MAGAZINE

BSD

FOR NOVICE AND ADVANCED USERS

Military Grade Data Wiping in FreeBSD With BCWipe

HAMMER File System Volumes Management and Pseudo

File Systems Mirroring in DragonFlyBSD

Advanced Unix Queuing Techniques

OpenBSD From a Veteran Linux User Perspective

VOL 11 NO 09
ISSUE 09/2017 (97)
ISSN 1898-9144

FREENAS MINI
STORAGE APPLIANCE

IT *SAVES* YOUR LIFE.

HOW IMPORTANT IS YOUR DATA?

Years of family photos. Your entire music and movie collection. Office documents you've put hours of work into. Backups for every computer you own. We ask again, *how important is your data?*

NOW IMAGINE LOSING IT ALL

Losing one bit - that's all it takes. One single bit, and your file is gone.

The worst part? **You won't know until you absolutely need that file again.**

Example of one-bit corruption

THE SOLUTION

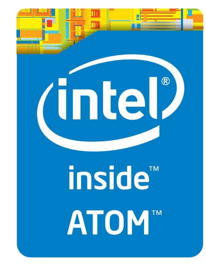
The FreeNAS Mini has emerged as the clear choice to save your digital life. **No other NAS in its class offers ECC (error correcting code) memory and ZFS bitrot protection to ensure data always reaches disk without corruption and *never degrades over time*.**

No other NAS combines the inherent data integrity and security of the ZFS filesystem with fast on-disk encryption. No other NAS provides comparable power and flexibility. The FreeNAS Mini is, hands-down, the best home and small office storage appliance you can buy on the market. **When it comes to saving your important data, there simply is no other solution.**

The Mini boasts these state-of-the-art features:

- 8-core 2.4GHz Intel® Atom™ processor
- Up to 16TB of storage capacity
- 16GB of ECC memory (with the option to upgrade to 32GB)
- 2 x 1 Gigabit network controllers
- Remote management port (IPMI)
- Tool-less design; hot swappable drive trays
- FreeNAS installed and configured

http://www.iXsystems.com/mini



Intel, the Intel logo, Intel Atom and Intel Atom Inside are trademarks of Intel Corporation in the U.S. and/or other countries.

FREENAS CERTIFIED STORAGE

With over six million downloads, FreeNAS is undisputedly *the* most popular storage operating system in the world.

Sure, you could build your own FreeNAS system: research every hardware option, order all the parts, wait for everything to ship and arrive, vent at customer service because it *hasn't*, and finally build it yourself while hoping everything fits - only to install the software and discover that the system you spent *days* agonizing over **isn't even compatible**. Or...

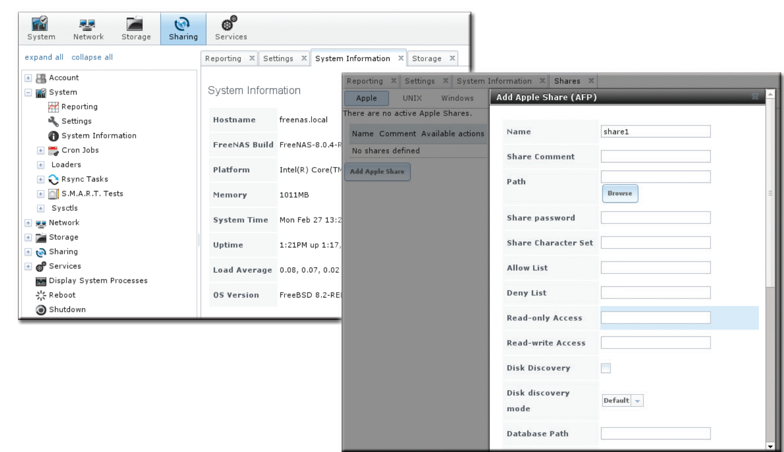
MAKE IT EASY ON YOURSELF

As the sponsors and lead developers of the FreeNAS project, iXsystems has combined over 20 years of hardware experience with our FreeNAS expertise to bring you FreeNAS Certified Storage. **We make it easy to enjoy all the benefits of FreeNAS without the headache of building, setting up, configuring, and supporting it yourself.** As one of the leaders in the storage industry, you know that you're getting the best combination of hardware designed for optimal performance with FreeNAS.

Every FreeNAS server we ship is...

» Custom built and optimized for your use case
» Installed, configured, tested, and guaranteed to work out of the box
» Supported by the Silicon Valley team that designed and built it
» Backed by a 3 years parts and labor limited warranty

As one of the leaders in the storage industry, you know that you're getting the best combination of hardware designed for optimal performance with FreeNAS. **Contact us today for a FREE Risk Elimination Consultation with one of our FreeNAS experts.** Remember, every purchase directly supports the FreeNAS project so we can continue adding features and improvements to the software for years to come. **And really - why would you buy a FreeNAS server from *anyone* else?**



FreeNAS 1U

- Intel® Xeon® Processor E3-1200v2 Family
- Up to 16TB of storage capacity
- 16GB ECC memory (upgradable to 32GB)
- 2 x 10/100/1000 Gigabit Ethernet controllers
- Redundant power supply

FreeNAS 2U

- 2x Intel® Xeon® Processors E5-2600v2 Family
- Up to 48TB of storage capacity
- 32GB ECC memory (upgradable to 128GB)
- 4 x 1GbE Network interface (Onboard) - (Upgradable to 2 x 10 Gigabit Interface)
- Redundant Power Supply

http://www.iXsystems.com/storage/freenas-certified-storage/

Intel, the Intel logo, the Intel Inside logo and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

Editor's Word

Dear Readers,

During such rainy days, I presume we all have a lot of time to read more and more. I hope so, and I wish that. However, I know that you are very busy at work, therefore, the weather is not a good indicator to reach this conclusion, especially now that we are almost ushering the year 2018. The days which are left are undisputedly the busiest days before us. However, I hope that the BSD Magazine will feature on your reading list since we believe you have been waiting for it.

In this issue, we present you an article written by Abdorrahman Homaei on *Military-Grade Data Wiping In FreeBSD With BCWipe*. You will not only learn how to install BCWipe but also to install BCWipe With Multithreaded Mode Enabled. You will know more about BCWipe advanced features and BCWipe in action. Besides, there is an article I would like to recommend which will give you more insight into a file system. The author, George Siju, will tell you about *HAMMER File System Volumes Management and Pseudo File Systems Mirroring in DragonFlyBSD*.

If you would like to learn more about Unix techniques, then Mark Sitkowski's article is worth a read. Reading his article will get you acquitted on how to examine the design of a queuing system, loosely modelled on that used by IBM, in its WebSphere MQ Series.

To add to your reading list, there is this interesting article titled *OpenBSD From a Veteran Linux User Perspective* by Carlos Fenollosa, solely based on his own experiences, which are quite fascinating and resourceful as well. His experience has been an eye-opener to many. That's why we thought it was worthwhile to share them with you, our reader. In his sharing, he considers himself an "old-school" Linux admin, and he's felt out of place with the latest changes to the system administration. It's interesting to see how he appreciates modernity. Carlos also agreed to answer a few interesting questions. Therefore, you have a chance to read a really great interview with him.

Of course, that is not all we have in this month's BSD issue. Please, go to the next page to see a full list of publication. I hope you will enjoy reading each one of them, and anticipating for the coming articles in October.

Lastly, I would like to express my sincere appreciation to the BSD team members for reviewing and proofreading, and iXsystems for their constant support and time to make this edition a success.

If you have any suggestions or advices that you want to share with the BSD readers, please, feel free to send your emails to me. And now, let's read the articles.

Enjoy!

Ewa & The BSD Team
ewa@bsdmag.org

Table of Contents

IN BRIEF

In Brief **07**

Ewa & The BSD Team

This column presents the latest news coverage of breaking news, events, product releases, and trending topics from the BSD sector.

SECURITY

Military Grade Data Wiping In FreeBSD With BCWipe **11**

Abdorrahman Homaei

Data wiping is a process of overwriting data on magnetic hard disk, SSD or USB flash by using zeros and ones on whole disk or specific zone. As a result, no one can recover the sensitive data and disk is still usable. You will not only learn how to install BCWipe, but also to install BCWipe With Multithreaded Mode Enabled. You will also know more about BCWipe advanced features and BCWipe in action.

DRAGONFLYBSD

HAMMER File System Volumes Management and Pseudo File Systems Mirroring in DragonFlyBSD **15**

George Siju

In DragonFlyBSD, the HAMMER filesystem allows us to perform functions akin to Logical Volume Management in Linux like adding and removing volumes from Volume Group or Pools, creating Logical Volumes with separate file systems, taking Logical Volume snapshot, etc. The reason for the choice of DragonFlyBSD and HAMMER file system was described in a previous article. A third article will follow describing PFS and snapshots management.

FREEBSD

Transparent Flow Mapping for NEAT **27**

Felix Weinrank

The NEAT library provides application developers with a unified and platform independent API for network communication, regardless of the underlying network protocol. Felix describes an approach to integrate multiplexing functionality into the NEAT library, giving application developers a simple way to use the benefits of mapping multiple data streams to a single transport connection without additional coding effort.

UNIX

Advanced Unix Queuing Techniques **37**

Mark Sitkowski

Following the discussion of some of the types of queuing mechanisms available on Unix, it may be beneficial to examine the design of a queuing system, loosely modeled on that used by IBM, in its WebSphere MQ Series.

BLOG PRESENTATION

OpenBSD
From a Veteran Linux User Perspective **51**

Carlos Fenollosa

For the first time I installed a BSD box on a machine I controlled. The experience has been eye-opening, especially since I consider myself an "old-school" Linux admin, and I've felt out of place with the latest changes on system administration.

INTERVIEW

Interview with Felix Weinrank **59**

Ewa & The BSD Team

Felix Weinrank is a computer scientist from Germany. He is currently a Ph.D student in the Department of Electrical Engineering and Computer Science at Münster University of Applied Sciences. His research interests include the SCTP transport protocol, low-latency Internet communication, and network emulation.

COLUMN

The gig economy giant, Uber, has had its operating licence suspended by Transport for London. Apart from concerns over the way the company operates, a more sinister reference was made to Greyball software which effectively tricks law enforcement and those Uber does not wish to deal with. Where should the line be drawn between good practice and deception? **61**

Rob Somerville

Among clouds Performance and Reliability is critical

Download syslog-ng Premium Edition product evaluation here

Attend to a free logging tech webinar here

The world's first High-Speed Reliable Logging™ technology

HIGH-SPEED RELIABLE LOGGING

- above 500 000 messages per second
- zero message loss due to the Reliable Log Transfer Protocol™
- trusted log transfer and storage

The High-Speed Reliable Logging™ (HSRL) and Reliable Log Transfer Protocol™ (RLTP) names are registered trademarks of BalaBit IT Security.

IN BRIEF

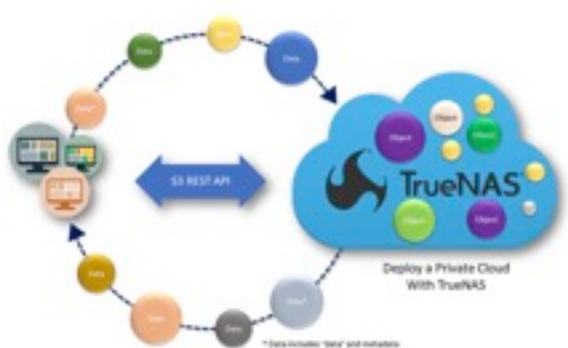
Building MariaDB on FreeBSD

You can just use the MariaDB ports in the FreeBSD ports tree using the following pattern: databases/mariadb(55)?-{client,server}. You can also use precompiled packages when available.

MariaDB can be compiled with spinlocks instead of mutexes. This option enables MariaDB "fast mutexes". These have disabled by default for a a number of years, and it's generally recommended not to use them.

Source:
https://mariadb.com/kb/en/library/building-mariadb-on-freebsd/

The Private Cloud Enabled by TrueNAS : Open for Business



March 14, 2006 marks an important date in the history of the IT abstraction known as the cloud. On this day, Amazon introduced the S3 (Simple Storage Service) to the world and things have never been the same. Fast forward to August, 2017 and Amazon's cloud service, of which S3 is a large part, is a $14.6B business and is growing at a rate of more than 50% per year. Unlike unified storage, which stores and manages data as files or blocks, S3 stores and manages data as objects.

2013 was the last time I found a public mention on how many objects are stored on the Amazon storage cloud, and that number was 2 trillion. While it is unclear what that number is today, one can assume it has since tripled to 6 trillion. To put this number in perspective, there are currently 7.5 billion people in the world. Each person could store 800 of those 6 trillion objects. A truly astounding number that will only continue to increase over time.

The success of the Amazon S3 is due in large part to the many IT and business benefits cloud storage provides. Up until very recently, TrueNAS was a fully unified storage solution providing file and block protocol support for NFS, SMB, AFP, iSCSI, and Fibre Channel. TrueNAS 11, released in early July, added object storage. This means that TrueNAS customers can now build on-premise clouds that are fully Amazon S3-compliant. It also means that services and applications developed for the S3 can be migrated to TrueNAS, bringing these customers the benefits of the public cloud in their own data center.

This is an important development on several fronts. Despite the rapid adoption of public cloud storage, many believe the adoption rate would be closer to universal if it were not for two concerns. One is the

lingering concern over security and the other is the cost at scale.

Let's discuss both in detail. Data stored on a public cloud is on infrastructure that belongs to and is owned by another entity. This would be Amazon for S3, Google for its Cloud Platform, and Microsoft for Azure. This loss of control is a source of concern for many IT professionals. If an enterprise owns the physical infrastructure on which its data lives, safeguards can be taken to prevent unauthorized access to the storage hardware and the data. Public clouds are multiuser systems where the data can be accessed by multiple users and organizations. While processes are designed and put in place to prevent the commingling of data, they sometimes fail. We all have seen or heard in the news where data in the cloud is exposed. In addition to the significant cost that a business can incur from data leakage (especially to a competitor), the business can also be subject to legal risk if the leakage involves certain classes of data.

The second concern deals with cost at scale. Since public cloud storage employs a pay-as-you-go model, it relieves new businesses from the burden of having to shell out a large amount of capital to build on prem storage infrastructure. Fledgling businesses can pay only for the storage they need and use. However, this model quickly breaks down as the business grows and there is more demand for ongoing storage. Case in point: A big cybersecurity organization moved from the cloud by using iXsystems storage and servers to build a private cloud and saved millions by cutting their Amazon S3 bill by over 80%. A recent iXsystems white paper covers the true cost of the public cloud in much greater detail.

