AsiaBSDCon 2013
Proceedings

March 14-17, 2013

Tokyo, Japan

Published in Japan, March 2013

INDEX

FreeNAS plugins (everything you ever wanted to know)

John Hixson

john@ixsystems.com
iXsystems, Inc

Abstract

When FreeNAS entered the 8.x series, many people were not happy that functionality that previously existed was no longer included. Such functionality was mainly multimedia focused and targeted at the home user. Services such as bittorrent, DLNA and iTunes media services are the primary examples. Beginning with FreeNAS 8.2.0, a plugin architecture was introduced. This architecture allows FreeNAS systems to be extended in any way that the user sees fit. The purpose of this paper is to describe the technical details of how the architecture works so that plugin authors have the knowledge to create new plugins. As a working example, the transmission bittorrent client plugin will be reviewed.

1 Introduction

FreeNAS is a very powerful open source operating system based on FreeBSD. However, once you get beyond all the great capabilities it offers, your options for extending it become limited. Your choices are using FreeBSD's built in package management system, or modifying the source code and building your own image.

Packages can be installed using FreeBSD's package management system, but care must be taken. You must be aware of what paths and files the package management system uses as well as the package itself. You have to very carefully select whats are used and where all the files go because once the system is rebooted, several key files can be overwritten or disappear.

FreeNAS creates memory disks for /var and /etc at boot time and copies the contents of /conf/base/var and /conf/base/etc to these file systems FreeBSD's package tools and ports work with files from /var/db/pkg and /var/db/ports. Also, the root file system is mounted read only. What this means is that when attempting to install a package, most files won't be allowed to be written to the system and the record in /var/db will be erased on boot. This can of course all be circumvented, but the point is that it's an involved process to get working right.

The major problem with using package management is that once you do an upgrade, everything you installed will get wiped out. Upgrades to FreeNAS only save the configuration and the volumes that are created, everything else is wiped clean.

The other option is to hack the build system to include the packages you want. This is certainly an option. The caveat with this is that you must have a FreeBSD system with development tools, an understanding of the build system and how it works, knowledge of what files to edit, and so on. This simply isn't feasible for most people. Most FreeNAS users simply aren't technical enough for this.

To address these problems and more, FreeNAS has introduced the plugin system. The plugin system is modular, self contained and allows everyday users to install programs that fit their needs onto FreeNAS from an easy to use interface. This also allows users to use their FreeNAS system as more than just a file server.

2 The plugins jail

In order to install plugins on FreeNAS, a plugins jail must first be installed, configured and running. A FreeNAS plugins jail is a standard FreeBSD jail packaged as a PBI and preconfigured with several necessary packages that allow the stock plugins to work. The plugins jail can be found in the plugins directory under the FreeNAS release directory that is being used.

To install a plugins jail, you must first upload it. This can be done from the web interface under services->plugins. You must specify where the jail will be stored temporarily when it's uploaded. The next step requires you to configure a jail path, a jail name, IP address and netmask, and a plugins archive path. The plugin jail configuration is stored in the database in the table services_pluginsjail. The following describes each column and what it is used for:

- jail_path – The file system path where the jail resides
- plugins_path – The file system path where the plugins reside
- jail_mac – MAC address for the jail interface (if configured)
- jail_ipv4address – The IPv4 address for the jail
- jail_ipv4netmask – The IPv4 netmask for the jail
- jail_name – The name of this plugins jail

Currently, Only a single IPV4 address is supported. In the future, multiple Ipv4 and Ipv6 addresses will be supported, as well as multiple plugin jails.

When the plugin jail is uploaded and configured, pbi_add is run and the jail is extracted to the jail_path + jail_name. Once this is done, the plugins jail is ready to

be run. When you turn the service on, /etc/rc.d/ix-jail is invoked. This script generates the proper /etc/rc.conf lines to configure the jail with vnet and allows /etc/rc.d/jail to start the jail. Once the jail is up and running, plugins are ready to be installed.

3 Installing a plugin

FreeNAS plugins use the PC-BSD PBI9 format. plugins are installed using the web interface. Installing a plugin is very easy, navigate to Services->plugins->Install Plugin. When a plugin is installed, the PBI information is stored in the database in the table plugins_plugins which has the following columns:

- plugin_version – plugin version number
- plugin_enabled – enabled/disabled status
- plugin_ip – fastcgi server IP address
- plugin_port – fastcgi server port
- plugin_arch – i386 or amd64
- plugin_api_version – RPC API version
- plugin_name – name of the plugin
- plugin_pbi_name – PBI file name as uploaded
- plugin_path – where in the file system the plugin is installed

Once the PBI information is saved, an oauth secret and key are generated record in the services_rpctoken table. This table contains the columns:

- secret – the oauth secret
- key – the oauth key

Once the PBI and Oauth information is recorded in the database, the following steps occur:

1. The PBI is installed into the plugins jail in /usr/pbi/${plugin}-${arch}/
2. The oauth key and secret are written into /usr/pbi/${plugin}-${arch}/.oauth
3. The plugin information is written into plugins.conf which is included by nginx.conf. This tells nginx that all URL's that specify the plugin path are to be passed to the plugins fastcgi server.
4. The plugins control script is started in the jail (/usr/pbi/${plugin}-${arch}/control start). This starts the plugin fastcgi server on the IP/port combination recorded in the database.
5. The web interface will refresh. The navtree makes a request to the plugin's _s/treemenu and treemenu-icon methods. The treemenu method returns a description of how to display the plugin information in the navtree. The treemenu-icon method passes the icon for the plugin to the navtree. Once these methods are called, the plugin appears in the navtree menu under Services->plugins->${plugin} with the plugin icon. The plugin will also appear under the Services->plugins menu in the main interface.

4 How they work

When the plugin icon is clicked, django matches the plugin URL and sends the request to the plugin fastcgi server. Requesting a plugin method is of the form:

- base_url + “/plugins/” + ${plugin} + “/” + ${method}

The methods that are available are:

- edit – edit the plugin configuration
- treemenu-icon – icon to be displayed in the navtree
- _s/treemenu – what/how to display in the navtree
- _s/start – start the plugin
- _s/stop – stop the plugin
- _s/status – status of the plugin

Plugins have access to the base system via RPC calls. All RPC requests are signed with the oauth credentials given to the plugin at install time. The following RPC methods are available:

- api.version() - get the plugin API version
- plugins.plugins.get() - get a listing of installed plugins
- plugins.jail_info() - get information about the plugins jail
- plugins.is_authenticated() - test if the plugin is currently authenticated
- fs.mountpoints.get() - get a listing of available files systems
- fs.mounted.get() - get a list of mounted file systems
- fs.mount() - mount a file system into the jail
- fs.umount() - unmount a jailed file system
- fs.directory.get() - get a directory listing
- fs.file.get() - get a file
- os.arch() - get OS architecture
- api.test() - verify RPC calls are working

When an RPC request to the base system takes place, the following thins happen:

1. An RPC request is built of the form: base_url + “/plugins/json-rpc/v1”
2. The RPC request is signed with the oauth credentials
3. The RPC request is sent with the requested method
4. The method is invoked if the oauth credentials are correct and the method exists. The results are then returned back to the plugin

The fastcgi server accepts the plugin request, then dispatches accordingly. This allows anything that anything that can communicate the fastcgi protocol to be a plugin, or even to manage plugins. Because of this flexibility, plugins can be developed using any language

or framework one wishes to use. All that is required for a FreeNAS plugin to work is that it implement the described methods and be packaged using the PBI9 format.

5 Making a plugin

Currently, making a plugin for FreeNAS is somewhat cumbersome. This process is expected to be streamlined in coming releases. While there are several methods to create a plugin, the one described was used to develop the 3 reference plugins included on Sourceforge.

Documentation for creating PBI files using the PBI9 format already exists, so only the FreeNAS specific portions will be covered. Creating a PBI for FreeNAS requires FreeBSD 8.x, PC-BSD 8.x, or FreeNAS 8.2.0 or higher. In all cases, pbi-manager and the ports collection must be installed. The basic procedure for creating a plugin is this:

1. Create plugin directory: myplugin
2. Create resource and scripts directories under this directory: myplugin/resources and myplugin/scripts
3. Create a PBI configuration file: myplugin/pbi.conf
4. Edit the pbi.conf file for your particular plugin Documentation for how to do this can be found at wiki.pcbsd.org “PBI Module Builder Guide”. It's relatively straight forward.
5. If there are any pre/post script needs, create the necessary scripts in the scripts directory as specified in the PBI module builder guide.
6. Invoke pbi_makeport to create the PBI

At this point, a PBI will have been created. Upload the PBI as previously described and it will be installed into the plugins jail. It will not be functional from within the web interface, but it is ready to be worked on from within the jail. This process can be repeated as the plugin is refined and developed.

Next, a control script must be created. The name of the script must be “control” and it must be located in the plugin directory (/usr/pbi/${plugin}-${arch}/. The control script takes 3 arguments, an action verb, an IP address, and a port. The purpose of the script is to start a fastcgi server on the specified IP address and port. The verbs that must be implemented are start, stop and status. The start verb starts the fastcgi server on the IP/port combination. The stop verb stops the server. The status verb exits with 0 if the server is running otherwise it exits with 1. This script is called from the main system when the plugin is enabled or disabled.

Once the control script is completed, the interface portion of the plugin can be worked on. The job of the interface is to export the methods needed by FreeNAS to integrate with the web interface as described in section 4. The start and stop methods must provide a means by which to start and stop the binary the plugin is in control of. This also includes any modifications to /etc/rc.conf if necessary. The treemenu method simply dumps out JSON. The treemenu-icon outputs the plugin icon. The workhorse of a FreeNAS plugin is the edit method. This is the method that presents the interface for configuring the plugin This generally entails saving and restoring state and generating and modifying configuration files.

6 An example - Transmission

When FreeNAS released 8.2.0, three reference plugins were also released. They were provided for two reasons: to provide the missing functionality that previously existed in FreeNAS 7.x, and to document and demonstrate how future plugins could be made.

One of the available plugins is Transmission. Transmission is a very popular bittorrent client. It's implementation is pretty simple and straight forward so it will be used for the example. Since Transmission is built into the build system, the build system configuration will be covered as well. Reviewing the build system process for making a plugin is recommended anyhow for plugin authors so they have a better understanding of how everything works. Here is an overview of the directory layout and key files for the transmission plugin:

${freenas}/nanobsd/plugins/

This is top level directory for all FreeNAS plugins. All plugin files are located in this directory. The build system will be aware of a plugin once it is placed in this directory. For Transmission, the following file is created:

${pluginroot}/transmission

This is the nanobsd configuration file for Transmission. It sets up the nanobsd environment for the Transmission build and provides function(s) for doing so. Since nanobsd is being used for the plugin build, a bit of trickery is done here. All of the nanobsd functions are overridden with stub calls except the last_orders() function. This is the function that makes the actual call to pbi_makeport and does the plugin build.

${pluginroot}/transmission_pbi/

This is the Transmission PBI directory. All plugins must have a PBI directory. Within this directory, two subdirectories must exist: scripts/ and resources/. A pbi.conf file must also exist.

${transmission_pbi}/pbi.conf

This file tells pbi-manager how to build Transmission. It contains information about the plugin such as the port(s) to be built, the icon(s) to be used, the make options for the binaries, etc.

${transmission_pbi}/resources/

This directory contains the bulk of the plugin interface. It

can be structured however the plugin author chooses. Since Transmission uses django, the django application resides in the directory along with an assortment of other scripts and programs.

${transmission_pbi}/resources/control.py

This is the transmission fastcgi server control program. As discussed previously, this program has three responsibilities: to start the fastcgi server, stop the fastcgi server and report the status of the fastcgi server. The start and stop methods also start and stop the django web server application. The django application exports all the required hooks that FreeNAS requires to interface with the plugin. A wrapper script that calls control.py is also in this directory. This “control” wrapper is the only mandatory file that needs to be known by the base system.

${transmission_pbi}/transmission

This is the RC script for transmission that controls the daemon. It's just like any other RC script that FreeNAS uses.

${transmission_pbi}/resources/tweak-rcconf

The job of this script is to modify /etc/rc.conf to enable or disable transmission.

${transmission_pbi}/scripts/

This directory contains hooks for different stages of the PBI build process and installation process. The possible scripts are pre-install.sh, post-install.sh, pre-portmake.sh, post-portmake.sh and pre-remove.sh. Pre-install.sh allows you to do customizations to the system prior to the plugin being installed, such as adding users and groups. Post-install.sh is run immediately after the plugin is installed. Some typical post install operations are database initialization and migrations. Pre-portmake.sh and post-portmake.sh allow you to do operations before and after port compile. Pre-remove.sh is run prior to plugin removal. Operations typically done by pre-remove.sh are user and group removal.

The other files in ${transmission_pbi}/resources are default.png, freenas and transmissionUI. Default.png is the default icon for the PBI. Freenas is a file that contains the plugin api version. TransmissionUI is the django application.

${transmission_ui}/freenas/

This is the django application. Under this directory are the typical django model, form, view and url files. In the urls.py file, the exported methods that FreeNAS requires are very visible and demonstrate what needs to be made available for a FreeNAS plugin to be functional.

To build the transmission plugin, run the command: ${freenas}/build/do_build.sh -t plugins/transmission. This will first create a ${freenas}/sbin directory and install pbi-manager into it. When pbi_makeport is invoked, it will compile a FreeBSD 8.x world and install it into a temporary directory which will later be tarred up and saved for future compilations. Once a world directory is ready, the ports that are needed to compile the plugin get compiled and installed. Any provided scripts get ran and then the PBI is made and placed in ${freenas}/${plugin}/${arch}/${plugin}.pbi. The plugin build is complete at this point.

To test and verify transmission works, upload the Transmission plugin through the FreeNAS web interface as previously described. Navigate to Services->plugins->Transmission from the navtree. Click on it and there should be an edit screen. There are default values filled in already but these can be modified and saved. Save the configuration and go to Services->plugins->Transmission from the main interface and turn the slider to on. At this point you can grab any torrent file, place it in the directory specified in the watch directory and watch it get downloaded in the download directory. Success!

8 Conclusion

FreeNAS plugins allow FreeNAS to be extended in ways anyone sees fit. They are very powerful in their flexibility and allow plugin authors to make a FreeNAS system into everything from a multimedia server to a print server. The purpose of this paper is to explain the plugin architecture so that more developers and people knowledgeable enough can make more plugins Happy hacking!

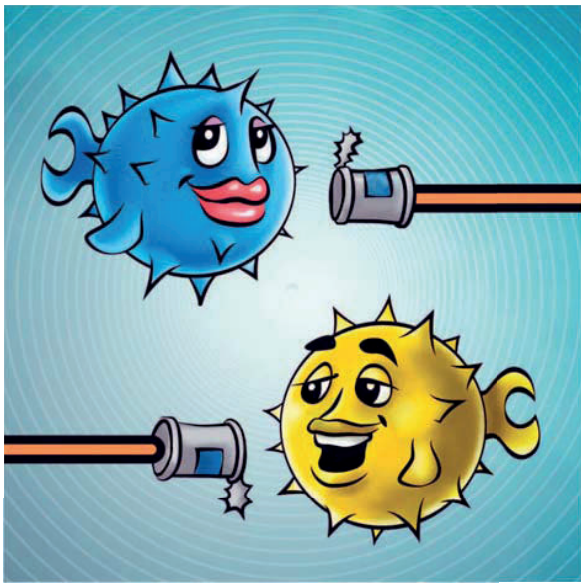
*Open*IKED

Reyk Floeter (reyk@openbsd.org)

February 2013

Abstract

This paper introduces the OpenIKED project[14], the latest portable subproject of OpenBSD[13]. OpenIKED is a FREE implementation of the most advanced Internet security Internet Key Exchange version 2 (IKEv2) Virtual Private Network (VPN) protocol using the strongest security, authentication and encryption techniques. The project was born in need of a modern Internet Protocol Security (IPsec) implementation for OpenBSD, but also for interoperability with the integrated IKEv2 client since Windows 7 and to provide a compliant solution for the US Government IPv6 (USGv6) standard. The project is still under active development; it was started by Reyk Floeter as iked for OpenBSD in 2010 but ported to other platforms including Linux, FreeBSD and NetBSD in late 2012 using the OpenIKED project name.



1 Introduction

This paper provides a brief description of the ***Open*IKED** project, including technical background, history and my personal motivation of creating it. It is inteded to provide some complementary information for my talk at AsiaBSDCon 2013 .

The project aims to provide a free implementation of the Internet Key Exchange (IKEv2) protocol which performs mutual authentication and which establishes and maintains IPsec VPN security policies and Security Associations (SAs) between peers. The IKEv2 protocol is de ned in Request for Comments (RFC) 5996, which combines and updates the previous standards: Internet Security Association and Key Management Protocol (ISAKMP)/Oakley (RFC 2408[8]), Internet Key Exchange version 1 (IKE) (RFC 2409[3]), and the Internet DOI (RFC 2407[17]). ***Open*IKED** only supports the IKEv2 protocol; support for ISAKMP/Oakley and IKE is provided by OpenBSD s isakmpd(8) or other implementations on non-OpenBSD platforms.

It is intended to be a lean, clean, secure, better con gurable and interoperable implementation that focusses on supporting the main standards and most important features of IKEv2. I primarily wrote ***Open*IKED** with signi cant contributions from Mike Belopuhov and various contributing OpenBSD hackers.

***Open*IKED** is developed as part of the OpenBSD Project. The software is freely usable and re-usable by everyone under an Internet Systems Consortium (ISC) license. The OpenBSD project sells CDs, T-Shirts and Posters. Sales of these items help to fund development.

2 Background & History

2.1 Why another VPN protocol?

There are a number of free implementations of VPN protocols available that can provide a su cient security for private network connections. But these protocols and implementations have di erent use cases, limitations, bene ts, and drawbacks. Most of them are specialized in one aspect but cannot provide a solution in another area.

An overview of VPN protocols

SSL-VPN is using the Secure Sockets Layer (SSL)/Transport Layer Security (TLS) protocols to securely tunnel network traffic on the "application layer". It is especially designed for "road-warriors" and is typically running on top of HyperText Transfer Protocol (HTTP) to pass web proxies in restricted corporate networks or public hotspots. There is no "SSL-VPN" standard, there are many different vendor-specific implementations, and it is suffering from some significant protocol overhead due to the SSL and HTTP encapsulation. SSL-VPN is typically not sufficient for high-performance VPN connections and its security is questionable due to its proprietary and vendor-specific nature.

OpenVPN is a well-known, open source SSL-VPN, that does not use an HTTP layer and can run on either TCP or UDP. It became very popular because it was easier to use with clients running various operating systems and because of its user-friendly Graphical User Interface (GUI) for most of these systems. OpenVPN made open source VPN useable for a huge user base, but it also had an negative impact on use and development of IPsec. And this is still the case: whenever someone asks deployment questions about (IPsec) VPN on the OpenBSD mailing lists, you can almost expect that someone will refer to OpenVPN instead. OpenVPN still suffers from some of the problems of SSL-VPN, it is not standards-based and all the portable versions are still based on a single implementation that is licensed under the terms of the GNU Public License version 2 (GPLv2). The "copy-left" limits the use in BSD environments and traditional BSD-based products. For the sake of IPsec, we're going to ignore OpenVPN here.

IPsec does a better job for performance, stronger security, and is very well reviewed by the security community. It is completely open and based on some RFC standards in the Internet community which enables interoperability between many different implementations. It is a real protocol stack and not just a strategy or "buzzword" like SSL-VPN. The biggest problems of IPsec are the limitations of its older IKE protocol and some obscurities and incompatible vendor-specific extensions in existing implementations.

VPN protocols in OpenBSD

OpenBSD already supports some different VPNs in the default installation of its "base" system, and there are many more options in its ports collection of 3rd party open source software.

npppd is a server-side Point-to-Point Protocol (PPP) implementation. The development was started as an internal project by the Japanese company Internet Initiative Japan Inc. (IIJ) and later continued in OpenBSD's source tree since January 2010. It was not enabled in the default OpenBSD build until recently and it will be included in the upcoming OpenBSD 5.3 release for the first time, that is expected around May 2013. It provides support for a number of different protocols that allow tunneling and private networking, including Layer 2 Tunneling Protocol (L2TP), Point-to-Point Tunneling Protocol (PPTP), and PPP over Ethernet (PPPoE).

The PPP protocols are generally used for Remote Access Service (RAS) solutions by Internet service providers with dial-in and mobile access offerings. The protocol stack provides a very flexible support for Authentication, Authorization and Accounting (AAA) that is critical in such environments. The protocols also provide great interoperability with a number of different platforms, including Microsoft Windows, Apple OS X, iOS, Google's Android and all free operating systems.

The PPTP and L2TP can be extended to provide additional VPN capabilities such as encryption and authentication of the tunnelled data. Nevertheless, PPTP has some serious security vulnerabilities and weaknesses that have been found in many security analyses. While it might still be a choice to connect mobile devices that do not support any other security protocols, it cannot be considered as a sufficiently secure VPN protocol.

The L2TP protocol is typically used as L2TP/IPsec to encapsulate the link layer 2 tunneling of the protocol in a "secure channel" that is provided by IPsec and negotiated by IKE. The major problem is the complexity of its protocol stack and the overhead that is caused by the encapsulation of different protocol layers.

isakmpd is a BSD-licensed implementation of the IKE protocol, the predecessor of IKEv2. It was written in 1998 by Niklas Hallqvist and Niels Provos for OpenBSD; an effort that was sponsored by Ericsson Radio Systems. It is widely used in the OpenBSD community and one of the major IKE implementations that was also ported to many other platforms and is the foundation of many proprietary implementations. ISAKMP/Oakley provides a protocol layer that was designed to allow multiple key exchange protocols, but IKE was the only protocol that ever implemented and supported by the daemon. The basic protocol is based on the standards RFC 2407, 2408, 2409 and many additional extensions.

The drawbacks of `isakmpd` are the limitations of the old IKE protocol and the complex configuration of the daemon itself. It uses an `.ini`-style configuration file and an additional KeyNote policy file that have to include a number of irritating and non-default

options. Configuring certificate-based IKE authentication was so difficult, that most users and tutorials simply used the lesser-secure pre-shared key authentication that was easier to configure with `isakmpd` and provided better interoperability with other IKE implementations.

In 2005, the `ipsecctl` tool was introduced to simplify the configuration of `isakmpd` in OpenBSD. `ipsecctl` is providing a single alternative configuration file, `/etc/ipsec.conf`, with a human-readable and modern grammar. Additionally, the tool is smart about some configuration defaults and dependencies. It parses its own configuration file and generates a more complex `.ini`-style configuration for `isakmpd` that is feeded into the running daemon through its First In - First Out (FIFO)-socket. It added another layer to the implementation but greatly reduced the complexity of deploying IPsec with `isakmpd` and significantly reduced the number of users in OpenBSD. However, it could not fix any of the limitations of the IKE protocol.

2.2 Internet Key Exchange version 2

IKEv2 is an important VPN protocol because it was designed for the strongest security requirements, the basic protocol is based on a single RFC 5996 standard[5], and it attempts to eliminate some of the worst limitations and misconceptions of IKE. It also aims to reduce the vendor-specific extensions by including them in a standardized way. Its importance in the "real world" is also increasing since Microsoft included it in its Windows 7 operating system, many other vendors started to migrate to IKEv2, and it became mandatory for compliance by standards like USGv6.

Changes in IKEV2

The IKEv2 protocol got improvements based on the experiences with the previous protocol version. It actually became a single protocol in contrast to IKE that was based on an protocol stack with the additional ISAKMP and DOI layers that have been removed in IKEv2. The format of encrypted messages has been modified to match Encapsulating Security Payload (ESP) and the initial handshake has been reduced to a single 4-way handshake that is replacing the previous "main" and "aggressive" modes. Support for road warriors has been greatly improved by adding more flexibility and reorganizing the protocol; road warriors with dynamic IP addresses can also use preshared keys now. Dynamic configuration of the clients is part of the standard using an integrated configuration payload that is inspired by the former IKE-CFG extension.

3 Design & Implementation

The implementation of ***Open*IKED** was originally started to get a future-proof IPsec keying daemon for OpenBSD. Only the IKEv2 protocol is implemented to get all the benefits of the improved version and to avoid the additional complexity of its predecessor IKE.

The daemon is using a layout that is used by OpenBSD's modern networking daemons. Additionally, further improvements of the layout and the related `privsep` and `imsg` frameworks are regularly merged between these daemons. The existing `isakmpd` still exists in OpenBSD and can be used for most legacy IPsec configurations.

3.1 Project Goals

The following statements have been picked to describe the project goals of ***Open*IKED** .

Lean: Provide a small and monolithic architecture that supports the main standards and most important features of IKEv2. Monolithic means that we do not even try to put lots of features in lots of dynamic libraries.

Clean: Write readable and clean code following strict coding style(9)[12] guidelines.

Secure: Implement secure code with strict validity checking, bounded buffer operations, and privilege separation to mitigate the security risks of possible bugs. Use strong cryptography with sane but secure defaults.

Interoperable: Provide good interoperability with other IKEv2 implementations, support non-standard extensions if it is required to interoperate with other major implementations.

Configurable: Make the configuration easy and nice with sane defaults, minimalistic configuration files and good documentation in the manual pages. Avoid the headaches of past and other IKE implementations.

Strong Crypto

***Open*IKED** supports strong crypto using modern cryptographic ciphers and algorithms that provide state-of-the-art security, performance, and possibly optimization for modern hardware. The implementation supports modern ciphers for IKESA (IKEv2 messages) and CHILDSAs (IPsec messages, e.g. ESP) including authentication and pseudorandom function with the SHA2 family and additional AES modes like AES-CTR or AES-GCM. The Diffie-Hellman key agreement

protocol has been extended with additional modes includes support for latest elliptic curve groups.

AES-GCM combines the authentication and encryption steps in the same AES block operation and allows to leave out any expensive HMAC calculation. Mike Belopuhov added support for AES-GCM and AES-GMAC (the authentication-only version) to ***Open*IKED** and the OpenBSD kernel and the ability to accelerate it on modern CPUs using Intel's AES New Instructions (AES-NI). This enables a significant performance improvement compared to traditional software-based AES-CBC-128 + HMAC-SHA2-256.

3.2 iked(8)

The `iked` program is the ***Open*IKED** daemon itself, accompanied by the `ikectl` control utility. It includes the IKEv2 implementation based on RFC 5996[5]. The implementation was created to get an IKEv2 IPsec daemon for OpenBSD based on the modern privilege separation model and the `imsg` framework, a well-defined configuration grammar for the `/etc/ipsec.conf` file and many other improvements over `isakmpd` like support for stateful configuration reloads, the ability to control the running daemon with the `ikectl` utility instead of a FIFO socket, better and more scalable support for gateway to gateway and especially road warrior scenarios, improved X.509 Certificate Authority (CA) usability and proper use of the OpenSSL API functions instead of custom crypto code.

Example Configuration

The grammar is based on `/etc/ipsec.conf` of `ipsecctl`, which loads its configuration and translates it into the `.ini`-style grammar of `isakmpd`. But `iked` is able to load and understand the grammar of the `/etc/iked.conf` configuration file directly without the need for an additional tool like `ipsecctl`. `iked`'s built-in support resolves many problems that appeared with the `ipsecctl` approach and allows features like stateful config reload.

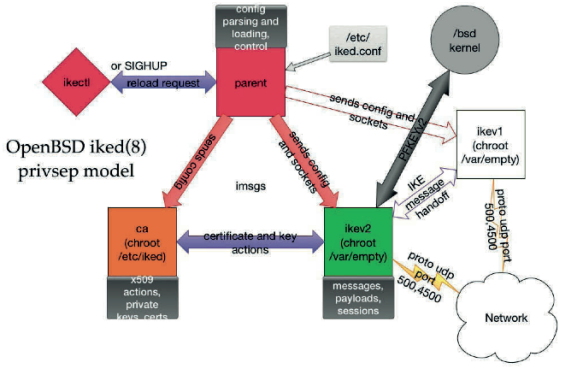
This is a"complex" `/etc/iked.conf` configuration file example for `iked`:

```
user "user1" "password123"
user "user2" "password456"

ikev2 "win7" passive esp \
        from 10.1.0.0/24 to 10.2.0.0/24 \
        local any peer any \
        eap "mschap-v2" \
        config address 10.2.0.1 \
        config name-server 10.1.0.2 \
        tag "$name-$id"

ikev2 esp \
        from 10.3.0.0/24 to 10.1.0.0/24 \
        from 10.5.0.0/24 to 10.1.0.0/24 \
        from 10.5.0.0/24 to 172.16.1.0/24 \
        local 192.168.1.1 peer 192.168.2.1 \
        psk "mekmitasdigoat"
```

Privilege Separation Model

The privilege separation model[18] was first defined for OpenSSH to restrict the effects of attacks and programming errors. A bug in the unprivileged child process does not compromise the security of the privileged part and the operating system itself. It is a similar approach to the modern "sandboxing" techniques but without the need for any custom or non-standard kernel extensions.

***Open*IKED** uses an emerged privsep model with the `imsg` message passing framework that was first implemented for OpenBGPD, copied between multiple daemons, and later imported into OpenBSD's `libutil`.



parent The parent process runs with full privileges to execute privileged operations for the children. It opens and binds to privileged sockets (e.g. UDP port 500), opens the PF_KEY Key Management API, Version 2 (PFKEYv2) socket, and loads the configuration file before sending the required information and resources to the unprivileged children.

ca The semi-privileged CA process handles everything related to certificates and private keys. The process runs in a chroot-environment of the `/etc/iked` directory (or ***Open*IKED** 's default configuration directory of the portable version) and accepts requests from the other processes. The idea behind CA is to isolate the private keys and any other confidential information in a dedicated process. The other processes can request CA lists and send signing or verification

requests of payload that is exchanged via `imsg` messages.

`iked`'s CA process has recently been adopted by OpenSMTPD[15], making it an SMTP implementation that strictly separates private keys from connection handling or email processing.

ikev2 The unprivileged IKEv2 process is the main actor in `iked`. It listens to requests from the network on User Datagram Protocol (UDP) ports 500 and 4500, parses and handles messages, handles IKEv2 sessions and PFKEYv2 communication with the kernel. It can forward IKE messages, version 1, to the IKEv1 process.

ikev1 The unprivileged IKEv1 process is currently an empty stub that does not implement the IKE protocol. It can accept messages, check the version, and forward IKEv2 messages to the IKEv2 process. The design intends to allow operation of both protocol versions on the same host, IP addresses and ports. In the future, the process could either use an implementation of the first protocol version or forward IKEv1 messages to `isakmpd` over an internal communication socket that could be added to both daemons.

ikectl The `ikectl` control utility communicates with the `iked` parent process by sending `imsg` messages over an UNIX socket attached to `/var/run/iked.sock`. It is used for status, reset and configuration reload commands.

3.3 ikectl(8) CA

In addition to status and reset commands, `ikectl` includes an isolated tool to simplify maintenance of the X.509 PKI and to set up a simple CA for `iked` and its peers. It is not intended to be a fully-featured CA toolset, but a set of commands that can create and maintain keys and certificates that are sufficient for medium or small installations of `iked`.

Example configuration of a local CA with two peers:

```
$ ikectl ca test create
$ ikectl ca test install
$ ikectl ca test cert 10.1.1.1 create
$ ikectl ca test cert 10.1.1.1 install
$ ikectl ca test cert 10.1.1.2 create
$ ikectl ca test cert 10.1.1.2 export
```

4 Portable version

Portability of networking software to different operating systems is a complicated task. Every system comes with differences in system APIs, headers, libraries, linker options and file locations. Even standards like POSIX only provide some limited portability if the target system follows that standard and is referring to the same revision. Additionally, some system features like cryptography and networking do not provide a fully-standardized API. The most common approach is to use automatic build tools like GNU automake and autoconf and many OS-specific `#ifdef`s in portable code.

4.1 OpenBSD's Portability Approach

There is a very important design decision in OpenBSD: *OpenBSD software is specifically written for OpenBSD*. It is not intended to have compatibility glue in the source code, neither OS-specific `#ifdef`s, abstraction layers or complex make files. Only external projects that get imported into the base system might use GNU autoconf tools, but software that was written for OpenBSD keeps on using the system's variant of BSD make. This makes the source code very clean and helps readability, maintenance, auditing and security.

OpenBSD distinguishes between "core" and "portable" versions of its subprojects. The main development usually happens in OpenBSD's CVS repository of the base system where the core version is specifically written for OpenBSD. The "portable" version adds a set of patches, a compatibility library, and portable build infrastructure using the GNU autoconf tools and is developed and maintained in different places outside of OpenBSD's CVS repository.

This approach was defined with the development of OpenSSH-portable and is described in Damien Miller's "Secure Portability"[10] paper. All the other portable OpenBSD projects, including OpenNTPD, OpenBGPD, OpenSMTPD and ***Open*IKED** are sharing the same principles and some autoconf and compatibility code that was created by the OpenSSH-portable team.

4.2 The Portable *Open*IKED Project

The project website is available at www.openiked.org[14]. In difference to the core version that is located in OpenBSD's CVS repository, the source code of the portable version is currently hosted at GitHub. Any changes on the portable code are pushed to GitHub from a private Git repository.

The source tree of OpenIKED contains the following directories:

- `openiked/`: Build scripts for automake/autoconf and README files.
- `openiked/ikectl/`: The control and status utility for iked.

- `openiked/iked/`: The IKEv2 daemon itself and some files that are shared with ikectl.

- `openiked/openbsd-compat/`: Portability glue and API functions for non-OpenBSD platforms.

Any changes in the core version are regularly merged into the portable version using a combination of CVS and Git. It is possible to compare the differences between these versions, by cloning the portable code from GitHub:

```
$ git clone git://github.com/reyk/openiked.git
```

And comparing the contents of the iked and ikectl directories in the git repository with the original sources in OpenBSD's (Anon)CVS repository by running the cvs diff command in these subdirectories. This will show the differences between to the core version:

```
$ cd openiked/iked/
$ cvs diff -Nup | tee ../iked.diff
$ cd ../ikectl/
$ cvs diff -Nup | tee ../ikectl.diff
```

4.3 Supported Operating Systems

***Open*IKED** currently runs on only a few major operating systems ("ports"), but it should theoretically run on any Unix-like system with kernel-based IPsec and PFKEYv2[9]. It should specifically run on all systems that are based on the Internet Protocol Version 6 (IPv6)/IPsec reference implementation of the KAME project[4], including all major BSD operating systems and their offsprings.

Nevertheless, the RFC document of the PFKEYv2 standard only specified an API for maintaining IPsec SAs, but it didn't specify an API for maintaining IPsec security policies. OpenBSD was the first open source operating system that supported IPsec and PFKEYv2 by default and the developers created its own API extensions in need for specifying security policies, or "flows". The KAME project implemented it differently using its own API of an "Security Policy Database (SPD)". Most other systems are using the KAME-based PFKEYv2 variant with minor individual differences, most notably the differences in supported crypto algorithms and extensions like NAT Traversal (NAT-T). Support for this variant has been added to the portable ***Open*IKED** version making it compatible with non-OpenBSD ports.

The software additionally depends on two libraries that are widely available for all of these operating systems: OpenSSL[16] 1.0 or later and libevent1[19] (libevent2 should theoretically work but was not tested as it is not used in OpenBSD).

Apple OS X (Darwin)

The Darwin operating system is originally based on FreeBSD, which includes a BSD-like system and a KAME-based IPsec stack and PFKEYv2 interface. Darwin was the first port because of its practical importance to run ***Open*IKED** on MacBooks with Apple OS X.

Since Apple deprecated OpenSSL in recent versions of Darwin/OS X, and their default installation only ships a pre-1.0 version, it is required to install OpenSSL 1.0 manually. I decided to use the MacPorts[7] system to install OpenSSL and libevent1 from its community-based collection of open source packages.

On the kernel side, Darwin provides support for NAT-T, but it is officially marked as "private" and hidden from the official `pfkeyv2.h` header file. The related information is found in the publicly available sources of the "XNU" Darwin kernel and NAT-T is supported by the port. Apple changed some of the standard BSD kernel APIs and header files that are related to networking and packet level bit and endianess operations. For example, FreeBSD's "htobe64" byte conversion macros has been replaced by "OSSwapHostToBigInt64" in a non-standard header file.

NetBSD & FreeBSD

Both systems provide OpenSSL and libevent1 in their package repositories and they are using an KAME-based IPsec stack. The NAT-T extension is currently not supported by the ports to these systems.

My biggest surprise was that they do not support IPsec in their default "GENERIC" kernels, not even as a loadable kernel module. It is required to compile a custom kernel with some additional options to enable IPsec and PFKEYv2.

