that generates the hotkey event is not necessarily the same device that gives control of the LCD backlight. A hotkey device driver simply uses the pmf_event_inject(9) function to inject a brightness control event (PMFE_DISPLAY_BRIGHTNESS_UP, PMFE_DISPLAY_BRIGHTNESS_DOWN) into the pmf event queue. A worker thread is then woken up, and scans the list of callbacks created using the pmf_event_register(9) and pmf_event_deregister(9) functions. Since this is an anonymous event (device_t argument to pmf_event_inject(9) is NULL), all callbacks registered for the appropriate PMF_DISPLAY_BRIGHTNESS_* event are executed. The hotkey driver does not need to know or care if the console driver, generic ACPI display driver, vendor-specific ACPI device driver, or other power control driver

will handle the event. As long as one of these drivers is present and working, the brightness key press will “just work”.

For events that are better handled in userland, hotkey support was added to sysmon_power(9) and powerd(8). This is typically used for associating a "lock screen" key with xscreensaver, a "display cycle" key with xrandr, "eject" key with a DVD-ROM tray, and so on.

2.4 Vendor and model specific information

Sometimes there is no choice but to only apply a quirk in a device driver on certain hardware platforms. A very simple API was added to the PMF to act as a dictionary for obtaining information about the running system from within device drivers. Platform dependent code is responsible for filling in these tables; on AMD64 and i386 information retrieved from DMI (aka SMBIOS) tables are used.

The sony_acpi(4) driver uses this to apply workarounds in its initialization routines on certain VAIO series notebooks.

3 ACPI improvements

3.1 ACPICA

The Intel ACPI Component Architecture (ACPICA) is an OS-independent reference implementation of the ACPI specification. NetBSD 4.0 shipped with the two year old 20060217 release of the ACPICA, so it was decided that the third party code should be updated to a newer release.

At the time of the ACPICA upgrade the latest version of ACPICA available from Intel was 20061109, but it was known that other operating systems were shipping newer releases. As the APIs are constantly evolving, a significant amount of integration effort would have been been required regardless of the release selected. It was decided to go with the release of ACPICA present in FreeBSD -CURRENT at the time, 20070320.

The majority of changes required for the new APIs were mechanical, related to renamed structures and structure members. Another issue involved a change in the way that tables are accessed; previous releases of ACPICA pre-parsed some tables and stored them in global pointers. This has been changed in some cases to store a copy of the table in a global structure, and in other cases the Operating System must map the table itself using AcpiOsMapMemory.

The operating system dependent (Osd) interfaces were also changed. The following functions had minor signature changes:

- AcpiOsMapMemory,
- AcpiOsWaitSemaphore,
- AcpiOsAcquireLock,
- AcpiOsDeleteLock,
- AcpiOsGetRootPointer,
- AcpiOsGetThreadId.

In addition, new functions such as AcpiOsValidateInterface and AcpiOsValidateAddress were required, and AcpiOsQueueForExecution was renamed to AcpiOsExecute.

In the process of tracing down various interrupt issues, the ACPI initialization sequence was updated. The initialization was split into two phases. The first phase is just long enough to load the MADT. That table is required for interrupt setup, especially when using the IOAPIC. The second phase can therefore directly hook up the ACPI System Configuration Interrupt and finish the initialization sequence. This replaces the lazy interrupt setup code in the AMD64 and i386 code.

This ACPICA update exposed a fundamental design flaw in the NetBSD Embedded Controller driver, requiring it to be rewritten from scratch.

Since this work has completed a new ACPICA web site, `http://www.acpica.org`, has appeared offering a 20080123 release for download.

3.2 Embedded Controller

The Embedded Controller (EC) is one of the central hardware components of ACPI. The ACPI virtual machine is using the EC to access various devices without requiring the attention of the CPU. It is also used by the hardware to notify the ACPI VM about changes in the system configuration using the System Configuration Interrupt (SCI).

The EC has a very simple interface using two one-byte ports. The first port is used to send commands to the EC and read back the current processing status. The second port (data port) is used for operands of the commands. The EC understands the five commands “query”, “read”, “write”, “enable burst” and “disable burst”. Operands like the address of a “read” or “write” or the return value for a “query” are transfered using an internal buffer in the EC. When this buffer is empty and new data can be send, a flag is set and an interrupt is sent. The same happens for reading data from the EC.