Public cloud storage providers charge for their services in many ways. One is by the amount of data stored on their cloud, and another is by the quantity of data retrieved from the cloud. For this exercise, let's just consider the Amazon S3 storage cost. If you were to just store 1TB of data on the S3, the monthly cost is $26.50. However, if you were to store 100TB of data, the monthly cost is now $2,872.18. Over three years, the cost of storing 100TB of data would be $103,398.48. This is based on information from the Amazon AWS Calculator for the US-West (Northern California) region. How does that compare to a physical array from iXsystems? The TrueNAS X10 has a starting price of $5,500.

So what does this all mean? If you have been using the public cloud as a data store and have concerns about security and cost, perhaps it is time to consider the private cloud option, particularly if you already own a TrueNAS appliance. Private cloud with a TrueNAS? Yes. It's here today and open for business.

Steve Wong, Director of Storage Product Management

Source: *https://www.ixsystems.com/blog/private-cloud-truenas/*

OpenSSH 7.6 Is Ready For Testing & Finishes Gutting SSHv1

A call for testing has been issued for the upcoming OpenSSH 7.6 with its release being imminent. This update to OpenSSH deletes the SSH protocol version 1 support. Besides removing the rest of SSHv1 support, OpenSSH 7.6 also nukes some other old code including support for

hmac-ripemd160 MAC, arcfour, blowfish, and CAST ciphers. RSA keys less than 1024 bits are also refused.

Source:
https://www.phoronix.com/scan.php?page=news_item&px=OpenSSH-7.6-Coming-Soon

SNIA SDC 2017: 20 Years and Still Going Strong

The Storage Networking Industry Association's Software Developer Summit (SNIA SDC) 2017 took place just after vBSDcon 2017 from September 11th through 14th in Santa Clara, California. Developers and decision makers from the largest storage vendors in the industry attended this event, and I found it invaluable to my role as iXsystems Senior Analyst to attend as well as to speak.

While flash-based storage in all its forms is a perennial hot topic at the SDC, it has an inevitable twist: we are steadily making our way back to byte-addressed persistent storage memory, not unlike the core memory of the earliest computers but in orders of magnitude bigger, faster, and cheaper. The first wave of this movement is the Non-Volatile Memory (NVM) programming model which is flash-native, doing away with many layers of abstraction that allow "spinning rust" to appear as block devices to a system.

Open-Source had its strongest presence at the SDC in the organization's 20-year history. I did my part to support this by opening the first day with two talks. The first demonstrated Open-Source as an ideal strategy for creating reference implementations of open standards. The second, "Mitigating Ransomware Attacks at the Block Level with OpenZFS", described why FreeBSD and FreeNAS are great Open-Source solutions for combating the real-world threat of ransomware while also serving as excellent reference implementations of open standards such as network protocols, plus techniques straight out of the SNIA Dictionary such as RAID and Replication.

SNIA has a natural preference for permissively-licensed reference implementations of the standards they develop, but not a consistent track record of delivering and maintaining them. This is changing with their Swordfish storage management stack that members have prototyped in Python and AngularJS under an MIT license. If Open-Source is music to my ears, this is the guitar solo and I am excited about the organization's sharpening focus on Open-Source.

The keynotes on the second day continued the Open-Source theme with Sage Weil from the Ceph project and Martin Petersen from Oracle with "Recent Developments in The Linux I/O Stack". OpenZFS specifically came up in an Intel talk when the researcher reported that the relatively larger 128K default block size of ZFS is optimal for use with Intel in-CPU encryption accelerators. Allan Jude of ZFS Book fame looks forward to seeing how even larger block sizes will perform with Intel crypto.

After flash storage and Open-Source, one hallway track theme stood out: the impending IoT and

ransomware bloodbath that will take place on consumer information devices. Today, from the recent massive Equifax leak to the barrage of ransomware attacks at all levels, there are clearly some valid concerns being raised that warrant changes in behavior by both users and vendors. One equal source of both hope and dread is the European Union General Data Protection Directive which can be thought of as the "strong crypto" of personal information privacy. Under the directive, E.U. countries will be required to establish a system that allows companies and organizations to report "potential" data breaches, and to provide their constituents the ability to erase themselves from any system containing Personally Identifiable Information (PII).

This "right to be forgotten" is so attractive that citizens are already beginning to exercise it and unfortunately, the directive offers little guidance in practical implementation and thus will be navigated in the courts. What is clear is that organizations will need to appoint a Data Protection Officer to assess the company's compliance with the GDPR and respond to GDPR-related inquiries. From an abuse perspective however, will criminals be able to strategically destroy evidence in the name of privacy? Will identity thieves double as identity assassins for want of well-defined and proven security mechanisms for validating information destruction requests? Will the arrival of employees at work constitute the potential for personal information exfiltration? With nine months remaining until the GDPR directive becomes fully enforceable, will company policies and vendor solutions be mature enough for widespread compliance? Finally, consider that U.S. companies like Google are not exempt from the GDPR if they collect personal information from E.U. citizens during the normal course of business. Rest assured, tools such as FreeNAS and TrueNAS are here to help both comply with the GDPR using per-user datasets and encryption at rest, plus mitigate ransomware attacks with block-level snapshots and clones.

The real-world challenge of political and mechanical compliance with the E.U. GDPR is only one example of the fascinating and timely topics of discussion within SNIA and is why I find participation in SNIA events so valuable. Many SNIA members occupy the top levels of their respective employers yet their passion drives them to volunteer with SNIA by giving talks, chairing committees, and organizing events like the SDC. If this balance of experience, passion, and willingness to leave your marketing guns at the door sounds like you, I invite you to learn more about SNIA and consider joining.

Michael Dexter, *Senior Analyst*, iXsystems

Source:
https://www.ixsystems.com/blog/snia-sdc-2017/

FreeBSD 10.4-RC1 Now Available

The developers of FreeBSD have made available the first RELEASE candidate for version 10.4. Check out the detailed list of changes at the mailing list page. Download the latest ISO here. Note that RC2 may be released shortly after the time of this posting.

Source:
https://www.freebsdnews.com/2017/09/22/freebsd-10-4-beta3/

Military Grade Data Wiping In FreeBSD With BCWipe

Inside

- What Is Data Wiping
- What Is BCWipe
- How To Install BCWipe
- How To Install BCWipe With Multithreaded Mode Enabled
- BCWipe Advanced Features
- BCWipe In Action

What Is Data Wiping

Data wiping is a process of overwriting data on magnetic hard disk, SSD or USB flash by using zeros and ones on whole disk or specific zone. As a result, no one can recover sensitive data and disk is still usable.

Varieties:

1. Software-based wiping

This type of wiping is carried out by a software that is installed on the drive.

2. Hardware-based wiping

This type of wiping needs some external device.

Don't confuse data wiping with file deletion.File deletion only removes direct pointers to the data and makes the data recovery possible with common software tools. Unlike degaussing and physical destruction, which render the storage media unusable, data wiping removes all information but still leaves the disk operable. Data erasure may not work completely on flash based media such as Solid State Drives and USB Flash Drives. This is because such devices can store remnant data which is inaccessible to the wiping technique. Moreover, the

data can be retrieved from the individual flash memory chips in the device.

Wiping software uses many techniques to ensure data is not recoverable like:

1. German BCI/VSITR 7-pass wiping
2. U.S. DoD 5220.22M 7-pass extended character rotation wiping with last pass verification
3. U.S. DoE 3-pass wiping
4. 35-pass Peter Gutmann's wiping
5. 7-pass Bruce Schneier's wiping
6. 1-pass wiping by zeroes

What Is BCWipe

BCWipe securely erases data from magnetic and solid-state memory. BCWipe repeatedly overwrites special patterns to the files or frees space to be destroyed. In normal mode, 35 passes are used (of which 8 are random).The p used were recommended in an article by Peter Gutmann entitled "Secure Deletion of Data from Magnetic and Solid-State Memory". In quick mode, U.S. DoD(Department of Defence) 5220.22-M standard is used with 7 pass wiping. In custom mode, U.S. DoD 5220.22-M standard is used with user defined number of passes.

How To Install BCWipe

BCWipe is available on FreeBSD ports tree, and you can easily install it.

```
# make -C /usr/ports/security/bcwipe install clean
```

Or, you can install BCWipe with PKG mechanism:

```
# pkg install bcwipe
```

How To Install BCWipe With Multi-threaded Mode Enabled

BCWipe has no compile option through FreeBSD port mechanism. Instead, you can rebuild BCWipe with multi-threading mode option :

```
# cd /usr/ports/security/bcwipe/

# make fetch extract

# cd work/bcwipe-1.9-9/

# ./configure –enable-pthreads

# make install clean
```

BCWipe Advanced Features

Bcwipe has useful features that make wiping process more suitable.

- `-n <delay>`

Wait delay seconds between wiping passes. Modern enterprise level storage systems (NAS, disk arrays etc.) employ powerful caches. To avoid undesirable caching effects, BCWipe allows users to insert adjustable delay between wiping passes. Please note that when wiping with delay between passes, the disk space is freed after the last pass.

- `-B   Disables direct IO mode when wiping block devices`

- `-t <threads>`

Wipes and verifies block devices in multi-thread mode. BCWipe runs worker threads. Useful for wiping multiple disk volumes.

- `-S (wipe file slack)`

Wipes files' slack. File slack is the disk space from one end of a file to the end of the last cluster used by that file. Cluster refers to the minimal portion of disk space used by the file system.

- -s Uses ISAAC random number generator by Bob Jenkins

Default is SHA-1 (Secure Hash Algorithm). ISAAC is random faster than SHA-1.

- -F (wipe free space) Wipes free space on specified filesystem.

- -b (block device) Wipes contents of block devices

BCWipe In Action

In this section, we describe a real scenario with BCWipe.

Issue the following command to get more information about BCWipe:

```
# bcwipe
```

Tip: in real-world scenario, people want to wipe out free space on whole mounted disks (/). However, the bcwipe command must be issued with caution.

To wipe free space:

```
# bcwipe -F /mnt/
```

This command will wipe out free space on /mnt/ path or whole mounted disks on this path.

```
# bcwipe -Fv -mt  /mnt/
```

This command wipes out free space on /mnt/ directory with 1-pass in verbose mode.

```
-mt refer to 1-pass.
```

To wipe a specific file:

```
# bcwipe -v -mz wipe.me
```

This command wipes wipe.me file with 1-pass wiping by zeroes in verbose mode.

```
# bcwipe -Fv -mg -t 5 /mnt/
```

This command wipes free space on /mnt/ directory with 35-pass Peter Gutmann's scheme by 5 threads in verbose mode.

To wipe a specific folder:

```
# bcwipe -rv /tmp/
```

This command wipes /tmp/ directory recursively with Peter Gutmann's scheme in verbose mode.

To wipe block device:

```
# bcwipe -v -mz -t2 -b /dev/da0
```

This command wipes /dev/da0 (USB flash) with 2 threads by 1-pass zeroes in verbose mode.

The point is, USB flash is not mounted and all of the data will be destroyed.

Conclusion

BCWipe along with FreeBSD give you military-grade functionalites, ensuring your sensitive data will not fall into the wrong hands.

Useful Links

http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-88r1.pdf

https://www.jetico.com/linux/bcwipe-help/wu_using.htm

https://www.nsa.gov/resources/everyone/media-destruction/

https://www.usenix.org/legacy/events/fast11/tech/full_papers/Wei.pdf

https://www.sans.org/reading-room/whitepapers/incident/secure-file-deletion-fact-fiction-631

About The Author

Abdorrahman Homaei has been working as a software developer since 2000. He has used FreeBSD for more than ten years. He became involved with the meetBSD dot ir and performed serious training on FreeBSD. He is starting his own company (corebox) in Feb 2017.

Full CV: http://in4bsd.com

His company: http://corebox.ir

HEY GOLIATH...

MEET DAVID

TRUENAS® PROVIDES MORE PERFORMANCE, FEATURES, AND CAPACITY PER-DOLLAR THAN ANY ENTERPRISE STORAGE ARRAY ON THE MARKET.

Introducing the TrueNAS X-Series: Perfectly suited for core-edge configurations and enterprise workloads such as backups, replication, and file sharing.

- ★ **Unified:** Simultaneous SAN, NAS, and object protocols to support multiple applications
- ★ **Scalable:** Up to 120 TB in 2U and 720 TB in 6U
- ★ **Safe:** High Availability ensures business continuity and avoids downtime
- ★ **Reliable:** Uses OpenZFS to keep data safe
- ★ **Trusted:** TrueNAS is the Enterprise version of FreeNAS®, the world's #1 Open Source SDS
- ★ **Enterprise:** Enterprise-class storage including unlimited instant snapshots and advanced storage optimization at a lower cost than equivalent solutions from Dell EMC, NetApp, and others

The TrueNAS X10 and TrueNAS X20 represent a new class of enterprise storage. Get the full details at iXsystems.com/TrueNAS.

Copyright © 2017 iXsystems. TrueNAS and FreeNAS are registered trademarks of iXsystems, Inc. All rights reserved.

HAMMER File System Volumes Management and Pseudo File Systems Mirroring in DragonFlyBSD

In DragonFlyBSD, the HAMMER filesystem allows us to perform functions akin to Logical Volume Management in Linux, like adding and removing volumes from Volume Group or Pools, Creating Logical Volumes with separate file systems, taking Logical Volume snapshot, etc. This activity part of this article was done on a DragonFlyBSD Installation on a Debian Stretch Linux Kernel Virtual Machine with one 100 GB and four 1 TB qcow2 hard disks. The reason for the choice of DragonFlyBSD and HAMMER file system was described in a previous article. A third article will follow describing PFS and snapshots management.