For FreeBSD[2]:

```
options IPSEC
#options IPSEC_DEBUG
device crypto
```

For NetBSD[11]:

```
options IPSEC
options IPSEC_ESP
```

DragonflyBSD

I started looking into a DragonflyBSD port, but I gave up quickly. Any efforts from the DragonflyBSD community would be appreciated.

GNU/Linux

Linux provides its own PFKEYv2 implementation that is mostly compatible with the KAME variant.

The autoconf framework and compatibility library that is based OpenSSH s portable version made it surprisingly easy to port ***Open*IKED** to Linux.

A drawback is the fact that the Linux kernel developers invented their own non-standard XFRM kernel API that is intended to replace PFKEYv2, which is considered to be obsolete. The PFKEYv2 interface still exists but is poorly maintained and lacks some features, like working support for HMAC-SHA2-256 HMAC authentication for IPsec. Linux originally added HMAC-SHA2-256 support based on the pre-standard speci cation with a truncation length of 96 bits that is incompatible to the standard length of 128 bits that is described in RFC 4868[6]. PFKEYv2 uses pre-de ned identi ers and attributes for algorithms, e.g. `SADB_X_AALG_SHA2_256` for HMAC-SHA2-256 with 128 bits truncation. The Linux kernel recognizes the `SADB_X_AALG_SHA2_256` identi er but assumes 96 bits truncation. The kernel developers never xed this obvious bug to keep compatibility with one or two other implementations that use the pre-standard version. They refer to the XFRM API that allows to set the algorithm, and the key and truncation lengths individually.

4.4 User-friendly GUI

There is actually no need to use a GUI to set up gateway to gateway connections. The con guration le, `iked.conf`, uses a well-de ned grammar that is easy to understand for system administrators and most users of ***Open*IKED** . But when connecting mobile users, or road warriors, to the gateway, an easy GUI is an important requirement for deploying the VPN. These users are most commonly non-technical laptop users that connect to a VPN gateway of their organization, university or company. It is most desirable that they can set up and debug the client-side of the VPN connection without much interaction from the IT department.

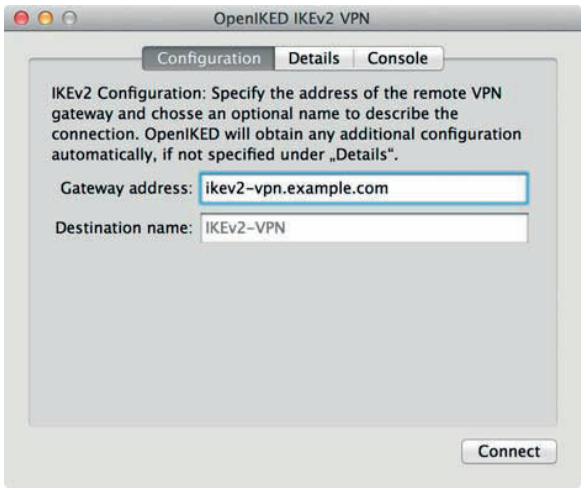
Microsoft Windows

Microsoft Windows 7 introduced an integrated IKEv2 client con guration dialog that is surprisingly easy to use for standard users. The con guration of traditional IPsec/IKE used to be very di cult under Windows but the IKEv2 client only requires installing the required certi cates, setting the remote gateway address, and specifying a user name and password for additional EAP-MSCHAPv2 authentication. And of course, ***Open*IKED** is a compatible gateway that can be used with the built-in Windows client.

OpenIKED.app

To get a similar level of IKEv2 user-friendliness on OS X, I started working on a simple GUI that is inspired by the Windows client. Accordingly, the goal is to provide a very simple tool that allows to set up IKEv2 client connections from Mac-based road warriors. The dynamic negotiation of IKEv2 and the secure defaults of ***Open*IKED** allows to reduce the required con guration settings to the minimum on the client side: remote gateway address and optional connection name. Other optional settings can be con gured in the Details tab. In di erence to the Windows client, additional user-based authentication is currently not supported as EAP-based authentication is only implemented for the server (responder) side.

The current version is a working proof of concept that requires manual installation of keys and certi cates into the con guration directory.



4.5 The Artwork

OpenBSD and its subprojects aren t just known for security, they re also known for their comic-style artwork. Each of these projects has a theme that is including the OpenBSD-styled logo and Pu y the pu er sh in some action. The artwork is used on T-Shirts, posters and CD covers and was originally designed by the Canadian artist Ty Semaka and some other artists today.

When I decided to turn iked into a portable project, it was clear that I needed a matching artwork. I had the idea of using a tin can telephone as a theme that represents VPN communication in an obscure way.

But I needed an artist to create a real Puffy theme and found Markus Hall from Sweden who kindly designed the logo and artwork for ***Open*IKED** . He also contributed the idea that puffy is talking to the cute blue pufferfish girl over the tin can phone.

5 Appendix

5.1 About the Author

Reyk Floeter[1] works as a freelance consultant and software developer with a focus on OpenBSD, networking, and security. He lives in Hannover, Germany, but works with international customers like IIJ in Tokyo. As a member of the OpenBSD project, he contributed various features, fixes, networking drivers and daemons since 2004, like OpenBSD's ath, trunk, vic, hostapd, relayd, snmpd, and iked. For more than nine years and until mid-2011, he was the CTO & Co-Founder of .vantronix where he gained experience in building, selling and deploying enterprise-class network security appliances based on OpenBSD.

References

[1] Reyk Floeter, *Reyk Floeter Consulting*, `http://www.reykfloeter.com/`.

[2] FreeBSD, *VPN over IPsec*, `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/ipsec.html`.

[3] Dan Harkins and Dave Carrel, *RFC 2409 - The Internet Key Exchange (IKE)*, `http://www.ietf.org/rfc/rfc2409.txt`, November 1998.

[4] KAME, *The KAME Project*, `http://www.kame.net/`, April 1998.

[5] Charlie Kaufman, Paul Hoffman, Yoav Nir, and Parsi Eronen, *RFC 5996 - Internet Key Exchange Protocol Version 2 (IKEv2)*, `http://www.ietf.org/rfc/rfc5996.txt`, September 2010.

[6] Scott Kelly and Sheila Frankel, *RFC 4868 - Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec*, `http://www.ietf.org/rfc/rfc4868.txt`, May 2007.

[7] MacPorts, *The MacPorts Project*, `http://www.macports.org`.

[8] Douglas Maughan, Mark Schertler, Mark Schneider, and Jeff Turner, *RFC 2408 - Internet Security Association and Key Management Protocol (ISAKMP)*, `http://www.ietf.org/rfc/rfc2408.txt`, November 1998.

[9] Daniel McDonald, Craig Metz, and Bao Phan, *RFC 2367 - PF_KEY Key Management API, Version 2*, `http://www.ietf.org/rfc/rfc2367.txt`, July 1998.

[10] Damien Miller, *Secure Portability*, `http://www.openbsd.org/papers/portability.pdf`, October 2005.

[11] NetBSD, *Configuring IPsec kernel*, `http://www.netbsd.org/docs/network/ipsec/#config_kernel`.

[12] OpenBSD, *style(9) - Kernel source file style guide (KNF)*, `http://www.openbsd.org/cgi-bin/man.cgi?query=style`.

[13] ———, *The OpenBSD Project*, `http://www.openbsd.org/`.

[14] ———, *The OpenIKED Project*, `http://www.openiked.org/`.

[15] ———, *The OpenSMTPD Project*, `http://www.opensmtpd.org/`.

[16] OpenSSL, *The OpenSSL Project*, `http://www.openssl.org`.

[17] Derrell Piper, *RFC 2407 - The Internet IP Security Domain of Interpretation for ISAKMP*, `http://www.ietf.org/rfc/rfc2407.txt`, November 1998.

[18] Niels Provos, Markus Friedl, and Peter Honeyman, *Preventing Privilege Escalation*, `http://www.citi.umich.edu/u/provos/papers/privsep.pdf`, August 2003.

[19] Niels Provos and Nick Mathewson, *libevent - an event notification library*, `http://libevent.org`.

NPF in NetBSD 6

S.P.Zeidler ⟨*spz@NetBSD.org*⟩
The NetBSD Foundation

Mindaugas Rasiukevicius ⟨*rmind@NetBSD.org*⟩
The NetBSD Foundation

Abstract

NPF has been released with NetBSD 6.0 as an experimental packet filter, and thus has started to see actual use. While it is going to take a few more cycles before it is fully ”production ready”, the exposure to users has given it a strong push to usability. Fixing small bugs and user interface intuitivity misses will help to evolve it from a theoretical well-designed framework to a practical packet filtering choice. The talk will cover distinguishing features of NPF design, give an overview of NPF's current practical capabilities, ongoing development, and will attempt to entice more people to try out NPF and give feedback.

1 Introduction

NPF is a layer 3 packet filter, supporting IPv4 and IPv6, as well as layer 4 protocols such as TCP, UDP and ICMP/IPv6-ICMP. NPF offers the traditional set of features provided by most packet filters. This includes stateful packet filtering, network address translation (NAT), tables (using a hash or tree as a container), rule procedures for easy development of NPF extensions (e.g. packet normalisation and logging), connection saving and restoring as well as other features. NPF focuses on high performance design, ability to handle a large volume of clients and using the speed of multi-core systems.

Various new features were developed since NetBSD 6.0 was released, and the upcoming 6.1 release will have considerable differences regarding their user interface and to a certain level regarding its capabilities.

2 What's special about NPF?

Inspired by the Berkeley Packet Filter (BPF), NPF uses ”n-code”, which is conceptually a byte-code processor, similar to machine code. Each rule is described by a sequence of low level operations, called ”n-code”, to perform for a packet. This design has the advantage of protocol independence, therefore support for new protocols (for example, layer 7) or custom filtering patterns can be easily added at userspace level without any modifications to the kernel itself.

NPF provides rule procedures as the main interface to use custom extensions. The syntax of the configuration file supports arbitrary procedures with their parameters, as supplied by the extensions. An extensions consists of two parts: a dynamic module (.so file) supplementing the npfctl(8) utility and a kernel module (.kmod file). Thus, kernel interfaces can be used instead of modifications to the NPF core code.

The internals of NPF are abstracted into well defined modules and follow strict interfacing principles to ease extensibility. Communication between userspace and the kernel is provided through the library libnpf, described in the npf(3) manual page. It can be conveniently used by developers who create their own extensions or third party products based on NPF. Application-level gateways (ALGs), such as support for traceroute(8), are also abstracted in separate modules.

2.1 Designed for SMP and high throughput

NPF has been designed so its data structures can use lockless methods where suitable and fine-grained locking in general. [1]

Ruleset inspection is lockless. It uses passive serialization as a protection mechanism. The reload of a ruleset is atomic with minimum impact on the active ruleset, i.e. the rule processing is not blocked during the reload. NPF rules can be nested, which is useful for grouping

[1] For the initial NPF release, some components are using read-write locks, although they could be lockless using a passive serialization interface. Since this interface was new in NetBSD 6.0, a conservative approach was taken. As of 6.1, those components have been converted to be lockless.

and chaining based on certain filtering patterns. Currently, the configuragion file syntax supports two levels of groups (having per-interface and traffic direction options), however there is no limitation in the kernel and syntax expansion is planned. As of NetBSD 6.1, dynamic NPF rules will be supported.

Efficient data structures were chosen for the connection (session) tracking mechanism: a hash table with buckets formed of red-black trees. The hash table provides distribution of locks which are protecting the trees, thus reducing lock and cacheline contention. The tree itself provides efficient lookup time in case of hash collision and, more importantly, prevents algorithmic complexity attacks on the hash table i.e. its worst case behaviour.

The session structure is relatively protocol-agnostic. The IPv4 or IPv6 addresses are used as the first set of IDs forming a key. The second set of IDs are generic. Depending on the protocol, they are filled either with port numbers in the TCP/UDP case or with ICMP IDs. It should be noted that the interface is also part of the key, as is the protocol. Therefore, a full key for a typical TCP connection would be formed from: `SRC_IP:SRC_PORT:DST_IP:DST_PORT:PROTO:IFACE`. This key is a unique identifier of a session.

Structures carrying information about NAT and/or rule procedures are associated with the sessions and follow their life-cycle.

NPF provides efficient storage for large volumes of IP addresses. They can be stored in a hash table or in a Patricia radix tree. The latter also allows to specify address ranges.

2.2 Modular design

NPF is modular, each component has its own abstracted interface. This makes writing NPF extensions easy. It also allows easy addition of filtering capabilities for layer 4 and higher. The implementer of a network protocol does not need to know anything about the internals of packet collection and disposal facilities. Services such as connection tracking are provided by a strict interface - other components may consider it as a black box.

The NPF extensions API will be fully provided with the NetBSD 6.1 release. Extensions consist of kernel modules and userland modules implemented as dynamically loadable libraries for npfctl, the NPF control utility. Extensions get configured as rule procedures and applied on the selected packets. They can take arguments in a key-value form. Extensions may rewrite packet contents (e.g. fields in the header) and influence their fate (block/pass).

There is a demo extension: the kernel part in `src/sys/net/npf/npf_ext_rndblock.c` and `src/lib/npf/ext_rndblock/npfext_rndblock.c` for npfctl. This extension simulates packet loss. The kernel part file is less than 180 lines long, the npfctl part is less than 100. Given that the copyright notice is a significant part of that, the 'administrative overhead' for a NPF extension is fairly low.

3 What can it do, at present?

The configuration file syntax is still avoiding to be Turing complete. In spite of the obvious temptation, it is planned to keep it that way. The config syntax has changed noticeably between what was released with NetBSD 6.0, and what will be in NetBSD 6.1. Further change is expected (of course, only to the better).

As mentioned, NPF is a stateful packet filter for IP (v4 and v6) and layer 4 - TCP, UDP, ICMP and ICMPv6, including filtering for ports, TCP states, ICMP types and codes are currently implemented.

Configuration example:

```
pass in family inet proto tcp \
  from $nicehost to $me port ssh
```

Rules get processed top to bottom as they are written in the config file, first through the interface specific group and then through the default group. Processing can be stopped early by using the tag `final` in a rule.

Addresses for filtering can be inferred from what is configured on an interface (own addresses), can be configured in the npf.conf configuration file, or can be fed into (and deleted from) tables defined in npf.conf using `npfctl table` commands.

NPF sidesteps the fragments issues by reassembling packets before further processing.

NPF supports various address and port translation variants. In NetBSD 6.0.x, it supports network address port translation ("masquerading"), port forwarding ("redirection") and bi-directional NAT in IPv4. There is an application-level gateway (ALG) for traceroute and ICMP translation; ALGs are provided as kernel modules.

NetBSD 6.0 configuration example:

```
# outgoing NAPT
map $ext_if dynamic 198.51.100.0/24 -> \
  $ext_if
# port forwarding
map $ext_if dynamic 198.51.100.2 port 22 \
  <- $ext_if 9022
```

Session state (including NAT state) can be dumped to file in a serialised form.

Packet normalization and logging are available since the NetBSD 6.0 release. From 6.1 onwards, they will be provided as NPF extensions. They get configured as procedures:

```
# procedure to log a packet and
# decrease its MSS to 1200 if higher
procedure "maxmss_log" {
  log: npflog0
  normalise: "max-mss" 1200
}
# in an interface group, apply the
# procedure to matching packets:
pass out family inet proto tcp flags S/SA \
  to $pmtublackholed apply "maxmss_log"
# continue to further processing
```

Rule reload builds a new configuration in the kernel, and switches from new to old atomically. NPF does not hold several configurations in the kernel to switch between. Instead, support for a temporary load with a timer and automatic rollback is planned.

4 Testing and debugging

As a good citizen of NetBSD, NPF has regression tests in the automated tests framework. These are implemented using the RUMP (Runnable Userspace Meta Programs) framework, which allows to exercise the kernel elements of NPF in userspace, without having to boot a test kernel (even in a VM). Regular tools like gdb can be used for debugging. Unit tests are implemented and available via the npftest program. Additionally, the program can load a custom configuration file and process packets from a pcap file. This allows developers to analyse NPF decisions and state tracking in userspace using a captured sample. As a result, debugging the problems experienced in other environments is easier.

Another debugging help is the npfctl debug option, which dumps the internal representation of a parsed config, as npfctl would have sent to the kernel, i.e. as disassembled n-code.

5 Meeting users

The NetBSD 6.0 configuration file syntax uses one label for a network interface, the addresses configured on that interface and the addresses configured for a specific address family on an interface; the meaning was inferred from context. This turned out to be hard to understand for humans (even if the software interpreted it just fine) and was changed for NetBSD 6.1 to explicitly require a function that takes the interface name as an argument, e.g. `inet($interface)` for all addresses configured on `$interface`.

Even before NetBSD 6.0 was shipped, the syntax expressing address and port translations was reworked to use `map`. It is bound to an interface, the crossing of which triggers the translation. By convention, the ”inside” network and optional port definition is on the left-hand side, and the ”outside” on the right side on the arrow that discerns the translation direction. This makes configurations easier to read (for the current sample of users).

Users also saw cosmetic issues that were previously missed, like printing garbage for linklocal scope for `npfctl show`.

Further improvement will be necessary in giving useful error messages for syntax problems in the configuration file. Getting feedback from users is very important to get NPF production ready, to oust the last small bugs and help identify the most desireable missing bits.

6 Whereto, my lovely?

NPF is still a moving target - there have been major changes and feature additions recently. Although the core APIs of NPF are becoming stable, they are still subject to extensions and minor changes. Therefore, when developing 3rd party extensions, it is recommended to follow source-changes@NetBSD.org since ”catching up” with the changes might be necessary.

NPF will also use BPF byte-code (first available in NetBSD 6.1), and BPF with just-in-time (JIT) compilation likely in NetBSD 7.0. In combination with the widely used pcap(3) library, it provides the ability to write sophisticated filtering criteria in an easy, expressive and very efficient way that is familiar to users.

A GSoC 2012 project produced patches for simple-case NPT66 (/48 only) and NAT64 (well-known prefix only). However, these still need to be expanded and integrated, and are therefore unlikely to make it into NetBSD 6.1 due to ”free time” constraints.

Further extensions are likely to arrive. Possible examples would be extensions for port knocking, traffic accounting and connection rate limiting.

Given interest, addition of further layer 4 protocols is likely.

Finally, NPF has design provisions to eventually enable active-passive and active-active firewall clusters; no timeline for implementing this exists yet.

64bit SMP NetBSD OS Porting

for TILE-Gx VLIW Many-Core Processor

Toru Nishimura

Sanctum Networks, Pvt. Ltd.

nisimura@sanctumnetworks.com

Abstract

Many-core processor is an attractive platform to run a general purpose OS like NetBSD. We, a team in Sanctum Networks, ported NetBSD 6.0 to 64bit VLIW many-core processor named TILE-Gx. In this paper we introduce distinctive features of TILE-Gx in some depth as the product remains less known in our engineering community. NetBSD porting was made well and smooth than anticipated. We realized that NetBSD 6.0 is a mature SMP OS which provides streamlined kernel structure and offers rich set of kernel API specifically designed for large degree SMP, beyond 32 processor, configuration. As a part of conclusion we mention about some of TILE-Gx NetBSD application area which we're willing to build.

1. Project outlook and time line

This porting project was initiated in mid June 2012 at Tokyo. The goal was to achieve full SMP capabilities on the 36 core TILE-Gx processor and understand the suitability of the TILE architecture for various application development.

- our porting target is Tilera Empower 1U computer with 36 core TILE-Gx processor.
- development environment with cross compiler was made at early July 2012.
- geographically separated two GIT repositories in push-pull synchronizing.
- Japan side hosts are in Sakura VPS and Amazon EC2.
- kernel image linking completed in late July 2012. It contained a lot of stab code guaranteed not to work.
- single core kernel was successful in running ramdisk sysinst program at 2012-10-30.
- since then, kernel stability, GbE driver and SMP has been persuaded.
- as of March 2013 porting project is going active. A number of functionalities, in particular ones for TILE-Gx unique features, are under development.

2. TILE-Gx features

TILE-Gx design was invented by an MIT professor, Dr. Anant Agarwal. It's the latest incarnation of a long time research since 1980s. TILE-Gx is the third generation product. There are two successors, TILE64 and TILEPro which are based on the same TILE architecture approach. Following to two 32bit designs, the third generation model is made 64bit processor.

TILE architecture emphasizes the scalability and low power with a unique on-chip inter-connection technology. TILE-Gx product family has 9 ~ 100 core configuration. Each of core is laid out in tiles-on-wall fashion. Core runs at 1.2GHz clock. Under work load with modest network activity 36 core TILE-Gx is said to achieve 25-30W power consumption in total.

TILE-Gx is a 64bit processor. It has 64bit integer and 64bit floating point operation, 64bit register and 64bit address space by 64bit pointer. These simple characteristics are quite familiar to any of plain old UNIX programmers since the time when MIPS R4000 was introduced at early 1990s.

2.1. Instruction set feature

TILE instruction set architecture is 3 way VLIW. Two or three instructions are in a single 64bit word which is in turn called “instruction bundle.” TILE can run up to three instructions simultaneously.

TILE-Gx has sixty four 64bit register file. Table 2-1 shows register definition. 58 of them, including two hardwired zero registers, are general purpose. It contains thread pointer tp register to facilitate thread programming. It shows ABI to define the common register usage program ought to follow.

register	mnemonic	type	usage
0 - 9	**r0 -r9**	saved by caller	arguments/return values
10 - 29	**r10 - r29**	saved by caller	“temporary”
30 - 51	**r30 - r51**	saved by callee	“safe across call”
52	**r52**	saved by callee	frame pointer
53	**tp**	dedicated	thread pointer
54	**sp**	dedicated	stack pointer
55	**lr**	saved by callee	return address
56	**sn**		always zero
57 58 59 60 61 62	**idn0** **idn1** **udn0** **udn1** **udn2** **udn3**	onchip network communication	I/O Dynamic Network 0 I/O Dynamic Network 1 User Dynamic Network 0 User Dynamic Network 1 User Dynamic Network 2 User Dynamic Network 3
63	**zero**		always zero

Table 2-1 register assignment for TILE-Gx ABI

Note that TILE offers a rather large number of, 10, arguments in register for function call. The return values are placed in the same set of register for arguments. This means r0 register content will be destroyed quite often when a function exits. Better to remind it as one of TILE debugging tips.

Six registers are reserved for inter-core communication via on-chip network named UDN and IDN. These registers work as FIFO ports much like FSL “fast-simplex-link” in MicroBlaze processor. Reading from empty register or writing on occupied register may cause the processor to stall until condition meets.

Registers are shared resource among VLIW concurrent execution flow. Each subroutine has a single entry point and a single exit. No parallel subroutines are in action. Register conflicts in parallel execution is considered program error. It brings undefined / unexpected values in register file. Register conflict avoidance is the programmer's responsibility.

TILE instruction has two different formats; X-format for 2 instruction in a bundle and Y-format for 3 instruction in a bundle. Basic arithmetics is done in 3 operand form. As register file is 64 in size, 3-operand instruction requires 18bit = 6 x 3 for register designation plus some more for opcode. 3-in-1 64bit bundle is considered rather tight encoding, however, contributes instruction density much. 2 register arithmetics have signed 8bit immediate or signed 16bit immediate value. The latter instruction belongs to 2-in-1 bundle X-format as it needs to have longer encoding. Unoccupied instruction slot in a bundle is filled with NOP which works as a flowing bubble in execution pipeline.

The most notable difference from conventional CISC / RISC instruction set is the lack of “register indirect with offset” addressing mode. TILE has no “LW R3, 0x178(R2)” style memory access. This means that local variable on stack and/or (C language) struct member must be accessed through an explicit pointer in a temporary register to refer the target address. It's a stark contrast to the case where “register indirect with offset“ addressing mode can achieve load / store operation with “base register + immediate value offset” very handy for local variable and struct member. This is another tip for TILE programming to remember.

2.2. Other useful instruction

TILE instruction set has an orthogonal set of atomic math / lock instructions.

fetchadd	Atomic addition
fetchand	Atomic logical AND
fetchor	Atomic logical OR
cmpexch	compare-and-swap

Table 2-2 atomic instructions

All of them have two variations for 8byte operation and 4byte operation. These instructions are comfortably useful to implement NetBSD atomic_ops(3) routines. They are well defined set of MP-safe atomic operations and widely used in SMP NetBSD kernel construct and/or parallel programming library like pthread(3).

Cmpexch instruction is for CAS “compare-and-swap” or TAS “test-and-set” operation. It works like as Intel cmpxchg instruction. Note that it's not based on LL/SC synchronize model found in MIPS, Alpha, PowerPC and ARM64. TILE cmpexch works with accompanying “CmpValue” SPR register. Locking primitives can be implemented with it in usual manner.

TILE-Gx has rich set of DSP and SIMD instruction. It also has some fancy instructions. A set of bit field operations, CLZ (count-leading-zeros) and CRC32 polynomial math for hashing / checksum and so on.

TILE-Gx floating point math does not have dedicated register set. FP instruction uses GP registers for source / destination operands.

2.3. Address space

TILE-Gx has 42bit effective address bit out of 64bit VA virtual address pointer. Virtual address is separated in upper 2TB space and lower 2TB space. There is a large void in between. VA[63:41] is either of all-0 or all-1, that is, sign extended from VA<41> value.

TILE-Gx has no MIPS KSEG0 / KSEG1 / XKPHYS like address segment exists to distinguish cache nature. Software is in charge of address inhabitation and cache nature control with help of smart TLB usage.

2.4. Layered protection

TILE architecture provides four level protection scheme. Level is ranging from 0 least protected to 3 most protected. It allows to build layered protection domains which run protected programs in each level.

PL0	User applications
PL1	Guest OS
PL2	Tilera Hypervisor
PL3	" virtual machine monitor" (who knows what really it is.)

Table 2-3 TILE protection level

Program runs in low order protection level is inhibited to touch resources in hight order level. Each core runs one of four protection level. Current protection level of individual core is called CPL. Control transfer is done by executing dedicated instruction; swint0, swint1, swint2, swint3. NetBSD uses swint1 instruction for system call to be issued by applications programs.

There is a large set of SPR registers. mfspr / mtspr instructions operate them. SPR number is encoded in 14bit. Most of SPR registers have their own ML "minimal protection level" value to arbitrate which level (0 ~ 3) of program can access to. MPL is the basis of layered protection for TILE runtime environment.

2.5. TLB and TSB

TLB plays a central role in TILE architecture. In TILE architecture TLB dost not just make VM virtual memory possible but also realizes chip-wide global cache coherency. TILE TLB entry is designed to be multi-core aware. TLB entry optionally holds the location of core in chip (in X-Y coordinate) to track and identify how TLB entry to tell VA-PA mapping is tied with a specific core.

Like as most of modern processors, TILE TLB is software managed. TILE-Gx has independent TLB stores for instruction and data; 16 entry iTLB and 32 entry dTLB. Note that TLB is a shared resource among programs which run in different protection domain. The 42bit VA space is also shared among them. Tilera Hypervisor reserves some of TLB entries for its own. Remaining is free for guest OS and application programs to use.

The TLB management strategy is modeled after SPARC processor. TILE uses "TSB" and "TTE" nomenclature for the very same purposes.

TSB "translation store buffer" is a software extension of TLB. TSB holds a super set of TLB in main memory. It works as a staging area to inject TLB entry into processor's iTLB or dTLB. HV is in charge for TLB miss handling. It always consults with TSB content in action. Guest OS can only operate TSB store. As TLB is one of highly sensitive shared resource among various programs, guest OS can not make access TLB. TSB is normally reserved inside protected guest OS memory area. TILE TSB is a unified one to hold iTLB entries and dTLB entries. The approach is different from SPARC64 which has iTSB and dTSB in parallel. TTE "translation table entry" is the software defined intermediate format of TLB entry.

On TLB miss HV takes control to run TLB refill operation. It searches first the offending TLB entry in TSB store. If the target entry is found, HV injects it to either of iTLB or dTLB and complete refill operation. If HV finds TSB has no such entry, then it posts a request for guest OS to come in and solve this "TSB miss" condition. Guest OS, in turn, responds to the TLB miss exception identifying it as genuine access error or recoverable fault condition. The rest of operation is identical to popular software managed TLB processors. If guest OS finds the exception is true TLB refill case, it adds the offending TLB entry into TSB store and returns. HV will take care the refilling. If guest OS finds the exception access error or protection violation, it performs its way to handle the cases.

TILE has ASID "address space identifier." ASID is to improve TLB hit ratio, that is, better VA->PA translation efficiency. It's as normal as and identical to other ASID processors. TILE ASID is 8bit, offering 256 individual address spaces to be distinguished for TLB lookup operation. Some literatures incorrectly mention that ASID is an extension of VA, like saying it realizes concatenated 8 + 42 address bit. ASID is to virtualize TLB, or to make imaginary multiple TLB stores which are numbered and iterated by ASID. ASID demands a smarter VM to operate. This topic will be discussed in a later section.

As other processors do, TILE processor handles many kinds of interrupt / exception. Device asynchronously posts variety of requests and different types of exception happens while a processor is in action. TILE uses IPI "inter-processor interrupt" not only for pure inter-processor messaging, but also for I/O device interrupt notification. As TILE integrated on-chip devices are located apart of core and notification comes across on-

chip network, it'd be reasonable to use IPI laminating many into a single form.

2.6. iMesh on-chip inter-connect

Processing core is laid out in a tiles-covering-wall fashion with mesh shape inter-connect to couple each other. Inter-connect has X-Y / street-avenue like layout. At each crossing is an independent switch processor to tie a computing node with the entire switch network. Tilera names it iMesh technology.

Switch processor is 16bit RISC to run low latency and high bandwidth switching function through limited number of signal connections. Besides of 4 paths for N, E, S and W directions to neighboring switches, one switch data-path is coupled with processor's L2 cache. Data stream travels through L2 first, then either of L1 iCache or dCache reaching to a processing core.
The inter-connect offers UDN "User Dynamic Network" and IDN "I/O Dynamic Network" for general purpose on-chip streaming and messaging communication. Total 6 register of TILE-Gx processor are assigned to accommodate the ease of programming.

It should be reminded that iMesh does not implement nor enforce any kind of "smart network topology." There was a number of massively-parallel multi-processor super computers built from time to time. All of them more-or-less persuaded a smarter topology for processor inter-connect to maintain low-latency and high bandwidth.

Notable examples are Cray T3D and SiCortex SC5832. T3D had a 3-dimensional "torus" graph topology to make Alpha processors tightly coupled each other. SC5382 had "Kautz graph" topology to inter-connect 6-core MIPS64 processor with the help with built-in DMA engine to talk with L2 caches and I/O devices.

In TILE architecture on-chip inter-connect is software defined. Switch processor can program the network topology to adapt varying demands. In this way, TILE architecture can maintain the flexibility and the scalability in parallel. It's unlikely "topology optimized" super computers can achieve both natures in balance.

iMesh API is provided to make finer control over on-chip network. Cores can be partitioned into groups which work parallel as if they are islands. This feature is implemented by switch network programmability

With help by "topology-aware" and "cache attribute aware" TLB entries, iMesh acts a central role for cache coherency.

2.7. Cache design and feature

Each core has 32KB iCache, 32KB dCache and 256KB i/d combined L2 cache. Either of L1 cache has VIPT "Virtual Index and Physical Tag" nature.

L1 iCache	32KB, 2 way associative, 64B line size.
L1 dCache	32KB, 2 way associative, 64B line size, write-through.
L2 cache	256KB, i/d combined. 8 way associative, 64B line size, write-back.

Table 2-4 cache characteristics

Some TILE processor literatures mention to "coherent L3 cache." It's somehow imprecise. The L3 functionality is achieved by a group of L2 cache. The scheme is called "cache homing." Let us start the explanation.

TILE L1 cache is inclusive to L2. L1 holds subset of L2 contents at any moment. L2 miss happens when offending cache line data is not found in L2. Core asks about the missing cache line data to "neighboring cores" which are grouped by HV for a single OS instance. If found there, cache line data is transferred to requesters L2 cache.

Foreign L2 caches work as an extension of local L2 cache. In other words, a group of cores share their L2 cache contents each other. This scheme is named "cache homing" and Tilera calls the group of L2 cache as "coherent L3" cache. 36 core Gx processor has "9MB coherent L3" = 36x 256KB L2.

Cache lines can be populated sparsely among different L2 to improve the cache efficiency. L3 cache homing is one page attributes. It's controllable by per-page basis.

3. TILE-Gx on-chip devices

integrated multiple DDR memory controller

2 controllers in 36 /16 core models, 1 in 9 core model. TILEPro, the successor of TILE-Gx, 64 core model had four DDR2 memory controller on chip. With dual controller configuration, memory can be driven in interleaved fashion.

mPIPE packet classifier

It's a programmable intelligent packet engine. It offers "frame parse" function to run "sieve-to-forward" classification on incoming Ethernet frame stream at line speed. mPIPE is tightly integrated with GbE / 10G Ethernet network interface.

- 4x 10G ports are available in 36 core model.
- Each port can be reprogrammed to host 4x GbE network interface.
- GbE-only ports are also available in 16 / 9 core model.

mPIPE has local buffer memory to handle incoming and outgoing Ethernet frames. mPIPE can perform load

balancing to distribute ingress frames to cores.

Core binds mPIPE device register set to a particular virtual address with a designated dTLB entry for control. mPIPE in turn holds an I/O TLB entry to access data which resides in target (~accelerating application or guest OS) address space so that it can understand VA->PA translation for frame data and accompanying descriptors.

mPIPE has its own 32bit instruction set. A special GCC toolchain is provided to program it.

- two or three operand instruction.
- 32x 32bit register file; 22 of them are general purpose.
- Private SPR registers with mfspr and mtspr to use.

MiCA crypto and compression engine

It's a standalone computing processor populated inside TILE-Gx. Multiple MiCA processors are on a single Gx. MiCA can copy data while encrypting and compressing operation in action. It's a streaming operation.

Core binds MiCA device register set to a particular virtual address with a designated dTLB entry for control. MiCA in turn holds an I/O TLB entry to access data which resides in target address space so that it can understand VA->PA translation for crypto / compression data.

Conventional I/O devices

There are some conventional I/O devices like PCIe, USB2.0 and I2C/SPI in our porting target computer.

PCIe controller works in either root-complex (host) or end-point (device) mode. USB2.0 is used for multiple purpose. It works as virtual console while in development and debug. It can also inject a binary image to Gx processor to run. The binary image consists of boot programs, HV image and guest OS in predefined format.

4. Tilera Hypervisor

HV utilizes TILE protection level feature. Guest OS has heavily limited access to SPR registers. Only handle number of SPR registers allowed to used by Guest OS.

HV is populated at the 1MB area in the upper 2TB space with a hardwired TLB entries.

HV has great control over the entire TILE processor complex. HV makes cores into groups which are manged in M x N rectangle shape to form OS instance.

HV assigns I/O devices to particular instances with I/O TLB entries. Tilera calls the scheme MMIO “memory mapped IO” scheme while SPARC names it “IOMMU.”

HV allows several guest OS'es to run simultaneously. Device and core grouping is defined a HV configuration at the machine startup. Because of it, HV is yet to be improved as flexible as what Xen can do in these days.

Two serial ports are provided in Gx processor. HV can dynamically bind one of serial ports to running OS instance as it console.

BME “bare metal environment”

BME is an API to build “light weight monitor” which runs designated core(s) run special purpose “driver” for data-plane processing. In general any BME program needs accompanying fully-featured OS, like Linux, as a control plane to manage the whole software complex.

iMesh messaging facility API is used by control OS to communicate with BME programs which run on separate core(s).

Several code examples are provided by Tilera;

- one TILE core runs “encryption server” on BME while Linux as “client” which receives the results from BME. In this example data transfer is done in a share page with help of UDN messaging between two.
- A number of Linux process get private cores to run and communicate each other with UDN messaging and shared pages.

5. NetBSD/tile

This port is based on NetBSD 6.0 STABLE code set. It's a 64bit SMP kernel and 64bit userland. The kernel runs as a guest OS conjunction with Tilera Hypervisor.

NetBSD/tile uses GCC 4.6.3 ported by Tilera. We have been using it as it is. As GCC 4.5 is still in use in NetBSD 6.0 code set, we integrated GCC 4.6.3 to start.

64bit pmap was implemented from scratch. It's modeled after Alpha pmap. Although TILE-Gx offers 13 different page sizes, HV employs much humble page size selection. We chose 64KB page for NetBSD/tile as it is parallel to Tilera Linux VM implementation. The virtual address partitioning is “10 + 8 + 8 + 16.”

NetBSD/tile utilizes SMP ready NetBSD6 kernel internal as large as possible. NetBSD5 introduced much sophisticated kernel constructs and API sets which are effective and useful for scalable SMP OS. Since then gradual streamlining has been done for fore-running SMP

NetBSD ports. Now NetBSD6 is a mature platform to make a jump start for fresh SMP porting.