This interface forces the driver to communicate asynchronously with the hardware. As this is often undesirable, the burst mode was added. It allows a driver to send commands and data back-to-back with the full attention of the EC. If the EC can’t process a command in a timely manner, it can disable the burst mode itself.

The old EC driver as inherited from FreeBSD tried to deal with this mess by using a mixture of interrupt mode and polling mode. It was very hard to follow the flow of control and add locking without introducing dead locks. For that reason, the EC driver was rewritten from scratch as part of the jmcneill-pm branch.

The first important observation for the new driver was the symmetry between the entry points. The driver has to deal with three request types: reads, writes and SCIs. SCIs are special events raised by the EC, read and write operations are originated in the system. The access to the driver can therefore be serialised by a single mutex shared between the three entry points. For the processing of the SCIs a kernel thread is the simplest solution, but a work queue could have been used as well.

The second observation for the driver design is that most of the complications in the interface are a result of not using the state flow of the EC in the driver. The actions in the driver are much easier to formulate as finite state machine.

As soon as one of the the entry points obtains the driver lock, it writes the address for read or write access and the command to process. Afterwards it just waits for completion. The interrupt handler drives the state machine. If it finds a request for a SCI, it signals the kernel thread to wake up. Depending on the state of the machine, it writes the address or transfers the data byte. When a command is done, it wakes up the blocking originating thread.

One problem of this approach is that it depends on the hardware properly sending interrupts. Many EC implementations don't do that though. Linux and FreeBSD dealt with this by using the burst mode. A simpler alternative is to just poll the hardware after some time using a callback. In other words, if the hardware doesn't send an interrupt, simulate it.

The second problem is that during early boot, suspend and resume neither interrupts nor timeouts are processed. Polling the EC directly is similiar again to how lost interrupts are processed. Experiments have shown that spinning a bit is generally helpful as most EC commands finish in less than 5ms. Polling has to be done with care though. On a Lenovo Thinkpad R52, a busy loop without delay completely kills the EC, it doesn't even provide the emergency powerdown. Similiar issues exist with other vendors. Tests have shown that at least 100ms intervals are needed for pure polling.

The new driver has proven to be robust and much easier to adopt to new vendor bugs.

3.3 Suspend support

Proper support for suspend to RAM is one of the most often requested features in the Open Source world. The Advanced Power Management interface provided this without much complexity in the Operating System. As Microsoft has pushed ACPI for a long time, vendor support for APM started to disappear. For ACPI based suspend to RAM the Operating System is responsible for most of work. The first part of the process is saving all device state and preempting them. This is part was addressed in section 2.2. The second part is saving the CPU state and calling the firmware. NetBSD inherited the S3 support for i386 from FreeBSD. The support was working, but lacking in two importants areas. S3 was not possible on AMD64 and it only worked with a Uniprocessor kernel. With the advent of Intel's Core 2 in consumer notebooks both limitations had to be fixed.

This was addressed in two parts. First, the ACPI wake code was ported to AMD64 and later it was extended to handle Application Processors (APs). The wake code is called by the firmware almost directly after the resume. At this point, the CPU is still running in Real Mode. For switching to Protected Mode part of the address space has to be mapped to the same address in virtual and physical address space. The code inherited from FreeBSD solved this by adding the address of the wakecode to the kernel map, even though the kernel map normally doesn't cover this address range. This was fixed by using a temporary copy of the Page Directory and a special Page Table, both located in low memory. The wakecode enables paging using this temporary copy and switches to the normal version in a second step. For the AMD64 port this was crucial as the switch to Long Mode needs the equivalent of the Page Directory under the 4GB limit. The only major surprise left for the AMD64 port was the trap when a page has the NX bit set and the feature was not enabled already.

The first attempt at multiprocessor support was to migrate all threads from the APs and let them just idle. On resume the bootstrap was repeated as it is done during the normal boot. This worked somewhat as the APs came back to live, but they hit an assertion in the process scheduler pretty soon. This assertion didn't make any sense as it essentially meant that the idle thread was not scheduled.