Formating the 4 1TB disks with HAMMER file system

First, let us see our System mount setup soon after installing the DragonFly on the 100 GB hard disk.

```
dfly1# df -h
Filesystem                Size   Used  Avail Capacity  Mounted on
ROOT                     76.5G  1267M  75.2G     2%    /
devfs                    1024B  1024B     0B   100%    /dev
/dev/serno/QM00001.s1a   1022M   362M   578M    38%    /boot
BUILD                    19.5G   219M  19.3G     1%    /build
/build/usr.obj           19.5G   219M  19.3G     1%    /usr/obj
/build/var.crash         19.5G   219M  19.3G     1%    /var/crash
/build/var.cache         19.5G   219M  19.3G     1%    /var/cache
/build/var.spool         19.5G   219M  19.3G     1%    /var/spool
/build/var.log           19.5G   219M  19.3G     1%    /var/log
/build/var.tmp           19.5G   219M  19.3G     1%    /var/tmp
tmpfs                     236M     0B   236M     0%    /tmp
procfs                   4096B  4096B     0B   100%    /proc
dfly1# mount
ROOT on / (hammer, noatime, local)
devfs on /dev (devfs, nosymfollow, local)
/dev/serno/QM00001.s1a on /boot (ufs, local)
BUILD on /build (hammer, noatime, local)
```

```
/build/usr.obj on /usr/obj (null)
/build/var.crash on /var/crash (null)
/build/var.cache on /var/cache (null)
/build/var.spool on /var/spool (null)
/build/var.log on /var/log (null)
/build/var.tmp on /var/tmp (null)
tmpfs on /tmp (tmpfs, local)
procfs on /proc (procfs, local)
```

Let us look at the hard disks available to us:

```
dfly1# ls /dev/ad*
/dev/ad0        /dev/ad0s1      /dev/ad0s1a     /dev/ad0s1b     /dev/ad0s1d     /dev/ad0s1e
dfly1# ls /dev/da*
/dev/da0        /dev/da0s0      /dev/da1        /dev/da1s0      /dev/da2        /dev/da2s0
/dev/da3        /dev/da3s0
```

IDE Disks start with **ad, whereas SATA** Disks start with **da.**

```
dfly1# dmesg | grep ad0
ad0: 102400MB <QEMU HARDDISK 2.5+> at ata0-master WDMA2
dfly1# dmesg | grep da
da0 at ahci0 bus 0 target 0 lun 0
da0: <SATA QEMU HARDDISK 2.5+> Fixed Direct Access SCSI-4 device
da0: Serial Number QM00005
da0: 150.000MB/s transfers
da0: 1024000MB (2097152000 512 byte sectors: 255H 63S/T 130541C)
da1 at ahci0 bus 1 target 0 lun 0
da1: <SATA QEMU HARDDISK 2.5+> Fixed Direct Access SCSI-4 device
da1: Serial Number QM00007
da1: 150.000MB/s transfers
da1: 1024000MB (2097152000 512 byte sectors: 255H 63S/T 130541C)
da2 at ahci0 bus 2 target 0 lun 0
da2: <SATA QEMU HARDDISK 2.5+> Fixed Direct Access SCSI-4 device
da2: Serial Number QM00009
da2: 150.000MB/s transfers
da2: 1024000MB (2097152000 512 byte sectors: 255H 63S/T 130541C)
da3 at ahci0 bus 3 target 0 lun 0
da3: <SATA QEMU HARDDISK 2.5+> Fixed Direct Access SCSI-4 device
da3: Serial Number QM00011
da3: 150.000MB/s transfers
da3: 1024000MB (2097152000 512 byte sectors: 255H 63S/T 130541C)
```

Let us now format the four **1TB** Hard Disks with the **HAMMER** file system. We need to specify a **LABEL** to each **HAMMER** file system during format. Here, we create a HAMMER file system spanning two physical volumes or disks.

```
dfly1# newfs_hammer -L POOL1 /dev/da0 /dev/da1
Volume 0 DEVICE /dev/da0        size   0.98TB
Volume 1 DEVICE /dev/da1        size   0.98TB
initialize freemap volume 0
initializing the undo map (1024 MB)
initialize freemap volume 1
---------------------------------------------
HAMMER version 7
2 volumes total size   1.95TB
root-volume:         /dev/da0
boot-area-size:       32.00KB
memory-log-size:     256.00KB
undo-buffer-size:      1.00GB
total-pre-allocated:   1.02GB
fsid:                e596db85-92ad-11e7-944b-535400e58e52

dfly1# newfs_hammer -L POOL2 /dev/da2
Volume 0 DEVICE /dev/da2        size   0.98TB
initialize freemap volume 0
initializing the undo map (1024 MB)
---------------------------------------------
HAMMER version 7
1 volume total size   0.98TB
root-volume:         /dev/da2
boot-area-size:       32.00KB
memory-log-size:     256.00KB
undo-buffer-size:      1.00GB
total-pre-allocated:   1.02GB
fsid:                536b2e56-92ae-11e7-944b-535400e58e52
```

Mounting the HAMMER File Systems

```
dfly1# mkdir /POOL1 /POOL2
dfly1# ls /POOL*
/POOL1:
/POOL2:
```

/etc/fstab entries for mounting the filesystems during **boot** or with **mount** command

```
# Our POOLs
/dev/da0s0:/dev/da1s0           /POOL1          hammer  rw              1       1
/dev/da2s0                                  /POOL2          hammer  rw              1       1
dfly1# mount -a
mount: Device busy
mount: Device busy
dfly1# mount
ROOT on / (hammer, noatime, local)
devfs on /dev (devfs, nosymfollow, local)
/dev/serno/QM00001.s1a on /boot (ufs, local)
BUILD on /build (hammer, noatime, local)
/build/usr.obj on /usr/obj (null)
/build/var.crash on /var/crash (null)
/build/var.cache on /var/cache (null)
/build/var.spool on /var/spool (null)
/build/var.log on /var/log (null)
/build/var.tmp on /var/tmp (null)
tmpfs on /tmp (tmpfs, local)
procfs on /proc (procfs, local)
POOL1 on /POOL1 (hammer, noatime, local)
POOL2 on /POOL2 (hammer, noatime, local)
dfly1# df -h
Filesystem               Size   Used  Avail Capacity  Mounted on
ROOT                    76.5G  1267M  75.2G     2%    /
devfs                   1024B  1024B     0B   100%    /dev
/dev/serno/QM00001.s1a  1022M   362M   578M    38%    /boot
BUILD                   19.5G   219M  19.3G     1%    /build
/build/usr.obj          19.5G   219M  19.3G     1%    /usr/obj
/build/var.crash        19.5G   219M  19.3G     1%    /var/crash
/build/var.cache        19.5G   219M  19.3G     1%    /var/cache
/build/var.spool        19.5G   219M  19.3G     1%    /var/spool
/build/var.log          19.5G   219M  19.3G     1%    /var/log
/build/var.tmp          19.5G   219M  19.3G     1%    /var/tmp
tmpfs                    236M     0B   236M     0%    /tmp
procfs                  4096B  4096B     0B   100%    /proc
POOL1                   1999G   203M  1999G     0%    /POOL1
POOL2                    999G   203M   999G     0%    /POOL2
```

Creating and mounting Pseudo File Systems

Initially, we will create Pseudo File Systems such that :

POOL1 houses Master PFSes

POOL2 houses Slave PFSes for mirroring

By convention, PFSes are created in the "pfs" directory under the Hammer Mother File system.

Creating Master PFS ISOs in POOL1

```
dfly1# cd /POOL1/

dfly1# ls

dfly1# mkdir pfs

dfly1# cd pfs

dfly1# hammer pfs-master ISOs

Creating PFS#1  succeeded!

ISOs

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x0000000100008020


shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52


unique-uuid=0ae191f8-92bb-11e7-944b-535400e58e
52

    label=""

    prune-min=00:00:00

    operating as a MASTER

    snapshots directory defaults to
/var/hammer/<pfs>
```

Null mounting PFS under the Mother file system

```
dfly1# cd /POOL1/

dfly1# ls

pfs

dfly1# mkdir ISOs
```

/etc/fstab entry for master PFS for mounting during **boot,** or with **mount** command.

```
# Our Master PFS mounts

/POOL1/pfs/ISOs                  /POOL1/ISOs
null    rw
```

Relevant output of "mount" command

```
dfly1# mount -a

dfly1# mount

POOL1 on /POOL1 (hammer, noatime, local)

POOL2 on /POOL2 (hammer, noatime, local)

/POOL1/pfs/@@-1:00001 on /POOL1/ISOs (null,
local)
```

Creating Slave PFS ISOs in POOL2

We will use these for mirroring data from Master PFS ISOs in POOL1.

For PFS mirroring to take place, **shared-uuid** of Slave PFS should be same as Master PFS

```
dfly1# hammer pfs-slave ISOs
shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52

Creating PFS#1  succeeded!

ISOs

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x0000000000000001


shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52


unique-uuid=5ed24fd0-92bc-11e7-944b-535400e58e
52

    label=""

    prune-min=00:00:00

    operating as a SLAVE

    snapshots directory defaults to
/var/hammer/<pfs>
```

Note : Slave PFSes should not be mounted!

Configure continuous mirroring using "mirror-stream"

```
dfly1# hammer mirror-stream /POOL1/ISOs/
/POOL2/pfs/ISOs

Prescan to break up bulk transfer

Prescan 1 chunks, total 0 MBytes (208)
```

Checking mirroring from POOL1 to POOL2 by looking at the number of files created

```
dfly1# cd /POOL1/ISOs/ && ls -l | wc -l && cd
/POOL2/pfs/ISOs/ && ls -l | wc -l

     281

     250
```

```
dfly1# cd /POOL1/ISOs/ && ls -l | wc -l && cd /POOL2/pfs/ISOs/ && ls -l | wc -l

     389

     358

dfly1# cd /POOL1/ISOs/ && ls -l | wc -l && cd /POOL2/pfs/ISOs/ && ls -l | wc -l

     420

     385

dfly1# cd /POOL1/ISOs/ && ls -l | wc -l && sleep 5 && cd /POOL2/pfs/ISOs/ && ls -l | wc -l

     507

     493

dfly1# cd /POOL1/ISOs/ && ls -l | wc -l && sleep 5 && cd /POOL2/pfs/ISOs/ && ls -l | wc -l

     516

     493

dfly1# cd /POOL1/ISOs/ && ls -l | wc -l && sleep 5 && cd /POOL2/pfs/ISOs/ && ls -l | wc -l

     536

     520
```

Mirroring can be continued even after reboot by adding the following entry to cron:

```
@reboot hammer mirror-stream /POOL1/ISOs/ /POOL2/pfs/ISOs
```

Creating Second Slave Mirror to do a one shot mirroring using "mirror-copy"

This operation ends once mirroring is complete, and does not go on continuously like "mirror-stream"

```
dfly1# cd /POOL2/pfs/

dfly1# ls

ISOs

dfly1# hammer pfs-status /POOL1/ISOs/

/POOL1/ISOs/    PFS#1 {

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x0000000100020e10

shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e52

unique-uuid=0ae191f8-92bb-11e7-944b-535400e58e52

    label=""

    prune-min=00:00:00

    operating as a MASTER

    snapshots directory defaults to /var/hammer/<pfs>

}

dfly1# hammer pfs-slave ISOs2 shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e52

Creating PFS#2  succeeded!

ISOs2

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x0000000000000001

shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e52

unique-uuid=ce39fcdb-92c6-11e7-944b-535400e58e52

    label=""

    prune-min=00:00:00

    operating as a SLAVE

    snapshots directory defaults to /var/hammer/<pfs>

dfly1# hammer mirror-copy /POOL1/ISOs/ /POOL2/pfs/ISOs2

Prescan to break up bulk transfer

Prescan 1 chunks, total 0 MBytes (442760)

Mirror-read /POOL1/ISOs/ succeeded

dfly1# cd /POOL2/pfs/ISOs2/ && ls -l | wc -l

    1384
```

Adding a second Physical Volume to POOL2

This will format the device and add all of its space to filesystem.

A HAMMER file system can use up to **256** volumes.

```
 dfly1# hammer volume-add /dev/da3s0 /POOL2

/dev/da2s0

/dev/da3s0

dfly1# df -h |grep POOL

POOL1                    1999G   203M  1999G    0%    /POOL1
```

```
POOL2                     1998G   203M  1998G
0%    /POOL2

/POOL1/pfs/@@-1:00001   1999G   203M  1999G
0%    /POOL1/ISOs
```

The new volume will be automatically mounted. So if you try to mount manually again, you will get the following message:

kernel: hammer_vfs_mount: The volumes are probably mounted

You can make changes permanent by editing "/etc/fstab" i.e. adding the new device to the existing "POOL2" entry as shown

```
/dev/da2s0:/dev/da3s0           /POOL2
hammer  rw               1       1
```

Check if the mirroring process is at work after boot

```
dfly1# ps aux | grep mirror

root    886  0.0  0.2   5012   2288  2  I0+
5:53AM   0:00.01 hammer mirror-stream
/POOL1/ISOs/ /POOL2/pfs/ISOs

root    887  0.0  0.1  21916   1340  2  I1+
5:53AM   0:00.03 hammer mirror-stream
/POOL1/ISOs/ /POOL2/pfs/ISOs

root    888  0.0  0.1   5528   1248  2  I1+
5:53AM   0:00.00 hammer mirror-stream
/POOL1/ISOs/ /POOL2/pfs/ISOs

root    897  0.0  0.1    636    504  3  R1+
5:55AM   0:00.01 grep mirror
```

Simple check to see if Master and slave are in sync

```
dfly1# cd /POOL1/ISOs/ && ls -l | wc -l &&
sleep 5 && cd /POOL2/pfs/ISOs/ && ls -l | wc
-l

    2279

    2279
```

Advanced check to see if Master and Slave are in sync

```
dfly1# hammer pfs-status /POOL1/ISOs | grep
sync

     sync-beg-tid=0x0000000000000001

    sync-end-tid=0x0000000100029880

dfly1# hammer pfs-status /POOL2/pfs/ISOs |
grep sync

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x0000000100029880
```

HowTo: Disaster recovery if the Master PFS /POOL1/pfs/ISOs is destroyed.

Destroying Master PFS /POOL1/pfs/ISOs mounted on /POOL1/ISOs

dfly1# hammer pfs-destroy /POOL1/pfs/ISOs

You have requested that PFS#1 () be destroyed

This will irrevocably destroy all data on this PFS!!!!!

Do you really want to do this? [y/n] y

This PFS is currently setup as a MASTER!

Are you absolutely sure you want to destroy it? [y/n] y

Destroying PFS#1 () in 5 4 3 2 1.. starting destruction pass

pfs-destroy of PFS#1 succeeded!