The following is the typical set of useful SMP API;
- atomic_ops(3)
- kcpuset(9)
- xcall(9)

The first group must be implemented in early kernel porting stage. In most cases they have to be written with assembler code to be best suited for particular processor nature. The latter two are pure software construct written in plain C code.

Parallel programming model is NetBSD pthread. NetBSD pthread is well organized to adapt various processors with minimum effort. We did not make particular modification for TILE-Gx support. It works just like as any other pthread implementations like one in Tilera Linux.

Very limited number of assembler files were written so far. One one for kernel; it's “locore.S” The file contains 4 well define major routines;
- CPU startup for primary core and secondary cores.
- Exception entry / dispatch / return
- CPU context switch
- fast software interrupt dispatch / return

Other assembler files are for libraries and a few application program like rtld(1). The following is the list of major TILE-Gx dependency in concern.
- src/common/lib/libc/arch/
- src/lib/libc/arch/
- src/libexec/rtld/arch/

5.1. Key design decisions

In this section we describe concisely about some design decisions to make a port realized.
- struct trapframe, struct switchframe and struct pcb.
- UPACE to hold kernel stack and struct pcb.
- pmap(9) to interface processor with NetBSD VM.
- Exception and interrupt handling to comply target processor design intent.
- IPI “inter-processor interrupt” which is essential to make SMP possible.

struct trapframe is a snapshot image of runtime context. One trapframe is always created at the high end address of USPACE. Actual kernel stack starts just below of it to grow downward. The reserved trapframe area is for user process context. Whenever user process gets interrupted by exception or device notification, the trapframe is to record the user context to resume later. This area is also used for system call. While in kernel mode, kernel gets interrupted by the same reasons as user mode process does. At the occasion, trapframe is created and pushed on kernel stack.

TILE architecture has 64x register file. 8 out of them are not a part of process context and to be excluded. We chose 64x 64bit = 512B size anyway for struct trapframe. In vacant fields we place some extra contexts for process to retain. They are exception return address, status register value at the time when exception happened, offending exception type and a value of a certain SPR, “CmpValue” indeed, for cmpexch instruction.

struct switchframe is for CPU context switch. NetBSD defines two context switch routines. cpu_switchto(9) and lwp_return(9) are the routine to perform context switch. TILE architecture has a large set of caller-saved register. Our switchframe is 25x 8B = 200B in size.

struct pcb is one of longest surviver among UNIX kernel primitives. It got smaller than used to be since the way how to run context switch made smarter. Our struct pcb is as small as just to hold struct switchframe and a bit extra.

USPACE size is 64KB as aligned with NetBSD/tile page size.

5.2. ASID management

ASID management is modeled after the way used for NetBSD/alpha and NetBSD/mips. In this section we explain it in larger degree.

Kernel has a variable for “ASID generation number” to make sure a unique ASID assigned for running process in processor. It's a central idea. Our ASID management algorithm works in this way.
- pmap_activate(9), one of NetBSD kernel API, switches processor's current ASID value whenever a new process is ready to take control.
- Switching current ASID is a light weight operation for OS as it eliminates the necessity of TLB flush at every context switch. ASID-less processors need to perform the whole scale TLB invalidation to discards all entries at every context switch. As TLB works as a cache for VM address translation, TLB flush hearts severely TLB hit ratio spoiling VM performance. ASID-aware processors just need to switch current ASID value. Changing processor current ASID can be considered to switch imaginary TLB store which exists for each ASID value.
- Every new born process has no ASID assigned. pmap_activate() chooses new one which is never allocated so far and assign it with the process. pmap_activate() also records the current ASID generation number in the process's pmap store.
- ASID is a small number to count only up to 255. If

pmap_activate() finds the 8bit gets exhausted, then it bumps ASID generation number in a kernel variable by 1 and chooses a new ASID wrapped to the least available number (normally 1 as ASID 0 is reserved for NetBSD kernel pmap). On this occasion, kernel makes full scale TLB invalidation to discard all TLB entries.

- Whenever pmap_activate() is about to switch current ASID, it checks ASID generation number in kernel variable matches the process's generation number recorded at ASID creation. If they differ, it means the process's ASID is no longer valid. pmap_activate() selects and assigns a fresh ASID for the process to run recording current ASID generation number too.

Given any moment every running process has its own unique ASID. The generation number scheme reduces the necessity of full scale TLB invalidation in great degree. TLB flush only happens when ASID range gets run out and ASID generation number is to be bumped.

5.3. TLB shootdown

TLB shootdown is the essential operation in any SMP kernel. Like as processor cache, TLB is a local resource to processor core. The way to invalidate local cache or local TLB is provided by a certain mechanism. In general invalidating remote TLB is as hard to archive as invalidating remote cache.

In SMP system, TLB invalidate operation must be propagated to multiple cores which have been running a particular process. Process's pmap must maintain a "processor set" to track which cores have run it. Here goes the explanation of remote TLB shootdown by ASID bump;

When pmap(9) detects the necessity to invalidate one or more TLB entry of particular process, kernel needs to run invalidate operation both for;

- the "local" core which happens to run the kernel on behalf of pmap() at the very moment, and
- all of "remote" cores which the process's pmap() are aware of.

The latter operation is named "TLB shootdown." It's implemented with IPI. It triggers a remote core action by inter-core message. TLB shootdown logic can be built in with help of xcall(9) "cross call" kernel API.

A smart ASID management can achieve remote TLB invalidation with a small cost.

- mark ASID in offending process's pmap() store "unassigned."
- broadcast a xcall(9) message to remote cores triggering IPI.
- When one of cores is about to run the process in the next scheduling, pmap_activate() will choose and assign a fresh ASID the offending process. The stale TLB entry with abandoned ASID gets invalidated at once.

5.4. Useful SMP facilities in NetBSD6

SMP NetBSD kernel provides the way to manage CPU in finer gain. There are less known set of useful commands. Let us mention about them in brief.

cpuctl(8) ... try "/usr/sbin/cpuctl list" on your modern Intel computers. It shows the list of CPU state which tells online / offline.

prset(8) ... try "/usr/sbin/prset -p" on your modern Intel computers. It can create arbitrary number of "processor set" which is bound with any process. CPU affinity is made possible by processor set binding. Would be possible to bind a processor set with a kthread (kernel thread) which runs specific kernel subsystem like GbE and/or disk drivers.

schedctl(8) ... try "/usr/sbin/schedctl -p 1" on your modern Intel computers.

- It assigns one of predefine scheduling policy to a process. It replaces nice(1) and renice(8) priority control commands.
- Three difference scheduling policies provided by NetBSD so far.
- Time-sharing which follows the tradition UNIX semantic used for long time.
- First-in, First-out
- Round-robin

5.5. Future development

This project is active. Here we try to make a summary about missing functionalities and future development in some arbitrary order.

Soon to use TILE-Gx native FP instructions. Currently the entire NetBSD including userland is made with "–DSOFTFLOAT" compile option.

Drivers for some conventional PCIe devices like SATA and/or 100M Ethernet NIC. Currently whole system code image is injected with USB debugging facility to run NFS diskless configuration.

iMesh communication API for NetBSD. It remain under research. For now there is no provision to utilize iMesh programming.

MiCA integration with a proper API. NetBSD kernel has pcu(9) "per-CPU-unit" framework. It's for the

encapsulation of CPU's hardware context to save / restore. It handsomely covers the cases beyond the general purpose register. The typical usage of pcu() is to manipulate FPU register set. We're considering whether pcu() can integrate multiple MiCA units to NetBSD kernel in sane manner.

NetBSD/xen allows dynamic attach / detach maneuvre while kernel is up-running. It allows core to attach / detach dynamically and allows block device attach / detach dynamically. We assume it'd be some difficult to implement similar functionality in TILE, however, it'd worth persuading the way to make them possible.

We're aware of Tilera HV has no provision to startup & tear down "targeted" core while up-running. HV source code is disclosed as a part of Tilera MDE development package. It's said that HV can be extended for customer's own needs.

LLVM transition from GCC4 is recognized mandatory as it would exploit the potential of TILE VLIW nature.

6. NetBSD TILE-Gx applications

We focus on compute-intensity markets. We're considering to engaged in SDN, VLDB search engine and desktop HPC.

SDN "Software Defined Network"

It's the third wave of virtualization technology; server virtualization, storage virtualization and then network virtualization. Industry trends predict that routers and firewall will vanish soon while they are morphing into big smart switches.

Bangalore team is now exploiting super fast frame forwarding algorithms. They are generalized for "search-and-lookup" computational complexity reduction problem. Problem statements are now being defined. The implementation of algorithms must be robust enough to handle incoming frame stream as fast as arriving in wire-speed rate. They must also be robust enough combination explosions of matching rules.

VLDB search engine

In these days Very Large scale DB are directly connected with Internet. It's working in real time manner. The typical case is SNS like FaceBook. "mem-caching" is now a common tactic to implement super fast search engine. We recognize many-core processor and GPGPU are now gathering industry attention as they would be good vehicle in engineering sense for VLDB search engine.

Desktop HPC "High Performance Computing"

It's a kind of human being's forever desire to own super computer at hand. TILE-Gx can be a handy basis of many-core 64bit general purpose computer. It's said that the next generation of Gx can be extended by wiring multiple processor with InterLaken inter-connect. Today a pair of Tesla GPGPU 16x lane PCIe cards can archive Tflops grade computing power. Then, how about making the twenty first century incarnation of desktop personal computer, let's say, whose outlook are just like as SGI Indigo or NextCube?

7. Conclusion

Poring NetBSD 6.0 to TILE-Gx is found easier than anticipated since NetBSD 6.0 provides SMP ready kernel constructs and API sets to use. The number of lines written in assembler was very small as the essential part of porting burden are well defined. VLIW nature of the processor is recognized not a hurdle.

Acknowledgment

Sanctum Networks wishes to express its gratitude to all members involved in this project, especially the members from Japan who contributed critically in the early stages.

Automating the deployment of FreeBSD & PC-BSD® systems

by

Kris Moore
kris@pcbsd.org

Introduction

When I had originally designed the **pc-sysinstall** system-installation utility for PC-BSD, one of the critical features was the ability to make every install a fully-scripted install. This would ensure that we could always replicate a particular type of installation and keep track of what steps were taken during the process. This functionality allowed us to easily develop and maintain a graphical front-end to the system installation process, by having it simply create a small configuration script of installation options. Over the years the pc-sysinstall utility has evolved to support more advanced features, such as support for ZFS, including RAIDZ and dataset control, package installation, FreeBSD dist-file support and much more.

With this functionality in mind, the pc-sysinstall backend could also be adapted to quickly automate the deployment of FreeBSD servers and PC-BSD desktops using a network boot environment. In PC-BSD & TrueOS™ 9.1 and higher, this functionality has been added in the form of the "**pc-thinclient**" utility. This utility walks you through the process of downloading and configuring the necessary software to 'PXE' boot your systems via the LAN interface, as opposed to using traditional media, such as DVD or USB. In order for clients to boot via PXE they will need a PXE capable network adapter, which is often a capability indicated in the specifications of your network card. In this paper we will take a look at the initial setup and configuration of your PXE installation server.

The initial PXE setup

To get started, you will need to have a system with two network interfaces running PC-BSD or TrueOS 9.1 and a complete ports tree in /usr/ports. If you do not have the ports tree installed, you can download it by running the command "**portsnap fetch extract update**" as root. With these pieces in place, open a root prompt and run the "**pc-thinclient**" command. The first screen you see will look something like this:

```
root@pxehost:/root # pc-thinclient
/usr/local/bin/pc-thinclient will install the components to convert this system
into a thin-client server.
Continue? (Y/N)
```

Enter "**y**" to continue, and the following screen will be shown:

```
Do you wish to make this a remote X desktop server or install server?
(r/i)
```

In this case you are going to be setting up a PXE installation server, so enter "**i**" to continue. (The "**r**" option can be used to make your system a X thin-client server. More information about this can be found on the wiki page at the end of the article.) After selecting your type of PXE system, the

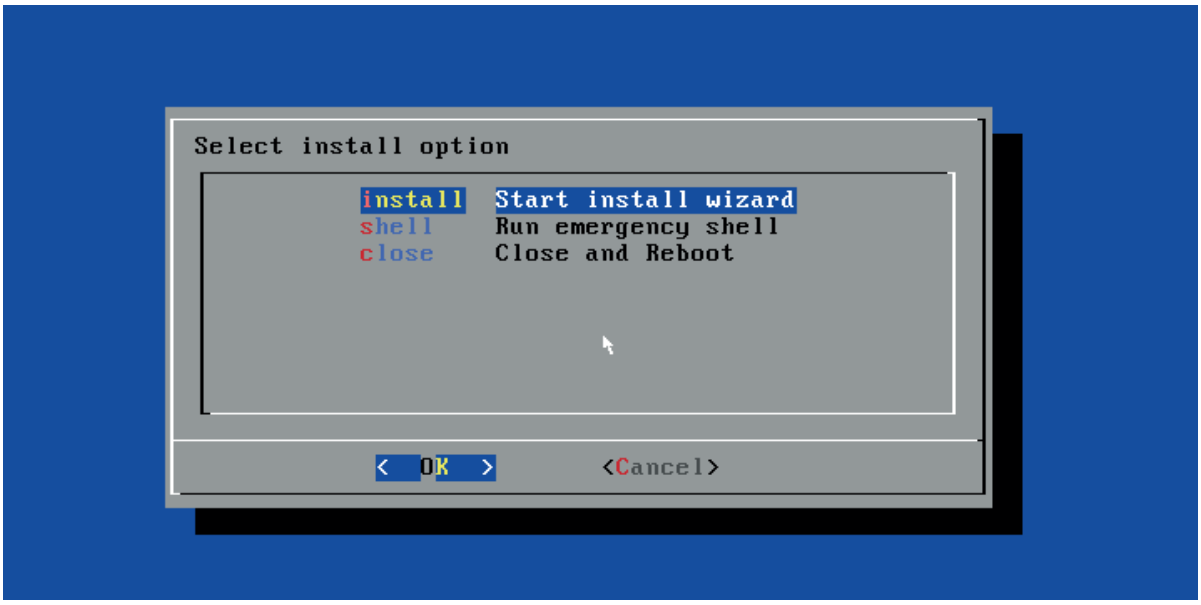
thin-client wizard will then begin to build the "**net/isc-dhcp42-server**" port. You will most likely only need the default port options, and can hit enter at any confirmation screens that appear. Once the port has finished installing, the thin-client setup will prompt you again for the PXE network interface to use:

```
Setting up system for PXE booting...
What NIC do you wish DHCPD to listen on? (I.E. re0)
nic)
```

Enter the interface name of the network card you wish to use as the PXE interface. This interface will be running the DHCP server, and should not be connected to an existing network with another DHCP server running. The wizard will then finish up the configuration for PXE booting, and display a message similar to this:

```
To perform system installations, place your custom pc-sysinstall scripts in:
/usr/home/thinclient/installscripts

An example script is provided in the above directory

For unattended installations, save your pc-sysinstall script as:
/usr/home/thinclient/installscripts/unattended.cfg

Your system is now setup to do PXE booting!
```

Your initial PXE setup is now complete! You may now try to PXE boot a client connected on the network interface you specified. If the client boots successfully, you should be presented with an installation screen like below:



By selecting the "install" option you will be presented with the example configuration option,

which can be used to install a basic FreeBSD system. (Below we will look at adding your own) By selecting it, you will be asked to confirm one last time that you wish to perform the installation using this configuration, and then the installation will proceed. In addition to performing installs, you can also access an emergency shell prompt. This can be very useful if you have a system which can no longer boot, and you wish to access the disk or attempt repairs of some kind.

Customizing the installation scripts

With your initial PXE configuration finished, you will now most likely want to create your own installation scripts. The thin-client wizard creates an example installation script in the **/usr/home/thinclient/installscripts/** directory, which is where you will want to place your custom scripts as well. Any scripts placed in this directory will be selectable as an installation option on the PXE client.

With the location of the install scripts in hand, lets now take a look at the provided **pc-sysinstall.example** file in that directory.

```
# Sample configuration file for an installation using pc-sysinstall

installMode=fresh
installInteractive=no
hostname=examplesystem

# Set the disk parameters
disk0=ada0
partition=all
bootManager=none
commitDiskPart

# Setup the disk label
# All sizes are expressed in MB
# Avail FS Types, UFS, UFS+S, UFS+J, ZFS, SWAP
# Size 0 means use the rest of the slice size
disk0-part=UFS+SUJ 1000 /
disk0-part=SWAP 2000 none
disk0-part=UFS+SUJ 0 /usr
commitDiskLabel

# Set if we are installing via optical, USB, or FTP
installType=FreeBSD
installMedium=local
localPath=/installarchive
packageType=tar
installFile=fbsd-release.txz

# Set the root pass
rootPass=root

# Setup our users
```

```
# Setup our users
userName=kris
userComment=Kris Moore
userPass=kris
userShell=/bin/csh
userHome=/home/kris
userGroups=wheel,operator
commitUser
```

The included comments in the example make most of the functionality fairly self-explanatory. However there are a few sections we will take a closer look at customizing. The first place to note is the "**installMedium=local**" and "**localPath=/installarchive**" options. The pc-sysinstall backend supports a number of methods of fetching the archive files for installation. In this case we are using the local file "**fbsd-release.txz**", stored in **/usr/home/thinclient/installarchive**. This directory will appear to PXE clients as **/installarchive**, allowing you on the host OS to supply your own tar archives of any FreeBSD / PC-BSD release or configuration you wish.

PRO Tip!

By using the **Warden** utility in PC-BSD / TrueOS, it is possible to setup your install environment inside a jail on the host system. Then when you are satisfied with the configuration, you can stop the jail and create the install archive on the host system with the "**tar**" command as shown below:

 # tar cvJf /usr/home/thinclient/installarchive/myarchive.txz -C /usr/jails/<jailip> .

After selecting your installation archive options you will most likely want to customize the disk layout of the new system. In the example provided we are performing a very simple full-disk installation to the disk ada0, using UFS with Soft Updates + Journaling. The resulting installation will contain a 1000MB root partition, a 2000MB swap, and use the rest of the disk space for /usr as seen below.

```
# Set the disk parameters
disk0=ada0
partition=all
bootManager=none
commitDiskPart

# Setup the disk label
# All sizes are expressed in MB
# Avail FS Types, UFS, UFS+S, UFS+J, ZFS, SWAP
# Size 0 means use the rest of the slice size
disk0-part=UFS+SUJ 1000 /
disk0-part=SWAP 2000 none
disk0-part=UFS+SUJ 0 /usr
commitDiskLabel
```

In addition to UFS support, the pc-sysinstall backend can also handle more advanced ZFS disk layouts in a similar manner:

```
# Setup the disk label
# All sizes are expressed in MB
# Avail FS Types, UFS, UFS+S, UFS+J, ZFS, SWAP
# Size 0 means use the rest of the slice size
disk0-part=ZFS 0 /,/usr,/var,/root (mirror: ada1)
commitDiskLabel
```

In the example above we have changed our disk layout from UFS to a single ZFS pool using the entire disk, and added the disk drive ada1 to the resulting zpool as a mirror device. While this example is fairly simplistic much more complex layouts can be created using ZFS, including raidz, dataset options and more.

PRO Tip!

Since every single install with pc-sysinstall is fully scripted, a good way to experiment with alternative ZFS layouts is using a virtualization tool such as VirtualBox and your PC-BSD installation DVD. By running the install GUI, you can customize your ZFS disk layout in an easy-to-use manner inside a virtual machine. Then when the installation is finished, a copy of the saved pc-sysinstall configuration file will be saved onto the installed system at /root/pc-sysinstall.cfg.

One last feature to look at with the thinclient utility is the ability to perform completely unattended installations. If you plan on only doing fully-automated installations via this PXE interface, you may do so simply by creating a configuration script named "**unattended.cfg**" and placing it in the **/usr/home/thinclient/installscripts/** directory alongside the example. When a client first boots it will check for this **unattended.cfg** file, and if found it will automatically use it for installation. Some caution should be taken when using this method, since simply plugging a PXE booting system into the wrong LAN cable could cause it to be re-installed.

Scalability

With the thin-client server now setup, one of the most often asked questions is that of scalability. How many systems can be installed at a time using this method? The answer of course will depend greatly upon your hardware setup. At its core the PXE server is just a large file-server, running DHCP for clients, and using NFS to share files. This is not very CPU intensive and will rather be more limited by the speeds of your disk drives, and networking systems. In order to scale up to doing larger volumes of system installations simultaneously, it may be wise to install your PXE server using multiple disk drives in a ZFS Mirror or RAIDZ configuration. On a system with RAM to spare it is even be possible to setup the **/usr/home/thinclient/installarchive/** directory as a ramdisk, removing the physical hard-disk bottleneck all together. This could be done using a command such as:

```
# mount -t tmpfs tmpfs /usr/home/thinclient/installarchive/
```

Since this is now running in system memory, you will need to take care that files placed here are not lost after a system reboot. From a configuration stand-point another limiting factor may be the

DHCP daemon itself. The default location for its configuration is **/usr/local/etc/dhcpd.conf**. The initial values provided by pc-thinclient will set up DHCP to serve up IP addresses starting at 192.168.2.100, and ending at 192.168.2.199. If you wish to support booting more clients these values may be adjusted to your particular needs.

Conclusion

We have briefly looked at some of the ways PC-BSD and TrueOS 9.1 make it easy to setup PXE booting and perform rapid deployment of FreeBSD based systems. We have also just begun to scratch the surface of the type of installations that can be performed with the pc-sysinstall backend. If you want to do further research into more complex installations please take a look at the included wiki references on pcbsd.org. We also will be happy to continue the discussion with you on the PC-BSD mailing lists or forums.

References

Wiki Articles:
http://wiki.pcbsd.org/index.php/Thin_Client
http://wiki.pcbsd.org/index.php/Creating_an_Automated_Installation_with_pc-sysinstall

Mailing Lists & Forums
http://lists.pcbsd.org
http://forums.pcbsd.org/

Perfect(ing) hashing in NetBSD

Jörg Sonnenberger

March 16, 2013

Abstract

Hash tables are one of the fundamental data structures used in the different parts of an Operating System from on-disk databases to directory caches in the kernel. To provide the desired performance characteristics, keeping the collision rate minimal is crucial. If the construction of the hash function guarantees that no collisions occur, it is a Perfect Hash Function. This paper gives a short introduction into the algorithms developed in the last 20 years for this task. It looks at the uses in NetBSD 6 like *nbperf* and the constant database *cdb* as well as work-in-progress code for routing and firewalling.

1 Introduction

Hash tables are data structures used to implement associative arrays, i.e. they provide a mapping of arbitrary complex keys to values or indices. They are one of the oldest data structures for this purpose and since their discovery in 1953, they have been studied extensively and can be found in many programs and libraries. The wide spread use is a result of the SMP friendliness and efficiency as ideally, inserts, lookups and removals are all constant time operations.

The core of every hash table implementation is the hash function. Typical examples are using the remainder of the division by a prime or Bernstein's string hash $h(x[0..n]) = 33 \times h(x[0..n-1]) + x[n]$ with $h(0) = 5381$[1]. All simple hash functions share one important property: certain input keys are mapped to the same index. This is called a hash collision and requires special handling. Techniques for dealing with them include linked lists between elements with the same hash, repeating the hashing with an additional counter as argument to find a different position or resizing the hash table.

In the last years, a number of so-called complexity attacks[5] have been published where an attacker explicitly creates hash collisions to force the target to waste time by slowing down the operations from expected constant time to linear in the number of keys. One way to address the complexity attacks is to move from hash tables to balanced trees at the cost of logarithmic complexity for most operations. The other alternative is deploying randomised hash functions.

Randomised hash functions are also the building block for more powerful hash table schemes: perfect hash functions. A Perfect Hash Function (PHF) is constructed in such a way, that any two keys of a known input set are mapped to different indices, i.e. that no hash collisions can exist. This makes them a perfect match for applications with a (mostly) fixed key set. If the PHF also maps the n keys to $0..n-1$, it is called a Minimal Perfect Hash Function (MPHF). The most well known program for creating PHF and MPHF is GNU gperf. It has been used for the keyword recognition in compilers like GCC.

This paper introduces the important developments in this area of research since 1990. As practical applications the *nbperf* program in NetBSD 6 is presented as well the new constant database *cdb*. An outlook to further work for using perfect hashing in the routing table and in NPF is also presented.

2 New algorithms for Perfect Hash Functions

This section looks at the early algorithms for PHF construction and challenges faced. It introduces the most noticable modern algorithms developed since 1990 and how they work.

The best algorithmic choice for a specific application depends on a number of common factors:

- Does the hash function preserve the key order? If it does, integration is easier as the existing table structures can be reused and the hash function works as additional secondary index.
- How much space does the hash function needs per key?
- What computations are needed for the hash function?
- If the algorithm constructs a non-minimal PHF, what key density can it achieve?

[1] This hash is also known as DJBX33A.

Comparing the individual properties and weighting them according the specific needs results in the correct choice.

2.1 Systematic construction of Perfect Hash Functions until 1990

Different authors have investigated construction mechanisms for Perfect Hash Functions since the invention of the hash table. Bostic published the predecessor of GNU gperf around 1984. Knuth discussed examples in The Art Of Computer Programming. A beautiful construction can be found in Pearson's paper "Fast Hashing of Variable-Length Text Strings" from 1990[7].

It is based on an 8-bit permutation table and traversed according to the XOR combination of the last hash value and the current input character. The design looks very similiar to the RC4 stream cipher. Pearson's paper explicitly discusses ways to systematically search for a permutation, but also the limitations. A trivial case of a problematic input is given where the target range has to be shifted to produce a MPHF and it can be easily seen that the algorithm doesn't scale very well with the number of keys.

The (lack of) scalability of the construction mechanism is a fundamental issue of the early approaches. If they work, they tend to provide very fast and moderately compact hash functions. For key sets larger than a few dozen keys at most, the construction will generally fail or require exponential construction time. For this reason, perfect hashing hasn't been deployed but for compiler construction for a long time.

2.2 Czech, Havas and Majewski's Minimal Perfect Hash Functions

The CHM construction was published in 1992[2]. It is one of the earliest, if not the earliest, expected linear time algorithm for the construction of MPHFs. Expected linear time in this case means that the algorithm uses randomised graphs with certain properties and try again, if a specific choice doesn't fit. Each run of the algorithm takes linear time and the chance of requiring more than one run is very low.

The resulting hash function has the useful property of being order preserving. This means that the hash function preserves the input order by mapping the n-th key is mapped to $n-1$. In practise this makes the hash function very easy to fit into existing code as e.g. tables mapping actions to handlers don't have to be reordered to fit the hash.

The algorithm depends one two central concepts. The first concept is the creation of a random graph by using two or more independent random hash functions. This graph has n edges and $m = cn$ vertices (for some $c \geq 1$). Each edge is created by taking the value of the chosen hash functions modulo m. If the graph is acyclic, the algorithm continues. Otherwise, another try is made with a different choice of random hash functions. If the constant c is chosen correctly, the graph is acyclic with a very high probability. When using two hash functions, c must be at least 2. When using three hash functions, c must be at least 1.24.

The second concept by Czech et al. is to continue by assigning a number to every vertex, so that the key number corresponding to each edge is the sum of the vertices of the edge modulo m. The initial value of all vertices is 0. An edge is chosen, so that one of the vertices has a degree of one. Without such an edge, the graph would contain a cyclic. Subsequently the value of the other vertex is updated, so that the edge fulfills the desired sum and the edge is removed afterwards.

The result requires storing m integers between 0 and $n-1$ as well as the parameters for the chosen random hash functions. It is the best known construction for order-preserving MPHF. The result hash function takes the compution of the two (three) chosen random hash functions, a modulo operation for each, two (three) table lookups, summing up the results and computing another modulo. The modulo operation itself can be replaced by two multiplications by computing the inverse. The storage requirement is at least $m \log_2 n$, but typically $m\lceil \log_2 n \rceil$ bits per key.

2.3 Botelho, Pagh and Ziviani's Perfect Hash Functions

The BPZ construction was published in 2007[1] and is very similar to the CHM algorithm. The main difference is the way numbers are assigned to the vertices. For BPZ, each edge is represented by one of its vertices. The sum of the corresponding numbers modulo 2 (3) gives the representative. As such the resulting hash function is not minimal by itself. At the same time, it requires much less space. When using three independent random hash functions, a value between 0 and 2 must be stored for all m vertices. One simple encoding stores five such values per byte ($3^5 = 243 < 256$). Using $c = 1.24$, this requires $1.24 \times n \times 8/5$ 1.98 bit storage per key with 24entries.

To obtain a MPHF, some post-processing is needed. The PHF can be reduced to a MPHF using a counting function. This function returns for a given index k how many "holes" the PHF has until k. This can be represented as bit vector with partial results ever so often memorised to keep the O(1) computation time. Careful choices require two additional table lookups and one 64 bit population count with a storage requirement of approximatily

2.79 bits per key using three random hash functions.

2.4 Belazzougui, Botelho and Dietzfelbinger's Perfect Hash Functions

"Hash, displace, and compress" or sort CHD was published in 2009[3] and is currently the algorithm known to create the smallest PHF. Using three random hash functions, a PHF can be constructed with 1.4 bit storage per key. The construction itself and the resulting hash function is a lot more complicated than BPZ though, so no further details will be provided.

3 nbperf

nbperf was started in 2009 to provide a replacement for GNU gperf that can deal with large key sets. cmph [2] was investigated for this purpose, but both the license and the implementation didn't fit into the NetBSD world. *nbperf* currently implements the CHM algorithm for 2-graphs and 3-graphs as well as the BPZ algorithm for 3-graphs. CHD wasn't available when the work on *nbperf* started and hasn't been implemented yet.

The output of *nbperf* is a single function that maps a pointer and size to the potential table entry. It does not validate the entry to avoid duplicating the keys or adding requirements on the data layout. If the callee of the hash function already knows that it has a valid key, it would also add overhead for no reason.

The most important option for *nbperf* is the desired construction algorithm. This affects the size and performance of the hash function. Hash functions using CHM are much larger. The 2-graph version requires two memory accesses, the 3-graph version three. The actual cache foot print depends on the number of keys as the size of the entries in the internal data array depends on that. For BPZ, the entries are much smaller, but some additional overhead is needed to provide a minimal hash function. As mentioned earlier, CHM is easier to use in applications, since it preserves the key order.

The following test case uses the Webster's Second International dictionary as shipped with NetBSD and the shorter secondary word list. They contain 234977 and 76205 lines. Each line is interpreted as one input key. *nbperf* is run with the "-p" option to get repeatable results. The "tries" column lists the number of iterations the program needed to find a usable random graph.

[2] http://cmph.sourceforge.net

Input	Algorithm	Tries	Run time in s
web2	CHM	1	0.58
	CHM3	39	0.85
	BPZ	11	0.51
web2a	CHM	12	0.35
	CHM3	7	0.17
	BPZ	18	0.16

The resulting code for CHM can be seen in listing 1

The "mi_vector_hash" function provides an endian-neutral version of the Jenkin's hash function[4]. The third argument is the chosen seed. The modulo operations are normally replaced by two multiplications by the compiler.

At the time of writing, two applications in NetBSD use *nbperf*. The first user was the new *terminfo* library in NetBSD 6 and uses it for the key word list of *tic*. The second example is *apropos*, which contains a stop word list (i.e. words to be filtered from the query). This list is indexed by a Perfect Hash Function.

Further work for *nbperf* includes investigating simpler random hash function families to provide results with performance characteristics similar to GNU gperf's hashes. An implementation of the CHD algorithm is also planned.

4 The NetBSD constant database

NetBSD used the Berkeley Database to provide indexed access for a number of performance sensitive interfaces. This includes lookups for user names, user IDs, services and devices. The Berkeley Database has a number of limitations for this applications, which opened up the question of how to address these:

- Lack of atomic transactions,
- Database size overhead,
- Code complexity,
- Userland caching on a per-application base,

The first item ensures that most use cases in the system deploy a copy-on-write scheme for the (rare) case of modifications. It also means that any program has to be able to regenerate the database content after a disk crash.

The second item matters for embedded systems as it limits what database can be shipped pre-built. The third item is somewhat related, if the disk image doesn't require write support, it still can't leave the relevant code out.

The last item increases the memory foot print and reduces sharing data. It also adds overhead for

Listing 1: CHM example

```
#include <stdlib.h>

uint32_t
hash(const void * __restrict key, size_t keylen)
{
  static const uint32_t g[469955] = {
    /* ... */
  };
  uint32_t h[3];

  mi_vector_hash(key, keylen, 0x00000000U, h);

  return (g[h[0] % 469955] + g[h[1] % 469955]) % 234977;
}
```

multi-threaded applications as the library has to avoid concurrent access to the internal block cache.

NetBSD has imported SQLite, which provides a higher level library including transaction support. This doesn't help with the items above, especially the third. A new library was created to complement SQLite: the constant database. This format provides a read-only storage layer with deterministic access time, lock-free operation and based on memory mapped files to fully utilize the kernel cache.

The constant database (CDB) consists of two parts. The first part is the value storage, allowing access to each record using the record number. It can be used to iterate over the content or to link entries together. The second part provides a Perfect Hash Function for additional key based access. The keys are not stored on disk, so the application is responsible for doing any validation. For most file formats, the key is part is of the record anyway and especially when using multiple keys for the same record, storing would increase the file size without justification. Key lookup requires one computation of mi_vector_hash for the given key and reading three locations in the on-disk hash description. Worst case is thus three page faults with loading the blocks from disk. That gives the index and one more access the actual data offset. The result is a pointer, size pair directly into the memory mapped area. Looking up the same key twice therefore doesn't result in any additional IO nor does it require any traps, unless the system is low on memory.

In terms of code complexity, the CDB reader adds about 1.6KB to libc on AMD64 and writer around 4.3KB. As the database files are often the same size or smaller than the corresponding text sources, dropping the text versions can result in an overall decrease in size.

For NetBSD 6 the device database, the service database and libterminfo use the new CDB format. The resulting databases are typically less than one fourth of the size of the corresponding Berkeley DB files. The creation time has also improved. Further work is required to convert the remaining users in libc, but also to provide access in other programs like Postfix.

5 Perfect hashing for the route lookup

Cleaning up the routing code and investigating new data structures and/or implementations is on-going work in NetBSD. David Young provided the first major part for this by isolating access to the radix tree and hiding it behind a clean interface. The pending work moves the preference selecting (i.e. which of two routes with the same netmask is chosen) and the special case of host routes out of the radix tree into the generic layer. This changes will allow replacing the old BSD radix tree with less generic, but faster code. It also makes it possible to switch the lookup data structure for the fast path.

The most interesting alternatives are compressed tries[6] (out of the scope of this paper) and multi-level hashing[8]. Multi-level hashing is based on the idea of performing the CIDR[3] lookup as binary search on the possible prefix lengths. For IPv4, this could mean starting with looking for /16 routes and depending on match or not, continue with /8 or /24 entries. This requires adding markers for more specific routes to direct the search.

Consider the following routing table:

[3] Classless Inter-Domain Routing

Destination network	Gateway
0/0	192.168.0.10
127/8	127.0.0.1
192.168.0/24	192.168.0.2
192.168.1/24	192.168.0.1
10/8	192.168.0.1
10.0.10/24	192.168.0.5
10.192/12	192.168.0.6
11.192/12	192.168.0.7

A search for 10.0.10.1 will start by looking for 10.0/16 in the hash table to be constructed. No such route exists, but the search has to continue with larger prefix length to find the correct entry 10.0.10/24. For this purpose, a marker has to be added with entry 10.0/16 and a reference to 10/8. The reference avoids the need for backtracking, i.e. when searching for 10.0.11.1. They can either reference the covering route or copy the corresponding gateway, depending on the granularity of traffic accounting. With the additional marker entries, the following content of the hash table is enough:

Destination network	Type	Data
0/0	GW	192.168.0.10
127/8	GW	127.0.0.1
192.168/16	R	0.0.0.0/0
192.168.0/24	GW	192.168.0.2
192.168.1/24	GW	192.168.0.1
10/8	GW	192.168.0.1
10.0/16	R	10/8
10.0.10/24	GW	192.168.0.5
10.192/12	GW	192.168.0.6
11/8	R	0/0
11.192/12	GW	192.168.0.7

For this specific case, three additional entries are enough as the marker for 10.192/12 is 10/8 and that's already present as route. Using perfect hashing ensures a predictable lookup cost as it limits the number of expensive memory accesses. Using the BPZ algorithm with a 2-graph and no post-filtering means a hash table utilisation of 50% and approximately 2 bit per key storage for the hash function itself. It is possible to use a single hash table for all entries or to use a /separate table for each prefix length. The latter allows using 64 bit per entry in case of IPv4 (32 bit network, 32 bit as the next-hop identifier) and between 64 bit and 160 bit for IPv6. Even for a core router in the Default Free Zone, 100,000 entries and more fit into the L3 cache of modern CPU.