The second try utilised the already working wakecode. The wakecode was changed to use storage in the CPU specific data instead of global variables. On suspend, the APs follow the same code path as the primary CPU and on resume, they are recovering exactly the state they were in before. After the first try on a Intel Core 2 system, the system crashed with the same assertion as during the first attempt.

Further debugging revealed that both cores disagreed on the state of the idle thread of the second core. This suggested a cache synchronisation problem and it turned out that the modifications of the second core where still in the L1 cache and not written back to main memory, when the suspend occured. The first core does an explicit invalidation before suspend, but it become obvious that the L1 cache of the second core was not affected by this. Adding the necessary flush before halting the second CPU fixed the problem.

At this point, an optimisation for the pmap module was added and the kernel changed to always use large pages to map the code segment, if the hardware supports it. This broke the i386 resume again. Just as the use of the NX bit, large pages had to be enabled earlier.

The wakecode is been improved in non-functional ways. One important change was to not use double return (like longjmp). The code flow was instead reorganised so that the suspend is entered from a leaf function. One side effect of this change is that the amount of state to save and restore has reduced as only caller-save registered are now volatile.

Further work in this area is to merge the MP bootstrap code with the ACPI wakecode. The former is almost a subset of the wakecode now. The change would allow moving the resource allocation and state setup from assembly code on the APs into the high-level code running on the AP, resulting in more simplifications.

4 Video card handling

Of all hardware in a modern PC, the most problematic part for a successful resume is the video card. One reason for this is the great variety of incompatible chips. Another reason is the complete lack of interface descriptions for many graphic chips.

The first approach to this problem is just calling the Power On Self Test (POST) code in the VGA BIOS before switching to protected mode. This has been available for a long time as option and works on a number of systems. The function depends on the firmware restoring the state of the main PCI bridges and the VGA device, which doesn't happen e.g. on Dell machines.

The second approach is a userland program called vbetool. The program either uses VM86 or a real mode software emulator to execute the POST code after the PCI code has restored the generic register set. This fixes the majority of the remaining systems. The biggest problem is that it doesn't allow you to

recover the display early enough to see and debug problems in the other drivers.

The third approach is to implement the necessary functions in chipset-specific drivers. This is actively worked on as part of the DRM code, but it is unlikely to address older, undocumented chips.

As part of the power management work a variation of the second approach was added. A size optimised version of x86emu (as used by XFree86 and vbetool) was added to NetBSD. This code allows doing the POST directly from within the kernel. Using VM86 mode would be an option for i386, but for AMD64. The CPU emulation is complemented by an emulation of the i8254 (the AT timer) to prevent the BIOS from destroying the kernel configuration.

The in-kernel VGA POST is still work-in-progress, but the goal is to completely replace vbetool and the early POST call.

5 Conclusion and future work

The jmcneill-pm branch and the related changes post-merge were a great success. Most laptops are now able to use ACPI based suspend-to-RAM. The number of systems that can't use ACPI for system configuration was greatly reduced as well. Work continues to fix any regressions left and identify remaining problems with ACPI on older hardware. Extending the ACPI S3 support is also part of the plan for NetBSD 5 to ship SMP enabled kernels by default.

The fine grained idle control of devices is still under investigation. The current implementation for audio devices has problems with uaudio(4) due to the architecture of the USB stack. Further extensions are planned though. The cardbus network drivers are currently powering down the card if it is not up. This is desirable for PCI devices as well.

The interface for device power management is focusing on making it easy to add support to an existing driver. It currently doesn't allow drive-specific logic for wake up events or multiple power states. The PCI support is currently limited to D0 and D3hot. Future work will exploit interface to support fast resume states like D1 and physically removing the power based on bridge logic (D3cold).

The event interface is used for handling many of the modern special buttons in the kernel. Future work will extend this to interact with userland components like Gnome.

The video BIOS access based on x86emu will be used for vesafb as well. Long term goal is making vesafb work on any platform with PCI devices.

To summarize the changes it is clear that NetBSD has caught up to Linux in many critical areas. The biggest remaining tasks are converting the various drivers in the kernel to support suspend/resume and to investigate the available mechanisms on other hardware architectures like ARM.