Ok, we still have the files on the Slave PFS **/POOL2/pfs/ISOs.** So, we can take the following steps for Disaster Recovery:

Upgrade Slave PFS to become a Master PFS.

Create a new Save PF .

Reconfigure mirroring from New Master PFS to new Slave PFS .

To start the disaster recovery process, first, we have to stop all mirroring processes to the Slave.

```
dfly1# ps aux | grep mirror

root    886  0.0  0.2   5012   2288  2  I0+
5:53AM   0:00.01 hammer mirror-stream
/POOL1/ISOs/ /POOL2/pfs/ISOs
```

```
root    887  0.0  0.1  21916   1376  2  I1+
5:53AM   0:00.06 hammer mirror-stream
/POOL1/ISOs/ /POOL2/pfs/ISOs

root    888  0.0  0.1   5528   1252  2  I1+
5:53AM   0:00.00 hammer mirror-stream
/POOL1/ISOs/ /POOL2/pfs/ISOs

root    925  0.0  0.1    636    500  3  R1+
6:13AM   0:00.00 grep mirror

dfly1# pkill -f "hammer mirror-stream
/POOL1/ISOs/ /POOL2/pfs/ISOs"

dfly1# ps aux | grep mirror

root    928  0.0  0.1   3524   1460  3  S0+
6:14AM   0:00.01 grep mirror
```

Upgrading Slave PFS to Master

The PFS will be rolled back to the current end synchronization transaction id

(removing any partial synchronizations), and it will then become writable. Slave PFSes are not writable in the normal way.

```
dfly1# hammer pfs-upgrade /POOL2/pfs/ISOs

pfs-upgrade of PFS#1 () succeeded
```

Mounting the new master and deleting the fstab entry for old master

```
dfly1# cd /POOL2

dfly1# ls

pfs

dfly1# mkdir ISOs
```

/etc/fstab changes

```
# Our Master PFS mounts

/POOL2/pfs/ISOs                  /POOL2/ISOs
null    rw
```

Unmounting old master and checking if the new Master PFS was mounted.

```
dfly1# umount /POOL1/ISOs

dfly1# mount | grep POOL

POOL1 on /POOL1 (hammer, noatime, local)

POOL2 on /POOL2 (hammer, noatime, local)

/POOL2/pfs/@@-1:00001 on /POOL2/ISOs (null,
local)
```

Creating a new slave in POOL1 to mirror from new master /POOL2/ISOs

```
dfly1# cd /POOL1

dfly1# ls

ISOs    pfs

dfly1# cd pfs

dfly1# ls

dfly1# hammer pfs-status /POOL2/ISOs/ | grep
shared-uuid


shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52

dfly1# hammer pfs-slave ISOs
shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52

Creating PFS#2  succeeded!

ISOs

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x0000000000000001


shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52


unique-uuid=81bdeecc-936c-11e7-b359-535400e58e
52

    label=""

    prune-min=00:00:00

    operating as a SLAVE

    snapshots directory defaults to
/var/hammer/<pfs>

dfly1# hammer mirror-copy /POOL2/ISOs/
/POOL1/pfs/ISOs

Prescan to break up bulk transfer

Prescan 1 chunks, total 0 MBytes (729160)
```

```
Mirror-read /POOL2/ISOs/ succeeded

dfly1# cd /POOL2/ISOs/ && ls -l | wc -l &&
sleep 5 && cd /POOL1/pfs/ISOs/ && ls -l | wc
-l

    2279

    2279

dfly1# hammer pfs-status /POOL2/ISOs/ | grep
sync

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x00000001000318a0

dfly1# hammer pfs-status /POOL1/pfs/ISOs/ |
grep sync

    sync-beg-tid=0x0000000000000001

    sync-end-tid=0x00000001000318a
```

Start mirroring from new Master PFS to new Slave PFS using "mirror-stream"

```
dfly1# nohup hammer mirror-stream /POOL2/ISOs/
/POOL1/pfs/ISOs &

[1] 1046

dfly1# ps aux | grep mirror

root   1046  0.2  0.2   5012   2304  2  S0
7:14AM   0:00.01 hammer mirror-stream
/POOL2/ISOs/ /POOL1/pfs/ISOs

root   1047  0.0  0.1  21916   1116  2  S1
7:14AM   0:00.01 hammer mirror-stream
/POOL2/ISOs/ /POOL1/pfs/ISOs

root   1048  0.0  0.1   5528   1160  2  S0
7:14AM   0:00.01 hammer mirror-stream
/POOL2/ISOs/ /POOL1/pfs/ISOs

root   1050  0.0  0.1   3480   1428  2  R1+
7:14AM   0:00.00 grep mirror
```

You can continue the mirroring operation even after a reboot using the following Cron entry:

```
@reboot hammer mirror-stream /POOL2/ISOs/
/POOL1/pfs/ISOs
```

Mirroring from a Master PFS to a Slave PFS then to another Slave PFS

You can do this to fulfill a scenario where you need replicated data both on premises as well as in another continent. The replication between **DragonFly** systems can take place securely on the internet using SSH. To accomplish this, all the **PFS**es involved should have the same "**shared-uuid".**

Let us create three such PFSes in our system. One Master in **POOL2** (already existing) and two Slaves in **POOL1**. As mentioned earlier, the second slave could be on a DragonFly system in another continent connected through the Internet using SSH.

```
dfly1# hammer pfs-status /POOL2/ISOs | grep
shared-uuid

shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52
dfly1# hammer pfs-slave /POOL1/pfs/ISOsSlave1
shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52
Creating PFS#1  succeeded!
/POOL1/pfs/ISOsSlave1
    sync-beg-tid=0x0000000000000001
    sync-end-tid=0x0000000000000001

shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52

unique-uuid=0604dc83-95d2-11e7-bf86-0100000000
00
    label=""
    prune-min=00:00:00
    operating as a SLAVE
    snapshots directory defaults to
/var/hammer/<pfs>
dfly1# hammer pfs-slave /POOL1/pfs/ISOsSlave2
shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52
Creating PFS#3  succeeded!
/POOL1/pfs/ISOsSlave2
    sync-beg-tid=0x0000000000000001
    sync-end-tid=0x0000000000000001

shared-uuid=0ae1913f-92bb-11e7-944b-535400e58e
52

unique-uuid=0e905044-95d2-11e7-bf86-0100000000
00
    label=""
    prune-min=00:00:00
    operating as a SLAVE
    snapshots directory defaults to
/var/hammer/<pfs>
```

Configuring mirroring from POOL2 Master to First Slave in POOL1

```
dfly1# nohup hammer mirror-stream /POOL2/ISOs
/POOL1/pfs/ISOsSlave1 &
[1] 1281
dfly1# Prescan to break up bulk transfer
Prescan 1 chunks, total 0 MBytes (729160)
dfly1# ps aux | grep mirror
root    1281  0.0  0.2   5012   2292  1  I0
8:18AM   0:00.01 hammer mirror-stream
/POOL2/ISOs /POOL1/pfs/ISOsSlave1
root    1282  0.0  0.2  21916   1748  1  S0
8:18AM   0:00.05 hammer mirror-stream
/POOL2/ISOs /POOL1/pfs/ISOsSlave1
root    1283  0.0  0.2   5528   1688  1  S0
8:18AM   0:00.03 hammer mirror-stream
/POOL2/ISOs /POOL1/pfs/ISOsSlave1
```

Checking Mirror Status

```
dfly1# cd /POOL2/ISOs && du -s && sync && cd
/POOL1/pfs/ISOsSlave1 && du -s
1749316 .
1749316 .
dfly1# cd /POOL2/ISOs && du -s && sync && cd
/POOL1/pfs/ISOsSlave1 && du -s
2170047 .
1749316 .
```

Configuring Slave to Slave mirroring

Now that we have some data in the slave PFS **/POOL1/pfs/ISOsSlave1,** we can start mirroring from it to the second slave **/POOL1/pfs/ISOsSlave2**. This is important because we will not be able to access a slave PFS until it has completed the first mirroring operation with it as the target (its root directory will not exist until then).

```
dfly1# nohup hammer mirror-stream
/POOL1/pfs/ISOsSlave1/ /POOL1/pfs/ISOsSlave2 &
[1] 911
dfly1# Prescan to break up bulk transfer

dfly1# ps aux | grep ISOsSlave2
root     911  0.0  0.2   5012   2304  1  I0
11:53AM   0:00.00 hammer mirror-stream
/POOL1/pfs/ISOsSlave1/ /POOL1/pfs/ISOsSlave2
root     912  0.0  1.0  30112   9868  1  D1
11:53AM   0:00.13 hammer mirror-stream
/POOL1/pfs/ISOsSlave1/ /POOL1/pfs/ISOsSlave2
root     913  0.0  0.1   5528   1096  1  I0
11:53AM   0:00.01 hammer mirror-stream
/POOL1/pfs/ISOsSlave1/ /POOL1/pfs/ISOsSlave2
root     915  0.0  0.4   5520   3528  1  D1V+
11:54AM   0:00.00 grep ISOsSlave2 (csh)
```

Checking Mirror Status

```
dfly1# cd /POOL1/pfs/ISOsSlave1 && du -s &&
sync && cd /POOL1/pfs/ISOsSlave2 && du -s
1749316 .
1749316 .
```

Removing a Volume from a HAMMER File System

Now, we will remove a physical disk from the HAMMER file system named POOL1. It is not possible to remove the root-volume as it contains filesystem metadata such as HAMMER's layer1 blockmap and UNDO/REDO FIFO. This command may also reblock the filesystem before it attempts to remove the volume if any data exists on the volume, and the volume is not empty.

```
dfly1# hammer info /POOL1
Volume identification
        Label               POOL1
        No. Volumes         2
        HAMMER Volumes
/dev/da0s0:/dev/da1s0
        Root Volume         /dev/da0s0
        FSID
e596db85-92ad-11e7-944b-535400e58e52
        HAMMER Version      7
Big-block information
        Total          255867
        Used             2357 (0.92%)
        Reserved           23 (0.01%)
        Free           253487 (99.07%)
Space information
        No. Inodes      20534
        Total size       2.0T (2146367963136
bytes)
        Used              18G (0.92%)
        Reserved         184M (0.01%)
        Free             1.9T (99.07%)
PFS information
          PFS#  Mode    Snaps
             0  MASTER      0 (root PFS)
             1  SLAVE       0
             2  SLAVE       0
             3  SLAVE       0
dfly1# hammer volume-del /dev/da1s0 /POOL1
/dev/da0s0
```

```
dfly1# hammer volume-list /POOL1
/dev/da0s0
```

You can make the changes permanent by editing the following line in /etc/fstab to:

```
# Our POOLs
/dev/da0s0          /POOL1          hammer
rw              1       1
```

Configuring Off Site Mirroring

Let us say a first Master PFS **/POOL1/Data1** is on a Server with IP address **111.111.111.111**

And a first Slave PFS **/POOL2/Data2** is on a Server with IP address **222.222.222.222** located on another floor of the same building as the server with the Master PFS.

Moreover, a second Slave PFS **/POOL3/Data3** is on a Server with IP address **333.333.333.333** situated in another continent.

Mirroring from Master 111.111.111.111 to first Slave 222.222.222.222 can be continued even after a reboot using the following Cron entry in 111.111.111.111

```
@reboot hammer mirror-stream /POOL1/Data1
root@222.222.222.222:/POOL2/Data2
```

Mirroring from first Slave 222.222.222.222 to second Slave 333.333.333.333 can be continued even after a reboot using the following Cron entry in 222.222.222.222

```
@reboot hammer mirror-stream /POOL2/Data2
root@333.333.333.333:/POOL3/Data3
```

Mirroring from a Slave reduces load on the Master and provides more throughput for the Master for writes & reads. For this to work, the SSH key based login for **root** user should be enabled on the Servers.

Characteristics of Physical Devices that can be added to a HAMMER master file system

It is possible to add IDE, SAS, SATA, and SCSI drives to the same pool or HAMMER master file system as long as the Operating System sees them as block devices. The devices are concatenated and remain unused until the previous block device is filled up. Therefore, the read/write speed will depend on the specific block device that is in use.

About The Author

Siju Oommen George started his career in 2001 as BSD/Linux/Windows/Mac sysadmin at hifx.in and has been working on related technologies since. Currently, he is working as CISO at Broadtech Innovations and also holds the post of a Technical Writer in hifx.in. More Information is available at https://www.linkedin.com/in/sijuoommengeorge/

APPLICATION DEBUGGING AND TROUBLESHOOTING

In this workshop you will see real life situations, where debugging skills will save you time, headaches and possibly find a solution with minimal amount of effort.

Debugging/Troubleshooting is a really useful skill when you are working in maintaining legacy applications, doing some small incremental changes to an old code base, where the code has been touched by so many hands over the years and it is becoming really a mess. So, management has decided that the code works as it is and you are not allowed to change it all over “the right way ™”.

https://bsdmag.org/course/application-debugging-and-troubleshooting-2/

Transparent Flow Mapping for NEAT

The NEAT library provides application developers with a unified and platform independent API for network communication, regardless of the underlying network protocol. NEAT's abstraction layer approach allows the integration of new network protocols and transport features, transparently to the user. With QUIC, RTMFP and WebRTC, several widely deployed protocols make use of mapping multiple data streams to a single transport connection. However, the usage of multiplexing requires application developers to spend additional effort and has to be supported by both endpoints. This paper describes an approach to integrate multiplexing functionality into the NEAT library, giving application developers a simple way to use the benefits of mapping multiple data streams to a single transport connection without additional coding effort. We describe our considerations about feature negotiation, connection handling and data transmission for multiplexed data streams, an introduction to the NEAT library, the implementation details as well as measurement results and future steps.

Introduction

The internet is dominated by two transport protocols, TCP and UDP, supported by nearly every operating system and network. However, within the last years, several new transport protocols have been developed to better address the needs of modern network communication, providing new features and improved techniques. Many of these protocols are developed on top of the existing TCP / UDP stack provided by the operating system, increasing the compatibility with existing networks. By deploying them in user space, shorter software update cycles can be realized. The usage of multiplexing to bundle several data paths to a single transport connection has become a key technology for many of these protocols. Adobe's Secure Real-Time Media Flow Protocol (RTMFP) [1], Google's Quick UDP Internet Connections (QUIC) [2] and Web Real-Time Communication Data Channel (WebRTC) [3] are some examples for widely used protocols and protocol stacks using multiplexing. Especially for delay-sensitive applications with a low transmission rate, multiplexing can be very beneficial. The inherent transport protocol mechanisms, like flow-control and congestion-control, improve their effectiveness when larger quantities of data have to be transmitted. In case of packet loss, a higher packet rate per flow will result in faster retransmissions and less application-to-application delay. Also, sharing a common congestion window (cwnd) is beneficial for newly created connections or connections with a low sending rate. Additionally, the reduced amount of parallel connections improves the capacity of servers.