The downside of using perfect hashing is the construction time. Investigations have to be performed on how critical the resulting update time is for busy routers.

Further optimisations can be deployed. The optimal branching is often not a static binary search, so storing hints for the next level to look at can be useful. Published research by Waldvogel et al. suggests that the average number of hash table probes can be much less than 2, when chosing the correct order. The lookup table itself can avoid redundant entries, i.e. if a more specific router and the immediate covered route have the same next-hop. This implies less precise accounting though.

6 Summary

Algorithms like CHM, BPZ and CHD provide a fast, practical construction of Perfect Hash Functions. This makes it possible to use them in different fields from performance critical read-mostly data structures, like the routing tables, to size sensitive on-disk databases. NetBSD 6 is the first BSD to use them in the real world and more areas will be covered in the future.

Areas for open research and practical implementations outlined in this paper include finishing the implementation in the network stack and finding fast simple random hash functions to replace the remaining use cases of GNU gperf.

References

[1] F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *In Proc. of the 10th Intl. Workshop on Data Structures and Algorithms*, pages 139–150. Springer LNCS, 2007.

[2] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43:257–264, 1992.

[3] F. Botelho D. Belazzougui and M. Dietzfelbinger. Hash, displace, and compress. In *Algorithms – ESA*, pages 682–693, 2009.

[4] B. Jenkin. Hash functions. *Dr. Dobbs Journal*, September 1997.

[5] A. Klink and J. Wälde. Efficient denial of service attacks on web application platforms, 2011.

[6] S. Nilsson and G. Karlsson. IP-Address lookup using LC-tries, 1998.

[7] P. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33:677–680, June 1990.

[8] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *Proc. ACM SIGCOMM*, pages 25–35, 1997.

The bhyve Operator's Manual

Michael Dexter

AsiaBSDCon 2013

OVERVIEW

bhyve is a legacy-free Type-2 hypervisor for FreeBSD that was imported into the mainline FreeBSD development repository in January of 2013 with svn revision r245652. A hypervisor allow for the operation of one or more guest operating systems within a host operating system.

As a legacy-free hypervisor, a bhyve host requires the Extended Page Tables (EPT) feature found on "Nehalem" and newer generations of Intel processors. This requirement eliminates the need for memory management routines that are traditionally implemented in software and yields virtually bare metal guest performance. A bhyve guest requires VirtIO network and block devices, which were already available in FreeBSD 8-STABLE, 9-STABLE and 10-CURRENT at the time of bhyve's import. If these two requirements are satisfied, the bhyve host and guests will operate in the established FreeBSD manner.

HARDWARE REQUIREMENTS

The presence of the Extended Page Table (EPT) feature can be determined by examining the host's `demesg(8)` output for the presence of the `POPCNT` (POP Count) feature as the two are coupled but not related. Established dynamic memory and storage requirements apply otherwise with the caveat that there is a 1:1 relationship between the deduction of dynamic memory from the host and its allocation to guests.

SOFTWARE REQUIREMENTS

A FreeBSD 10-CURRENT system from svn revision r245652 onward will include all of the necessary bhyve host components: the `vmm(4)` kernel module, the `libvmmapi` library and the `bhyveload(8)`, `bhvye(8)` and `bhyvectl(8)` utilities.

A suitable FreeBSD 8-STABLE, 9-STABLE or 10-CURRENT guest can exist in a disk image or any valid storage device and only requires a modified `/etc/ttys` entry to work. All other options can be specified at runtime at the loader prompt. Permanent configuration changes however are generally desired and will be demonstrated.

Permanent `/etc/ttys` configuration (can be appended):

```
console "/usr/libexec/getty std.9600"   vt100   on secure
```

Boot time or permanent `/etc/fstab` configuration for a MBR-partitioned device:

```
# Device                Mountpoint      FStype  Options         Dump    Pass#
/dev/vtbd0s1a           /               ufs     rw              1       1
```

or for a GPT-partitioned device:

```
# Device                Mountpoint      FStype  Options         Dump    Pass#
/dev/vtbd0p1            /               ufs     rw              1       1
```

Example runtime or permanent `/etc/rc.conf` networking configuration:

```
ifconfig_vtnet0="DHCP"
```

Depending on how your guest is built, the `/boot/loader.conf` may require:

```
virtio_load="YES"
if_vtnet_load="YES"
virtio_pci_load="YES"
virtio_blk_load="YES"
```

BHYVE OPERATION

At a minimum, a bhyve host requires that the `vmm.ko` kernel module be loaded:

```
su kldload vmm
```

To enable guest networking, load the `if_tap.ko` kernel module and create a `tap(4)` interface (`tap0` on `em0` as an example):

```
su kldload if_tap
su ifconfig tap0 create
su ifconfig bridge0 addm tap0 addm em0 up
su ifconfig tap0 up
```

The host and guest are now configured for bhyve operation. The bhyve guest bootstrap process is a two-stage operation not unlike that of the host. The `bhyveload(8)` utility loads the guest kernel into memory and the `bhyve(8)` utility executes it. These commands should be run in tight sequence with identical memory, storage device and guest name parameters.

All configuration parameters forward will be highly site-specific.

```
su /usr/sbin/bhyveload -m 256 -M 0 -d mydiskimage myguestname

su /usr/sbin/bhyve  -a -A -m 256 -M 0 -I -H -g 0 \
-s 0:0,hostbridge \
-s 1:0,virtio-net,tap0 \
-s 2:0,virtio-blk,mydiskimage \
-S 31,uart,stdio \
myguestname
```

Assuming a successful boot, the guest should behave like a standard FreeBSD system and '`shutdown -p now`' will cleanly shut it down.

If you wish to terminate a guest at the loader prompt, simply type '`quit`'.

Executing guests can be viewed by name on the host system with:

```
su ls /dev/vmm
```

The above bootstrap procedure will not free the memory allocated to the guest but it can be freed using `bhyvectl(8)`:

```
su /usr/sbin/bhyvectl --vm=myguestname --destroy
```

PRAGMATIC BHYVE GUEST OPERATION

The above procedures do not address the subjective matter of populating and configuring the guest userland or automating the guest operation lifecycle.

Two efforts exist to automate the building and operation of bhyve guests.

bhyve developer Neel Natu provides a script named '`vmrun.sh`' that works in conjunction with a FreeBSD 10-CURRENT installation iso image named '`release.iso`' to automate the creation of a bootable disk image named '`diskdev`' that can be populated using the standard FreeBSD installer. This script provides an 8G disk image and 512M of dynamic memory by default but these parameters can be easily modified at runtime or by modifying the script.

The '`vmrun.sh`' script requires only a guest name such as '`vm1`':

```
su vmrun.sh vm1
```

Mr. Natu's script and instructions can be obtained from:

```
http://people.freebsd.org/~neel/bhyve/vmrun.sh
http://people.freebsd.org/~neel/bhyve/bhyve_instructions.txt
```

A suitable 10-CURRENT 'release.iso' image can be retrieved from:

http://ftp4.us.freebsd.org/pub/FreeBSD/snapshots/amd64/amd64/ISO-IMAGES/10.0/

Note that the versioning prefix must be removed for the iso to work.

Alternatively, the bhyve.org web site provides a series of scripts that facilitate the creation and operation of bhyve guests in a highly-customizable manner.

While evolving, at the time of writing these scripts include:

```
0-make-softdevice.sh        Create and mount a disk image or zvol
1-format-device.sh          Format the disk image or zvol
1-format-zvol-gpt.sh        Format a zfs zvol with a GPT layout
2-install-guest.sh          Populate the disk image with the OS
3-host-prep.sh              Prepare the host for bhyve and networking
4-boot-guest.sh             Boot the guest
5-cleanup-guests.sh         Clean up after the guest
mount-diskdev.sh            Mount a specified disk image
```

In addition to the basic functionality provided by 'vmrun.sh', these scripts create and format disk images in MBR and GPT format, format zvol, iSCSI and hardware devices, populate FreeBSD 8-STABLE, 9-STABLE and 10-CURRENT userlands from the official mirrors and facilitate guest operation.

Any and all jail(8) and NanoBSD configuration strategies should be applicable to bhyve and the creation of "thick" and "thin" bhyve hosts and guests.

THE BHYVE DEVELOPMENT ENVIRONMENT

As a native FreeBSD 8-STABLE and later on FreeBSD 10-CURRENT computing environment, bhyve provides the ideal infrastructure for all categories of FreeBSD development. The hard partitioning provided by the vmm virtual machine manager allows the "unstable" FreeBSD 10-CURRENT environment to be used for "stable" FreeBSD 8 and 9 development with far fewer potential risks than emulated or jailed environments. Common development nuisances such as library incompatibilities are eliminated by the virtual machine divide.

The inclusion of dtrace(1M) in FreeBSD 10-CURRENT provides an unprecedented level of system introspection that should prove invaluable to FreeBSD, bhyve and future foreign guest operating systems.

Some rudimentary DTrace examples:

```
su kldload dtraceall
```

```
su dtrace -n 'io:::start /execname == "bhyve"/ { @[ustack()] = count(); }'

              libc.so.7`0x800ed3fda
              bhyve`pci_vtblk_qnotify+0x59
              bhyve`pci_vtblk_write+0x220
              bhyve`pci_emul_io_handler+0x16e
              bhyve`emulate_inout+0x197
              bhyve`vmexit_inout+0x155
              bhyve`vm_loop+0x118
              bhyve`fbsdrun_start_thread+0x87
              libthr.so.3`0x800c53413
              505
```

The `sysutils/DTraceToolkit` port provides FreeBSD-specific DTrace scripts.

```
su /usr/share/dtrace/toolkit/hotuser -p `pgrep -n bhyve`

Sampling... Hit Ctrl-C to end.
^C
FUNCTION                                                  COUNT   PCNT
bhyve`vm_loop                                                 1   1.8%
libvmmapi.so.5`0x800838920                                    1   1.8%
libthr.so.3`0x800c59491                                       1   1.8%
libthr.so.3`0x800c58df1                                       1   1.8%
libvmmapi.so.5`0x800838950                                    1   1.8%
libthr.so.3`0x800c5992f                                       1   1.8%
libc.so.7`0x800f83bb3                                         1   1.8%
bhyve`emulate_inout                                           1   1.8%
libc.so.7`0x800ed3fd0                                         1   1.8%
libthr.so.3`0x800c594c1                                       1   1.8%
bhyve`pci_vtblk_write                                         1   1.8%
bhyve`pci_vtblk_proc                                          1   1.8%
libvmmapi.so.5`0x800838a78                                    2   3.6%
libvmmapi.so.5`0x800838a2e                                    5   8.9%
bhyve`vmexit_inout                                            6  10.7%
libc.so.7`0x800f8875a                                        31  55.4%
```

```
su /usr/share/dtrace/toolkit/hotkernel

Sampling... Hit Ctrl-C to end.
^C
FUNCTION                                                COUNT   PCNT
vmm.ko`vm_guest_msrs                                        1   0.0%
vmm.ko`vcpu_set_state                                       1   0.0%
vmm.ko`vmx_setreg                                           1   0.0%
vmm.ko`vm_nmi_pending                                       1   0.0%
vmm.ko`vm_get_register                                      1   0.0%
vmm.ko`lapic_intr_accepted                                  1   0.0%
vmm.ko`vm_lapic                                             1   0.0%
vmm.ko`vlapic_op_mem_read                                   2   0.0%
vmm.ko`lapic_mmio_write                                     2   0.0%
vmm.ko`vlapic_intr_accepted                                 2   0.0%
vmm.ko`vm_set_register                                      2   0.0%
vmm.ko`lapic_pending_intr                                   3   0.0%
vmm.ko`vmm_fetch_instruction                                3   0.0%
vmm.ko`vmmdev_ioctl                                         3   0.0%
vmm.ko`vmmdev_ioctl                                         3   0.0%
vmm.ko`vm_exitinfo                                          3   0.0%
vmm.ko`vcpu_stats                                           3   0.0%
vmm.ko`vmm_emulate_instruction                              4   0.0%
vmm.ko`vmx_getreg                                           4   0.0%
vmm.ko`lapic_timer_tick                                     5   0.0%
vmm.ko`vm_gpa2hpa                                           7   0.0%
vmm.ko`vlapic_pending_intr                                  7   0.0%
vmm.ko`vlapic_op_mem_write                                  7   0.0%
vmm.ko`vmm_decode_instruction                              19   0.0%
vmm.ko`vmcs_read                                           20   0.0%
vmm.ko`ept_vmmmap_get                                      21   0.0%
vmm.ko`vlapic_timer_tick                                   29   0.0%
vmm.ko`vm_run                                              30   0.0%
vmm.ko`restore_guest_msrs                                  32   0.0%
vmm.ko`restore_host_msrs                                   42   0.0%
vmm.ko`vlapic_update_ppr                                  142   0.1%
vmm.ko`vmx_run                                          33275  16.3%
kernel`acpi_cpu_c1                                     168750  82.6%
```

In addition to the DTrace suite, the `bhyvectl(8)` command provides extensive bhyve-specific profiling information:

```
su bhyvectl --get-lowmem --vm=guest0
lowmem          0x0000000000000000/268435456

su bhyvectl --get-stats --vm=guest0
vcpu0
number of ticks vcpu was idle           5244097
number of NMIs delivered to vcpu        0
vcpu migration across host cpus         1186676
vm exits due to external interrupt      2229742
number of times hlt was ignored         0
number of times hlt was intercepted     2158532
vcpu total runtime                      288974908299
```

BHYVE REFERENCE

The following options are available to the `bhyveload(8)` and `bhyve(8)` commands:

```
Usage: bhyveload [-d <disk image path>] [-h <host filesystem path>] [-m
<lowmem>] [-M <highmem>] <vmname>
```

```
Usage: bhyve [-aehABHIP] [-g <gdb port>] [-z <hz>] [-s <pci>] [-S <pci>]
[-p pincpu] [-n <pci>] [-m lowmem] [-M highmem] <vm>
       -a: local apic is in XAPIC mode (default is X2APIC)
       -A: create an ACPI table
       -g: gdb port (default is 6466 and 0 means don't open)
       -c: # cpus (default 1)
       -p: pin vcpu 'n' to host cpu 'pincpu + n'
       -B: inject breakpoint exception on vm entry
       -H: vmexit from the guest on hlt
       -I: present an ioapic to the guest
       -P: vmexit from the guest on pause
       -e: exit on unhandled i/o access
       -h: help
       -z: guest hz (default is 100)
       -s: <slot,driver,configinfo> PCI slot config
       -S: <slot,driver,configinfo> legacy PCI slot config
       -m: lowmem in MB
       -M: highmem in MB
       -x: mux vcpus to 1 hcpu
       -t: mux vcpu timeslice hz (default 200)
```

FUTURE DEVELOPMENTS

bhyve contains experimental PCI device pass-through support and is scheduled to include:

- AMD Virtualization Extensions
- Foreign Guest Operating System Support
- ACPI Guest Suspend and Resume
- Thin Provisioning of Memory
- Generalization of CPUID Features for Guest Migratability
- Sparse Image Support such as QCOW, VDI and VMDK
- Porting to other Host Operating Systems

bhyve is arguably the most compelling FreeBSD development since `jail(8)` and continues FreeBSD's tradition of providing innovative multiplicity options to operators and developers. `jail(8)` and bhyve are by no means mutually-exclusive technologies and should provide value when used in conjunction with one another in parallel or via encapsulation. bhyve also promises to make the most of recent FreeBSD features such as the DTrace, the ZFS filesystem and FreeBSD’s virtual network stack. The fundamental point to remember about bhyve is that a FreeBSD bhyve system is simply a FreeBSD system that will fully leverage your FreeBSD administration and development experience.

OpenSMTPD : We deliver!

Éric Faurot
eric@openbsd.org

February 8, 2013

Abstract

In this paper we present the OpenSMTPD daemon: a simple, modern and portable mail server implemented using privilege-separation and messaging passing. Among different features, it comes with a notably simple configuration file format, and it offers very powerful deployment options.

We describe the internal organisation of the daemon in different processes with very specific roles. We examine the workflows for the main server tasks: enqueueing mails from external sources, delivering to the local users, relaying to external host and generating bounces. Finally, we discuss the server modularity, especially the table and backend APIs.

1 Introduction

Although several mail server implementations exist, they are not always satisfying for various reasons: complexity of the configuration, aging design which make it difficult to add support for new features, or inappropriate licensing terms.

The aim of the OpenSMTPD project is to provide a simple, robust and flexible implementation of the SMTP protocol, as defined in by RFC 5321[2] and other related RFCs. It is available under the liberal ISC license. It is being developed as part of the OpenBSD project. The development has started a few years ago, and has been very active in the last months. This paper presents an overview of the OpenSMTPD daemon design.

The first section will describe the configuration principles. In the next section we present the internal design of the daemon, based on privileged-separation and message-passing. The following section illustrates the workflow for the five main tasks : enqueueing, scheduling, delivering, relaying and bouncing. In the fourth section we will briefly discuss the flexibility of the daemon through filters and backends.

2 Features and Usage

2.1 Configuration

One of the most distinctive feature of OpenSMTPD is the simplicity of the configuration file, especially in comparison with other alternative mail server implementations. It can describe complex setups in a clear and concise manner, which makes the maintenance easy for the administrator, and helps to prevent misconfigurations with unexpected side-effects. Basic examples of configuration examples are described here.

The philosophy and syntax is largely inspired by the pf[1] configuration. It is based on the definition of a rule set: each *input* passes through the rule-matching engine to decide what action to take. Unlike pf, OpenSMTPD uses a first-match wins strategy.

```
listen on lo0
accept for local deliver to mbox
accept for any relay
```

The first line tells the server to listen for SMTP connections on the loopback interface. Then comes the rule-set definition. The first rule matches mails sent for the local machine, which means that the destination domain is the machine hostname (or localhost). The action to take is to deliver the mail to the user mbox. In this case, the user part of the destination address is expected to be a local user. The second rule matches all mails regardless of the destination domain, and use standard relaying for all domain.
In this example, the rules will implicitly reject mails not originating from the local machine. The updated configuration which follows allows to also listen on external interfaces (egress group), and accept mails from external sources for local deliveries using an alias file. Relaying to external domain is still there but only for mails originating from the local machine :

```
listen on lo0
listen on egress

table aliases file:/etc/mail/aliases

accept from any for local alias <aliases> deliver to mbox
accept for any relay
```

A common use-case is to route mail for outside through a specific host. The following example is a typical setup for a home user relaying its mail through his ISP, with authentication.

```
listen on lo0

table aliases file:/etc/mail/aliases
table secrets file:/etc/mail/secret

accept for local alias <aliases> deliver to mbox
accept for any relay via tls+auth://myisp@smtps.my.isp auth <secrets>
```

2.2 Administration

The administrator may interact with the daemon using a simple `smtpctl` control program. This program connects to the daemon control socket, turns the user request into specific internal control messages, forwards them to the control process and reports the result. Currently, the program supports the following operations:

- inspecting the current state of the queue: list of messages and scheduling information,
- pausing or resuming subsystems such as listeners for incoming mails, outgoing transfers or local deliveries,
- scheduling pending messages for immediate delivery,
- removing specific messages,
- retrieving various internal statistic counters,
- monitoring the server activity in real time.

2.3 Other features

Only a few simple cases have been described here, to give an overview of the philosophy behind the OpenSMTPD configuration. It supports many other features that are expected from any decent SMTP server implementation, among which :

- support for TLS and SMTPS,
- user authentication,
- backup server for a domain,
- primary and virtual domains,
- completely virtual users,
- local delivery to mbox, Maildir or external program.

The following example, adapted from a real-world configuration, shows a more complex setup using many of the OpenSMTPD features. The details won't be discussed here, but it shows how very powerful setups can be achieved, while the configuration file remains reasonably easy to read.

```
listen on lo0
#   Tag traffic from the local DKIM signing proxy
listen on lo0 port 10029 tag DKIM
#   Listen for incoming mail on standard smtp port and smtps port with ssl.
listen on egress ssl certificate mail.mydomain.org
#   Listen on a different port, enable tls and require auth
listen on egress port submission tls certificate mail.mydomain.org auth

table sources           { xxx.xxx.xxx.44, xxx.xxx.xxx.45 }
table aliases           "/etc/mail/smtpd/aliases"
table aliases2          "/etc/mail/smtpd/aliases2"
table pdomains          "/etc/mail/smtpd/primary-domains"
table vdomains          "/etc/mail/smtpd/virtual-domains"
table vusers            "/etc/mail/smtpd/virtual-users"
table bdomains          "/etc/mail/smtpd/backup-domains"

#   Deliver local mails with aliases
accept for local alias <aliases> deliver to maildir

#   Accept external mails for our primary domains.
accept from any for domain <pdomains> alias <aliases> deliver to maildir

#   Accept external mails for a specific primary domain with other aliases.
accept from any for domain "other.dom" alias <aliases2> deliver to maildir

#   Virtual domains, with a mapping from virtual users to real mailboxes
accept from any for domain <vdomains> virtual <vusers> deliver to maildir

#   Act as a backup server for the given domains, only relay to servers with
#   lower MX preference than the one specified.
accept from any for domain <bdomains> relay backup mx1.mydomain.org

#   Relay all signed traffic, using multiple source addresses (round robin).
accept tagged DKIM for any relay source <sources>
#   Relay outgoing mails through a DKIM signing proxy.
accept for any relay via smtp://127.0.0.1:10028
```

3 Internal Design

3.1 Fundamental concepts

The most important "object" around which the OpenSMTPD daemon is organised is the *envelope*. An envelope describes a message that is in transit within the server. It is always associated with a body, or content. Different envelopes can be associated with the same underlying content, to form what we refer to as a *message*, as illustrated in figure 1.

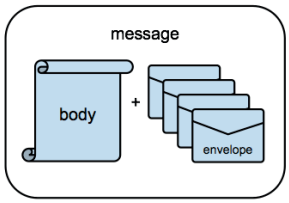


Figure 1: A message

An envelope has an origin, the sender, which is shared between all envelopes within the same message. An envelope also has a single destination, the recipient. A message sent to multiple recipient will have one envelope for each of them. The envelope also contains administrative information like the creation time, the expiry time, some internal flags, information on the enqueueing, etc. It also contains routing information, which describes how the associated content is to be transfered or delivered. An envelope is uniquely identified with its envelope ID, a 64 bits number that is generated internally.

Envelopes are created by and SMTP clients and the associated messages are stored (queued) temporarily in the server, until they can reach their next or final destination. There are three types of envelopes: local envelopes which are to be delivered to a local user on the host, relay envelopes, which must be transfered to another host, and bounce envelopes which are internal, and are meant to generate a report.

The OpenSMTPD server is implemented as a set of dedicated processes communicating using the ISMG(3) framework, and working together, exchanging envelopes, to drain accepted messages out. This design has several advantages. Defining specific roles and isolating functionalities in different processes lead to a design that is globally simpler to understand, and more reliable in the sense that it prevents layer violation. The processes can also run at different privilege levels, providing extra protection in case one of them (especially those directly exposed to the internet) is compromised.

Besides the short-lived ones that are temporarily forked, there are 9 processes :

- scheduler
- queue
- mail transfer agent
- mail delivery agent
- mail filter agent

- lookup agent
- smtp
- control
- parent

In the rest of the section we describe the different processes and their role in the system.

3.2 Process layout

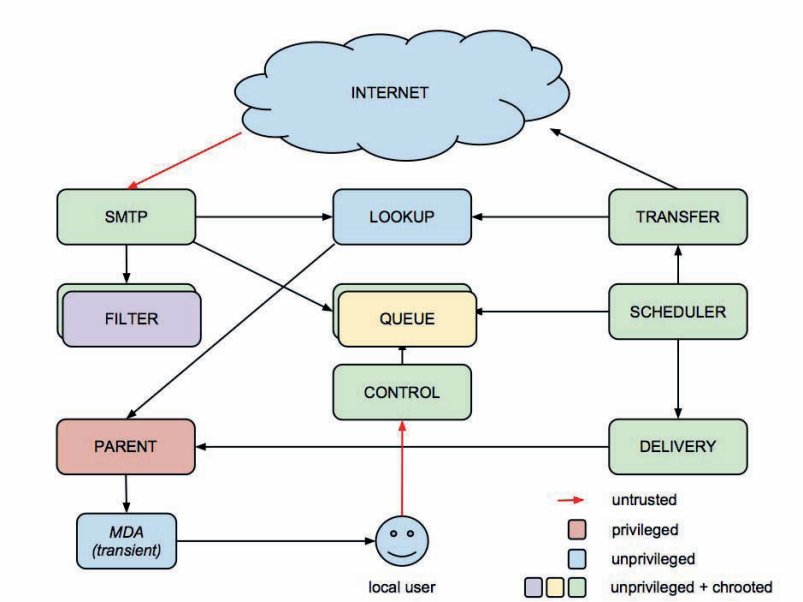


Figure 2: OpenSMTPD process layout

3.2.1 Scheduler

The scheduler is the process that knows all existing envelopes, and which takes the decision of scheduling for delivery, relaying, expiring them. It does not know the full detail of the envelopes, but only the necessary scheduling information: creation date, expiry date, retry count and a few flags which are required to take scheduling decision.

The scheduler only maintains a runtime state: it gets notified about new envelopes, dispatch work to be done when an envelope is schedulable, and wait for notification of the delivery outcome, to update the envelope runtime state.

It runs fully unprivileged and chrooted to an empty directory.

3.2.2 Queue

The queue process is responsible for storing envelopes and messages on permanent storage reliably, and reloading them on demand. For practical reason it is also the process that re-inject bounced envelopes. When the daemon starts, the queue process is also responsible for reloading existing

envelopes off permanent storage and providing them to the scheduler. It is the only process that is supposed to maintain an offline state.

One important duty of the queue process for reliability is to make sure that incoming messages and envelopes have reached permanent storage before being acknowledged, and that they are not removed or altered before being confirmed as delivered.

The queue process runs unprivileged, and chrooted into the smtpd spool directory.

3.2.3 SMTP

The SMTP process implements the SMTP protocol. It manages clients connections and push incoming messages to the rest of the system. As for the rest of the daemon, it runs fully event-based. Due to its design which relies heavily on fd-passing, the daemon is very sensible to file descriptor exhaustion situations. To prevent this from happening, which would impair the already running sessions, the daemon automatically disable input sockets when system limits are reached or close to be. It also implements simple counter-measures against clients that hog a connection by not doing any useful work, i.e. actually delivering mails.

It receives the listening sockets through dedicated imsg from the parent process, which already bound them to reserved ports. So it can run fully unprivileged and chrooted to an empty directory.

3.2.4 Mail transfer agent

The mail transfer agent (mta), is the process that handles relaying to other hosts.

The process managed destination relays to which mails must be sent. It tries to establish connections to these relays and drain pending messages. It handles connections limit, such as the number of mails sent per session, the number of recipients per mail, the number of parallel connections to the same host or domain, the maximum number of connections.

Like the SMTP process, it runs unprivileged and chrooted to an empty directory.

3.2.5 Mail delivery agent

The mail delivery agent (mda) is responsible for managing local deliveries. It does not do the delivery itself, but asks the parent privileged process to execute the actual mail delivery program on behalf of the local user to which a mail is destinated. The process makes sure to limit the number of running delivery processes on a global and per-user basis.

This process also runs unprivileged and chrooted to an empty directory,

3.2.6 Lookup agent

The lookup agent (lka) acts as a proxy for all kinds of lookups that other processes wants to do: credentials for authentication, user information, DNS resolving, which is a very important part of an SMTP daemon. The DNS queries are done asynchronously using an async variant of the resolver API. Multiple queries can run in parallels within the same process, which avoids having to block the whole process or rely on cumbersome work-arounds like thread pools or forks.

The lookup agent is also the process that does the rule-set evaluation for incoming envelopes, as we will see in the next section.

The main reason for having a separate process doing the queries is isolation: it allows other

processes to run with very constrained environment. The lookup agent runs unprivileged, but is not chrooted since it must be able to access system resources like /etc/resolv.conf.

3.2.7 Mail filter agent

The mail filter agent (mfa) manages the pluggable filter processes and drives the filtering process. It maintains a chain of filters through which SMTP requests and events are passed for processing, before being sent back to the SMTP process.

3.2.8 Control

The control process is the one that listens on the UNIX socket for request `smtpctl` program. It forwards requests from the administrator to the relevant process: pausing/resuming some agent, force the scheduling or removal of a message, get information about the runtime state of a message or envelope, etc, and deliver the responses to the client.

It is also responsible for collecting counter updates. All processes can report information about their internal activity for statistical analysis or runtime monitoring, in the form of named counter increments/decrements. For example, the number of active clients connections or queued envelopes.

3.2.9 Parent

This is the master process. It is the only process that needs to run as superuser. After the configuration file is loaded, it forks all the above processes, send them their runtime configuration, and waits for queries from other processes.

This process can perform the following privileged operations on behalf of other process: open a user .forward file for the lookup agent when expanding envelopes, perform local user authentication for the smtp process, execute or kill an external delivery program on behalf of the mail delivery process.

4 Operational workflows

We will describe here the internal exchanges that occurs to fulfill different services that the server must provide.

4.1 Enqueueing

The enqueueing task consists in accepting messages from outside. The sequence is depicted in figure 3. We suppose that all SMTP negotiation is already done, and the client wants to start a new transaction.

When a new MAIL FROM command is received it tells the queue process to create a new incoming message, and gets a unique message id in return. For each recipient (RCPT TO), a query is sent to the lookup agent, which tries find a matching rule. If found the recipient is *expanded* according to that rule (by looking for aliases or forward). The complete expand process is not detailed here, but it creates one or more final envelopes corresponding to the given recipient. These envelopes are sent to queue which stores them for the incoming message, and forward them to the scheduler. When all envelopes for a recipient are expanded, the SMTP process is notified that the

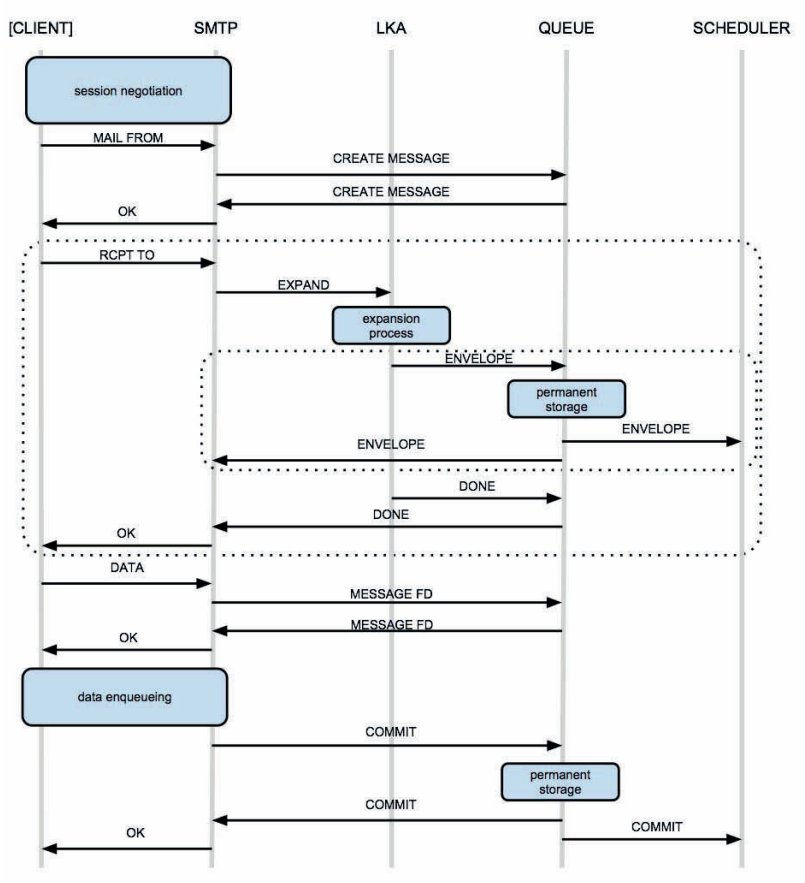


Figure 3: Enqueueing sequence diagram

RCPT is accepted. The client can then issue new RCPT request, repeating the expand sequence.

When the client wants to start sending the message content (using the DATA command), the SMTP process request a file descriptor from the queue process for writing the incoming message. When all data is written, it asks the queue to *commit* the message, which means moving it from incoming to permanent queue, and notifying the scheduler. When the SMTP receives the confirmation from the queue that the message is committed, it notifies the client.

If anything fails at some point of the process, the incoming message is cancelled. The sequence ensures that the envelopes and message have always reached permanent storage before the client and the scheduler are notified. So the whole process is fully transactional.

4.2 Scheduling

The scheduler receives new envelopes for incoming envelopes from the queue process. When a message is committed, it starts scheduling the associated envelopes. The life-cycle of these envelopes in the scheduler is shown on diagram 4.

The scheduler creates batches of schedulable envelopes id which are sent to the queue process

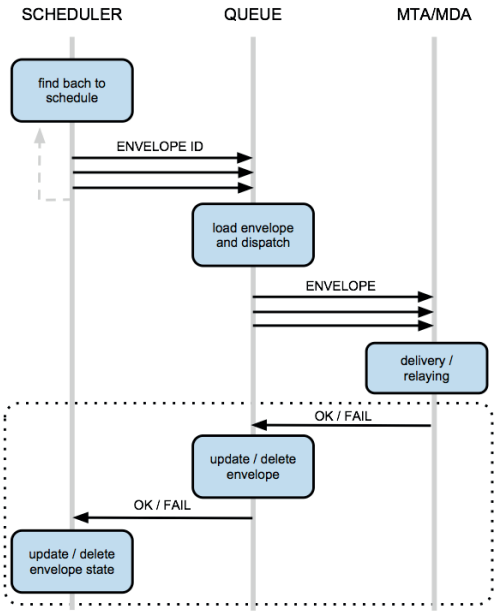


Figure 4: Scheduling sequence diagram

for delivery or relaying, and marked as "in-flight". The queue process loads the full envelopes and it dispatches them between the two mda an mta agent, depending on the envelope type.

When the agent is done with an envelope, it reports the outcome to the queue process, which updates the envelope state or delete it on success. The scheduler is then notified, so that it can update the state of the "in-flight" envelope accordingly: either remove it, or re-schedule later in a new batch.

4.3 Local deliveries

The detail of the local delivery operation is depicted on figure 5. When a message is to be delivered locally, the delivery process first retrieves the user information from the lka. It queries the queue process to obtain a file descriptor to read the message content from. When received, it sends a fork request with given user info to the parent process, which returns a file descriptor to write the message body to. When done, it waits for the notification from parent that the forked mda process has correctly exited, and if so, it reports the envelope as delivered to the queue process, which removes the envelope and informs the scheduler. In case of error, a failure report is sent to the queue.

4.4 Relaying

Relaying is the process by which responsibility for a message in the queued is transfered to another server, normally through the SMTP protocol. The internal interaction between smtpd process is shown on figure 6.

When receiving a envelope to relay, the transfer process first determines the relay to use. If necessary, it requests the credentials from the lookup agent. It also requests the list of MX hosts to use. These information are cached as long as the relay is currently referenced, so new envelopes

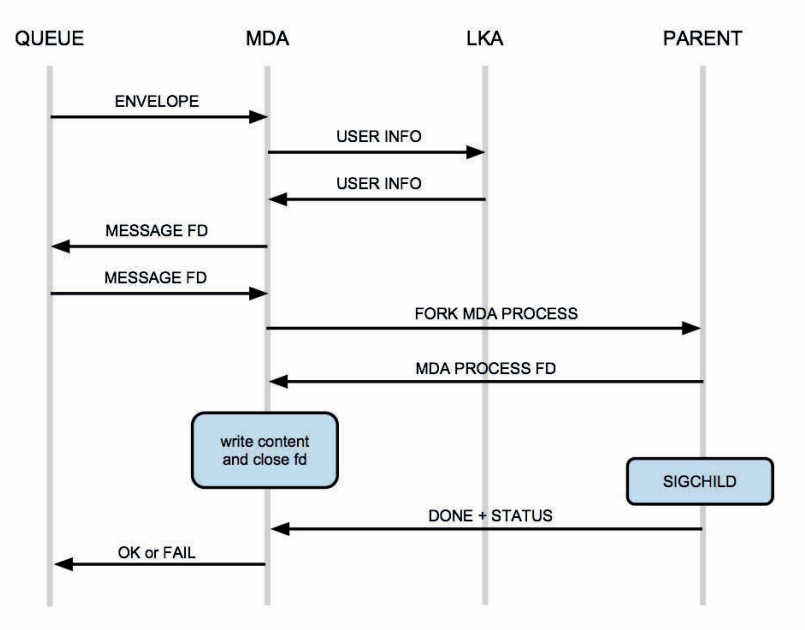


Figure 5: Local delivery sequence diagram

for the same relay in the mean time will not trigger these queries again. Received envelopes are grouped into tasks (messages for the same relay and for the same message) and queued on the relay for delivery.

When ready, the mail transfer agent will try to establish connections for the relay by trying to connect the MXs. Once a session is established, the reverse DNS entry is searched and the initial SMTP negotiation happens. At this point the agent might want to get the specific HELO string to use from the lka. When the session is ready to send mails, it dequeues the next task, retrieves the file descriptor for the message body from the queue, and starts the SMTP transfer. When the remote host has acknowledged (or rejected) the message transfer, the queue is notified for each envelope. The session can than proceed to the next task, if any.

If no connections could be established at all, all envelopes for that relay are sent back to the queue process with a failure report.

4.5 Bounces

Bounces are report messages that are sent by the mailer daemon to report temporary or permanent failure when trying to deliver a mail it has accepted. There are two kinds of bounces which can occurs: either a delay notification that is generated when an envelope has been queued for a certain time, or a failure report when the envelope has been refused or has expired, in which case the associated envelope is discarded.

A bounce is always associated to a message, it is internally handled as a special type of envelope for that message. It is created by the queue process as any other envelopes, expect that it is usually created on an already committed message (as opposed to incoming). Once created, the bounce envelope is sent to the scheduler, as any other envelope.

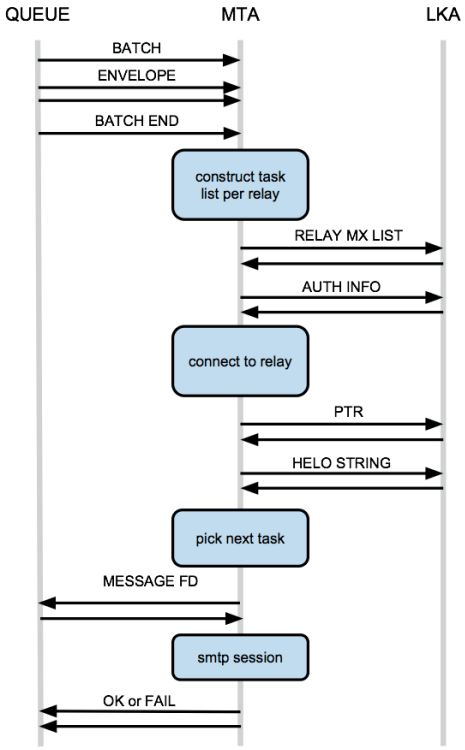


Figure 6: Relaying sequence diagram

It is processed by the queue process which will create regular envelope targeted at the original sender (return path) with a report and the bounced message. So the queue process request an internal socket from the smtp process to establish a local SMTP session, and re-inject the bounce message using the SMTP protocol. It creates a message with a single envelope, which is the original sender for the bounced envelope, and the body of the message contains the failure report for the original recipient.

Bounces are grouped when possible: when receiving a bounce envelope to re-inject, the queue process will delay the SMTP session lightly waiting for other bounces for the same message. The figure 7 illustrates the successive states of the queue when bouncing and discarding two envelopes for the same message.

5 Modularity and deployment options

Besides the flexibility of the configuration which allows the creative minds to easily build complex processing chains (relaying outgoing mail through a DKIM proxy is a 4 liners), OpenSMTPD provides two mean of extending its functionalities: input filters and internal backends.

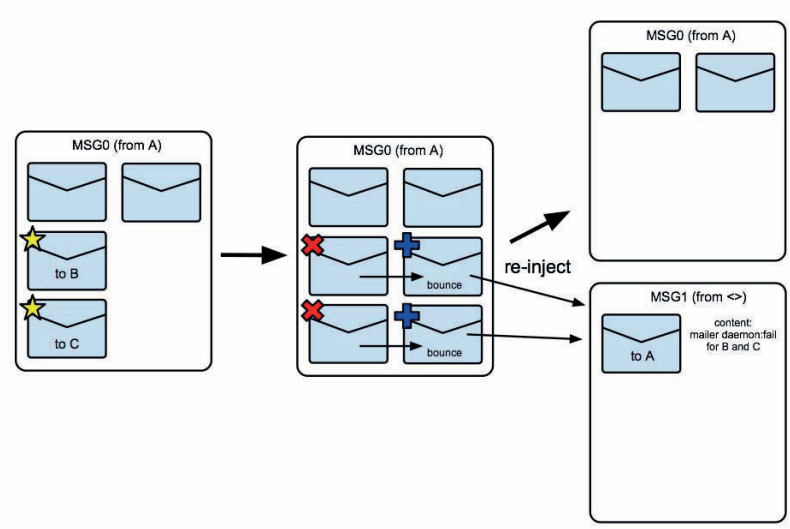


Figure 7: Bounce creation and re-injection

5.1 Filtering

The goal is to provide a powerful way to let external programs filter the incoming SMTP sessions and messages, in order to reject some possibly altering the recipients and/or the message contents.

A filtering mechanism has been in place since the beginning, but it has not been a high priority until more fundamental parts of the daemon were in place. It has been completely re-designed recently to fix some early limitations.

Filters are implemented as external programs which are started with the daemon. They are bound to the `mfa` process, and communicate with it via dedicated IMSG. The filter will registers various *hooks* for events or queries it wants to intercepts. An API and library is provided for writing filters, hiding the internal imsg machinery.

At the moment, the integrated filter support is still under development, and not officially supported yet and will not be part of the first release. It is however possible achieve various kind of filtering using a proxy loop: incoming mails are relayed through a specific SMTP proxy, such as a DKIM signing program or spam filter, and re-injected on a specific tagged listener.

A specific `milter` glue filter is planned, to allow using filters implemented with the milter API to run with OpenSMTPD. The ability to filter outgoing mail (at the transfer agent level) is also planned but not implemented yet.

5.2 Backends

Whereas filters operates on users inputs to alter the smtpd behaviour, the backend mechanism is a mean to completely change the deployment strategy.

Various parts of the daemon internally relies on abstractions for specific operations, which are defined as concise and strict API, which are implemented by *backends*. From the point of view of the calling process, this is completely hidden, so any implementation of these interfaces with

the required semantics, can be used. This has two advantages: it prevents layer violation in the design, and it makes the implementation very flexible, once the correct abstractions are found. The three most important backends are presented here.

5.2.1 Table backends

Almost all lookups in the daemons are done using the *table* abstraction. Depending on the context, a table can be either a set, or an associative array. The types of the elements also depends on the context. Tables are said to provide one or more *service*, and are implemented by a backend. The different services used are :

- ALIAS: association of a user name to mail aliases, used for alias lookups.
- DOMAIN: list of domain names: used to match destination or source domains.
- CREDENTIALS: association of a label to a pair of username/password, for SMTP authentication.
- NETADDR: list of network addresses, used for rule matching on the connection.
- USERINFO: association of a user name to its system uid/gid and home directory.
- SOURCE: list of local IP addresses, used by the MTA for outgoing connections.

The existing backends are :

- static: in memory content, or loaded from a file,
- db: using a db(3) database,
- getpwnam: this special backends implements only the credentials and userinfo service,
- ldap and sqlite: those are experimental backends

Any table backend able to implement a given service can be used in all contexts where this service is expected. It is then trivial to run completely virtual mail servers.

5.2.2 Scheduler backend

The scheduler backend API only have a few function to insert new envelopes fetch the list of envelopes to schedule, which are dispatched to the relevant agents, and update their state depending on the delivery outcome. There is a couple of administrative functions too. The current scheduler operates entirely in memory.

5.2.3 Queue backend

The queue process relies on a storage backend which role is to implement a set of well-defined atomic operation on envelopes and messages. The default out-of-the box backend uses bucket-based disk storage and rely on file-system operations. OpenSMTPD also ships with a RAM-based queue, which keeps every everything in memory. Of course, the storage is not permanent anymore, but it is a good way for testing things, and it is also useful, when designing the API and implementation, to make sure that the abstractions make sense and that no dangerous assumptions are made on the queue implementation.

As another Proof-of-Concept, we have written a completely alternative storage backend for the queue , which used ReST queries to store envelopes on an external *cloud* storage. The performances were of course terrible in the example setup, but the reliability of the whole delivery process was intact, since every thing is still transactional.

6 Conclusion

OpenSMTPD provides a simple and powerful implementation of a mail server, with a relatively small codebase. The design principles behind it make it very easy to use and deploy in a large range of environments. Performances issues have not been discussed here. It is a bit early to focus on performance tuning. Nonetheless, early tests shows that the server behaves correctly under load, and is able to process envelopes at very satisfying rate.

The daemon is actively maintained. Primary development is done on OpenBSD, and a portable version for other UNIX systems is provided too. It's been tested successfully on various flavours of Linux, other BSDs, and MacOS-X.

Some features are not there yet: for example support for the PIPELINING extensions is missing, and we need a way to quarantine some messages/envelopes. However we believe that the current features are enough to handle most situations. We are now focusing on internal cleanups and testing. A first official release is to be expected very soon.

References

[1] Pf: The openbsd packet filter. `http://www.openbsd.org/faq/pf/index.html`.

[2] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), October 2008.

Implements BIOS emulation support for BHyVe: A BSD Hypervisor

Abstract

Current BHyVe only supports FreeBSD/amd64 as a GuestOS.
One of the reason why BHyVe cannot support other OSes is lack of BIOS support.
My project is implementing BIOS emulator on BHyVe, to remove these limitations.

1. Background

1.1 History of virtualization on x86 architecture

There's a famous requirements called "Popek & Goldberg Virtualization requirements"[1], which defines a set of conditions sufficient for an architecture to support virtualization efficiently.
Efficient virtualization means virtualize machine without using full CPU emulation, run guest code natively.
Explain the requirements simply, to an architecture virtualizable, all **sensitive instructions** should be privileged instruction.
Sensitive instructions definition is the instruction which can interfere the global status of system.
Which means, all sensitive instructions executed under user mode should be trapped by privileged mode program.
Without this condition, Guest OS affects Host OS system status and causes system crash.
x86 architecture was the architecture which didin't meet the requirement, because It had non-privileged sensitive instructions.
To virtualize this architecture efficiently, hypervisors needed to avoid execute these instructions, and replace instruction with suitable operations.
There were some approach to implement it:
On VMware approach, the hypervisor replaces problematic sensitive instructions on-the-fly, while running guest machine. This approach called **Binary Translation**[2].
It could run most of unmodified OSes, but it had some performance overhead.
On Xen approach, the hypervisor requires to run pre-modified GuestOS which replaced problematic sensitive instructions to dedicated operations called **Hypercall**. This approach called **Para-virtualization**[3].
It has less performance overhead than Binary Translation on some conditions, but requires pre-modified GuestOS.
Due to increasing popularity of virtualization on x86 machines, Intel decided to enhance x86 architecture to **virtualizable**.
The feature called **Intel VT-x**, or **Hardware-Assisted Virtualization** which is vendor neutral term.
AMD also developed hardware-assisted virtualization feature on their own CPU, called **AMD-V**.

1.2 Detail of Intel VT-x

VT-x provides new protection model which isolated with **Ring protection**, for virtualization.
It added two CPU modes, **hypervisor mode** and **guest machine mode**.
Hypervisor mode called **VMX Root Mode**, and guest machine mode called **VMX non Root Mode**(Figure 1).

Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. Commun. ACM 17, 7 (July 1974), 412-421. DOI=10.1145/361011.361073 http://doi.acm.org/10.1145/361011.361073

[2] Brian Walters. 1999. VMware Virtual Platform. Linux J. 1999, 63es, Article 6 (July 1999).

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 164-177. DOI=10.1145/945445.945462 http://doi.acm.org/10.1145/945445.945462

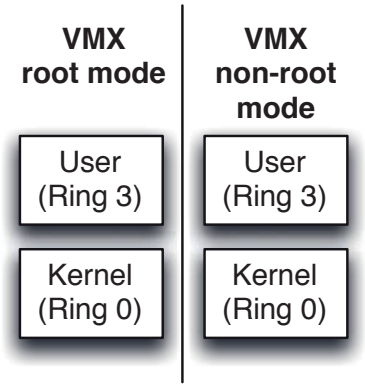


Figure 1. VMX root Mode and VMX non-root Mode

On VT-x, hypervisor can run guest OS on VMX non Root Mode without any modification, including **sensitive instructions**, without affecting Host OS system status.
When sensitive instructions are being executed under VMX non Root Mode, CPU stops execution of VMX non Root Mode, exit to VMX Root Mode.
Then it trapped by hypervisor, hypervisor emulates the instruction which guest tried to execute.
Mode change from VMX Root Mode to VMX non-root Mode called **VMEntry**, from VMX non-root Mode to VMX Root Mode called **VMExit**(Figure 2).

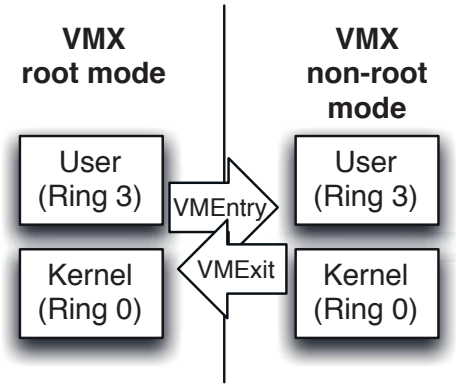


Figure 2. VMEntry and VMExit

Some more events other than sensitive instructions which need to intercept by hypervisor also causes VMExit.

For example, IN/OUT instruction causes VMExit, and hypervisor emulates virtual device access.
VT-x defines number of events which can cause VMExit, and hypervisor needs to configure enable/disable on each VMExit events.
Reasons of VMExit is called **VMExit reason**, it classified by genres of events.

Here are VMExit reason list:

- Exception or NMI
- External interrupt
- Triple fault
- INIT signal received
- SIPI received
- SM received
- Internal interrupt
- Task switch
- CPUID instruction
- Intel SMX instructions
- Cache operation instructions(INVD, WBINVD)
- TLB operation instructions(HNVLPG, INVPCID)
- IO operation instructions(INB, OUTB, etc)
- Performance monitoring conter operation instruction(RDTSC)
- SMM related instruction(RSM)
- VT-x instructions(Can use for implement nested virtualization)
- Accesses to control registers
- Accesses to debug registers
- Accesses to MSR
- MONITOR/MWAIT instructions
- PAUSE instruction
- Accesses to Local APIC
- Accesses to GDTR, IDTR, LDTR, TR
- VMX preemption timer
- RDRAND instruction

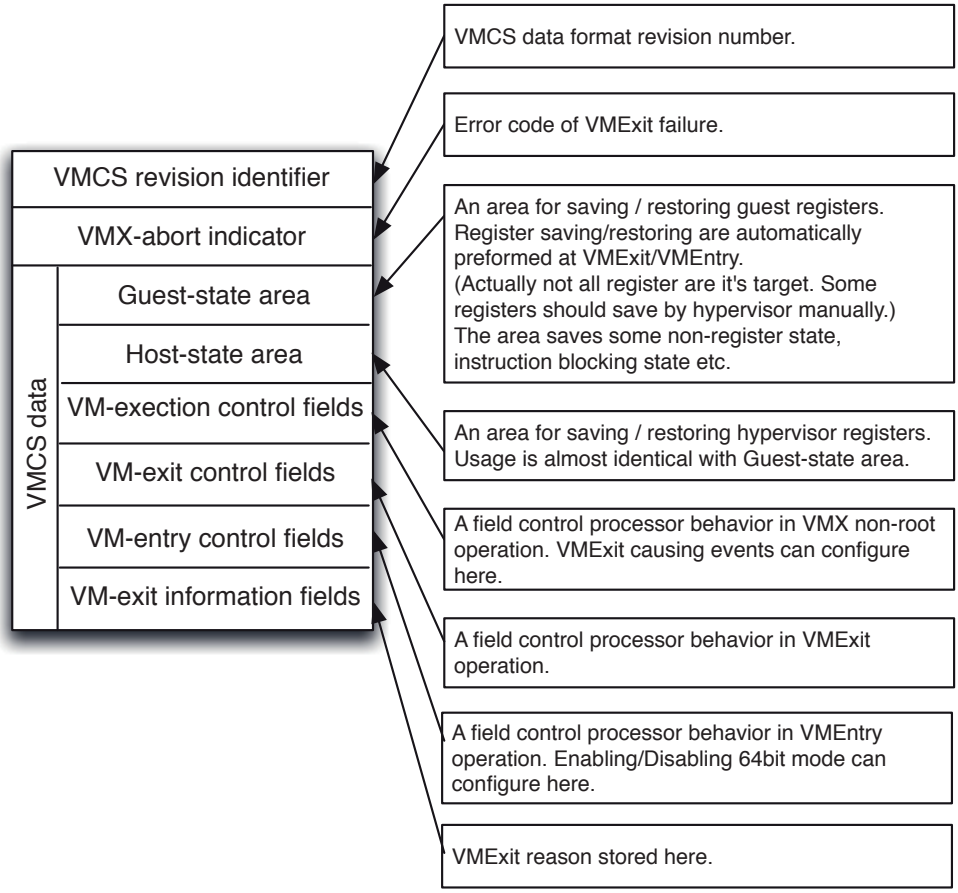


Figure 3. Structure of VMCS

All configuration data related to VT-x stored to **VMCS(Virtual Machine Control Structure)**, which is on memory data structure for each guest machine[4].
Figure 3 shows VMCS structure.

1.3 VT-x enabled hypervisor lifecycle

Hypervisors for VT-x works as following lifecycle (Figure 4).

1. VT-x enabling
   It requires to enable at first to use VT-x features.
   To enable it, you need set VMXE bit on CR4 register, and invoke VMXON instruction.
2. VMCS initialization
   VMCS is 4KB alined 4KB page.
   You need to notify the page address to CPU by invoking VMPTRLD instruction, then write initial configuration values by VMWRITE instruction.
   You need to write initial register values here, and it done by /usr/sbin/bhyveload.
3. VMEntry to VMX non root mode
   Entry to VMX non root mode by invoking VMLAUNCH or VMRESUME instruction.
   On first launch you need to use VMLAUNCH, after that you need to use VMRESUME.
   Before the entry operation, you need to save Host OS registers and restore Guest OS registers.
   VT-x only offers minimum automatic save/restore features, rest of the registers need to take care manually.
4. Run guest machine
   CPU runs VMX non root mode, guest machine works natively.

[4] If guest system has two or more virtual CPUs, VMCS needs for each vCPUs.

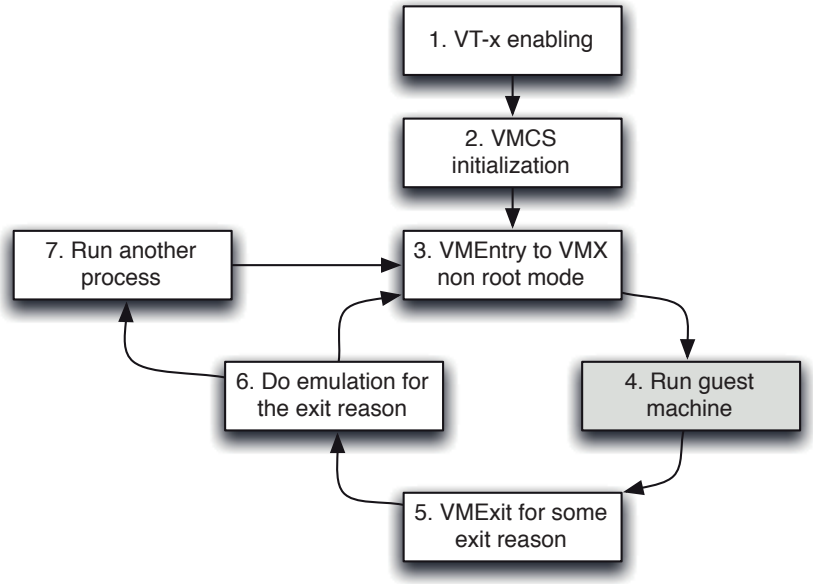


Figure 4. VT-x enabled hypervisor lifecycle

5. VMExit for some reason
   When some events which causes VMExit, CPU returns to VTX root mode.
   You need to save/restore register at first, then check the VMExit reason.
6. Do emulation for the exit reason
   If VMExit reason was the event which requires some emulation on hypervisor, perform emulation. (Ex: Guest OS wrote data on HDD
   Depending Host OS scheduling, it may resume VM by start again from 3, or task switch to another process.

1.4 Memory Virtualization

Mordan multi-tasking OSes use paging to provide individual memory space for each processes.
To run guest OS program natively, address translation on paging become problematic function.
For example (Figure 5):
You allocate physical page 1- 4 to Guest A, and 5-8 to GuestB.
Both guests map page 1 of Process A to page 1 of guest physical memory.
Then it should point to:

- Page 1 of Process A on Guest A ->
  Page 1 of Guest physical memory ->
  Page 1 of Host physical
- Page 1 of Process B on Guest B ->
  Page 1 of Guest physical memory ->
  Page 5 of Host physical

But, if you run guest OS natively, CPU will translate Page 1 of Process B on Guest B to **Page 1 of Host physical memory**.
Because CPU doesn't know the paging for guests are nested.

There is software technique to solve the problem called **shadow paging** (Figure 6).
Hypervisor creates clone of guest page table, set host physical address on it, traps guest writing CR3 register and set cloned page table to CR3.
Then CPU able to know correct mapping of guest memory.
This technique was used on both Binary translation based VMware, and also early implementation of hypervisors for VT-x.
But it has big overhead, Intel decided to add nested paging support on VT-x from Nehalem micro-architecture.

EPT is the name of nested paging feature (Figure 7),
It simply adds Guest physical address to Host physical address translation table.
Now hypervisor doesn't need to take care guest paging, it become much simpler and faster.

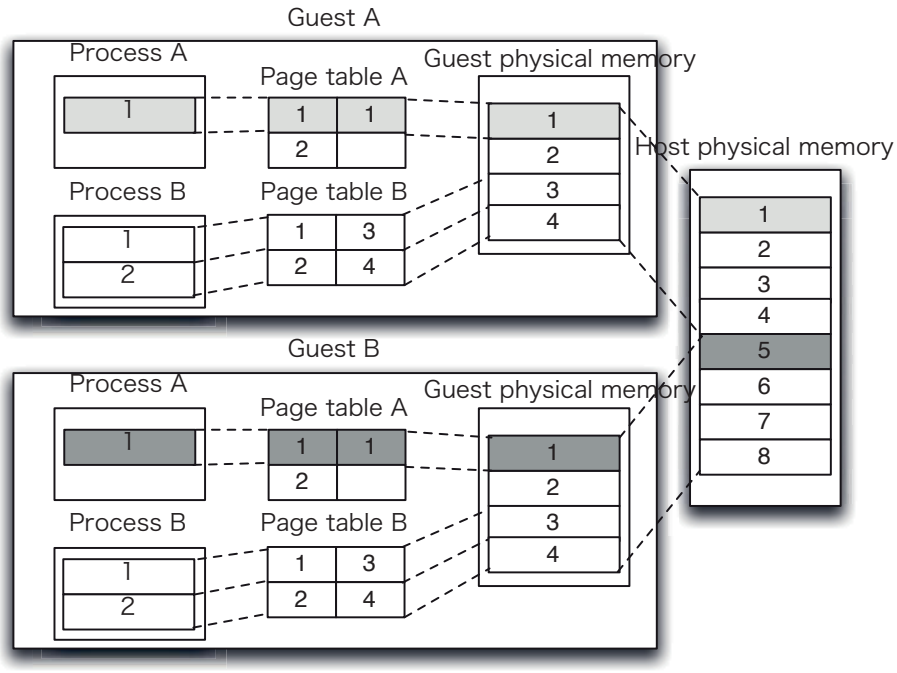


Figure 5. Problem of memory virtualization

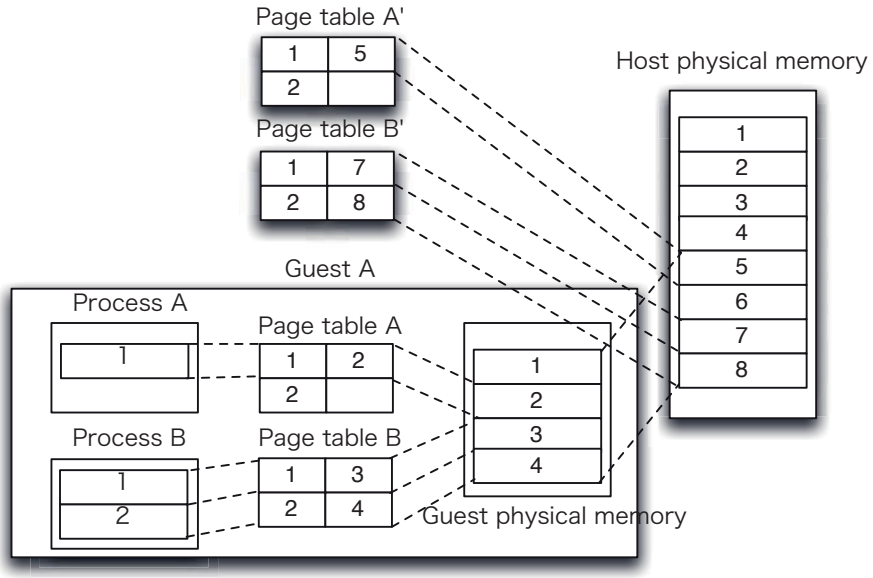


Figure 6. Shadow paging

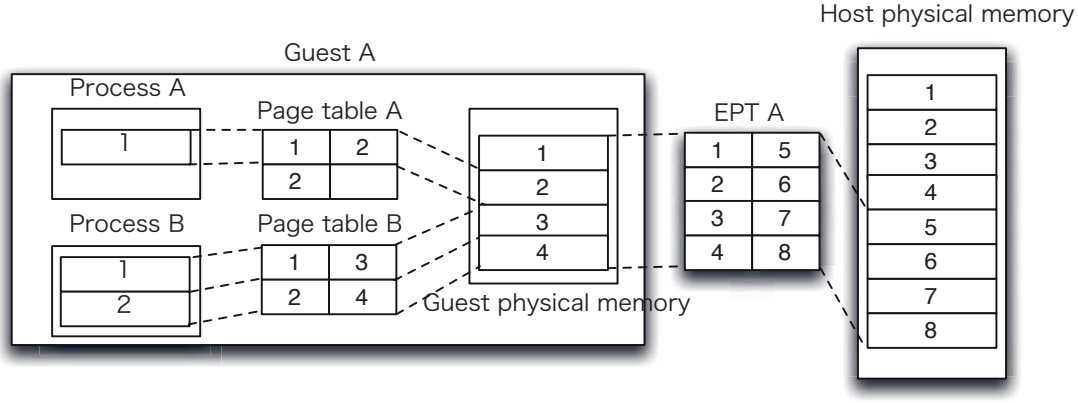


Figure 7. EPT

Actually, not all VT-x supported CPUs supports EPT, on these CPUs hypervisors still need to do shadow paging.

2. BHyVe: BSD Hypervisor

2.1 What is BHyVe?

BHyVe is new project to implement a hypervisor witch will integrate in FreeBSD.
The concept is similar to Linux KVM, it provides “hypervisor driver” to unmodified BSD kernel running on bare-metal machine.
With the driver, the kernel become a hypervisor, able to run GuestOS just like normal process on the kernel.
Both hypervisors are designed for hardware assisted virtualization, unlike Xen’s para-virtualization and VMware’s binary translation.
The kernel module only provides a feature to switch CPU modes between Host mode and Guest mode, almost all device emulation is performed in userland process.

2.2 Difference of approach between Linux KVM and BHyVe

Linux KVM uses modified QEMU[5] as the userland part[6].
It’s good way to support large coverage of Guest OSes, because QEMU is highly developed emulator, many people already confirmed to run variety of OSes on it.
KVM could support almost same features what QEMU has, and it just worked fine.
BHyVe’s approach is different.
BHyVe implements minimum set of device support which required to run FreeBSD guest, from scratch.
In the result, we could have completely GPL-free, BSD licensed, well coded hypervisor, but it only supports FreeBSD/amd64 as a Guest OS at this point.
One of the reason why BHyVe cannot support other OSes is lack of BIOS support.
BHyVe loads and executes FreeBSD kernel directly using custom OS loader runs on Host OS, instead of boot up from disk image.
With this method, we need to implement OS loader for each OSes, and currently we don’t have any loader other than FreeBSD.
Also, it doesn’t support some OSes which calls BIOS function while running.
So I started the project to implementing BIOS emulator on BHyVe, to remove these limitations.

2.3 Hardware requirements

BHyVe requires an Intel CPU which supports Intel VT-x and EPT.
It means you will need Nehalem core or later Intel CPUs, because EPT is only supported on these processors.
Currently, AMD-V is not supported.
Installing on physical machine is best choice, but it also works on recent version of VMware, using **Nested virtualization feature**[7].

2.3 Supported features

BHyVe only supports FreeBSD/amd64 8-10 for guest OS.

---

[5] Original QEMU has full emulation of x86 CPU, but on KVM we want to use VT-x hardware assisted virtualization instead of CPU emulation.
So they replace CPU emulation code to KVM driver call.

[6] Strictly speaking, KVM has another userland implementation called Linux Native KVM Tools, which is built from scratch - same as BHyVe’s userland part.
And it has similar limitation with BHyVe.

[7] The technology which enables **Hypervisor on Hypervisor**. Note that it still requires Nehalem core or later Intel CPUs even on VMware.

It emulates following devices:

- HDD controller: virtio-blk
- NIC controller: virtio-net
- Serial console: 16550 compatible PCI UART
- PCI/PCIe devices passthrough (VT-d)

Boot-up from virtio-blk with PCI UART console is not general hardware configuration on PC architecture, we need to change guest kernel settings on /boot/loader.conf(on guest disk image).
And some older FreeBSD also need to add a virtio drivers[8].
PCI device passthrough is also supported, able to use physical PCI/PCIe devices directly.
Recently ACPI support and IO-APIC support are added, which improves compatibility with existing OSes.

2.4 BHyVe internal

BHyVe built with two parts: kernel module and userland process.
The kernel module is called vmm.ko, it performs actions which requires privileged mode (ex: executes VT-x instructions.
Userland process is named /usr/sbin/bhyve, provides user interface and emulates virtual hardwares.
BHyVe also has OS Loader called /usr/sbin/bhyveload, loads and initializes guest kernel without BIOS.
/usr/sbin/bhyveload source code is based on FreeBSD bootloader, so it outputs bootloader screen, but VM instance is not yet executing at that stage.
It runs on Host OS, create VM instance and loads kernel onto guest memory area, initializes guest machine registers to prepare direct kernel boot.
To destroy VM instance, VM control utility / usr/sbin/bhyvectl is available.
These userland programs are accesses vmm.ko via VMM control library called libvmmapi.
Figure 8 illustrates overall view of BHyVe.

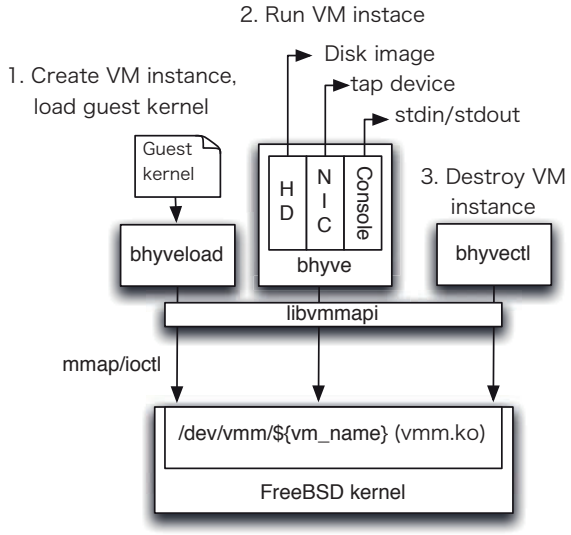


Figure 8. BHyVe overall view

3. Implement BIOS Emulation

3.1 BIOS on real hardware

BIOS interrupt calls are implemented as software interrupt handler on real mode(Figure 9).
CPU executes initialization code on BIOS ROM at the beginning of startup machine, it initializes real mode interrupt vector to handle number of software interrupts reserved for BIOS interrupt calls(Figure 10).
BIOS interrupt calls aren't only for legacy OSes like MS-DOS, almost all boot loaders for mordan OSes are using BIOS interrupt call to access disks, display and keyboard.

3.2 BIOS on Linux KVM

On Linux KVM, QEMU loads **Real BIOS(called SeaBIOS)** on guest memory area at the beginning of QEMU startup.
KVM version of SeaBIOS's BIOS call handler accesses hardware by IO instruction or memory mapped IO, and the behavior is basically same as BIOS for real hardware.
The difference is how the hardware access handled.
On KVM, the hardware access will trapped by KVM hypervisor driver, and QEMU emulates

[8] virtio is **para-virtual driver** which designed for Linux KVM. para-virtual driver needs special driver for guest, but usually much faster than full emulation driver.

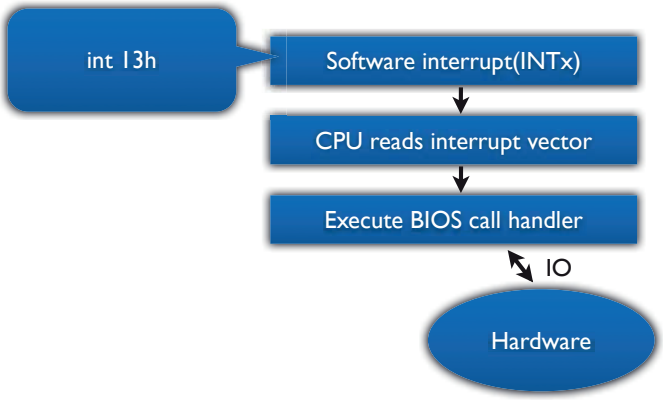


Figure 9. BIOS interrupt call mechanism on real hardware

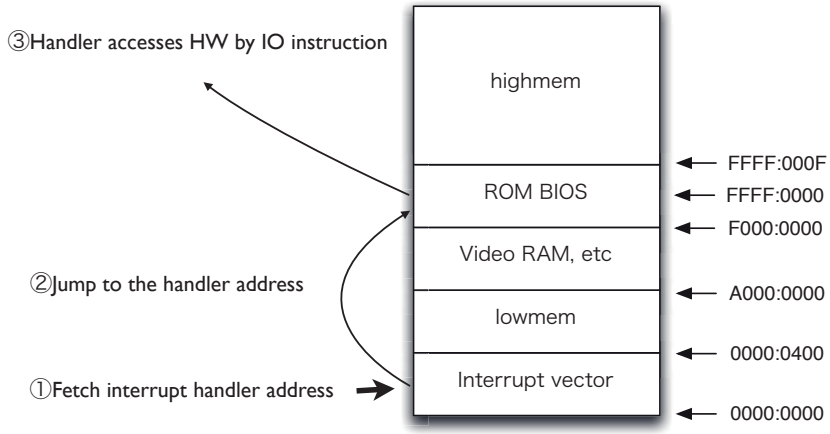


Figure 10. Memory map on real hardware

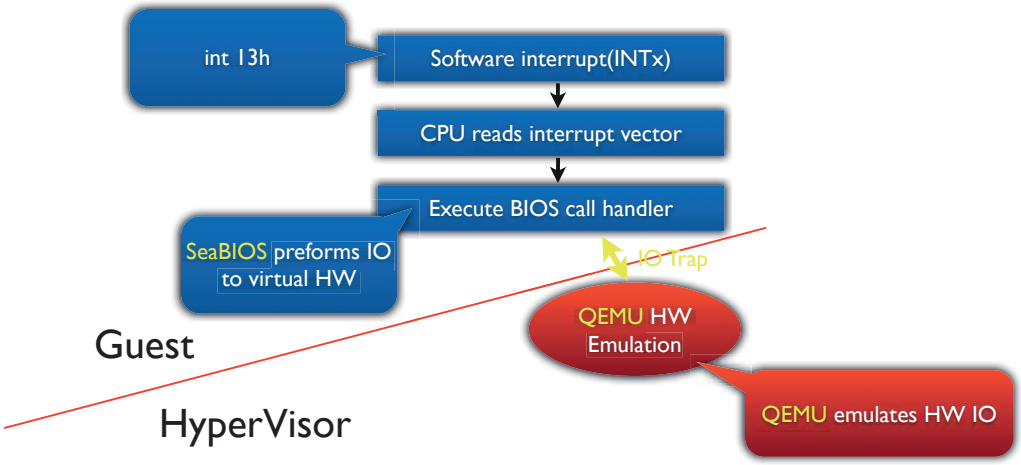


Figure 11. BIOS interrupt call mechanism on KVM

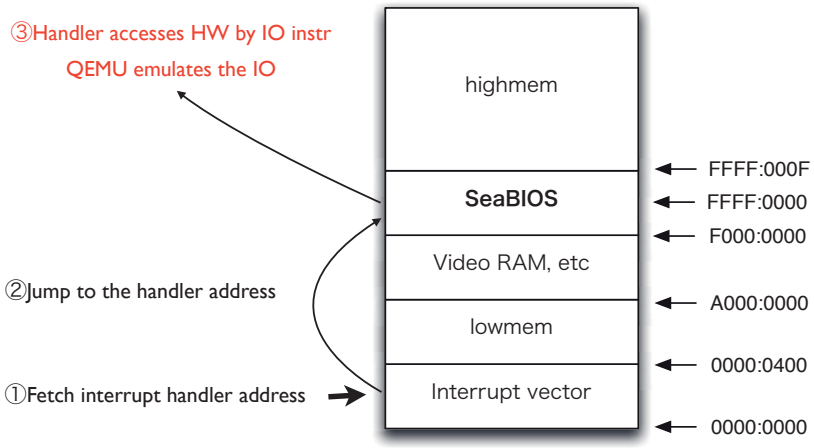


Figure 12. Memory map on KVM

hardware device, then KVM hypervisor driver resume a guest environment(Figure 11).
In this implementation, KVM and QEMU doesn't trap BIOS interrupt calls, it just loads real BIOS on guest memory space(Figure 12) and emulates hardware device.