Having the choice between several network protocols with specific characteristics give application developers the ability to use the best matching solution for their use case, but also causes new difficulties. Every protocol, regardless of whether accessed via the operating systems socket API or by a third party library on application level, requires a different API usage.

The NEAT library [4] addresses this issue by offering a unified and cross-platform API for network communication. This includes not only transport protocols offered by the underlying operating system, like TCP, UDP or Stream Control Transmission Protocol (SCTP), but also network protocols which operate at application level. For example, on

platforms without native support for the SCTP protocol, like macOS and NetBSD, NEAT can seamlessly include an SCTP userland implementation [5].

Multiplexing has to be implemented at application level on top of the network stack of the operating system, requiring additional coding effort. The developer either has to implement it from scratch or use an existing library providing an application level protocol which includes this feature. Especially when the application has only limited knowledge of its peer's multiplexing capabilities, a fallback solution is required to guarantee a successful communication. Even if the effort for multiplexing is high and its usage is not beneficial for all traffic patterns, previous investigations [6] have shown that the advantages outweigh the disadvantages for many use-cases.

Our work introduces a multi-purpose multiplexing solution for the NEAT library, providing application developers with the benefits of multiplexing without additional effort. This includes an automatic negotiation mechanism which ensures a maximum of compatibility and transparency to the application. After introducing the NEAT library, its concept of flows and how they map to transport connections, we will explain the concept and implementation of transparently mapping multiple flows to a single SCTP transport connection without prior knowledge about the peer's capabilities. The section is followed by some of our measurement results, considerations about alternative transport protocols and an outlook for our ongoing and future work.

NEAT Library

The NEAT library offers application developers a new interface for network communication. Instead of using the traditional socket API, which requires a lot of protocol and platform specific coding effort, NEAT provides a unified cross-platform API for network communication. This includes DNS-name resolution, connection handling, buffer management and encryption. NEAT is built on top of the libuv [7] event-loop library, and, therefore, it offers a non-blocking and callback-based API. Additionally to the functionality provided by the NEAT library, the developer has full access to the libuv library's functionality. A detailed insight about the concept and architecture of NEAT has been given in [8].

Instead of specifying a transport protocol, the developer specifies his requirements for the properties provided by the transport service for every path. These requirements are for example ordered/unordered delivery, message preservation or reliability. Taking the preferences and requirements of the application into account, the NEAT library chooses the best matching protocol at run-time and cares for the protocol specific connection handling. In addition to the widely used TCP and UDP protocols, NEAT supports the SCTP protocol, the native SCTP implementations on FreeBSD and Linux, as well as the SCTP userland implementation on platforms not having a native support for SCTP, such as macOS and NetBSD.

NEAT Flows

In NEAT, a communication channel between two application endpoints is called *flow*. Flows offer applications a bi-directional data transmission interface to the network.

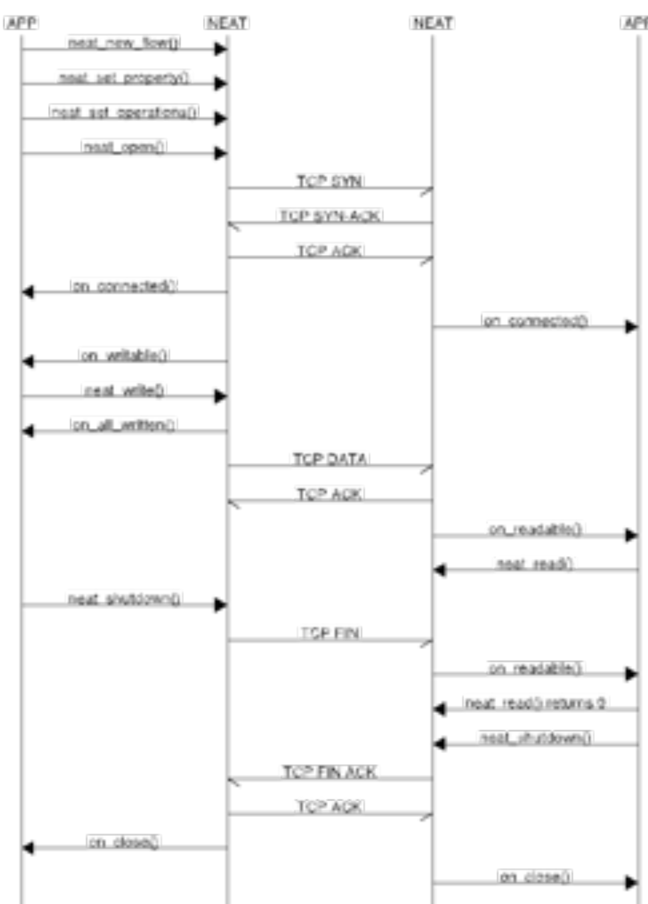


Figure 1. NEAT message and function sequence example using TCP

In order to create a new flow, the client application provides a DNS-name or IP-address and the port-number of the remote endpoint and an optional set of properties. These properties offer a flexible way of configuring the requirements for the new flow, allowing a high level transport feature requirements specification. This includes demanding a reliable data transport and message preserving boundaries, as well as a lower level approach by setting the transport protocol(s) or protocol features, like SCTP's multihoming, and encryption. There is a distinction between required and optional flow properties. For example, an application may require the SCTP protocol for a flow and optionally enable SCTP's multihoming feature. Flows are assigned to flow groups where they have a specific priority within the group, affecting the share of the available bandwidth. If not specified, all flows are assigned to flow zero.

Based on this information and collected data from previous connections, available address-protocol candidates are built, and the NEAT library tries to establish a connection, based on the flow specific properties.

If multiple address-protocol candidates are available, NEAT probes all available candidates by using the Happy-Eyeballs algorithm [9]. In case of several successfully established connections, NEAT will select the best matching connection and close all spare connections. This selection is based on the flows properties, taking the transport protocol specific characteristics and user specified priorities into account. For example, the TCP connection setup takes less round-trips than the SCTP connection setup. If the NEAT library probes TCP and SCTP candidates, the TCP connection will probably be established before the SCTP connection. To overcome this disadvantage for SCTP, NEAT will wait for an additional period of time before evaluating the results. When a connection for a candidate has successfully been established, the NEAT flow changes its state from *connecting* to *open*, and the application will be notified by means of the *on_connected* callback. Figure 1 illustrates the usage and operation of a NEAT flow using the TCP protocol.

When NEAT uses the stream-based TCP transport protocol, the flow is reported ready for data transmission to the application by calling the *on_connected* callback, right after the network socket becomes writable, followed by calling the *on_writable* callback. The application may now send and receive data via the flow's data transmission functions. Due to NEAT's non-blocking-io constraint, applications can write data to connected flows at any time. The NEAT library will try to send the data directly to the network. However, if the socket is not writable or the amount of data cannot be sent at once, the unsent data is buffered in a dedicated flow buffer. The data will be sent as soon as the underlying network socket becomes writable again. When all data has been transmitted to the network and no outstanding data is left in the outgoing flow buffer, NEAT will notify the application by calling the *on_all_written* callback. This callback allows applications to saturate a network connection without bloating the outgoing flow buffer.

When the flow's network socket becomes readable, the NEAT layer notifies the application by triggering the *on_readable* callback. The application can now read data from the flow by using the *neat_read* function and by providing a read buffer with a given size. If the amount of received data exceeds the provided buffer size, the *on_readable* callback will be triggered again until all received data has been handed over to the application. Internally, the application reads directly from the flow's underlying network socket without additional buffering by NEAT. Applications may close a NEAT flow at any time by calling *neat_close* or initiate a graceful connection shutdown by using *neat_shutdown*. The library will transmit all outstanding data to the remote peer, handle the connection closing procedure and trigger the *on_close* callback when all operations have been finished. After the *on_close* callback has been triggered, no flow specific callbacks will be triggered by the library and subsequent calls to read- or write-functions on the flow will result in an error.

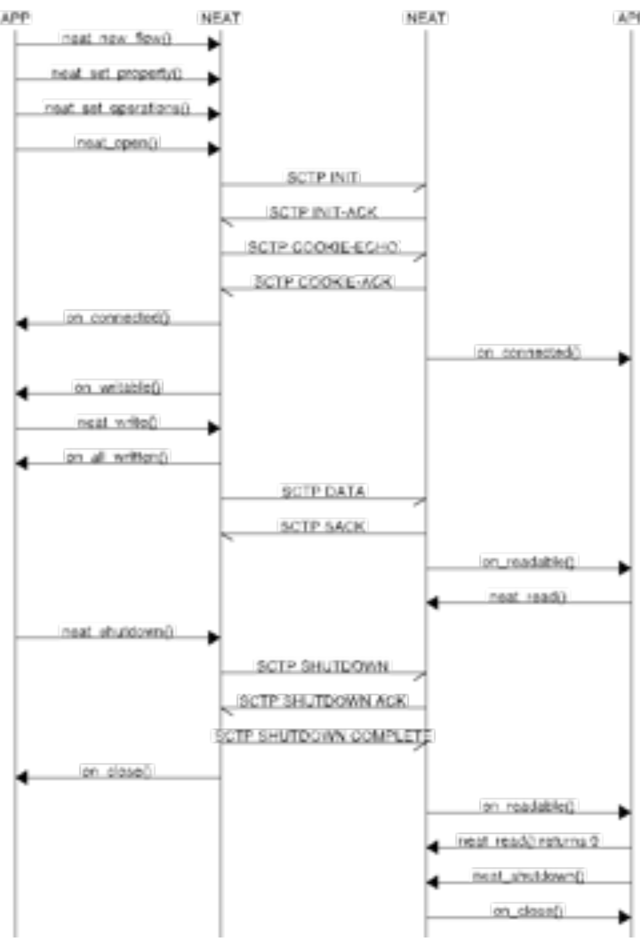


Figure 2. NEAT message and function sequence example using SCTP

The usage of message oriented protocols like SCTP or UDP within NEAT differs internally from stream-based protocols like TCP, but operates transparently to the application. Figure 2 illustrates the usage and operation of a NEAT flow using the SCTP protocol. Once a SCTP based transport connection is established, NEAT will evaluate SCTP specific connection parameters and extensions before announcing the flow's *open* state to the application. The parameters and supported extensions are important for the further usage of the flow. They include the amount of available SCTP streams and support of explicit end of record (EOR) marking, which allows the transmission of arbitrary large user messages by the application. Once all SCTP notifications have been read, NEAT will trigger the *on_connected* callback to notify the application that the flow is ready for data transmission. When the application writes data to the flow, it will be sent to the network or buffered within a flow specific send buffer, similar to TCP, as explained before. In contrast to stream based protocols, NEAT buffers unsent data in a message preserving way by using a message queue. Every user message is buffered in a distinct entry within the queue. When incoming data is available at the network socket, NEAT will read the incoming message into the flow specific receive buffer. If the message is a fragment of a larger user message, NEAT receives and reassembles all fragments before announcing the complete message to the application via the *on_readable* callback. NEAT will only buffer a single user message, no further messages are read from the socket until the buffered message has been read by the application, in order to avoid bloating the incoming buffer on the receiver side. Closing SCTP based flows is similar to the procedure of flows using TCP. NEAT, like SCTP, does not support TCP's half-closed feature, in order to keep the promise of a unified API.

Transparent Flow mapping

Transparent flow mapping hooks into NEAT's abstraction layer approach by multiplexing multiple NEAT flows to a single transport connection without additional actions of the application. If both endpoints support multiplexing and the applications have enabled the support for transparent mapping in their settings, NEAT will automatically use the transparent mapping. As shown in Figure 3, the flows still show up as they would when using a dedicated transport connection, providing the same API and functionality.

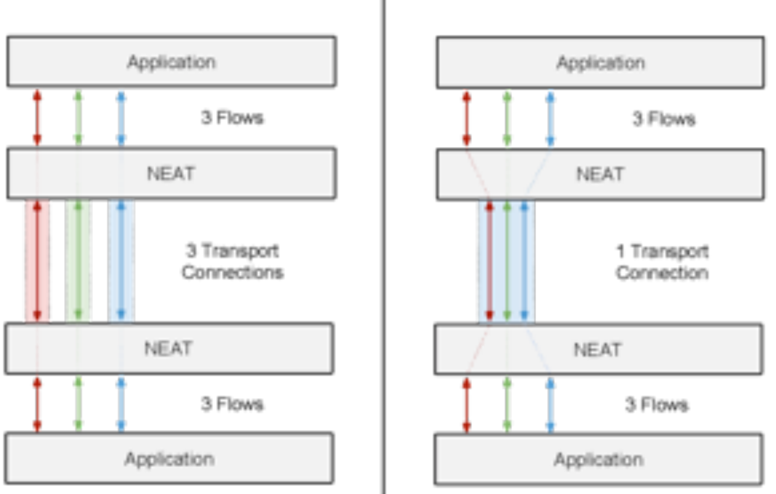


Figure 3. NEAT flows - comparison of 1:1 and transparent flow mapping

Requirements and Negotiation

Before the transparent mapping can be used, both peers have to fulfill some requirements and negotiate the support of the feature. NEAT requires some SCTP specific extensions to be supported by the

network stacks on both sides, including the support for Stream Reconfiguration [10]. The user may require a flow to preserve data message boundaries. In this case NEAT requires the support for the SCTP User Message Interleaving (I-DATA) [11] extension, in order to prevent a sender side head-of-line blocking. If the local requirements are fulfilled, NEAT has to negotiate the multiplexing capabilities with its peer. This is achieved by using SCTP's adaptation layer indication. The NEAT specific adaptation layer indication value is exchanged within SCTP's connection setup procedure and provided as an SCTP notification on both sides, once the connection has been established. If all requirements are met, the transport connection is marked as usable for transparent flow mapping. Otherwise the NEAT library continues operating in regular mode and maps every flow to a separate transport connection. This approach has the advantage of being fully interoperable with peers not using the NEAT library.