3.3 Emulating BIOS on BHyVe

3.3.1 doscmd

Port SeaBIOS on BHyVe and implement hardware emulation was an option, and it was probably best way to improve compatibility of legacy code, but SeaBIOS is GPL'd software, it's not comfortable to bring in FreeBSD code tree.
And there's no implementation non-GPL opensourced BIOS.
Instead, there's BSD licensed DOS Emulator called **doscmd**.
It's the software to run old DOS application on FreeBSD using virtual 8086 mode, similar to DOSBox(but DOSBox is GPL'd software).

The emulator mechanism is described as follows:

1. Map pages to lowmem area (begin from 0x0), load the DOS application on the area.
2. Enter virtual 8086 mode, start executing the DOS application.
3. DOS application invokes BIOS interrupt call or DOS API call by INTx instruction.
4. DOS Emulator traps software interrupt, emulate BIOS interrupt call or DOS API call.
5. Resume DOS application.

It traps BIOS interrupt calls and DOS API calls and emulate them on FreeBSD protected mode program.
I decided to port the BIOS interrupt call emulation code to BHyVe and trap BIOS interrupt call on BHyVe, instead of porting real BIOS.

3.3.2 Run real mode program on VT-x

On older implementation of VT-x enabled CPU doesn't allow to VMEnter the guest which doesn't enable paging.
Which means real mode program cannot run on VT-x, and hypervisors needed to virtualize real mode without VT-x.
Linux KVM used full CPU emulation using QEMU to virtualize real mode.
Some other hypervisors are used virtual 8086 mode.
This issue was resolved by extending VT-x features.
Intel added **unrestricted guest mode** on Westmere micro-architecture and later Intel CPUs, it uses EPT to translate guest physical address access to host physical address.
With this mode, VMEnter without enable paging is allowed.
I decided to use this mode for BHyVe BIOS emulation.

3.3.3 Trapping BIOS interrupt call

VT-x has functionality to trap various event on guest mode, it can be done by changing VT-x configuration structure called VMCS.
And BHyVe kernel module can notify these events by IOCTL return.
So all I need to do to trapping BIOS call is changing configuration on VMCS, and notify event by IOCTL return when it trapped.
But the problem is which VMExit event is optimal for the purpose.
It looks like trapping software interrupt is the easiest way, but we may have problem after Guest OS switched protected mode.
Real mode and protected mode has different interrupt vector.
It's possible to re-use BIOS interrupt call vector number for different purpose on protected mode.
Maybe we can detect mode change between real mode/protected mode, and enable/disable software interrupt trapping, but it's bit complicated.

Instead of implement complicated mode change detection, I decided to implement software interrupt handler which cause VMExit.

The handler doesn't contain programs for handling the BIOS interrupt call, just perform VMExit by **VMCALL instruction**.
VMCALL causes unconditional VMExit.
It's for call hypervisor from guest OS, such function is called **Hypercall**.

Following is simplest handler implementation:

```
VMCALL
IRET
```

Even program is same, you should have the handler program for each vector.
Because guest EIP can be use for determine handled vector number.

If you place BIOS interrupt call handler start at 0x400, and program length is 4byte for each (VMCALL is 3byte + IRET is 1byte), you can determine vector number from hypervisor with following program:

```
vector = (guest_eip - 0x400) / 0x4;
```

BHyVe need to initialize interrupt vector and set pointer of the handler described above.
In this way, it doesn't take care about mode changes anymore.

Figure 13 shows BIOS interrupt call mechanism on my implementation.
On the implementation, it traps BIOS interrupt call itself, emulates by hypervisor.

4. Implementation
Most of work are rewriting doscmd to fit BHyVe interface, from FreeBSD virtual 8086 API.

- Code was 64bit unsafe

doscmd was designed only for 32bit x86, and BHyVe is only for amd64.
So I need to re-write some codes to 64bit safe.

ex:

```
u_long
↓
uint32_t
```

- Guest memory area started from 0x0

To use virtual 8086, doscmd places guest memory area from 0x0.
But BHyVe's guest memory area is mapped to non-zero address, we need to move all address to BHyVe's guest memory area.

ex:

```
*(char *)(0x400) = 0;
      ↓
*(char *)(0x400 + guest_mem) = 0;
```

- Interface with /usr/sbin/bhyve

I don't wanted to mix doscmd's complicated source code with /usr/sbin/bhyve's code, so I modified doscmd's Makefile to build it as a library.
And named it **libbiosemul**.

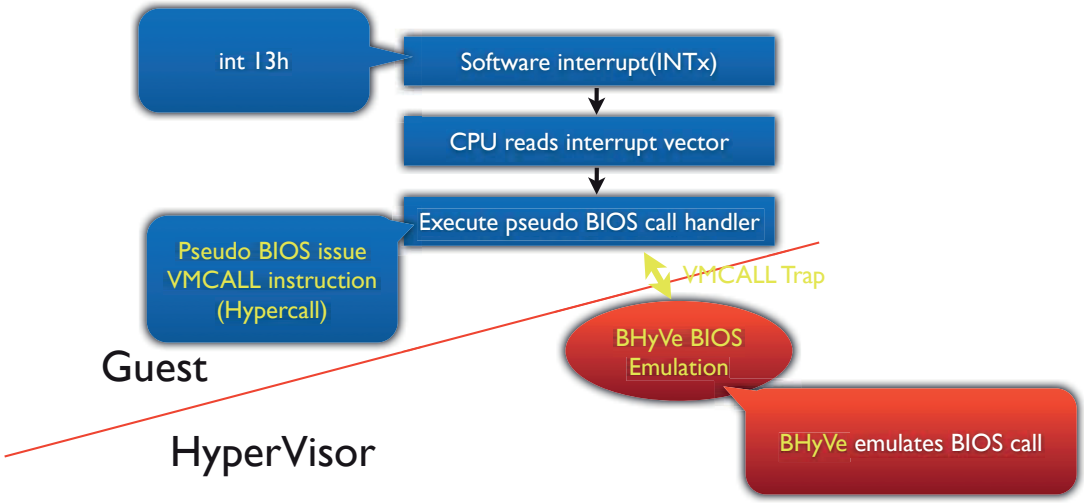


Figure 13. BIOS interrupt call mechanism on BHyVe

It exposed only few functions:

```
void biosemul_init(struct vmctx
*ctx, int vcpu, char *lomem, int
trace_mode);

int biosemul_call(struct vmctx
*ctx, int vcpu);
```

biosemul_init is called at initialization.
biosemul_call is main function, which called at every BIOS call.

- Guest register storage

doscmd stored guest register values on their structure, but BHyVe need to call ioctl to get / set register value.
It's hard to re-write all code to call ioctl, so I didn't changed doscmd code.
I just copy all register values to doscmd struct at beginning of BIOS call emulation, and copyback it the end of the emulation.

- Instruction level tracing

I implemented instruction level tracer to debug BIOS emulator.
It's also uses psuedo BIOS interrupt call handler to implement.

5. Development status

It still early stage of development, none of OSes boots up with the BIOS emulator.
I'm focusing to boot-up FreeBSD/amd64, now mbr and boot1 are working correctly.

Using BGP for realtime import and export of OpenBSD spamd entries

Peter Hessler
phessler@openbsd.org
OpenBSD

Bob Beck
beck@openbsd.org
OpenBSD

1 Introduction

In the battle against spam, many mail server admins collect and distribute IP addresses of systems that have sent them spam. However, distribution of these lists are traditionally limited to 2 methods. Method #1 is periodically downloading this list from a source, usually a web server - which is subject to load issues on the target web server. #2 is a real-time lookup against an external provider (such as dns-rbls) so your response time is dependent upon how fast they respond or timeout.

OpenBSD spamd[1] is typically used as a daemon to stop the "low hanging fruit" of spam, by subjecting previously unseen hosts to `greylisting` to allow time to identify it they are a real mailserver. Real mailservers are whitelisted locally after having passed greylisting, with their connections to the real mailservers monitored via `spamlogd`[2] . As such spamd keeps a list in a database of what it believes to be currently active "real" mailservers.

This paper suggests and discusses a 3rd solution: using BGP[3] to distribute the IP addresses in a real-time manner. By doing so we can take advantage of OpenBSD `spamd's` information to distribute two useful lists via BGP:

1. Each participant can share their TRAPPED entries from spamd(8) - hosts which the local spamd has determined should not be allowed to pass greylisting. Other hosts can use these lists to also prevent such hosts from passing greylisting in the short term.
2. By taking advantage of the information kept in `spamdb` - each participant can share a subset of their WHITE entries from `spamdb`, chosen based on criteria that makes them very likely to be real mail servers that are continuing to exchange mail with the participating domain on a regular basis. By doing this all participants can use this information to build a `bypass` table in `pf`[4] so that all such "real mailservers" talking to any participant in the network are not subjected to greylisting.

Having a greater amount of information is, of course, a great boon to a mail server administrator. This paper will show how an admin can use blacklist entries to not only block access from badly behaving mail servers, but, more importantly, allow access from so-called "known good" mail servers.

1.1 Traditional use of spamd(8)

Traditionally, OpenBSD users will use the spamd(8) daemon included with OpenBSD. This daemon will keep track of hosts it has communicated with, and put them in one of 3 lists. GREY, WHITE, and TRAPPED which are tracked in spamd's `spamdb`[5] database.

Whitelisted (WHITE) hosts do not talk to spamd(8), and are instead sent to the real mailserver.

Greylisted (GREY) hosts are not on the WHITE or TRAPPED lists. Normally these are hosts for which no SMTP[6] activity has been seen previously. These connections are redirected to spamd(8), and are given a temporary failure message when they try to deliver mail. Greylisted become Whitelisted after they retry delivery of the same message more than 3 times and are still retrying after 30 minutes of delay.

Trapped (TRAPPED) hosts are hosts which have been identified as doing bad things during the time they are Greylisted. Typically this is mailing to an unknown or spamtrap user, mailing with invalid smtp usage, or other spam signature activity. These hosts can be TRAPPED, typically for 24 hours, and will not be allowed to pass Greylisting during that time. This can be quite powerful because the trapping is only applied to hosts for which mail has never been exchanged before. As an example, it is not unusual for a legitimate mail server to mail to a large number of unknown users. However, it *is* unusual for a real mail server for which we have never exchanged mail with before to suddenly start mailing unknown users on its first communication with us.

Spamd also has the ability to use external blacklists, where clients on such a list will be given a rejection message specific to that list. This allows the default Greylisting behaviour to be combined with external lists of spam sources. The `spamd-setup(8)`[7] utility sends external blacklist data to `spamd`, as well as configuring mail rejection messages for blacklist entries. This utility uses method #1 to retrieve the list of blacklist entries.

In our case, we use BGP to distribute TRAPPED lists so that they may be used as external BLACK lists - as well as distributing selected WHITE entries so they can be shared among sites.

2 Definitions: Client vs Route Server vs Spamd Source

In this paper, we will discuss a reference implementation network. The authors will implement this network and it will be available for public use at "`rs.bgp-spamd.net`".

There are three important parts of the reference implementation network.

`Spamd Source`: These systems are feeding the Route Server with IP addresses fed from GREYTRAP and WHITE lists. They are the source of our `spamd` list information. Client systems are not able connect directly to the Spamd Source systems, their information will be sent via the Route Server.

Our reference implementation uses: University of Alberta (aka uatraps), Bob Beck (aka obtuse.com), and Peter Hansteen (aka bsdly.net) as our Spamd Sources.

`Route Server`: This system is the center hub of the network. Both the Spamd Sources and the Client systems connect. This system sorts and redistributes the BGP feeds from the Spamd Sources and distributes them to the Clients.

Our implementation uses the reference server, "rs.bgp-spamd.net"

`Client`: Any end-user.

Originally using sources from OpenBSD's `/etc/mail/spamd.conf`

3 Using BGP to distribute spamd lists

3.1 Basic explanation of BGP

In a traditional BGP network, Router A will send a list of IP networks assigned to itself, to all of its peers. Router B will also distribute its IP networks, as well as the IP networks assigned to Router A. Router A can mark specific IP networks with attributes, known as Communities, as a way to communicate with Router B some intentions for

these routes.

Some common examples of community meanings include “do not redistribute to a specific geographical area”, “Prefix this route 3 times (make it less desirable to the peers)”, and “Blackhole this network”.

3.2 Our use of BGP

In this paper, we will use the fact that BGP is designed to distribute IP networks with community attributes to distribute TRAPPED and certain WHITE entries from `spamd`. We want to do this for two reasons:

1. We distribute TRAPPED entries from trusted sources to use as BLACK lists - we are assuming that our trusted sources have a reasonable criteria for trapping hosts, and that TRAPPED hosts are short lived.
2. We distribute a subset of the WHITE entries from `spamd` - Our goal is to distribute entries that we are confident are “real” mailservers based on the information in the `spamdb` database. This list of “likely good mailservers” can be used by participants to establish a `bgp-spamd-bypass` table managed by `bgpd`[8] in `pf`. The advantage of this is that “real” mailservers communicating regularly with *any* participant will not be subjected to greylisting delays at *all* participants.

We have chosen some arbitrary BGP Communities for use in marking BLACK list and WHITE entries. The authors have chosen `$AS:666` for BLACK list entries, and `$AS:42` for WHITE entries. In this case, `$AS` will be expanded to the AS of the originating system. For additional filtering capabilities we will also copy this Community and mark it with the AS on the Route Server.

While it is possible for a regular BGP router to be a member of this experiment, the authors recommend against it. Clients MUST NOT make routing decisions based on the routes that we send, a large amount of host-specific routes will be injected into the our BGP network, and care needs to be taken to ensure that the behaviour of one BGP network does not affect the behaviour of the other. With our sample configuration files, client and server entities do not need to be directly connected, and can easily be routed over the regular internet, even through NAT.

3.3 Distributing spamd bypass lists

When a previously unseen host hits OpenBSD spamd, it is normally subject to greylisting - meaning initial connections will be delayed and forced to retry for 30 minutes. After 30 minutes, if the same tuple[9] of mail is still being retried the connecting host passes greylisting, and is whitelisted. Whitelisted hosts are kept for 30 days from the time traffic from them is last seen, and spamd counts number of connections passed, keeping whitelisted hosts from being subject to greylisting again as long as they continue to periodically send mail.

Some sites already use the recommendation of a `<nospamd>` table to bypass spamd for trusted networks - often those may include things like Google’s mailserver networks, etc.

What we want for a spamd bypass list from other sources is not to know “this host passed greylisting” - but rather “I am pretty sure this is a real mail server”. Using BGP puts a new and powerful dimension on this, as participating systems can send real-time information about hosts that send mail to us.

OpenBSD spamd’s database stores the time entries were first whitelisted as well as how many connections have been seen to pass to the real mailservers - this can be seen in the `spamdb` output.

We want to avoid putting “one off” servers - potentially spam software that retries connections, in the spamd bypass list - So instead what we do is we feed BGP a list of whitelist connections that have been around considerably longer than our expiry period (of 30 days), and who have made more than a trivial number of smtp connections. At the moment we have chosen any host that has been whitelisted for more than 75 days, and who has made at least 10 successful smtp connections. These hosts we share out in BGP so that they can be used by Client systems to bypass spamd checks as they can be considered “trusted”.

The power of this is then "real" mailservers who frequently exchange mail with one participant will not see greylisting delays with other participants who share the same bypass list distributed by BGP - effectively a well chosen bypass list coming from BGP amounts to a dynamic list of established mailservers communicating with all participants contributing to the list.

In our sample configuration, Client systems WILL NOT be allowed to make changes to the distributed lists. The Route Server WILL reject and ignore all submitted routes from Client systems, and all Spamd Sources are configured to reject connections from unknown systems. Additionally, the connections between the Spamd Sources and the Route Server are protected with a variety of extra mechanisms, to further prevent generic BGP hijacking or TCP layer attacks. These policies are to guarantee the integrity of the IP address lists, and to make sure that incorrect information is not distributed.

3.4 Blacklist Source Selection Criteria

In selecting our sources for addresses to use for our blacklist, the authors chose to be very conservative. All upstream Blacklist Sources ONLY select IP addresses that have sent emails to specific spamtrap addresses or have otherwise been marked as spam by the mail server administrator. These IP addresses are automatically expired after 24 hours. If the same IP address attempts to send an email to a spamtrap address during that time, the time counter will be reset.

While manual adding of addresses is possible, this is generally avoided.

The list of Spamd sources was selected by the authors, to be trusted systems with personally known administrators. The authors are concerned about invalid or malicious information being added to the network so care has been made that all information injected into this network will be based on trusted information.

All IPv4 addresses marked with the BLACK list community are /32 routes, which are IPv4 host-specific routes. This prevents us from accidentally blocking systems that have not actually sent us any spam, but may be a network "neighbour" of a spammer. This is enforced both on the Spamd source, and the Route Server.

As a side note, many of these sources are also found in the default OpenBSD `/etc/mail/spamd.conf` configuration file, and are generally well-known to the OpenBSD community.

3.5 Transmission Path from Spamd Source to Clients

When a Spamd Source wishes to add a specific IP address to the distributed whitelist, they run the equivalent of this command:

```
bgpctl network add 192.0.2.55/32 community $AS:42
```

where `192.0.2.55` is the desired IP address, and where `$AS` is the AS number assigned to this Spamd Source.

Once this address is added to the system, the Spamd Source BGP process will see the new route, and tell all of its peers (including the Route Server) about this new route. When the Route Server receives this, it will then also notify all of its peers, including the Client systems. A Client system will receive it, and use the "`match ... set pftable`" filter rule to add the IP address to the appropriate PF table.

A script to update the lists distributed by BGP on the Spamd Source is available in the Appendix.

4 Sample client configuration

Before we show a sample Client system configuration, the authors wish to show a sample of the output of `bgpctl show rib detail` for a single entry, as the information contained is the basis of our filtering. This example is a host route for `192.0.2.55` from `AS 65043`. The "Communities" entry shows that it has been marked with the "`65066:42`" and the "`65043:42`" Communities, which means we can make decisions based on one or both of them.

```
BGP routing table entry for 192.0.2.55/32
    65043
    Nexthop 198.18.0.191 (via ???) from 203.0.113.113 (203.0.113.113)
    Origin IGP, metric 0, localpref 100, weight 0, external
    Last update: 02:10:26 ago
    Communities: 65066:42 65043:42
```

4.1 Sample client pf.conf

This section is used to declare what filters will be applied to the various lists in PF. In this sample configuration, we will add a rule for the WHITE list `bypass`, to the default spamd(8) ruleset. In this sample, the default spamd(8) ruleset is indented for easy identification.

```
        table <spamd-white> persist
        # local bypass file.
        table <nospamd> persist file "/etc/mail/nospamd"

 # new bypass file from BGP.
 table <bgp-spamd-bypass> persist

 # we add this line
 pass in quick log on egress proto tcp from <bgp-spamd-bypass> to any port smtp
 # everything else goes to spamd

         # Exiting spamd(8) configuration
         pass in quick on egress proto tcp from <nospamd> to any port smtp
         pass in quick log on egress proto tcp from <spamd-white> to any port smtp
         pass in quick on egress proto tcp from any to any port smtp \
             rdr-to 127.0.0.1 port spamd
         pass out log on egress proto tcp to any port smtp
```

4.2 Sample client bgpd.conf

This section is used to connect to the Route Server and fetch the lists. After the lists are fetched, a filter is used to add and remove the IP addresses to the specified PF tables.

/etc/bgpd.conf

```
# Begin bgpd.conf

spamdAS="65066"

AS 65001
fib-update no   # Do not change the routing table

group "spamd-bgp" {
        remote-as $spamdAS
        multihop 64
        enforce neighbor-as no

        # rs.bgp-spamd.net
        neighbor 81.209.183.113
        announce none
}

# 'match' is required, to remove entries when routes are withdrawn
match from group spamd-bgp community $spamdAS:42  set pftable "bgp-spamd-bypass"

# EOF
```

4.3 Using spamd.conf to block BLACK list entries

A naive implementation can simply use PF to block BLACK list entries. This has the obvious disadvantage that any host that is blocked, will not know it is being blocked and can simply assume that the destination system is offline. Additionally, there will not be any way for the sender to know it is being blocked on purpose. This information is necessary for several corner cases, where a mail server triggered the BLACK list entry, but is still legitimate. In such a case, telling the sending server that they are on the BLACK list, allows for the administrator to use alternate means of contact to explain why their system should not be blocked.

For these reasons, the authors strongly recommend that Client systems use `spamd.conf`[10] as a means to inform systems on the BLACK list, that they are - in fact - blacklisted.

4.3.1 Blocking of Combined BGP blacklists

Here is a method to simply block all addresses on the BLACK list. This has the advantage of being simple for the Client system administrator, however it require that the Client system administrator determine the reasons why any address was blocked.

Below is a simple script for cron to update BLACK lists. It will print the host IP address of each entry to the `/var/mail/spamd.black` file, then run `spamd-setup` (which will be configured next). Here, we use the fact that the Route Serve marks all distributed BLACK list routes with the Community string `65066:666`. It is designed to be run from `cron` at a frequency faster than the greylist pass time, (for example, every 20 minutes) so that the trapped lists are updated on a regular basis.

```
   /usr/local/sbin/bgp-spamd.black.sh

#!/bin/sh
AS=65066

bgpctl show rib community ${AS}:666 | awk '{print $2}' | \
        sed 's/\/.*$//' > /var/mail/spamd.black

/usr/libexec/spamd-setup

# EOF
```

And a `spamd.conf` configuration file, for `spamd-setup`.

```
   /etc/mail/spamd.conf

# Configuration file for spamd.conf

all:\
        :bgp-spamd:

bgp-spamd:\
         :black:\
         :msg="Your address %A has sent mail to a spamtrap\n\
          within the last 24 hours":\
         :method=file:\
         :file=/var/mail/spamd.black:

# EOF
```

4.3.2 Separation of BGP blacklists

Here is a method to use the Communities attribute to separate the blacklists into their original sources. This method has the advantage of informing the sender exactly which list they were listed on so the sender can contact the originator of the filter list, instead of every mail administrator using these lists. However, the main disadvantage of this is that the Client systems will need to know some internal information about the BGP network, and keep an up-to-date list in their `spamd.conf` and helper scripts.

In this example script, the ASes "65042", "65043" and "65513" are used instead of the Route Server's AS of "65066". This is so we can split out these specific upstream Spamd Source that are providing the information. By specifically enumerating which lists that will be used, this will explicitly ignore any additional ASes that may be providing us with BLACK list hosts. Users will need to adjust their scripts for local preferences.

The following script is appropriate to update `spamd` from the BGP BLACK list entries. It is designed to be run from `cron` at a frequency faster than the greylist pass time, (for example, every 20 minutes) so that the BLACK lists are updated on a regular basis.

/usr/local/sbin/bgp-spamd.black.sh

```
#!/bin/sh

for AS in 65042 65043 65513; do
    bgpctl show rib community ${AS}:666 | awk {'print $2}' | \
        sed 's/\/.*$//' > /var/mail/spamd.${AS}.black
done

/usr/libexec/spamd-setup

# EOF
```

And a spamd.conf configuration file, for spamd-setup(8).

/etc/mail/spamd.conf

```
# Configuration file for spamd.conf

all:\
        :bgp65042:bgp65043:bgp65513:

bgp65042:\
         :black:\
         :msg="Your address %A has sent mail to a foad.obtuse.com spamtrap\n\
          within the last 24 hours":\
         :method=file:\
         :file=/var/mail/spamd.65042.black:
bgp65043:\
         :black:\
         :msg="Your address %A has sent mail to a ualberta.ca spamtrap\n\
          within the last 24 hours":\
         :method=file:\
         :file=/var/mail/spamd.65043.black:
bgp65513:\
         :black:\
         :msg="Your address %A has sent mail to a bsdly.net spamtrap\n\
          within the last 24 hours":\
         :method=file:\
         :file=/var/mail/spamd.65513.black:

# EOF
```

4.4 Non-OpenBSD Clients

While this paper focuses on using OpenBSD for all 3 types of systems, non-OpenBSD clients can also use these lists for their own anti-spam systems. While specific examples will not be discussed here, any user will need to filter based on BGP Communities, and insert/remove those addresses into their preferred system - provided their configuration does not alter the routes of the Client system.

Any reimplementation of this network can be done, as long as the Blacklist Source and Route Server systems are able to add large amounts (150k +) of arbitrary IP host-nets with specific BGP Communities. This is normal and very common when administering BGP networks, and should be possible with nearly all BGP server implementations.

4.5 Possible Risks to Client systems

While the risks of this configuration are minimal, there are still some possible issues.

1. Use of system resources. On a test system ran by one of the authors, the current (as of 2013-02-08) list of 103k WHITE entries, and 50k BLACK list entries, the `bgpd` process uses 43.5M of memory, and the `bgp-spamd-bypass` PF table is using approx 16M of memory. This can be a problem for low memory machines.

2. When the `bgpd` process ends, it will empty the `bgp-spamd-bypass` PF table and no longer update the spamd.conf BLACK list files. This will cause the amount of whitelisted systems to return to only what has been seen locally, and the age of the BLACK list entries will quickly grow stale and invalid. The Authors recommend that Client systems monitor and ensure that the `bgpd` is running.

5 Sample Route Server configuration

Here we describe an example configuration for the Route Server. In it, we connect to two Spamd Source systems, and we also provide access for Client systems. Connections to the Spamd Sources are protected. For the Spamd Source peer "upA", TCP MD5 signatures are used. For connections to Spamd Source peer "downB", we will use IPsec with dynamic keying. The `OpenBGPd` daemon will set up the flows, and uses `isakmpd(8)`[11] to manage the session keys.

bgpd.conf follows on next page

```
myAS="65066"

AS $myAS
router-id 203.0.113.113
fib-update no
nexthop qualify via default
transparent-as yes
socket "/var/www/logs/bgpd.rsock" restricted
socket "/logs/bgpd.rsock" restricted

group blacklist-sources {
        multihop 64
        announce none

        # Neighbor upA - John Q Public - john.public@example.com
        neighbor 198.51.100.198 {
                dump all in "/tmp/upA-all-in-%H%M" 3600
                descr "upA"
                remote-as 65198
                tcp md5sig key deadbeef
        }
        # Neighbor downB - Mike Bolton - mike@bolt.example.net
        neighbor 198.18.0.191 {
                dump all in "/tmp/downB-all-in-%H%M" 3600
                descr "downB"
                remote-as 65191
                ipsec ah ike
        }
}

group RS {
        announce all
        set nexthop no-modify
        enforce neighbor-as no
        announce as-4byte no
        multihop 64
        ttl-security no
        holdtime min 60
        softreconfig in no
        maxprefix 1 restart 5

        neighbor 0.0.0.0/0 {  passive  }
}

deny from any
allow from group blacklist-sources
allow to any

# Ensure that an IP to be blacklisted is only a host entry
deny from group blacklist-sources inet  \
        community neighbor-as:666 prefixlen < 32
deny from group blacklist-sources inet6 \
        community neighbor-as:666 prefixlen < 128

# Set my own community, so clients have an easy way to filter
match from group blacklist-sources \
        community neighbor-as:666 set community $myAS:666
```

6 Security Concerns

This is a completely different BGP network from the global routing table. Any routes added here will not be visible to or modify the global routing table, and will not prevent any IP traffic from flowing. Using the included configurations will not change the routing table of any connected system, and will not attempt to steal or re-route any traffic. Additionally, routes that are delivered to the Client systems are configured in such a way that routes will not be accepted by the kernel, because the desired `nexthop` will not be directly reachable.

These routes are intended to be added to the PF table as a way to assist `spamd` in being more effective - both in catching hosts that should not make their way through greylisting via a short term block list, and by allowing real mailservers through via a `bypass` pf table to avoid greylisting delays. These routes are not intended for use to restrict general connectivity.

6.1 Security Concerns for Spamd Sources

The sources for `spamdb` information used must be trusted to provide only valid information to the participants. It must be agreed ahead of time what the selection criteria for a "real" mailserver is - as well as what the criteria for trapping hosts is.

If a participant in the system is trapping hosts long term, this will affect all participating sites (for example, if one participant summarily decides `google.com` is evil and always traps their hosts, all people using their traplist to blacklist hosts will be affected). Similarly, if a participant in the system does not apply a sufficient degree of care to ensure entries published to the bypass list are "real" smtp servers - you run the risk of more spam leakage coming through.

Additionally, all sources must themselves be kept secure. Someone with nefarious intent who can manipulate one of the participant's BGP servers can easily publish erroneous information to either prevent mail from coming through at all, or allow lots of spam to bypass spamd. (Imagine if a participant were to advertise `0.0.0.0/0` as a member of the bypass or trapped lists.)

In order to protect the integrity of the IP address lists, it is recommended that Spamd sources protect their BGP sessions to the Route Servers with security features such as TCP MD5 signatures or IPSec tunnels.

6.2 Security Concerns for Route Servers

The Route Servers MUST NOT accept a `default` or `0.0.0.0/0` route from Spamd Sources. The Route Server needs to ensure that the entire internet is not either WHITE listed, nor BLACK listed.

The Route Server SHOULD NOT accept non-host routes from Spamd Sources. The authors strongly recommend that each and every host to be BLACK listed or WHITE listed instead be explicitly enumerated. Local implementations may adjust this for their own needs, but the authors recommend that sites be conservative with their lists, to allow actual behaviour dictate which list a host should be on.

The Route Server MUST NOT accept any entries from Client systems. Client systems are unknown and untrusted, and the administrator does not know of the quality of their lists.

Like all publicly facing systems, Route Servers SHOULD install security patches and be generally kept up to date.

6.3 Security Concerns for Clients

Client systems SHOULD NOT modify local routing table based on received routes and SHOULD block only `smtp` traffic based on the received routes. The ability to ping or connect to a website on the same IP address as a BLACK list host is valuable, as well as emailing the administrator of the BLACK list host. Additionally, WHITE entries

should not modify the routing table, as we are only listing host IP addresses, and not a “better” route for these hosts.

7 Future Work

Future work on this topic include collecting statistics about the addresses being distributed, as well as the churn rate. Additionally, work on simplifying inserting the traplist addresses into spamd(8) is desired.

8 Acknowledgements

Many thanks to Peter N.M. Hansteen of BSDly.net, Bob Beck of obtuse.com, and the University of Alberta at ualberta.ca for being sources of spamdb information.
Thanks to Claudio Jeker and Stuart Henderson for assistance with BGP design and behaviour.

9 Availability

This paper, as well as client configuration examples are available at

http://www.bgp-spamd.net/

It is planned that the reference server “rs.bgp-spamd.net” will provide this service for the entirety of calendar year 2013. In December 2013, there will be an announcement about the future status of this project.

Please note that this reference server is an experiment and is subject to modification and closing. The authors will attempt to provide reasonable notice before closing the list, however no guarantees can be made.

An announcement mailing list will be available via the website, for important Client system announcements, as well as the general status of the reference implementation.

Appendix

The following is appropriate to update the BGP source from spamd output. It is designed to be run from cron at a frequency faster than the greylist pass time, (for example, every 10 minutes) so that the trapped lists are updated before machines are passed on other sites.

```
#!/usr/bin/perl
# Copyright (c) 2013 Bob Beck <beck@obtuse.com>  All rights reserved.
#
# Permission to use, copy, modify, and distribute this software for any
# purpose with or without fee is hereby granted, provided that the above
# copyright notice and this permission notice appear in all copies.
#
# THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
# WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
# ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
# WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
# ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
# OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

# perl to parse spamdb output and deal with bgpd. can take multiple
# spamdb output from multiple mail servers on command line, talks
# to bgpd on localhost to update whitelist.
```

```
# for typical use from a number of spamd collectors collect spamdb output
# in a number of files, and then run this script on the bgpd machine with
# all the files as command line args to the script.

# so typical use from cron is either
#
# ssh machine1 "spamdb" > machine1.spamdb
# ssh machine2 "spamdb" > machine2.spamdb
# bgpspamd.perl machine1.spamdb machine2.spamdb
#
# If bgpd is not running on the spamd machine, or if spamd and bgpd are
# only running on the same machine
#
# spamdb | bgpspamd.perl
#

use integer;

# AS to use - please change from this value.
my $AS = 65535;
# community string for spamd bypass - where trusted whitelist entries get sent.
my $CS = 42;
# community string for spamd traps - where we send traps.
my $TCS = 666;

# These two control how we pick only good whitelist entries to
# recommended for spamd bypass - we want to only send things we are
# relatively certain are "real" mailservers - not leakage from
# spam software that retries. for this to be effective we do have
# to be logging real connecitons to our mailservers and noticing
# them with spamlogd.
#
# Only distribute white entries that are older than 75 days
my $agelimit = 3600 * 24 * 75;
# Only distribute white entries that have made more than 10 smtp connections.
my $maillimit = 10;

my %ips;
my %tips;
my $key;

while (<>) {
    my $now = time();
    if (/^WHITE/) {
        chomp;
        my @line = split(/\|/);
        if (($line[5] < $now - $agelimit) && $line[8] > $maillimit) {
            $ips{"$line[1]"}=1;
        }
    } elsif (/^TRAPPED/) {
        chomp;
        my @line = split(/\|/);
        $tips{"$line[1]"}=1;
    }
}
open (BGP, "bgpctl show rib community $AS:$CS|") || die "can't bgpctl!";
while (<BGP>) {
    if (/^AI/) {
        chomp;
        my @line = split;
        my $ip = $line[1];
```

```
        $ip =~ s/\/.*$//;
        $ips{$ip}--=1;
    }
}
close(BGP);

open (BGP, "bgpctl show rib community $AS:$TCS|") || die "can't bgpctl!";
while (<BGP>) {
    if (/^AI/) {
        chomp;
        my @line = split;
        my $ip = $line[1];
        $ip =~ s/\/.*$//;
        $tips{$ip}--=1;
    }
}

close(BGP);

foreach $key (keys %ips) {
    if ($ips{$key} > 0) {
        system "bgpctl network add $key community $AS:$CS > /dev/null 2>&1\n";
    } elsif ($ips{$key} < 0) {
        system "bgpctl network delete $key community $AS:CS > /dev/null 2>&1\n";
    }
}

foreach $key (keys %tips) {
    if ($tips{$key} > 0) {
        system "bgpctl network add $key community $AS:$TCS > /dev/null 2>&1\n";
    } elsif ($tips{$key} < 0) {
        system "bgpctl network delete $key community $AS:$TCS > /dev/null 2>&1\n";
    }
}
```

Notes

[1] spamd(8), spamd - spam deferral daemon, OpenBSD manual pages
[2] spamdlogd(8), spamlogd - spamd whitelist updating daemon, OpenBSD manual pages
[3] RFC4271, Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)", January 2006
[4] pf(4), pf - packet filter, OpenBSD manual pages
[5] spamdb(8), spamdb - spamd database tool, OpenBSD manual pages
[6] RFC 822, Postel, J., "SIMPLE MAIL TRANSFER PROTOCOL", August 1982
[7] spamd-setup(8), spamd-setup - parse and load file of spammer addresses, OpenBSD manual pages
[8] bgpd(8), bgpd - Border Gateway Protocol daemon, OpenBSD manual pages
[9] connecting IP address, HELO/EHLO, envelope-from, and envelope-to of the connection
[10] spamd.conf(5), spamd.conf - spamd configuration file, OpenBSD manual pages
[11] isakmpd(8), isakmpd - ISAKMP/Oakley a.k.a. IKEv1 key management daemon, OpenBSD manual pages

Calloutng: a new infrastructure for timer facilities in the FreeBSD kernel

Davide Italiano
The FreeBSD Project
davide@FreeBSD.org

Alexander Motin
The FreeBSD Project
mav@FreeBSD.org

ABSTRACT

In BSD kernels, timers are provided by the callout facility, which allows to register a function with an argument to be called at specified future time. The current design of this subsystem suffer of some problems, such as the impossibility of handling high-resolution events or its inherent periodic structure, which may lead to spurious wakeups and higher power consumptions. Some consumers, such as high-speed networking, VoIP and other real-time applications need a better precision than the one currently allowed. Also, especially with the ubiquity of laptops in the last years, the energy wasted by interrupts waking CPUs from sleep may be a sensitive factor. In this paper we present a new design and implementation of the callout facility, which tries to address those long standing issues, proposing also a new programming interface to take advantage of the new features.

Categories and Subject Descriptors

[**Operating systems**]; [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Kernel, Timers, C, Algorithms, Data structures, POSIX

1. INTRODUCTION

A certain number of computer tasks are generally driven by timers, which acts behind the scenes and are invisible for user. For example, network card driver needs to periodically poll the link status, or the TCP code needs to know when it is time to start retransmission procedures after acknowledgement not received in time. In FreeBSD [1] and other BSD kernels, the interface that allows programmer to register events to be called in the future takes the name of callout. The design of this subsystem was not modified for years and suffers of some issues, such as measuring time in fixed periods, waking up the CPU periodically on each period even if there are no events to process, inability group together close events from different periods, etc.
The calloutng project aims to create a new timer infrastructure with the following objectives:

- Improve the accuracy of events processing by removing concept of periods
- Avoid periodic CPU wakeups in order to reduce energy consumption
- Group close events to reduce the number of interrupts and respectively processor wakeups
- Keep compatibility with the existing Kernel Programming Interfaces (KPIs)
- Don't introduce performance penalties

The rest of the paper is organized as follows: Section 2 gives a relatively high-level description of the design as originally proposed and some improvements have been done during the years in FreeBSD-land, while Section 3 deeply analyzes the problems present in the current approach. In Section 4 the new design and implementation are explained and its validity is shown via experimental results in the successive section. The work is concluded with some considerations and future directions.

2. STATE OF THE ART

The callout code in FreeBSD is based on the work of Adam M. Costello and George Varghese. A detailed description can be found in [2]. The main idea is that of maintaining an array of unsorted lists (the so-called callwheel, see Figure 1) in order to keep track of the outstanding timeouts in system. To maintain the concept of time, the authors used a global variable called *ticks*, which keeps track of the time elapsed since boot. Historically, the timer was tuned to a constant frequency hz, so that it generated interrupts *hz* times per second. On every interrupt function *hardclock()* was called, incrementing the ticks variable.
When a new timeout is registered to fire at time t (expressed in ticks), it's inserted into the element of the array, addressed by (t *mod* callwheelsize), where callwheelsize is the size of the array and *mod* indicates the mathematical modulo operation. Being the lists unsorted, the insertion operation takes minimal constant time. On every tick, the array element pointed by (ticks *mod* callwheelsize) is processed to see if there are expired callouts, and in such cases the handler functions are called. If callwheelsize is chosen to be comparable to the number of outstanding callouts in the system, lists will be (in average) short.
The main advantage of using this strategy relies on its simplicity: the callout code has O(1) average time for all the operations, and O(1) worst case for most of them. In order to process events, the code needs only to read a global kernel variable, which is very cheap, in particular compared to hardware timecounter, which can be quite expensive. Also, this design requires neither specific hardware architecture

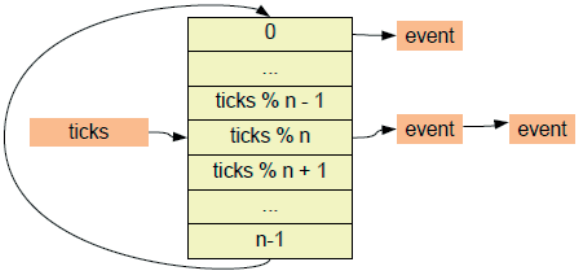


Figure 1: The callwheel data structure

nor specific OS infrastructure. It needs only a timer (no restrictions, because almost any timer has the ability to periodically generate interrupts or simulate this behaviour), and a subsystem for handling interrupt.

While the current FreeBSD implementation shares basic concepts with the mechanisms described above, it evolved quite a bit during the years, introducing substantial improvements in term of performances/scalability and specifying a different programming interface. About the former, the single callwheel was replaced by a per-CPU callwheels, introducing respective migration system (2008).

For what concerns the latter, it is provided via an additional function that consumers can use to allocate their callouts, namely *callout_init()* and its locked variants (*callout_init_mtx()*, *callout_init_rw()* ...). This way the callout subsystem doesn't have to deal internally with object lifecycle, passing this responsibility to the consumer.

Also, KPI was extended, in order to reflect the introduction of per-CPU wheels, adding a new *callout_reset_on()* function, that allows to specify a CPU to run the callout on. It worth to say that the callout processing threads are not hard bound to each CPU, so they can't be starved into not processing their queues. They are *medium* bound by ULE and probably tend to run where they are scheduled on 4BSD but there is no guarantee. The ULE will run them on their native CPU if it's running something of lower priority or idle. If it's running something of greater priority, it will look for a more idle CPU. This prevents the softclock thread from being starved by other interrupt thread activity.

3. PROBLEMS WITH CURRENT IMPLEMENTATION

. The constant frequency *hz* determines the resolution at which events can be handled. On most FreeBSD machines this value is equal to one thousand, which means that one ticks corresponds to 1 millisecond. This way, even a function which specify timeout using accurate time intervals with resolution of one nanosecond (e.g. nanosleep) will see this interval rounded up to the nearest tick, i.e. it will be like the resolution is one millisecond. As soon as many callers require interval to be no shorter than specified, the kernel has to add one more tick to cover possibility that present tick is almost over. As result, the best we can get is resolution of one millisecond with one millisecond of additional latency.

Also, independently from the time for which the next event is scheduled, CPU is woken up on every tick to increment the global ticks variable and scan the respective bucket of the callwheel. These spurious and useless CPU wakeups directly translate in power dissipation, which e.g. from a laptop user point of view results in a reduction of the on-battery time. Stiil talking about power consumption, the actual mechanism is unable to coalesce or defer callouts. Such a mechanism could be useful for istance in low power modes, where the system could postpone check for a new CD in the drive if CPU is sleeping deep at the moment.

The last but not the least, all the callouts currently run from software interrupt thread context. That causes additional context switches and in some cases even wakeup of other CPUs.

4. A NEW DESIGN

In order to address the first of the present problems, the interfaces to services need some rethinking. For what concern the kernel, all the Kernel Programming Interfaces dealing with time events are built around the concept of 'tick'. All consumers specify their timeouts via the callout_*() interface as relative number of ticks. 'ticks' variable is a 32-bit integer, and it is not big enough to represent time with resolution of microseconds (or nanoseconds) without quickly overrunning (overflowing). Change is definitely needed there in order to guarantee higher granularity.

For what concern the userland, the Application Programming Interfaces (APIs) and data types are standardized by Portable Operating System Interface (POSIX), so they cannot be changed without losing compliance. Luckily, most of the functions provide variants with acceptable degree of precision.

Let's take, as an example, the nanosleep() system call. This service asks the kernel to suspend the execution of the calling thread until either at least the time specified in the first argument has elapsed, or the call will be aborted by signal delivered to the thread. The function can specify the timeout up to a nanosecond granularity.

Other services (e.g. select()), which monitor one or more file descriptor to see if there's data ready, can limit the amount of time they monitor. This time is specified in one function argument. Differently from what happen in the previous case, the granularity is that of microseconds. Unfortunately, as long as all those system calls dealing with timeouts rely on callout(9) at the kernel level, they're limited by the precision offered by the kernel. So, as first step a new data type is needed to represent more granular precision.

From a system-wide perspective, FreeBSD has the following data types to represent time intervals:

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
};

struct timeval {
    time_t tv_sec;
    long tv_usec;
};

struct bintime {
    time_t bt_sec;
    uint64_t bt_frac;
};
```

The first type allows to specify time up to microseconds granularity, while the second up to nanosecond one. The common part between the two representation is that they're pseudo-decimal. Conversely, the bintime structure is a binary number, and this has some advantages, first of all making easier mathematical operations on the type. Depending on the underlying architecture, these struct have different sizes. In particular, they range respectively from 64 to 128 bit (for timeval and timespec) and from 96 to 128 bit (for bintime) [3].
The binary nature of bintime could make it suitable for our purposes (in fact it was originally chosen as default data type), but there are some problems that should be considered. First of all, using 128 bit (on amd64, probably the most widely spread platform) is overkill, because at best hardware clocks have short term stabilities approaching 1e-8, but likely as bad as 1e-6. In addition, compilers don't provide a native *int128_t* or *int96_t* type.
The choice made was that of using A 32.32 fixed point format, fitting it into an *int64_t*. Not only this makes mathematical operations and comparison trivial, but allows also to express 'real world' units for time such as microseconds or minutes trivially.

```
typedef sbintime_t   int64_t;
#define SBT_1S  ((sbintime_t)1 << 32)
#define SBT_1M  (SBT_1S * 60)
#define SBT_1MS (SBT_1S / 1000)
#define SBT_1US (SBT_1S / 10000000)
#define SBT_1NS (SBT_1S / 10000000000)
```

Now that the type is decided, the programming interface in the kernel requires to be adapted. Changing existing functions arguments is considered discouraging and intrusive, because it creates an huge KPI breakage for third-party software, considering the popularity of callout in many kernel subsystems, including device drivers. So it's been decided to maintain the old set of functions, introducing a new set of alternative functions, which rely on the aforementioned chosen type to represent time. Other than changing a field in order to specify the expiration time, the new field has been introduced in order to specify precision tolerance. This field may be provided by the consumer to specify a tolerance interval it allows for the scheduled callout. In other words, specifying a non-zero precision value, the consumer gives an hint to the callout backend code about how to group events. To sum up, the resulting KPI takes the following form:

```
int callout_reset_sbt_on (..., sbintime_t
     sbt, sbintime_t precision, int flags
    );
```

Some functionalities, e.g. the precision-related bits, may be useful also in the old interface. A new KPI for old customers also has been proposed, so that they can take advantage of the newly introduced features.

```
int callout_reset_flags_on (..., int
    ticks, ..., int flags);
```

Thus, an extension was made to the existing condvar(9), sleepqueue(9), sleep(9) and callout(9) KPIs by adding to them functions that take an argument of the type sbintime_t instead of int (ticks to represent a time) and a new argument to represent desired event precision. In particular:

```
int cv_timedwait_sbt (..., sbintime_t sbt
    , sbintime_t precision);
int msleep_sbt (..., sbintime_t sbt,
    sbintime_t precision);
int sleepq_set_timeout_sbt (...,
    sbintime_t sbt, sbintime_t precision)
    ;
```

In parallel to the KPI changes, there's need to adapt the callout structure in order to store the added bits. The struct callout prior to changes had the following form:

```
struct callout {
    ...
    int c_time;
    void  *c_arg;
    void  (*c_func)(void *);
    struct lock_object *c_lock;
    int c_flags;
    volatile int c_cpu;
};
```

As the reader can easily imagine, the c_time field, which stores the expiration time need to be replaced to sbintime_t in order to keep track of the new changes. Similarly, to store precision, a new field need to be added. Although the callout interface doesn't require direct access to the elements of the 'struct callout', which is an opaque type, clients need to allocate memory for it using the callout_init functions.. The result is that the size of the struct callout is a part of the KBI. The use of larger data types leads to an increase in the size of the structure and inevitably break the existing KBI. This change requires rebuilding of almost all the kernel modules and makes impossible to merge these changes to the existing STABLE branch, because the FreeBSD policy requires (or at least recommends) to preserve binary compatibility among different releases with the same major number (in this case FreeBSD-9).

Changes to the backend data structure

During the initial planning stage we've analyzed possibility of using different tree-based data structures to store events. They can provide much better results on operation of fetching the first event, but have more complicated insertion and removal operation. As soon as many events in system are cancelled and never really fire (for example, device timeouts, TCP timeouts, etc.), performance of insert/remove operations is more important. But tree-like algorithms typically have O(log(n)) complexity for such operations, while callwheel has only O(1). Also tree-like algorithms may require memory (re-)allocation during some insert operation. That is impossible to do with present callout locking policy, that allows to use it even while holding spin-locks.

As result, the underlying design has been maintained, but slightly refreshed. The wheel is going to use as hash key subset of time bits, picking some of them from the integer part of sbintime_t and some from the fractional, so that the key changes sequentially approximately every millisecond. The new hash function is translated into code as follows:

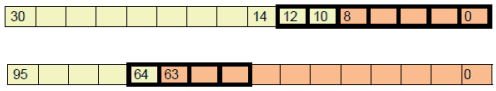


Figure 2: Old and new callout hash keys compared

```
#define CC_HASH_SHIFT   10

static inline int
callout_hash(sbintime_t sbt)
{

    return ((int)(sbt >> (32 -
        CC_HASH_SHIFT))) & callwheelmask;
}
```

where the last bitwise AND operation realizes modulo being callwheel size constrained to be a power-of-two. The compared behaviour of the hash in the two cases is shown in Figure 2. The following motivations affect callwheel parameters now:

- The callwheel bucket should not be too big to not rescan events in current bucket several times if several events are scheduled close to each other.
- The callwheel bucket should not be too small to minimize number of sequentially scanned empty buckets during events processing.
- The total number of buckets should be big enough to store (if possible) most of events within one callwheel turn. It should minimize number of extra scanned events from distant future, when looking for expired events.

As result, for systems with many tick-oriented callouts, bucket size can be kept equal to the tick time. For systems with many shorter callouts it can be reduced. For mostly idle low-power systems with few events it can be increased.

Obtaining the current time

The new design poses a new challenge. The conversion to struct bintime instead of tick-based approach to represent time makes impossible to use low-cost global variable ticks to get the current time. There are two possibilities to get time in FreeBSD: the first, going directly to the hardware, using binuptime() function, the second, using a cached variable which is updated from time to time, using getbinuptime(). It was decided to use a mixed approach: in cases where high precision needed (e.g. for short intervals) use heavy but precise function binuptime(), and use light but with 1ms accuracy getbinuptime() function in cases where precision is not essential. This allows to save time by using better accuracy only when necessary.

Accuracy

In particular situations, e.g. in low-power environments, for the correct and effective use of resources, events need to have information about the required accuracy. The new functions introduced in kernelspace allow to specify it as absolute or relative value. In order to store it, the callout structure has been augmented by adding a 'precision' field of type struct bintime, which represents the tolerance interval, allowed by the consumer of callout. Unfortunately, this is not possible for the old interface. None of existing functions in kernel space and none functions in userspace have such (or similar) argument, and therefore only an estimation might be done, based on the timeout value passed and other global parameters of the system (e.g. hz value). The default level of accuracy (in percentage, for example) can be set globally via the syscontrol interface.
The possibility of aggregation for events is checked every time the wheel is processed. When the callout_process() function is executed, the precision field is used together with the time field to find a set of events which allowed times overlap, which can be executed at the same time.

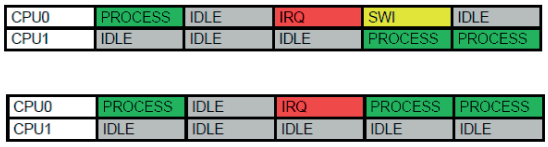


Figure 3: Traditional and SWI-less callout comparison

CPU-affinity and cache effects

The callout subsystem uses separate threads to handle the events. In particular, during callout initialization, a software interrupt threads (SWI) for each CPU are created, and they are used to execute the handler functions associated to the events. Right now, all the callouts are executed using these software interrupts. This approach has some advantages. Among others, that it is good in terms of uniformity and removing many of limitations, existing for code, executing directly from hardware interrupt context (e.g. the ability to use non-spin mutexes and other useful locking primitives). However, the use of additional threads complicates the job of the scheduler. If the callout handler function will evoke another thread (that is a quite often operation), the scheduler will try to run that one on a different CPU, as long as the original processor is currently busy running the SWI thread. Waking up another processor has significant drawbacks. First of all, waking up another processor from deep sleep might be a long (hundreds of microseconds) and energy-consuming process. Besides, it is ineffective in terms of caches usage, since the thread scheduled on another CPU, which caches unlikely contain any useful data.
Let's see an example. Some thread uses the tsleep(9) interface to delay its execution. tsleep(9) at a lower-level relies on the sleepqueue(9) interface, which uses callout(9) to awake at the proper time. So what's happening in the callout execution flow: the SWI thread is woken up for the single purpose of waking up the basic thread! And as described above, very likely wake it up on another processor!
The analysis shows, that example above is relevant to all customers using sleep(9), sleepqueue(9) or condvar(9) KPIs, and respectively all userland consumers which rely on those

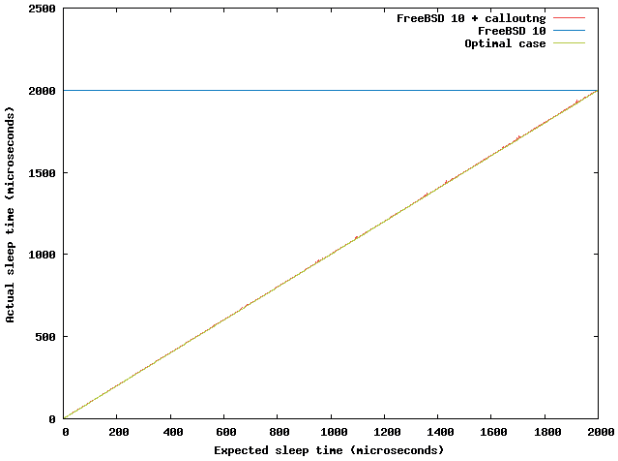


Figure 4: Calloutng performances on amd64: absolute precision

primitives: poll(2), select(2), nanosleep(2), etc.
In order to solve this problem the mechanism of direct execution has been implemented. Using the new C_DIRECT_EXEC flag the callout(9) consumer can specify that event handler can be executed directly in the context of hardware interrupt. This eliminates the use of SWI for this handler by the cost of enforcing additional constraints. According to our tests, use of this technique, where applicable, allows significantly reduce resource consumption and latency of the event processing, as well as, in theory, improve efficiency and reduce pollution of the CPU caches.

5. EXPERIMENTAL RESULTS

When using TSC timecounter and LAPIC eventtimer, the new code provides events handling accuracy (depending on hardware) down to only 2-3 microseconds. The benchmarks (real and synthetic) shown no substantial performance reduction, even when when used with slow timecounter hardware. Using the direct event handlers execution shown significantly increased productivity and accuracy, while also improving system power consumption.
The following tests have been run measuring the actual sleep time of the usleep() system call, specifying as input a sequential timeout starting from one microseconds and increasing it. The system used was Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz, equipped with 8GB of RAM. The couple LAPIC+TSC has been used. In the graphs the x-axis represent the input passed to usleep() and the y-axis represent the real sleep time, both expressed in microseconds. Figure 4 shows how calloutng behaves when the users requires absolute precision, while Figure 5 show the result when the precision tolerance has been set to 5 percent, in order to reduce the number of interrupts and exploit the benefits of aggregation. The same two tests have been repeated also on ARM hardware, SheevaPlug – Marvell 88F6281 1.2GHz, 512MB RAM, and the corresponding results are shown in Figure 6 and Figure 7.

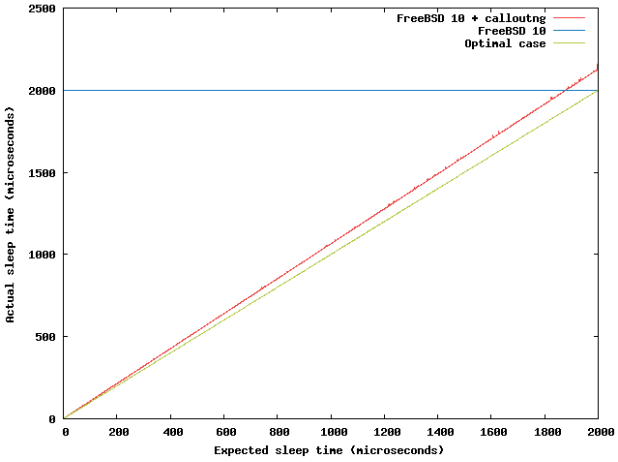


Figure 5: Calloutng performances on amd64: 5 percent precision tolerance

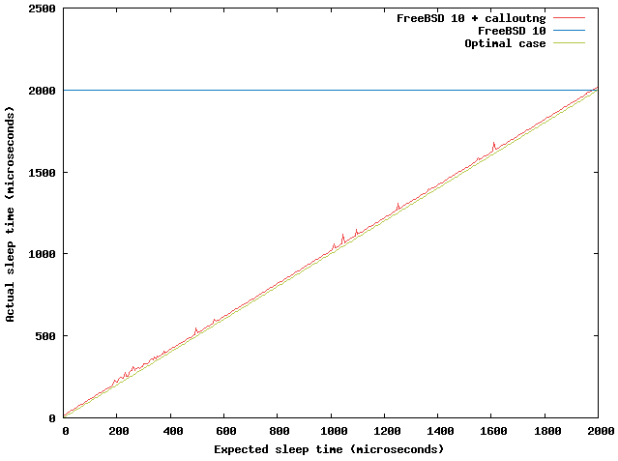


Figure 6: Calloutng performances on ARM: absolute precision

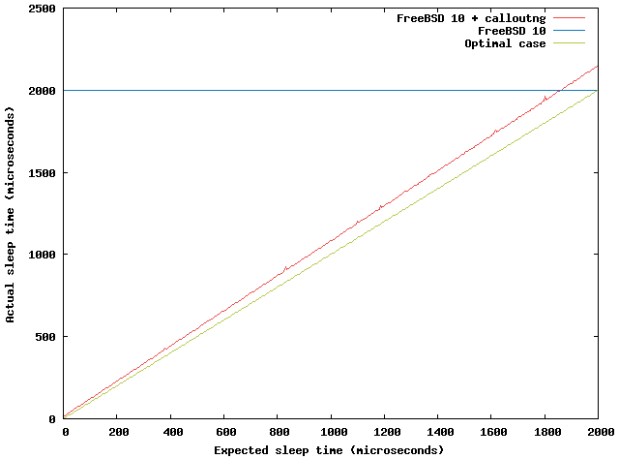


Figure 7: Calloutng performances on ARM: 5 precent precision tolerance

6. CONCLUSIONS

The code written as part of this project going to be the part of the forthcoming FreeBSD 10.0 release. Further wider deployment of newly implemented APIs in different kernel subsystems will provide additional benefits. Some user-level APIs could be added later to specify events precision for the specified thread or process.

7. ACKNOWLEDGMENTS

We would like to thank Google, Inc. (Google Summer of Code Program, 2012 edition) and iXsystems, Inc. for sponsoring this project, as well as the FreeBSD community for reviewing, discussing and testing the changes we made.

8. REFERENCES

[1] The FreeBSD Project. `http://www.freebsd.org/`.

[2] A. M. Costello and G. Varghese. Redesigning the BSD Callout and Timer Facilities. November 1995.

[3] P.-H. Kamp. Timecounters: Efficient and precise timekeeping in SMP kernels. June 2004.

SCTP in Go

Olivier Van Acker
Department of Computer Science and Information Systems
Birkbeck University of London
London, United Kingdom
Email: olivier@robotmotel.com

***Abstract*—This paper describes a successful attempt to combine two relatively new technologies: Stream Control Transmission Protocol (SCTP) and the programming language Go, achieved by extending the existing Go network library with SCTP.**

SCTP is a reliable, message-oriented transport layer protocol, similar to TCP and UDP. It offers sequenced delivery of messages over multiple streams, network fault tolerance via multihoming support, resistance against flooding and masquerade attacks and congestion avoidance procedures. It has improvements over wider-established network technologies and is gradually gaining traction in the telecom and Internet industries.

Go is an open source, concurrent, statically typed, compiled and garbage-collected language, developed by Google Inc. Go's main design goals are simplicity and ease of use and it has a syntax broadly similar to C. Go has good support for networked and multicore computing and as a system language is often used for networked applications, however it doesn't yet support SCTP.

By combining SCTP and Go, software engineers can exploit the advantages of both technologies. The implementation of SCTP extending the Go network library was done on FreeBSD and Mac OS X - the two operating systems that contain the most up to date implementation of the SCTP specification.

***Index Terms*—Stream Control Transmission Protocol (SCTP); Transmission Control Protocol (TCP); Go; Networking;**

I. Introduction

This paper will look into combining two relatively new technologies: a network protocol called SCTP and the programming language Go. Both technologies claim to offer improvements over existing technologies: SCTP does away with the limitations of the widely used TCP protocol **(author?)** [8]; and Go was designed with simplicity and minimized programmer effort in mind, thereby preventing a forest of features getting in the way of program design: Less is exponentially more **(author?)** [6]. The current version of the Go network library does not support the SCTP protocol and this paper examines how easy it is to extend the Go network library with this protocol. The work in this paper is based on the dissertation submitted for an MSc in Computer Science at Birkbeck University in London and is available as an open source project.

A. Relevance

After ten years SCTP as a technology is becoming more and more relevant. One notable implementation is the use of SCTP as a data channel in the Web Real-Time Communication (WebRTC) standard **(author?)** [11], a HTML 5 extension to enable real time video and audio communication in browsers. Google inc. and the Mozilla Foundation are each planning to release a browser this year implementing the WebRTC standard. Go as a language is very new and it is too early to say what impact it will have. It does however receive a lot of media attention since it is developed by Google. It is also growing in popularity because of its ease of use as a concurrent system language **(author?)** [1].

B. Outline

Section II presents an overview of Go and SCTP, followed by (section III) a description of how the TCP socket API is integrated in the Go networking library. This is a starting point for the design of an SCTP extension to the network library in Go, described in section IV. Section V explains the implementation. Section VI analysis the results and VII concludes.

II. Technology overview

In this section I will give an overview of the main features of SCTP and Go.

A. SCTP

SCTP was initially developed in response to the demands of the telecoms industry, which was not satisfied with the reliability and performance of TCP **(author?)** [10, p. 15]. During the design phase the decision was made to make SCTP a less telephony-centric IP protocol **(author?)** [10, p. 16] so that it could also be used for more generic data transport purposes.

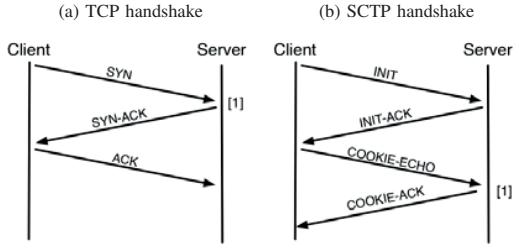
1) Comparison with TCP: It is fruitful to compare TCP and SCTP, as TCP is the most widely-used protocol **(author?)** [5] and SCTP is very similar to it:

2) Multihoming: A host is said to be multihomed if it has multiple IP addresses which can be associated with one or more physical interfaces connected to the same or different networks**(author?)** [2]. TCP can only bind to one network address at each end of the connection. In the event of a network failure there is no way to stay connected and send the data over another physical path without setting up a new connection. SCTP natively supports multihoming at the transport layer. This makes it possible to connect and transfer data to and from multihomed hosts, so that when one network path fails, the connection seamlessly fails over using the remaining paths. And together with concurrent multipath transfer (CMT) **(author?)** [4] it is possible to increase data throughput by transferring data simultaneously over multiple links.

3) In-built message boundaries: In TCP there are no markers to indicate the start or end of a specific piece of data (a user message). All data sent over the socket is converted into a byte stream and an application must add its own mechanism to be able to reconstruct the messages. In SCTP the message boundaries of the data sent are preserved in the protocol. In the event of the message being larger than the maximum package size a notofication is sent to the application layer more is on its way.

4) Protection against Denial of Service (DOS) attacks: Another disadvantage of TCP is that it is open to 'SYN flood attack', a specific type of attack which can drain the server of resources, resulting in a denial of service (nothing else can connect). To prevent this, SCTP uses a four-way handshake to establish the connection, whereas TCP only uses a three-way handshake. With a four-way handshake the originator has to double-acknowledge itself ("is it really you?") by resending a cookie it previously received from the destination server before that server will assign resources to the connection. This prevents timeouts on the server side and thus makes this type of denial of service impossible. To reduce start up delay, actual data can also be sent during the second part of the handshake.

Figure 1: Initiating a network connection



5) SCTP Multistreaming: SCTP user messages in a single SCTP socket connection can be sent and delivered to the application layer over independent streams. In case of two sets of user messages A and B, each set delivered sequentially, the messages of set A can be sent over a different stream than B. And in case a messages in set A gets missing and part of the sequence needs to be resent this will not effect the data of set B if it is sent over a different stream. This tackels the 'head of line blocking' problem (figure 2) where messages already delivered need to be redelivered because they have to arrive in order and becuase one message did not arrive.

6) Associations: A connection between a SCTP server and client is called an association. An association gets initiated by a request from a SCTP client. So a listening server can accept incoming requests from multiple clients. Messages sent over the assocation have an association id attached to them. to make it possible to know where they come from. Within a single association you can have multiple streams for data transmission (See figure 3)

7) Two programming models: SCTP has two interfaces for implementation in a networked application: one-to-one and

Figure 2: Head of line blocking

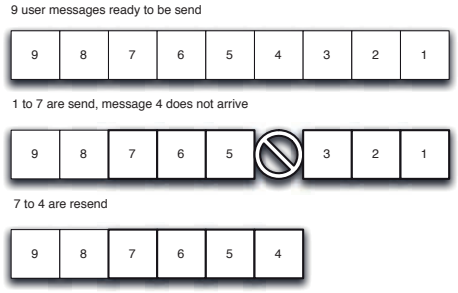
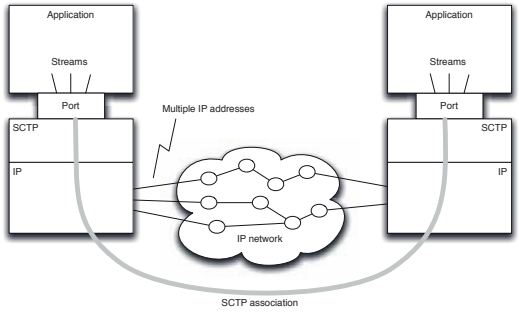


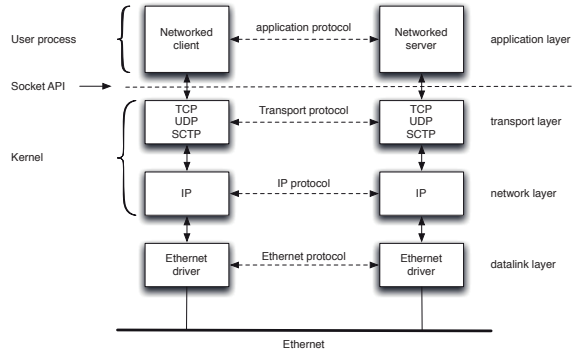
Figure 3: Associations and streams



one-to-many. A single socket in a one-to-many model can have multiple incoming associations, meaning that multiple clients can connect to a single server listening on a single socket. The one-to-one model can only have a single association per socket. The one-to-one model is for easy migration of existing TCP applications, it maps one-to-one to the system calls TCP makes to establish a connection. But it can only have one connection per association, and thus only a single client can connect to a server. The one-to-one interface makes migrating an existing application a relatively painless exercise. If you want to upgrade your existing TCP application to a one-to-many SCTP application, significant retooling is needed. **(author?)** [7, p. 267]

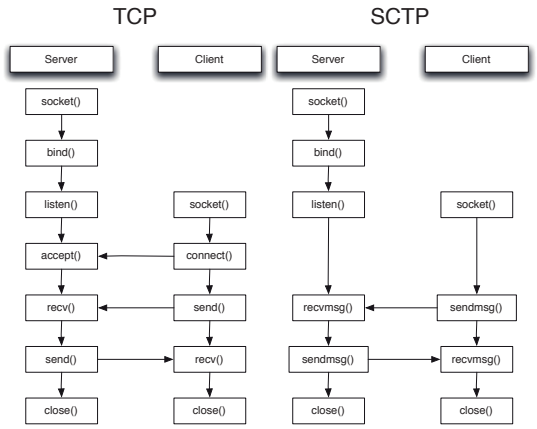
8) Socket API: Figure 4 is an overview of two applications communicating over an IP network using a transport protocol (TCP, UDP or SCTP). The software engineer writing the client and server in the application layer only has to know about the function calls exposing the functionality in the transport layer. This collection of functions is called the socket API. A socket is an endpoint for networked communication; multiple processes can open sockets via the API and data written into one socket can be read from another. The socket API consist of functions which can open and close the socket, send data over it and set the behavior of a socket using 'socket options'. An example of such behavior is the enabling or disabling of data buffering before sending to reduce the number of packets to sent over the network and therefore improve efficiency

Figure 4: Socket API



(Nagle's algorithm).

Figure 5: TCP and SCTP socket API



9) SCTP Socket API: Figure 5 is an overview of both the TCP and SCTP socket APIs and gives the order in which the system calls would be used in a simple client server application sending and receiving messages. The server first creates a socket, this returns a file descriptor which can be used by the bind function to to make a request to assign a address to it. After this the server can start listening to incoming connections with the listen() function. After this point TCP is different from SCTP. The TCP client actively connects to the peer and the server accepts the connection with the accept() and connect() functions. With SCTP the connection set up hand handshake happens implicitly when sending and receiving a message. At this point the server and client can exchange messages and finally the connection terminates with the close() function.

10) Socket I/O functions and ancillary data: The sendmsg() and recvmsg() functions (see appendix D for definition) are the most general of all socket I/O functions **(author?)** [7, p. 390] and can be used in combination with all protocols in the transport layer. Both functions have room for message data

Figure 6: Ancillary data embedded in sendmsg() data structure

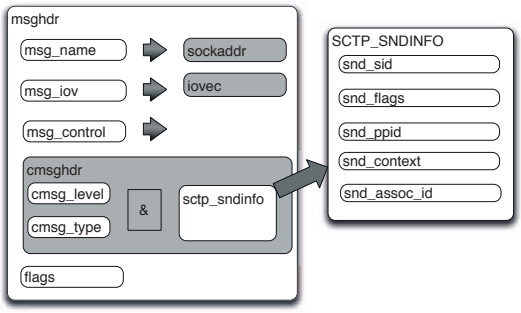


Table I: Ancillary data mappings

sid	stream identifier
ssn	stream sequence number
ppid	identifier set by peer
aid	association id

and ancillary data (see appendix E). SCTP adds extra meta data which contains information about the message being sent and its connection. Table I gives an overview of the ancillary data mapped to the return variables.

B. Go language overview

In this section I will give a short overview of the main features and characteristics of the Go language and provide more detail where relevant to the project.

1) Data: Go has a syntax broadly similar to the C and Java languages. Variables are declared either by variable name followed by data type, or by name only with data type inferred (known as 'type inference').' With the initialization operator := variables can be declared and initialized in a single statement. If not initialized explicitly, the variable is set to a default value. Here are some examples:

```
var number int
var first_name, last_name string
var greeting = hello("olivier")
fractal := make([]uint64, 10)
```

Go has two ways to create data structures: i) with 'new', where the data structure is initialized to zero; ii) with 'make', where the data structure is initialized to a specified value.

```
var p *[]int = new([]int)
var v  []int = make([]int, 100)
```

An asterisk indicates that the variable is a pointer.

2) Functions: Unlike most commonly-used languages Go functions and methods can return multiple values. The following function expects a string and integer as input parameters and returns a string and an error. If the count integer is zero the function will return a nil and an error, otherwise it returns the greeting and nil as error.