Flow creation

Creating a new flow triggers the NEAT library to search for an established SCTP association with a matching tuple of destination address, port, properties and support for transparent flow mapping. Only flows belonging to the same flow group are taken into this survey, allowing the application to prevent multistreaming for a flow by assigning it to an empty flow group.

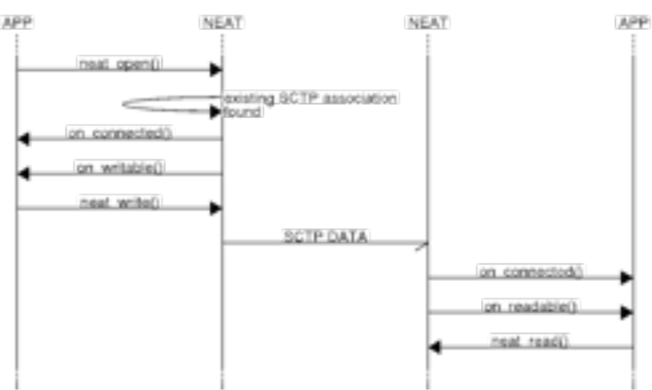


Figure 4. Transparently mapped flow creation procedure

If NEAT discovers a matching SCTP association, the new flow is mapped to it instantly and all ongoing connection establishment procedures for other address-protocol-candidates are stopped. The mapping is realized by assigning the new flow to a dedicated SCTP stream of the established association. The amount of flows per transport connection is limited by the number of available incoming and outgoing SCTP streams per association. SCTP itself supports up to 65535 streams per association. As shown in Figure 4, the NEAT library will notify the application instantly by triggering the *on_connected* and *on_writable* callbacks. If multiple SCTP associations are available for a transparent mapping, NEAT takes the first one to bundle as many flows as possible.

The first flow, for which the SCTP association has initially been created, will always use stream id zero. All additional flows are assigned to unused stream ids. To avoid a glare situation, occurring when both endpoints map new flows simultaneously, the peer which initiated the transport connection will use even stream numbers whereas the remote side will map its flows to odd stream numbers. Both sides maintain a status map of the assigned stream numbers.

Due to the lack of a connection setup procedure on the network, the creation of a new flow is signalized to the remote side by sending the first data message. Transparently mapped flows are instantly ready for data transmission without additional round-trips and, superior to the TCP fast open mechanism [12], the amount of outgoing data is not limited. When receiving an SCTP message on a previously unused stream id, the receiver creates a new incoming flow and triggers the same callbacks as if a new connection using a native transport connection had been opened. Using an implicit flow setup restricts the usage of transparently mapped flows for use cases where the server starts transmitting data to the client without receiving a request, for example a daytime-server. A possible approach to overcome this limitation is the explicit connection setup by sending a control message with a specific Payload Protocol Identifier (PPID) to trigger the incoming flow procedure on the receiver side.

Data transmission

One of the most challenging parts of transparently mapped flows is the handling of incoming and outgoing data. Sharing a network socket between multiple flows requires the NEAT library to cope with scheduling and buffer management techniques. When a shared socket becomes writable, NEAT

schedules over all assigned flows in a round-robin manner. Beginning with the first flow, the library transmits scheduled data from the outgoing flow buffer before triggering the flow's *on_writable* callback. When the flow neither has outstanding data in the buffer nor received new data from the application, the library will continue with the same procedure for the next flow. As mentioned in the negotiation section, applications may send arbitrary large messages and require message boundary preservation. To transmit user messages larger than the maximum segment size (MSS), SCTP supports fragmentation and reassembly. The sender fragments the user message in multiple DATA chunks for transmission which are reassembled by the receiver. If the sender starts transmitting a large user message, consisting of several data chunks, transmissions on all other streams are blocked until all fragments of the user message have been transmitted. To overcome this sender side head-of-line-blocking when transmitting large user messages, NEAT uses the SCTP I-DATA extension. I-DATA solves the sender side head-of-line-blocking issue by supporting message interleaving [11] and is also used in the WebRTC protocol for the same purpose [3]. Another major change for multistreaming affects the receiver side. Whereas a one-to-one style mapped flow only buffers a single incoming user message, a socket used for multistreaming reads messages from the underlying SCTP socket until all assigned flows have at least one user message in their receive buffer. If the sender transmits data on two or more flows and the receiver does not read data from one particular flow, NEAT buffers this data to prevent other multistreamed flows from being blocked by this flow. Limiting the maximum amount of buffered data on the receiver side would either result in dropping data for the particular flow or in blocking all incoming messages for every transparently mapped flow on the affected SCTP socket, both cases are undesirable. A possible approach to overcome this limitation would be application based flow control per transparently mapped flow. Here the receiver signalizes the increasing flow buffer by sending a specific control message to the sender to prevent further transmissions on this particular SCTP stream.

Teardown

Analogous to the creation of a transparently mapped flow, NEAT cannot make use of SCTP's native closing procedure for teardown. Instead, NEAT uses the SCTP Stream Reconfiguration extension for the closing procedure. When the application calls the *neat_shutdown* function for a flow to initiate a graceful shutdown, all outstanding data will be sent and the application may still receive data from its peer, shown in Figure 5. Internally, the flow is marked as closing by the library and once the outgoing flow buffer has been drained, NEAT will trigger the SCTP stream reset procedure for the outgoing stream. After calling the *neat_shutdown* method, the application cannot write any additional data to the flow, the *on_writable* event will not be triggered any more and calling *neat_write* will cause an error.

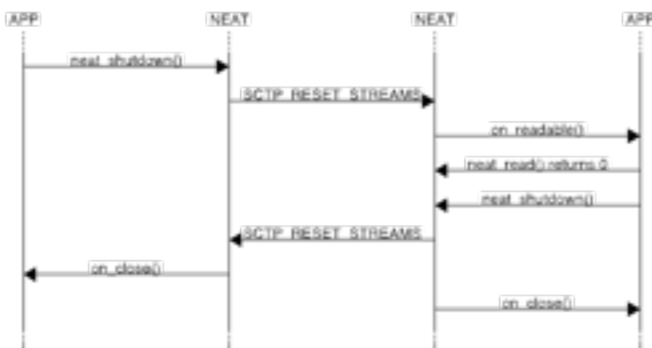


Figure 5. Transparently mapped flow shutdown procedure

Upon receiving an SCTP Stream Request for an incoming stream, NEAT indicates the event by a return value of null when the application calls the *neat_read* function. The flow will not accept new data via the *neat_write* function for transmission. When the remote endpoint also responds with a Stream Reset Request for the incoming stream, the closing procedure of the flow has finished and all resources may be freed. This behavior reflects the connection teardown process for unmapped flows. An application may also use the *neat_close* function. In contrast to *neat_shutdown* the closing procedure resets the outgoing as well as the incoming SCTP streams. Once the closing procedure for a flow has been finished, the SCTP stream id may be reassigned to a new multistreamed flow. Both endpoints maintain an SCTP association assigned status map for every stream id.

Measurements

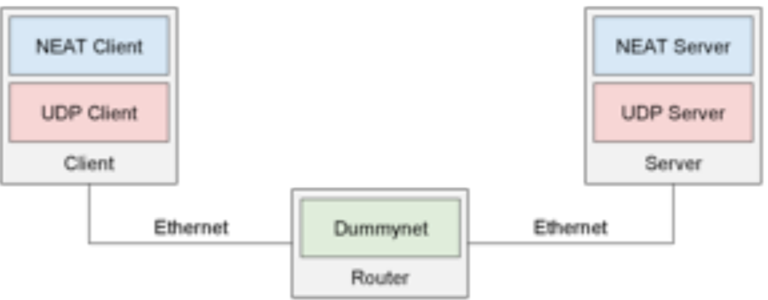


Figure 6. NEAT flow mapping - 1:1 mapping and transparent flow mapping

To examine the advantages and disadvantages of a transparent mapping, we used a client-router-server scenario. All machines are physical nodes running FreeBSD 12 with a *GENERIC-NODEBUG* kernel. As shown in Figure 6, the NEAT Client and the NEAT Server are connected via the router which emulates various network conditions between the two peers by using FreeBSD's builtin dummynet [13] network emulation tool. The router emulates different network conditions by adding delay and packet loss to the path between the server and the client. To achieve some randomness during the measurements, the client transmits a low amount of random UDP messages to the server. Our benchmarking tool, using the NEAT library, is designed to measure a variety of parameters, including the application-to-application-delay between both applications for every flow.

The scenario compares the impact of packet loss and link delay for mapped and unmapped streams concerning application-to-application delay. The NEAT Client opens two SCTP based flows to the NEAT Server and sends small messages between 100-200 bytes periodically with a low rate on each flow to simulate an application using multiple flows for data transmission. This behavior is typical for many use-cases like a browser scheduling requests over multiple connections or control systems reporting data to a central instance.

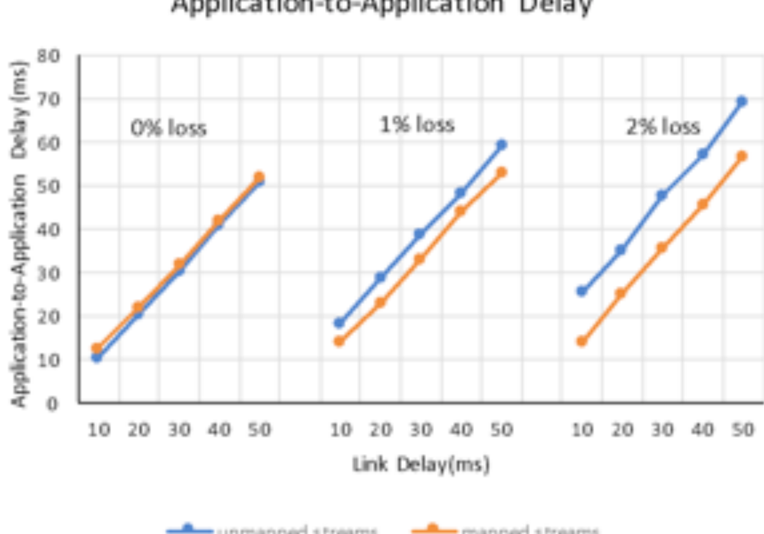


Figure 7. Measurement results comparison of mapped and unmapped flows

We varied the link delay, starting with 10 milliseconds in steps of 10 milliseconds to a delay of 50 milliseconds and used loss rates of zero, one and two percent on the link. Every measurement ran for 60 seconds and was repeated ten times. As shown in Figure 7, the results show a slightly higher application delay for multiplexed flows on connections without loss, resulting from internal data handling within the NEAT library. In case of packet loss, the transparently mapped flows show a lower delay compared to regular flows. This is a result of better utilizing the transport protocols loss detection algorithms.

Our results show a significant application-to-application delay improvement for transparently mapped flows in comparison to regular flows, fulfilling our expectations.

Alternative transport protocol considerations

As mentioned in the previous sections, transparent flow mapping is not tied to the SCTP protocol. In addition to the SCTP protocol, Google's QUIC protocol also covers many requirements for the transparent mapping of multiple flows and, since it is layered on top of UDP, it can seamlessly be integrated into NEAT's abstraction layer approach. Mainly developed to replace TCP as the underlying transport protocol for HTTP2, QUIC is not tied to this use-case and may be used by any other application for generic purposes. Similar to SCTP, QUIC uses

multiplexed streams and does not suffer from head-of-line blocking. In contrast to SCTP, QUIC supports zero-RTT connection setup and uses encryption by default. Due to QUIC's early stage of development and the lack of a specification, QUIC is a candidate for future work. Another candidate is Adobe's RTMFP protocol which is also UDP based and multiplexes multiple flows over a single transport connection. Although specified in by an RFC [1], no official RTMFP library is available and the development has been discontinued.

Conclusion and outlook

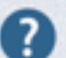
While multiplexing of several data streams on a single transport connection has become a feature more and more popular due to its usage within new protocols, it still requires additional effort for application developers. Especially when the application has no knowledge about its peer. Even if the developer uses a userland implementation of a transport protocol that supports multiplexing, it still remains an additional coding effort, especially when a fallback solution is desired. With NEAT's approach of creating an abstraction layer on top of the different network protocol APIs to give developers a unified way of accessing transport function. We were able to seamlessly integrate a transparent flow mapping feature which gives application developers the benefit of multiplexing without additional coding effort and still being fully compatible. We introduced our approach for multiplexing using SCTP, the integration in the NEAT library and the techniques for feature negotiation, flow handling and data transmission. Our measurements show advantages of transparently mapped flows over regular flows in usual use-cases. In our ongoing work, we are focusing on improving the buffer management and scheduling of concurrent multiplexed flows. Additionally, we will add support for WebRTC Data-Channels [3] to the NEAT library. This allows developers to use NEAT not only for client-server communication but also for building peer-to-peer applications.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334

BSD Certification

The BSD Certification Group Inc. (BSDCG) is a non-profit organization committed to creating and maintaining a global certification standard for system administration on BSD based operating systems.



WHAT CERTIFICATIONS ARE AVAILABLE?

BSDA: Entry-level certification suited for candidates with a general Unix background and at least six months of experience with BSD systems.

BSDP: Advanced certification for senior system administrators with at least three years of experience on BSD systems. Successful BSDP candidates are able to demonstrate strong to expert skills in BSD Unix system administration.

WHERE CAN I GET CERTIFIED?

We're pleased to announce that after 7 months of negotiations and the work required to make the exam available in a computer based format, that the BSDA exam is now available at several hundred testing centers around the world. Paper based BSDA exams cost $75 USD. Computer based BSDA exams cost $150 USD. The price of the BSDP exams are yet to be determined.

Payments are made through our registration website:
https://register.bsdcertification.org//register/payment

WHERE CAN I GET MORE INFORMATION?

More information and links to our mailing lists, LinkedIn groups, and Facebook group are available at our website:
http://www.bsdcertification.org

Registration for upcoming exam events is available at our registration website:
https://register.bsdcertification.org//register/get-a-bsdcg-id

(NEAT). The views expressed are solely those of the authors.

[1] M. Thornburgh, “Adobe’s Secure Real-Time Media Flow Protocol.” Internet Engineering Task Force; RFC 7016 (Informational); IETF, Nov-2013.

[2] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, “QUIC: A udp-based secure and reliable transport for http/2,” IETF Secretariat; Working Draft, Internet-Draft draft-hamilton-early-deployment-quic-00, July 2016.

[3] R. Jesup, S. Loreto, and M. Tuexen, “WebRTC data channels,” IETF Secretariat; Working Draft, Internet-Draft draft-ietf-rtcweb-data-channel-13, January 2015.

[4] NEAT Project, “A New, Evolutive API and Transport-Layer Architecture for the Internet,” *Available at https://www.neat-project.org/*, 2017.

[5] “usrsctp - a portable SCTP userland stack,” *Available at https://github.com/sctplab/usrsctp*, 2017.

[6] M. Welzl, F. Niederbacher, and S. Gjessing, “Beneficial Transparent Deployment of SCTP: the Missing Pieces,” *IEEE Globecom 2011 proceedings*, 2011.

[7] “libuv — Cross-platform Asynchronous I/O.” [Online]. Available: https://libuv.org/.

[8] N. Khademi *et al.*, “NEAT: A Platform- and Protocol-Independent Internet Transport API,” *IEEE Communications Magazine*, 2017.

[9] D. Wing and A. Yourtchenko, “Happy Eyeballs: Success with Dual-Stack Hosts.” Internet Engineering Task Force; RFC 6555 (Proposed Standard); IETF, Apr-2012.

[10] R. Stewart, M. Tuexen, and P. Lei, “Stream Control Transmission Protocol (SCTP) Stream Reconfiguration.” Internet Engineering Task Force; RFC 6525 (Proposed Standard); IETF, Feb-2012.

[11] R. Stewart, M. Tuexen, S. Loreto, and R. Seggelmann, “Stream schedulers and user message interleaving for the stream control transmission protocol,” IETF Secretariat; Working Draft, Internet-Draft draft-ietf-tsvwg-sctp-ndata-08, October 2016.

[12] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, “TCP Fast Open.” Internet Engineering Task Force; RFC 7413 (Experimental); IETF, Dec-2014.

[13] “The dummynet project,” *Available at http://info.iet.unipi.it/~luigi/dummynet/*, 2017.

Disclaimer

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

Copyrights for components of this work owned by others than IFIP must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

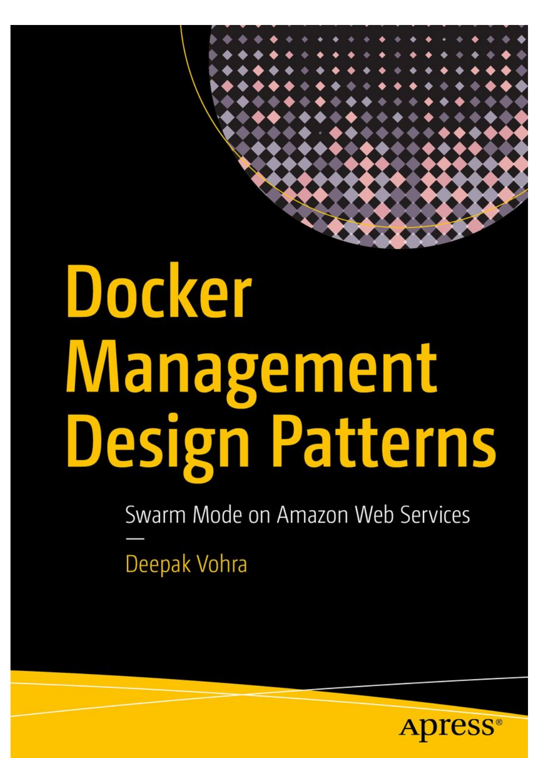
16th International IFIP TC6 Networking Conference, Networking 2017, Stockholm, June 12-15, 2017
ISBN 978-3-901882-94-4 © 2017 IFIP

About The Author

Felix Weinrank is a computer scientist from Germany. He is currently a Ph.D student in the Department of Electrical Engineering and Computer Science at Münster University of Applied Sciences. His research interests include the SCTP transport protocol, low-latency Internet communication and network emulation.

apress.com

Apress®



1st ed., XVIII, 320 p. 239 illus., 210 illus. in color.

A product of Apress

Softcover

- **36,99 € | £27.99 | $39.99**
- ***39,58 € (D) | 40,69 € (A) | CHF 41.00**

Available from your library or

- **springer.com/shop**

Printed eBook for just

- **€ | $ 24.99**
- **springer.com/mycopy**

D. **Vohra**

Docker Management Design Patterns

Swarm Mode on Amazon Web Services

- **Master Docker management design patterns and how to use them to develop applications**
- **Utilize Swarm Mode features to manage a cluster of containers distributed across multiple machines**
- **Learn to use Docker for AWS managed services**

Master every aspect of orchestrating/managing Docker including creating a Swarm, creating services, using mounts, scheduling, scaling, resource management, rolling updates, load balancing, high availability, logging and monitoring, using multiple zones, and networking. This book also discusses the managed services for Docker Swarm: Docker for AWS and Docker Cloud Swarm mode.

Docker Management Design Patterns explains how to use Docker Swarm mode with Docker Engine to create a distributed Docker container cluster and how to scale a cluster of containers, schedule containers on specific nodes, and mount a volume.

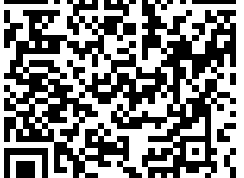
You will learn to provision a Swarm on production-ready AWS EC2 nodes, and to link Docker Cloud to Docker for AWS to provision a new Swarm or connect to an existing Swarm. Finally, you will learn to deploy a Docker Stack on Docker Swarm with Docker Compose.

You will:

- Appl

y Docker management design patterns

- Use Docker Swarm mode and other new features introduced in Docker 1.12 and 1.13
- Create and scale a Docker service
- Use mounts including volumes
- Configure scheduling, load balancing, high availability, logging and monitoring, rolling updates, resource management, and networking
- Use Docker for AWS managed services including a multi-zone Swarm
- Build Docker Cloud managed services in Swarm mode



Order online at springer.com ► or for the Americas call (toll free) 1-800-SPRINGER ► or email us at: customerservice@springer.com. ► For outside the Americas call +49 (0) 6221-345-4301 ► or email us at: customerservice@springer.com.

The first € price and the £ and $ price are net prices, subject to local VAT. Prices indicated with * include VAT for books; the €(D) includes 7% for Germany, the €(A) includes 10% for Austria. Prices indicated with ** include VAT for electronic products; 19% for Germany, 20% for Austria. All prices exclusive of carriage charges. Prices and other details are subject to change without notice. All errors and omissions excepted.

Standard Apress Distribution

Advanced Unix Queuing Techniques

Following on from the discussion of some of the types of queuing mechanisms, available on Unix, it may be beneficial to examine the design of a queuing system, loosely modeled on that used by IBM, in its WebSphere MQ Series.

The MQ paradigm is based on the concept of a Queue Manager, which controls a set of local queues, which are grouped into Channels, which latter are of the 'send' or 'receive' variety. These queues are disk based files, each located under a directory which bears the name of the queue. Normally, there will be several inbound and outbound queues combined into a receiving or sending Channel, respectively. Each Channel has associated with it an IP address of the remote machine. In operation, each queue manager performs as a simple TCP/IP server, and listens for connections on its separate port, whose number is around 1414. Connections arrive from other machines and, in the traditional manner, the server forks to service each connection, transferring double-byte data from the socket to the disk. Although this, in essence, appears to be a classic client-server system, the connections are, actually, from server to server, or queue manager to queue manager. Most of the queue manager's time is spent listening for incoming connections, and polling its local outgoing queue directory, waiting for some local application to place data on a queue.

When this happens, the local queue manager forks and makes a connection to the remote machine, whose IP address and port number are available from the channel definition. While the child process is transferring data, the parent continues to check the directory and listen to port 1414.

It may be instructive to follow this sequence through.

- Machines A and B both listen on port 1414.
- Machine A's queue manager is informed that an application has placed a message on the 'send' queue.
- The queue manager forks a child process, which makes a socket connection to machine B, on port 1414.
- Machine B's queue manager accepts the connection, and forks a child, to handle the connection. The child uses port 32768.
- Data is transferred, from A to B.

Note that there is no conflict in terms of the port number since both managers fork child processes, which use dup'd ports.

Unix Shared Memory-Based Queue Functions

There are only four system calls associated with memory-based queuing functions:

msgget() – which creates or identifies queues, but does not get messages.

msgsend() – which sends a message to a queue.

msgrecv() – which fetches messages from a queue.

msgctl() – which either interrogates or deletes the queue.

We create a queue like this:

```
int qd, qe;

    if((qd = msgget(IPC_PRIVATE,
IPC_CREAT|0777)) == -1){

        perror("msgget");

    }
```

The IPC_PRIVATE parameter has the same significance as it creates shared memory segments, and can be any unique identifier contained in an int – or ‘key_t’ since it is defined in the header. The IPC_CREATE | 777 flags are identical to those used with the open() system call for creating files, and have the same meaning. The return value, qd, looks and works like a file descriptor. It is the queue identifier which we will use when accessing this queue. Now that we have an identifier, we can read from and write to the queue but first, we need to define a queue element structure.

As with other Unix queues, this structure has to look like this:

```
struct {
    long typ;
    char txt[length];
} msg;
```

Meaning that, when it is lying about in the memory, or in flight down a TCP/IP connection, it looks like a packet with a four-byte header and a ‘length’ byte payload.

More importantly, the queue functions expect to read a piece of contiguous memory, which conforms to this definition.

The significance of this is that, we cannot dynamically allocate the memory for the ‘txt’ member since this would make our message look like an int, followed by a pointer. At best, we would enqueue or dequeue 8 bytes and at worst, we would get a segmentation error.

As usual, there is a work-around for this apparent limitation.

Basically, we don’t define a structure at all, but concentrate on the packet concept.

If we declare a pointer to unsigned char,

```
unsigned char *xsg;
```

then, we can send any type of data, and of any length. The only thing we need to remember is always to allocate 4 bytes more memory than we need, and to commence writing our four bytes data past the starting address of our allocated memory. By way of an example, let us assume that we wish to enqueue an array of double-byte characters, which look like int’s on a Solaris system and shorts on most other Unix systems.

```
wchar_t  wchars[255];
```

For convenience, we would rather declare a dummy pointer to an int so that we can fool the system into letting us add our ‘type’ member to the first 4 bytes of our unsigned char array. The ‘type’ member contains an arbitrary, user-defined integer, which we can use later to identify our message in the queue.

```
Int *xint;
```

Now, we can send the array to a queue. However, we first need to allocate memory:

```
  if((xsg = (char *)malloc(sizeof(wchars)+4))
== NULL){
        printf(“Memory allocation error\n”);
    }
```

Take note of how we ask for an extra 4 bytes to cater for our ‘type’ member.

Next, we set our pointer to the first location in the array:

```
  xint = (int *)&xsg[0];
```

Next, we can insert our integer:

```
  xint[0] = 9;
```

and load the newly-acquired memory with our wide-character data:

```
memcpy((char *)&xsg[4], (char
*)wchars), sizeof(wchars));
```

Finally, we send it to the message queue:

```
  if(msgsnd(qd, (void *)xsg, sizeof(numbers),
0) == -1){
        perror(“mq_send”);
    }
```

The syntax for msgsnd() is fairly obvious, and follows the pattern of write() to a file, or that of send() to a socket. The first argument is our queue descriptor, the second is a pointer to the data, and the third is its length. Finally, we have the flag which defines what to do with the message if the queue either contains the maximum number of bytes, or the maximum number of messages:

- If the flag is IPC_NOWAIT, the call returns immediately and the message is not sent.

- If the flag is zero, msgsnd() will hang until the queue is no longer full, or the queue gets deleted, or the current process catches an interrupt.

For our purposes, a flag of zero makes for more reliable delivery. So, that is what we use.

Now for the receiving function, msgrcv().

The syntax is, as with msgsnd(), similar to the read() and recv() system calls.

Given a receive buffer definition similar to

unsigned char data[8192];

and type and flag parameters defined as

Int, typ, flg;

We can see how familiar the code is:

```
   If(msgrcv(qd, (void *)&data, sizeof(data), typ, flg)) ==  -1){

        perror("msgrcv");

    }
```

The msgctl() system call returns information about the queue.

The syntax is:

```
msgctl(queue, flag, structure_pointer);
```

where, the flag is either IPC_STAT, for retrieving data, or IPC_SET, for altering queue characteristics.

In practical terms, the only items we can alter are the permissions on the queue, unless we run as root, when we can change the maximum queue size.

Accordingly, information retrieval is this function's greatest value. The structure_pointer mentioned above points to the msqid_ds structure, defined in sys/msg.h as:

```
struct msqid_ds {

        struct ipc_perm msg_perm;           /*
operation permission struct */

        struct msg       *msg_first;
/* ptr to first message on q */

        struct msg       *msg_last;
/* ptr to last message on q */

        msglen_t          msg_cbytes;
/* current # bytes on q */

        msgqnum_t      msg_qnum;           /*
# of messages on q */

        msglen_t          msg_qbytes;
/* max # of bytes on q */

        pid_t               msg_lspid;
/* pid of last msgsnd */

        pid_t               msg_lrpid;
/* pid of last msgrcv */

        time_t             msg_stime;
/* last msgsnd time */

        time_t             msg_rtime;
/* last msgrcv time */

        time_t             msg_ctime;
/* last change time */

        short               msg_cv;
/* not used */

        short               msg_qnum_cv;
/* not used */

};
```

whose members are filled in by the system call.

Putting it all together gives us a typical call as:

```
   struct msqid_ds atr;

    if(msgctl(qd, IPC_STAT, &atr) == -1){

        perror("msgctl STAT server");

    }
```

from which, we can get useful information, such as:

```
printf("Current queue length: %d\n", atr.msg_qnum);
```

Implementing Some MQ Functionality

The MQ model is very efficient in terms of use of machine resources. The functionality is spread over many processes, and is a good example of a

Multi-Dimensional Architecture. However, since MQ needs to use persistent queues, it throws away all of its performance advantages, by doing very heavy disk I/O. This is largely irrelevant for short messages but, if it is used for performing large data extracts, from large databases, the delays become significant. For the sake of this exercise, we will assume that, the speed of operation is of paramount importance. However, the normal reliability of delivery is adequate and a machine crash is tolerable. Accordingly, we will design and partially code a simple queue manager which implements most of the functionality of the MQ manager, but uses kernel queues. The implementation of any refinements is left, as they say, to the reader.