```
func hello(name string, count int) (greeting
     string, err error) {
    if count = 0 {
        return nil, errors.New("Cannot␣say␣
             hello␣zero␣times")
    }
    greeting = "Hello" + name, nil
    return
}
```

This hello function can be be called in the following manner:

```
greeting, err := hello("paard", 0)
if err != nil {
    println("error!")
} else {
    println(greeting)
}
```

Or if we are not interested in one of the return values, a blank identifier _ can be used to discard the value:

```
greeting, _ := hello("paard", 1)
```

3) Object and methods: In Go data structures (objects) are defined as follows:

```
type Person struct {
    name string
    age int
}
```

Functionality can be added to an object by defining methods associated with it. A difference between Go and other object-oriented languages like Java is that in Go this functionality is defined outside of the object, for example:

```
func (p Person) SayHello(name String) {
    return "Hello␣" + name ",␣my␣name␣is␣" + p.name
}
```

In the above example the SayHello method has a string as input and is associated with a Person object p.

Methods can be associated with any type, not just objects. In the following example the Add() method is associated with an Array of strings:

```
type NameArray []string
func (na NameArray) Add(name string) []string {
    ...
}
```

It's worth noting that Go has objects but not inheritance.

4) Interfaces : As with other object oriented languages, you can specify behavior of an object by using interfaces and also implement multiple interfaces per object. The interfaces does not have to be explicitly named: as soon as the type implements the methods of the interface the compiler knows the relation between the two. In the following example, in function main, a cat object gets created, it talks and then gets cast (type changed) to an animal object:

```
type animal interface {
    Talk()
}

type Cat

func (c Cat) Talk() {
    fmt.Println("Meow")
}

func main() {
    var c Cat
    c.Talk()
    a := animal(c)    // Cast from cat to animal
    a.Talk()
}
```

5) Pointers: Although Go has pointers (references to memory locations), they are implemented differently than in a language like C. Pointer arithmetic is not possible so Go pointers can only be used to pass values by reference. At a lower level Go pointers can be converted to C pointers.

6) Concurrency: Go has a basic primitive for concurrency called a goroutine. The name is not only a play on a programming component called coroutine but is also its implementation of it in the language. Coroutines are methods which call on each other, data comes in, gets processed and gets passed to a the next coroutine. One of the advantages of coroutines is they are generally easier to create and understand [Knuth V1 p193]. But the main advantage is that it lends itself very well for distributed computing where data gets passed around from one processor to another, maybe even on a different computer.

In Go every function can become a goroutine by simple putting go in front of it. A gorouting can than communicate its input and output via channels. This concept is called Communicating sequential processes (CSP) and is surprisingly versatile in its use. **(author?)** [3] Here an example:

```
1  package main
2
3  func receiveChan(ci chan int) {
4    for {
5      i := <-ci
6      println(i)
7    }
8  }
9
10 func main() {
11   ci := make(chan int)
12   go receiveChan(ci)
13
14   for i := 0; i < 10; i++ {
15     ci <- i
16   }
17 }
```

The receiveChan() function has as input a channel of integers. On line 4 an endless for loop starts where line 5 waits for an integer to come in on the channel. The main functions first creates a channel of integers. Line 12 starts the function receiveChan as a Go routine in the background. This is followed by a loop sending 10 integers over the channel to

the receiveChan function.

7) Much more: There is more to the language like garbage collection, first class functions, arrays, slices, however the explanation of this falls outside the scope of this paper. More information can be found on the Go website [1].

III. Go networking

The following section contains the findings of my research on how Go combines the system level TCP socket API into an easy-to-use network library and how that works internally. Much of the TCP functionality is very similar to the SCTP stack and this will serve as an example of how to implement SCTP in Go.

A. The network library

Go provides a portable interface for network I/O. Basic interaction is provided by the Dial, Listen, ListenPacket and Accept functions which return implementations of the Conn, PacketConn and Listener interfaces. The Listen function is for byte stream data connections and the ListenPacket is for data transfer which includes messages boundaries, like UDP or Unix domain sockets. These functions simplify the access to the socket API, but if needed the application developer can still access the socket API in more detail.

Here is an example of a typical TCP client and server application. The client sends a single message and the server waits to receive a message, prints it out after receiving and starts listening again. First a simple client:

```
1  package main
2  import "net"
3
4  func main() {
5      conn, err := net.Dial("tcp", "localhost:1234")
6      if err != nil {
7          return
8      }
9      defer conn.Close()
10     conn.Write([]byte("Hello world!"))
11 }
```

The function main() is the entry point of the program. The first step the client performs is 'dialing' a server with the TCP protocol (line 5). The Dial() function returns a connection which is an implementation of the Conn interface. After checking for errors (6-8) the defer keyword before the connection close command indicates the connection can be finished as soon as it is no longer needed. In this case it happens immediately after the write so it does not make much sense to defer it, but in larger programs with multiple exit points you only need a single (deferred) close statement, which makes it easier to understand and maintain the program.

Next the server:

[1] http://golang.org/doc/effective_go.html

```
1  package main
2  import "net"
3
4  func main() {
5      listen, err := net.Listen("tcp", "localhost:1234"
           )
6      if err != nil {
7          return
8      }
9      buffer := make([]byte, 1024)
10     for {
11         conn, err := listen.Accept()
12         if err != nil {
13             continue
14         }
15         conn.Read(buffer)
16         println(string(buffer))
17     }
18 }
```

The server gets created with the Listen() method and returns an object which implements the Listener interface. On line 9 a byte array buffer gets created for the received data. The following for loop (10 - 17) will continuously wait to accept an incoming connection (11), check for errors after connect, read from it (15) and convert the byte array to a string before printing it (16).

B. Under the hood

Go's network communication library uses the same socket API as any C program. In this section I will examine what happens when a network connection is set up and which socket API calls are made at what point. The example uses TCP. To illustrate how Go accesses the socket API I will take the TCP client described in the previous section as an example and show how it interacts with the kernel by opening a connection.

Socket and Connect: For a TCP client to create a connection and send some data the following system calls need to be made in this order:

1) resolve IP address
2) socket()
3) setsockopt() (optional)
4) connect()

In Go all this is wrapped in the net.Dial() call.

Figure 8 shows a sequence diagram of method calls after a client calls Dial(). First the hostname:port gets parsed and resolved to an IP address (1.1)[2]. Net.dialAddr() (1.3) determines the transport protocol type and calls the relevant method, net.DialTCP() in this case (1.3.1). Next net.internetSocket() gets called which internally calls socket() in the syscall package. Syscall.socket() is an auto-generated method based on C header files which describe the socket API.

Every network protocol in Go has its own connection type. As you can see in figure 8 the generic Dial() method eventually reaches the specific DialTCP() method which returns a TCP-specific connection type. This type gets cast to the more generic Conn type which can be used in the client application. If TCP-specific functionality is needed the Conn type can be

[2] There are more method calls behind this but they are not relevant for this example

recast to a TCPConn which then makes it possible to access TCP-specific functionality.

C. Auto-generation

Native Go functions to access kernel system calls and data structures can be auto-generated via scripts. In FreeBSD, if the description of the system call is present in the syscalls.master file [3], the function will be available through the syscall package. Most system calls which are used in Go are wrapped in methods to make them fit better the language. The structures which are passed to the system calls are created in the same way. The source code directory of the syscall package contains a file with a link pointing to a header file which describes the structures.

D. Go non-blocking networking

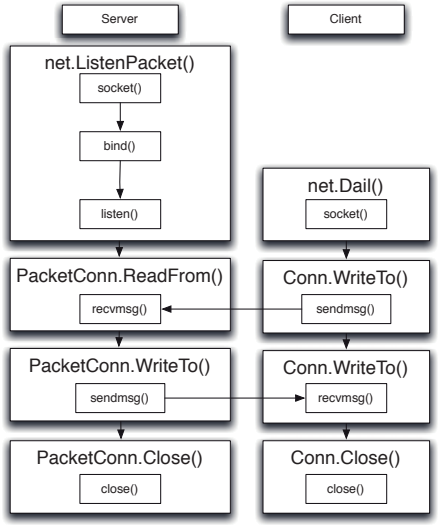
All networking in Go is non blocking which is not the default in the TCP/IP model. As soon as a application tries to retrieve data from a socket, e.g. via readmsg(), it will not block until some data comes in, instead it will immediately move on in the program. Reading data from the socket is normally done in an endless loop. To make Go actually wait for sending data back to the application Go implements the reactor design pattern. The implementation of this software pattern makes use of a mechanism in the kernel where you can ask the kernel to send a signal back to the application if certain events occur whilst you keep doing other things in the program. For example data being written to the socket descriptor. On BSD the system call kqeueu() and kevent() are used to register and queue events. The functionality of this is handled by Go's runtime.

IV. Design

In the following section I will describe how the different Go network methods will map to the SCTP socket API.

[3]System call name/number master file: http://fxr.watson.org/fxr/source/kern/syscalls.master

Figure 7: Go SCTP network API



A. Mapping APIs

The SCTP socket API will follow closely the wrapping of the TCP socket API in the Go network library as described in section III-B. Figure 7 show the mapping of the socket API to the different functions and methods. At the server side the socket(), bind() and listen() functions are bundled into the ListenPacket() function which resides in the network package (net). The ListenPacket() function returns an implementation of the PacketConn interface (See appendix B). The client wraps the socket() function into the Dail() function. This functions returns a Conn (connection) interface which can used for writing messages (Conn.WriteTo()) and these user messages can be received vie the ReadFrom() method on the PacketConn interface.

B. SCTP specific

1) Receive SCTP specific information : To access SCTP specific functionality, such as which stream the message has been sent on, or the association id, net.ListenSCTP() can be used. This method returns a SCTP specific type (SCTPConn) which has the method ReadFromSCTP() and WriteToSCTP() added to it. These methods return and set the information contained by the SCTP receive information structure, added as ancillary data when the system call recvmsg() returns.

2) Send SCTP specific information: To be able to set SCTP specific send information such as stream id or association id via the SCTP Send Information Structure, the several methods on the SCTPConn object can be used (See table II):

Table II: Initialization parameters

Number of output streams	(*SCTPConn) InitNumStreams(n int) error
Max number of input streams	(*SCTPConn) InitMaxInStream(n int) error
Max number of attempts to connect	(*SCTPConn) InitMaxAttempts(n int) error
Timeout	(*SCTPConn) InitMaxInitTimeout(n int) error

A typical server which has access to SCTP specific functionality would look like this:

```
package main
import (
    "net"
    "strconv"
)

func main() {
    addr, _ := net.ResolveSCTPAddr("sctp", "localhost:4242")
    conn, _ := net.ListenSCTP("sctp", addr)
    defer conn.Close()
    for {
        message := make([]byte, 1024)
        _, _, stream, _ := conn.ReadFromSCTP(message)
        println("stream " + strconv.Itoa(int(stream)) + ": "
            + string(message))
    }
}
```

In this program ListenSCTP returns a SCTP connection type. This type implements Conn and PacketConn interface and has the ReadFromSCTP method added to it. The println() functions prints the stream id and the user message.

V. Implementation

In this section I will describe how the SCTP network functionality can fit into the existing Go network framework. The main goal of the SCTP application programming interface (API) design is to combine lower-level system calls in an easy-to-use framework. This framework hides the underlying complexity of the socket API but if needed gives access to all the functionality provided by the protocol. To make sure SCTP fits in the Go design philosophy, less is more, I will make use as much as possible of the existing components and interfaces in the Go network package. In the following section I'll e

A. Server

For a server to be able to set up a SCTP association it needs to create a socket, bind to it, optionally set some socket options and start listening to it. A typical server will access the socket API in the following sequence:

1) socket()
2) bind()
3) listen()
4) recvmsg()

The socket(), bind() and listen() functions will be wrapped into a Listen() method which returns a connection type. There are three variations: net.Listen(), net.ListenPacket() and net.ListenSCTP(). The Go network library provides the net.ListenPacket() method for packet-oriented networking like UDP, and net.Listen() for stream-oriented networking like TCP. SCTP which is packet-oriented, can also make use of the net.ListenPacket() method. ListenPacket() returns an implementation of the PacketConn interface which can be used to read and write messages (recvmsg()). A simple SCTP echo server using the PacketConn interface might look like this:

```
package main
import "net"

func main() {
    conn, _ := net.ListenPacket("sctp", "
        localhost:4242")
    defer conn.Close()
    message := make([]byte, 1024)
    conn.ReadFrom(message)
    print(string(message))
}
```

After the the main entry point a connection object is created via the ListenPacket() method. The parameters of this method indicate that the connection should use the SCTP protocol and listen on localhost port 4242. The next line defers the closing of the connection when it is not needed anymore. After creating a byte array buffer to store incoming data a message is read from the connection. The ReadFrom() method will block until a message is received. Finally the message is printed and the program ends.

Receive SCTP-specific information: To access SCTP-specific functionality, such as which stream the message has been sent on, or the association id, net.ListenSCTP() can be used. This method returns a SCTP-specific type (SCTPConn) which has the method ReadFromSCTP() added to it:

```
(*SCTPConn).ReadFromSCTP(message *string)(sid
    int, ssn int, ppid int, aid int, addr
    SCTPAddr, err error)
```

The ReadFromSCTP() method returns the information contained by the SCTP receive information structure, added as ancillary data when the system call recvmsg() returns. A typical server which has access to SCTP-specific functionality would look like this:

```
package main
import (
    "net"
    "strconv"
)

func main() {
    addr, _ := net.ResolveSCTPAddr("sctp", "localhost
        :4242")
    conn, _ := net.ListenSCTP("sctp", addr)
    defer conn.Close()
    for {
        message := make([]byte, 1024)
        _, _, stream, _ := conn.ReadFromSCTP(message)
        println("stream " + strconv.Itoa(int(stream))
            + ": " + string(message))
    }
}
```

In this program ListenSCTP() returns a SCTP connection type. This type implements the Conn and PacketConn interfaces and adds the ReadFromSCTP() method.

B. Client

In Go a client connection sets itself up with a call to the Dial() method. The Dial() method returns the generic Conn

interface. Every network protocol in Go has its own Dial() method which returns a protocol-specific connection type. In the case of SCTP this is the PacketConn type which has underneath it a specific SCTP connection type (SCTPConn). A simple SCTP client sending a single message would look like this:

```
package main
import "net"

func main() {
    addr, _ := net.ResolveSCTPAddr("sctp", "localhost
        :4242")
    conn, _ := net.DialSCTP("sctp", nil, addr)
    defer conn.Close()
    message := []byte("paard")
    conn.WriteTo(message, addr)
}
```

The DialSCTP() method creates the socket and sets default socket options. Sending the message via WriteTo() will implicitly set up the connection.

Send SCTP-specific information: To be able to set SCTP-specific send information such as stream id or association id via the SCTP Send Information Structure, the WriteToSCTP() method can be used:

```
(*SCTPConn).WriteToSCTP(message *string, addr
    SCTPAddr, sid int, ssn int, ppid int, aid int,
    err error)
```

Creating and binding of the socket : The sequence diagram in figure 9 gives an overview of how a socket is created and bind to. At point 1.1.3 in this figure net.internetSocket() returns a socket descriptor which is used to create the actual SCTP connection type. At this point the SCTP initialization structure is set to its default values together with the NODELAY socket option. The 'no delay' socket option disables buffering of bytes sent over the connection. This is also the default setting for the TCP implementation in Go.

VI. Analysis

A. Performance

For performance testing a client-server application was designed, with the client sending messages and the server receiving them. Both a C and a Go version of this application were created and compared and run against each other. The data throughput of an SCTP client-server application written in Go is approximately twice as high as the same program written in C. Most of the delay happens in the Go client. Unlike the C implementation, the Go application needs to go through some additional method calls before it reaches the system call which sends the data. Go is also garbage-collected, which causes an extra delay because several data structures are redeclared each time a message is sent. Another factor to consider is that the implementation of SCTP in Go is against version 1.0.2 of Go. This version does not have a compiler which is good at optimizing. Newer versions of Go address this issue.

B. Fitting like a glove

Since Go already provides methods and interfaces for message-based data transmission that could be reused, and because of the SCTP socket API's similarity to the TCP socket API, making SCTP available in the Go network library was a relatively straightforward task. I was able to reuse the ancillary data structure from the Unix socket API and had only to add the SCTP-specific ancillary data to the structure. It was easy to follow the design philosophy 'less is exponentially more': the SCTP socket API could be wrapped in an easier-to-use API, just as it is done with TCP. This resulted in a Go SCTP API which can be used in the most simple way, hiding all complexity of the protocol or, if needed, it is possible to dive deeper and make use of more specific SCTP functionality.

C. Issues during implementation

During implementation there were two major changes that caused unexpected setbacks. The first, as previously mentioned, was the changing SCTP socket API, which made large parts of the implementation already in place obsolete, forcing me to rewrite the majority of the implementation. The second issue was the first official release (1.0) of Go. Until that release I had sporadically synchronized my version of Go with the latest changes of the main development tree of the canonical Go source code repository. Building up to the official release the Go development team did a considerable amount of work. With the 1.0 release a large number of changes had to be incorporated into my own branch. As there were many changes to the internals of Go, this resulted in many merge conflicts in certain areas, specifically around the implementation of the generic Dial and Listen interfaces. Most of the work in this area had to be redone.

D. Extensions

There are many extensions to SCTP described in multiple RFCs. A complete SCTP implementation should include **(author?)** [9]:

1) RFC4960 (basic SCTP)
2) RFC3758 (partial reliability)
3) RFC4895 (authentication)
4) RFC5061 (dynamic addresses)

The last three in this list are not included in this implementation.

VII. Conclusion

Because of its similarity to existing protocols available in the Go networking library, SCTP fits easily into it. The biggest challenges of this project were the ongoing work on the SCTP specification and Go itself which made successful implementation a moving target. More recently (mid-way 2012) the APIs of Go and SCTP have been stabilized. It should be noted however that there are many extensions to SCTP described in multiple RFCs. The work in this paper only looks at the bare minimum needed to make SCTP work in Go.

A. Future work

User land implementation of SCTP in Go: Not all operating systems support the SCTP protocol natively. It is however possible to have SCTP running on top of the UDP protocol, outside the kernel (user land). To make this work a user land implementation of SCTP on top of UDP needs to be added to Go. Once this is done SCTP could become available on all different platforms supported by the Go language.

REFERENCES

[1] Why we need go - OReilly radar. http://radar.oreilly.com/2012/09/golang.html.

[2] R. Braden. Requirements for internet hosts - communication layers. http://tools.ietf.org/html/rfc1122.

[3] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[4] Janardhan R. Iyengar, Paul D. Amer, and Randall Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.*, 14(5):951–964, October 2006.

[5] B. Penoff, A. Wagner, M. Tuxen, and I. Rungeler. Portable and performant userspace SCTP stack. In *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1 –9, August 2012.

[6] Rob Pike. command center: Less is exponentially more. http://commandcenter.blogspot.co.uk/2012/06/less-is-exponentially-more.html, June 2012.

[7] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *Unix Network Programming: Sockets Networking API v. 1*. Addison Wesley, 3 edition, November 2003.

[8] R. Stewart. RFC 4960 - stream control transmission protocol, motivation. http://tools.ietf.org/html/rfc4960#section-1.1.

[9] Randall Stewart, Michael Tuexen, and Peter Lei. SCTP: what is it, and how to use it? *BSDCan 2008*, 2008.

[10] Randall R. Stewart and Qiaobing Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison Wesley, 1 edition, October 2001.

[11] Michael Tuexen, Salvatore Loreto, and Randell Jesup. RTCWeb datagram connection. http://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-00.

APPENDIX

A. Conn interface

```
type Conn interface {
    Read(b []byte) (n int, err error)

    Write(b []byte) (n int, err error)

    Close() error

    LocalAddr() Addr

    RemoteAddr() Addr

    SetDeadline(t time.Time) error

    SetReadDeadline(t time.Time) error

    SetWriteDeadline(t time.Time) error
}
```

B. PacketConn interface

```
type PacketConn interface {

    ReadFrom(b []byte) (n int, addr Addr, err error)

    WriteTo(b []byte, addr Addr) (n int, err error)

    Close() error

    LocalAddr() Addr

    SetDeadline(t time.Time) error

    SetReadDeadline(t time.Time) error

    SetWriteDeadline(t time.Time) error
}
```

C. Listener interface

```
type Listener interface {
    Accept() (c Conn, err error)

    Close() error

    Addr() Addr
}
```

D. Socket IO function definition

```
ssize_t sendmsg(int s, const struct msghdr *msg, int flags)

ssize_t recvmsg(int s, struct msghdr *msg, int flags)
```

E. Message header structure and ancillary data

```
struct msghdr {
  void         *msg_name;       /* optional address */
  socklen_t     msg_namelen;    /* size of address */
  struct iovec *msg_iov;        /* scatter/gather array */
  int           msg_iovlen;     /* # elements in msg_iov */
  void         *msg_control;    /* ancillary data */
  socklen_t     msg_controllen; /* ancillary data buffer len */
  int           msg_flags;      /* flags on message */
};
```

The msg_control argument, which has length msg_controllen, points to a buffer for other

Figure 8: TCP Client setting up connection

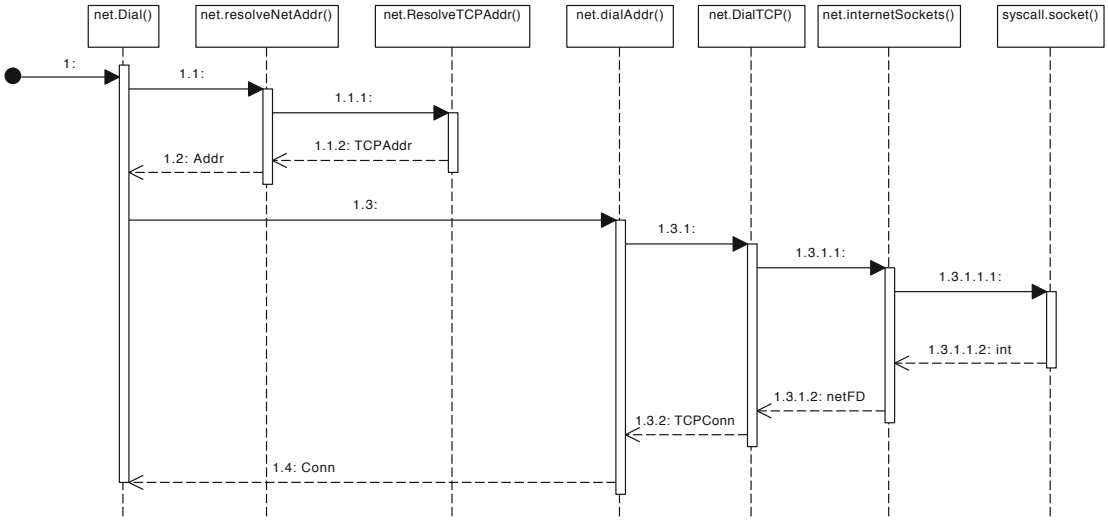
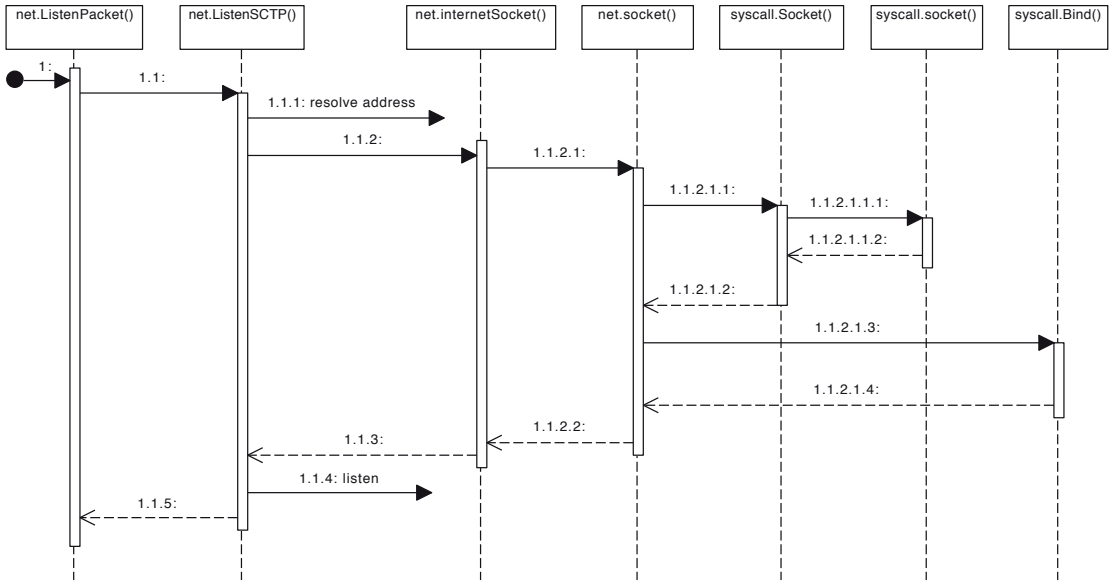


Figure 9: Creating and bind



protocol control related messages or other miscellaneous ancillary data. The messages are of the form:

```
struct cmsghdr {
  socklen_t  cmsg_len;    /* data byte count, including hdr */
  int        cmsg_level;  /* originating protocol */
  int        cmsg_type;   /* protocol-specific type */
 /* followed by u_char cmsg_data[]; */
};
```

The surprising complexity of TCP/IP checksums in the network stack

Henning Brauer
BS Web Services

Abstract

TCP and IP have well known and well understood checksum mechanisms. The actual checksum math is easy and, from a performance standpoint, so cheap that it can be considered free. In the process of improving the use of hardware checksum offloading engines, recalculating the IP checksum has been found to be essentialy free. However, that is not the case for the TCP and UDP checksums, which was partially expected. On further inspection a surprising complexity in dealing with the protocol checksums has been found.

We'll look at how these checksums are calculated, where the complexity comes from, how an ancient BSD performance hack made it into hardware offloading engines, the stack interaction and issues with hardware offloading engines.

1 Introduction

For a long time I had been annoyed by the checksum handling in pf. pf used to fix up checksums on the fly, as in, whenever it modified a packet - which it would do for all forms of NAT, for example - it adjusted the original checksum for the applied word delta. This is not only annoying and error prone in many places, it also lead to deeply nested calls to pf_cksum_fixup. When Theo de Raadt somewhen in 2009 or 2010 pointed me to one of these super ugly and not exactly efficient nested pf_cksum_fixup calls, i knew it was time to look deeper into the issue.

On top of this, there is a long standing bug with said checksum fixup and packets pf redirected to localhost and checksum offloading, a bug that Christian Weisgerber (naddy@) has explained to us pf people repeatedly over the years, but it was neither easy to follow nor to fix.

Eventually, at the k2k10 hackathon in Iceland, I started looking into our checksum handling in general, and the findings were quite interesting.

2 Checksum Calculation

The actual calculation of the checksum is quite simple. The checksum is the lowest word of the one-complement sum of all the words the checksum covers, basically.

3 General Performance Considerations

After years of profiling our network stack it is clear that the actual math here is so cheap that it can be considered free on every halfway modern system. The actual integer units are never our bottleneck, the limiting factors are latency

and bandwidth to caches, memory and devices.

Thus, a checksum covering a few fields that have been touched very recently and thus are in cache is almost free. A checksum covering a whole bunch of data that hasn't been accessed yet at all is expensive, since the data has to be read from RAM, which is relatively slow.

The actual checksum algorithm has optimized assembler implementations on many platforms we support. Wether these hand-optimizied versions are actually faster than the generic C version is an interesting question that has not been evaluated here.

4 The IP Checksum

The checksum in the IP header (referred to as the IP checksum) covers the IP header. It has to be updated by each router forwarding the packet as it updates the ttl field. IPv6 does not have this checksum.

Since this checksum only covers the relatively small IP header and several fields of that header have just been accessed before, recalculating it is pretty much free. The performance advantage of offloading it to a capable NIC is so small that it gets lost in the noise when trying to measure it.

5 IP Checksum Implementation in OpenBSD

In our network stack - and similiar in the other BSD-derived stacks - an incoming IP packet is handled in ip_input(). Besides a lot of validity checks, ip_input() decides wether that packet is to be delivered locally, in which case it is handed off to upper layers, to be forwarded or to be dropped as undeliverable, e. g. when forwarding is off. The inbound pf_test() call, which makes pf examine the packet, is also here.

In the forwarding case, the packet is handed off to ip_forward, which deals with routing and ttl decrementation. The actual route lookup and some other related tasks are already done in ip_input(), as an implementation side-effect. If the packet is to be forwarded it gets handed off to ip_output().

ip_output() checks and, for locally generated packets, fills in a lot of the IP header fields. The outbound pf_test() call is here as well. Right after the pf_test call the ip checksum is recalculated unconditionally, last not least to cover the ttl decrement for forwarded packets, possible changes done by pf. Locally generated packets do not even have a checksum at this point and get it filled in.

At this point it seems obvious that the ip checksum fixup done all over the place in pf is useless work, since the ip checksum is recalculated just after pf in ip_output anyway, and inbound the check happens before pf. However, pf is not only called from ip_input() and ip_ouput(). There also is the bridge case - the bridge calls pf_test() too, and the bridge does of course not decrement the ttl, nor does it make other changes to the IP header, thus it does not recalculate the ip checksum after pf. This is also the reason why the bridge is special-cased all over the stack.

The solution to this problem is to make the bridge behave like a regular output path. To complicate matters, checksum offloading enters the picture.

6 IP checksum offloading

Pretty much every network interface chip/card made in the last decade is capable of performing the ip checksum calculation in hardware. To make use of that, our stack has been modified a long time ago to delay the actual checksum calculation up until we definately know on which interface the packet is going to be sent out. We can then check wether the interface in question has matching offload capabilities, indicated via interface flags. If so we don't need

to do much more but to mark the packet for hardware checksumming. If not, we calculate the ip checksum in software.

To know wether a packet needs checksumming at all we use a flag in the mbuf packet header, a structure attached to the packet data for bookkeeping in throughout the stack. Everywhere we know the packet needs checksumming we plain set this flag.

This works fine for all the regular output pathes. It doesn't for the bridge, due to its lack of checksum handling alltogether.

The bridge code is quite old and not exactly an example for good programming. It is hard to follow. Adding the missing checksum handling in its output pathes - there is unfortunately even more than one - turned out to be not so easy. Once this was done and the bridge special casing all over the stack removed, things mostly worked. Some weird behaviour was eventually tracked down to problems with broadcase packets, and upon closer inspection the bridge uses a gross hack to shortcut broadcast processing, so that a packet that supposedly goes out to a checksum offloading capable interface can get copied and sent out on another interface, potentially without matching offloading capabilities. This resulted in packets being sent out uncheck-summed in that case.

Fixing the broadcast hacked was not straightfoward, this needs to be adressed at a later time. The special casing in the stack had to stay. However, with that basic checksum handling and the special casing in place, we were able to stop doing any ip checksum handling in pf, since now all output pathes recalculate the checksum if necessary.

Since recalculating the ip checksum is so cheap even in software on any halfway modern system performance improvements from this change were to small to be really measurable.

7 The TCP and UDP checksums

The tcp and udp, often referred to as protocol checksums, are quite a different beast from the ip checksum. They only cover a few ip header fields, that part is called pseudo header checksum, the tcp/udp header and the entire payload. Due to the full payload coverage recalculating the protocol checksum is not as cheap. While chances are good that the payload is still in cache for locally generated packets, the forwarding case almost certainly means fetching the payload from RAM, since we don't touch it for pure forwarding otherwise.

As with the ip checksum, pf used to update the protocol checksum on the fly, with the same problems as with the ip checksum, just in more places.

8 protocol checksums in the OpenBSD network stack

The procotol checksum handling is much more complex than the ip checksum. As all BSD-derived network stacks OpenBSD used protocol control blocks, in short pcbs, to track connections. Even for udp, which is a connectionless protocol - connectionless on the wire doesn't mean that the stacks don't have some kind of state. The pcbs are looked up using hash tables, or, in OpenBSD, by following the link to them from the pf state.

When a socket is opened, a template pcb for connections from or to this socket is created. The known parts are already filled in and checksummed. Once a connection is made using that socket, the template pcb is copied, the other side's information is added, and the checksum updated for that. This only covers the ip header parts, not the protocol header, and forms the pseudo header checksum. The packet is marked for needing checksumming at this point and then passes on to get protocol header and payload.

Eventually, late in the outbound path, the flag indicating a checksumming need is evaluated. If the interface that this packet should go out on has matching offloading capabilities, we don't need to do anything, otherwise we do the checksumming in software. That's the theory, at least.

The early calculated pseudo header checksum is a hack that might have made sense on a hp300 or a vax, but is counterproductive on any halfway modern system and foremost complicated things considerably. This is where the pf redirect to localhost problem comes from. Some network interface cards - last not least intel and broadcom - implemented their offloading engines so that they rely on this hack, by relying on the pseudo header checksum being there.

However, when such a packet passes through pf, we don't know that it just has a partial checksum, and happily update it for fields it doesn't cover. On a packet that originated from localhost and gets rewritten by pf - prime example being replies to packets that have been redirected to localhost - we have exactly that and end up with a broken pseudo header checksum. For cards that rely on and just update it we end up with broken checksums on the wire. Both the software engine as many other interface hardware recalculate the entire checksum and don't care about the existing pseudo-header checksum.

As with the ip checksum the bridge code had to be updated for this output path to behave.

9 protocol checksum offloading

As with the ip checksum, almost all halfway recent network interface chips and cards support tcp and udp checksum offloading, at least for IPv4. Things are considerably more complicated here tho, since we have to deal with 3 offloading cases: no offloading, pseudo header checksum required, and full offloading. Since the pseudo header case was broken due to the pf handling, protocol checksum offloading is disabled in the drivers in this case.

Unfortunately we have seen many silicone bugs with offloading. While that mostly affects early implementations and is long history, we have recently seen a case with an Intel 10GE chip that corrupted ospf packets - ospf is neither udp nor tcp! - when the tcp/udp offloading engines were turned on.

10 changing the stack to make better use of offloading engines

The basic principle is easy: work under the assumption that we have always have offloading engines. If we hit a path that doesn't, provide a software fallback.

To accomodate for this, the actual checksumming has been moved far down in the stack and is done pretty much as late as possible now, when we know for sure wether we have an outgoing path with offloading capabilities. That allows for removal of pretty much all checksum handling everywhere else in the stack, we just have to set the flag now to ask for checksumming.

Subsequently, all ip and protocol checksum adjustments have been removed from pf, with just the flag modifications remaining. This has interesting consequences. Since we are not updating but recalculating the checksum for forwarded packets that are modified by pf now, that case suffers if there is no checksum offloading available, since we have to access the entire payload. With pretty much any system made in the last 10 years supporting offloading, only the case where pf modifies forwarded packets (that means NAT mostly) being affected, and preventing this - by calculating the pseudo header checksum before and after pf making changes and applying the delta to the existing checksum - hurts machines with offloading capabilities, this seems acceptable.

Since we are not updating the checksum any more but recalculating, we have to consider the case of broken checksums. Adjusting a broken checksum leads to a broken checksum, so all's good - but if we're recalculating, we have ti verify the old checksum first. Again, in many cases we already have that check performed by the hardware offloading engines already, if not, we have to fall back to software. This again punishes hardware without offloading capabilities, but is not fixable without punishing the offloading case substantially. On a halfway modern i386 or amd64 system with disabled offloading capabilities the total hit is in the 5 to 10traffic mix - exact numbers vary a lot with specific hardware implementations. And these systems have offloading capable interfaces.

Last not least this finally allows us to enable the protocol checksum offloading on the interfaces requiring the pseudo header checksum, since pf won't munge the checksum any more.

11 ICMP

ICMP has a checksum too. The icmp checksum handling has been remodeled after the tcp and udp case, instead of calculating the checksum early on and every time changes are being made, we just set the flag. Late and way down in the stack we check the flag and calculate the checksum. Should there ever be hardware that can offload the icmp checksum calculation it is trivial to hook in now.

ICMP shows another interesting case. ICMP errors quote the packet they refer to, usually that is tcp or udp. And this quoted packet of course has a checksum as well. In most cases the quoted packet is truncated and the checksum can't be verified anyway, so there is no real need to adjust it. pf used to do it nontheless. In the cases where the quoted packet is not truncated, we can recalculate the inner checksum as usual, just without being able to use any offloading. In return the quoted, not truncated packets are tiny, so the cost is minor. In my current implementation this is not done, since nothing in the real world cares about the inner checksum. The only case I am aware of is scapy, a program generate and verify packets.

12 performance considerations

On a modern amd64 system we could not see any performance benefit from these changes, not even from offloading in general. On older systems (Pentium M) I have seen around 7indicates that the modern Xeon has so much raw computing power and large caches that other factors, like latency to the devices, hides the saved work completely when just doing packet forwarding.

However, these changes simplify the checksum handling a lot, shortening the code considerably and allow for even more simplifications by followup cleanups.

13 future work

The gross bridge broadcast hack mentioned earlier needs to be fixed so that the special casing all over the stack can go away.

There is no point in the partial early checksum calculations in the pcb templates and upon establishment of a new connection any more, we can simplify things by just calculating the pseudo header checksum in the output routines where we figure out wether we have offloading or use the software engine.

14 Acknowledgments

A lot of help and support, last not least with testing, came from Christian Weisgerber (naddy@) and Mike Belopuhov (mikeb@).

The IPv6 protocol checksum software engine was written by Kenneth R Westerback (krw@).

My trip to AsiaBSDcon has been funded by the conference.

15 Availability

The ip checksum handling changes are part of OpenBSD since 5.1. The protocol checksum parts should be committed soon.

This paper and the slides from my presentation will be availabe from the papers section on

`http://www.bulabula.org`

and be linked from OpenBSD's paper section on

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