The Queue Manager

First, let us remind ourselves of the two prime functions of our queue manager:

- Listen for incoming connections
- Check local queues for data needing transmission to remote sites

There are two possible solutions to this apparent dichotomy

- Run two processes, one possibly being the child of the other.
- Run one process, within which we have two threads of execution.

Since MQ series was born at a time (and on a machine) where pthreads weren't even a glimmer on the horizon, IBM uses multiple processes.

From a performance perspective, this would represent the best practice for us, too. But first, let us specify, in detail, what happens.

If our server is sitting there, with its ear glued to port 1415, then how is it going to transmit data through that port? The answer is that it isn't.

What happens is that, whatever checks the local queues and decides to send the messages to the far corners of the earth, will make a connection to port 1414, in the same way as the server on the far side of the moon, with a message for us. This means that our server only has to differentiate between an inbound and an outbound connection and take the appropriate action. Once the connection is established, and the server has forked a child, the child will have its brand-new socket, a port number, and can send or receive data at will. The sequence of events in our server is now looks like this:

- Listen to port 1414
- If the incoming connection is local, it is outbound.
- Fork process for reading messages from queues
- Inside this process, send outbound queue data to remote address
- Child terminates
- If the incoming connection is remote, it is inbound data.
- Fork process for enqueuing messages
- Inside this process, enqueue incoming messages
- Child terminates.

Now, we nearly have a complete picture. However, we are pretending that we only have one queue manager and a pair of queues at each end. In reality, we could have many, going to many destinations. Also, following the MQ policy of 'guaranteed delivery', we would need to re-send any messages which failed on the first attempt.

What about the User Application?

The whole purpose of a queuing system is to make data flow from one place to another. So, how does the user application fit in? In an MQ environment, the application is always written such that it;

- Calls CONNECT to connect to a queue manager.
- Makes a request to GET inbound messages or PUT outbound messages
- Does its thing.
- Quits.

Note that, given the above scenario, it is evident that the application doesn't need to directly execute the queuing system calls. The CONNECT() function, from the MQ library, hides the mechanics of the TCP/IP connection to the queue manager, and the GET() or PUT() function calls merely pass the data down the socket connection, for the queue manager to enqueue or dequeue.

The Channel Manager

Everything seems to be automatic. Do we need a Channel Manager?

It is the responsibility of the application to remove inbound messages, and the responsibility of the queue manager to forward outbound messages. This means that, for inbound messages:

- Server handles connection from remote site.
- Server enqueues inbound message.
- Server waits for the application to connect with a GET request.

For outbound messages:

- Server handles connection from application, with a PUT request.
- Server enqueues message.
- Server forwards the message to the remote site.

It may be seen that, inbound messages are event driven and need no external agent. Outbound messages are enqueued, and then forwarded immediately by the child process which enqueued them. Accordingly, unless there is a communications failure, we may assume that the queue manager server will itself, immediately, empty the outbound queue. It is this latter condition which necessitates the use of a channel manager.

Each channel manager will be associated with a queue manager, and both will be handling traffic to and from a given destination, (the 'channel'). When the channel manager sees a message count of some number more than zero, it connects to its appropriate queue manager, on the appropriate port number, to tell it that the queue needs servicing, and the IP address where the data should be sent.

Its operation would follow this pattern:

- Check the length of the outbound queue.
- If it is greater than zero, we need to send a message.
- Connect to port 1414.
- Give the queue manager server details of which queue, and where to send it.
- Go to sleep for a predetermined period.

The Protocol

We will also need to design a primitive protocol, contained within the message header, so that the server can differentiate between an inbound and an outbound connection and for safety, can differentiate them from a spurious connection. Within the protocol, there has to be a slot for identifying the queue and the destination, if appropriate.

While keeping within the constraints of the basic queue structure, we will add some fields to it to make it better suited for our purpose.

We need the following information:

- Type of operation – 'GET' or 'PUT', symbolized by '1' or '0', as 1 character.
- Name of queue – 23 characters, although MQ permits 40.
- Length of following message + length of an int.
- Type – an integer, as defined in msg.h
- Data

Which gives us a 32-byte header, which fits on either a 4-byte or 8-byte boundary, such that TCP/IP will not perform any padding.

We can deduce from all of the aforegoing that, multi-threading is not going to be of any use to us. We will have two families of processes, associated with the queue manager and channel manger, and it

would be inefficient to attempt to run them as two threads. Readers who disagree are invited to code a threaded version and compare response times.

Server Code

Since this is a test program, we will begin it by creating one outbound and one inbound queue. Ordinarily, these will have been created earlier, and their descriptors saved, for passing to the queue manager. We will need the following definitions:

```
struct xqdata{
    long typ;          /* 0=1st on queue, n=1st of type n, -n=1st of <n */
    char msg[8192];
};
struct xqdata qdata;                /* data from msg queues */


struct xmsg {
    char mode;              /* 1=GET 0=PUT */
    char qname[23];
    int length;
    struct xqdata mesg;
};
struct xmsg *msg;


int qd, qe;            /* outbound, inbound queues */
int *xint;
char *xsg;
main()
{
/* and we can now create our queues: */
/* outbound queue */
    if((qd = msgget(IPC_PRIVATE, IPC_CREAT|0777)) == -1){
        perror("msgget");
    }
/* inbound queue */
    if((qe = msgget(IPC_PRIVATE, IPC_CREAT|0777)) == -1){
        perror("msgget");
    }
/* now the TCP/IP furniture */
    memset(&sa, 0, sizeof(struct sockaddr_in));
    if(gethostname(hname, sizeof(hname)) != 0){
        printf("Can't determine our own host name. Quitting\n");
        quit(-1);
    }
    if((hp = gethostbyname(hname)) == NULL){
        return(-1);
    }
    if((svc = getservbyname("ingreslock", "tcp")) == NULL){
        printf("%s doesn't exist\n");
        quit(-1);
    } else {
        portnum = svc->s_port;
    }
    printf("Server on machine %s, listening to port %d\n", hname, portnum);
    sa.sin_family = hp->h_addrtype;
    sa.sin_port = htons(portnum);
    if((s = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        quit(-1);
    }
if(bind(s, (struct sockaddr *)&sa, sizeof(struct sockaddr_in)) < 0){
        printf("Can't bind to port %d\n", portnum);
        perror("bind");
        close(s);
        quit(-1);
    }
    listen(s, 20);
    while(1){
        inlength = sizeof(sa);
        if((xs = accept(s, (struct sockaddr *)&sa, &inlength)) < 0){
            break;
        } else {
```

```
            p = (char
*)inet_ntoa(sa.sin_addr);

            printf("\nIncoming connection from
>%s<\n", p);

            fflush(stdout);


            switch((ppid = fork())){
/* make our child process */

                case -1:

                    perror("fork");

                    exit(-1);

                break;

                case 0:

                    if(setsid() == -1){

                        perror("setsid");

                    }

                    i = 0;

                    memset(buf, '\0',
sizeof(buf));

                    while((rval = recv(ds,
buf, sizeof(buf), 0)) > 0){

                        printf("Server read
(%d):>%s<\n", rval, buf);

                        /*

                         * Concatenate until
we have it all

                         * data[] should be
dynamically allocated!

                         */

                        memcpy(&data[i], buf,
rval);

                        i += rval;

                        if(i >= 8192) break;

                        rval = 0;

                        memset(buf, '\0',
sizeof(buf));

                    }

                    /* we only do 'GET' and
'PUT' */

                    if(data[0] == '0' ||
data[0] == '1'){

                        if(readtoken(ds, data)
== 0){

                            printf("Server
%d:Done\n", getpid());

                            if(close(ds) !=
0){

                                perror("Server
close socket error");

                            }

                            exit(0); /* child
can quit now */

                        } else {

                            printf("Server
%d:Finished with errors\n",

getpid());

                            if(close(ds) !=
0){

                                perror("Server
close socket error");

                            }

                            exit(-1);

                        }

                    } else {

                        printf("Server
%d:Received corrupt message\n",

getpid());

                    }


                break;

                default:

                    printf("Server
%d:Listening for next connection\n",

getpid());

                    waitpid(ppid, &status, 0);

                break;

            }




        }

    }

    /* we've finished with the queues, let's
delete them */

    if(msgctl(qd, IPC_RMID, NULL) == -1){

        perror("msgctl RMID");

    }

    if(msgctl(qe, IPC_RMID, NULL) == -1){

        perror("msgctl RMID");
```

```
    }
    printf("Server done\n");

}
/* main */
```

Additionally, we're going to need some other functions, to do the housekeeping for us.

First, the message parser:

/***
****

 * Parse the message header, and extract the message. The raw format is:

GET or PUT	Queue name	Length of msg + type	type	msg.....
1 byte	23 bytes	4 bytes	4 bytes	| 8192 bytes...|
0	1 23	24 27	28 31	| 32 .....

 * Which is defined as:

 * struct xmsg {

 * char mode; -> '1' = GET, '0' = 'PUT'

 * char qname[23];

 * int length;

 * unsigned char mesg[8192];

 * };

 *

 * Note that the 'mesg' member is what we actually need to enqueue the msg,

 * and is in the format:

 *

```
 * struct {
 *    int type;
 *    char txt[8188];
 * } qdata;
 *
*******************************************************************
*****************************/
```

```
readtoken(s, token)                    /*
readtoken */

int s;
unsigned char *token;

{

pid_t pid;
unsigned int priority;
int rval;
int nwrite;
char name[25];
char mode;
int length;
int i;

    printf("\nChild server PID %d running\n",
getpid());

    msg = (struct xmsg *)token;

    strcpy(name, msg->qname);
    mode = msg->mode;

    length &= 0x00;              /* make a
number out of chars */
    rval = 0x00 | token[24];
    rval = rval << 24;
    length |= rval;

    rval = 0x00 | token[25];
    rval = rval << 16;
    length |= rval;

    rval = 0x00 | token[26];
    rval = rval << 8;
    length |= rval;
```

```
    length |= token[27];

    printf("Token data: queue >%s< mode >%c<
length %d\n", name, mode, length);

    for(i = 28; i < rval; i++){

        printf("%c", token[i]);

    }

    printf("\n");

/*
```

* This is artificial, since we need to examine the channel

```
    * details, to find out which queue goes
where

     */

    if(mode == '1'){
/* GET (De-queue) */

        if(strcmp(name, "MQ1") == 0){
/* Inbound queue */

            dequeue(qd, s);

        }

        else if(strcmp(name, "MQ2") == 0){
/* Outbound queue */

            printf("Doing GET from outbound
queue\n");

            dequeue(qe, s);

        }

    } else {
/* PUT (Enqueue) */

        if(strcmp(name, "MQ1") == 0){
/* Inbound queue */

            printf("Doing PUT to inbound
queue\n");

            enqueue(qd, length, &token[28]);

        }

        else if(strcmp(name, "MQ2") == 0){
/* Outbound queue */

            enqueue(qe, length, &token[28]);

        }

        /* We now need to forward this message
to the address at the other

         * end of this channel

         */

        forward(token, name);

    }

    return(0);

}                            /* readtoken */
```

Now, here are the enqueuing and dequeuing functions:

```
enqueue(qdd, length, token)              /*
enqueue */

int qdd;

int length;

unsigned char *token;

{

unsigned char *xsg;

struct msqid_ds atr;

    printf("Server enqueuing messages...\n");

    if(msgsnd(qdd, (void *)token, length, 0)
== -1){

        perror("msgsnd");

    }

    memset((char *)&atr, '\0', sizeof(struct
msqid_ds));

    if(msgctl(qdd, IPC_STAT, &atr) == -1){

        perror("msgctl STAT server");

    } else {

        if(atr.msg_qnum > 0){

            printf("Placed %d messages (%d
total bytes) on q %d\n", atr.msg_qnum

, atr.msg_cbytes, qdd);

        }

    }
```

```
}                          /* enqueue */

dequeue(qqd, s)                        /* dequeue
*/

int qqd;

int s;

{

struct xmsg ndata;         /* data from TCP/IP
*/

int i;

int rval;

int nwrite;

long typ = 0;  /* get 1st available msg */

int flg = 0;   /* block until msg arrives */

struct msqid_ds atr;

    printf("Client collecting messages from
queue...\n");

    memset((char *)&qdata, '\0',
sizeof(qdata));

    while((rval = msgrcv(qqd, (void *)&qdata,
sizeof(qdata), typ, flg)) != -1){

        printf("Server read %d bytes of type
%d queue %d data:\n",

rval, qdata.typ, qqd);

        sprintf(ndata.qname, "MQ%d", qqd);

        ndata.length = rval;

        ndata.mode = 0;

        ndata.mesg.typ = 9;

        memcpy(ndata.mesg.msg, qdata.msg,
rval);

        for(i = 0; i < rval; i++){

            printf("%c", qdata.msg[i]);

        }

        printf("\n");

        if(rval > 0){    /* no point writing
nothing */

if((nwrite = send(s, (void *)&ndata, rval, 0))
< rval){

                perror("Server: write to
socket");

                free(xsg);

                return(-1);

            } else {

                printf("Server %d: sent %d
byte msg to client\n",

getpid(), nwrite);

            }

        } else {

            printf("Strange: zero-length
message on queue...\n");

        }

        memset((char *)&qdata, '\0',
sizeof(qdata));

        memset((char *)&ndata, '\0',
sizeof(ndata));

        memset((char *)&atr, '\0',
sizeof(struct msqid_ds));

        if(msgctl(qqd, IPC_STAT, &atr) != -1){

            if(atr.msg_qnum == 0){

                printf("No more messages\n");

                break;

            } else {

                printf("%d messages left (%d
bytes) on q %d\n", atr.msg_qnum, at
r.msg_cbytes, qqd);

            }

        } else {

            perror("msgctl STAT dequeue");

        }

    }

    printf("Server de-queued q %d\n", qqd);

    free(xsg);

}                          /* dequeue */
```

The Client Code

Our queue manager, as coded above, will work stand-alone, and GET and PUT messages to the two queues it created. What it won't do, is to forward the messages to remote sites. This is because, it needs to behave as a client in order to do that.

The following code creates a standalone client, which we can be used in place of the 'user application', and which the reader is encouraged to incorporate, as a set of functions, into the queue manager.

First, we need much the same definitions, as we did with the server:

```
struct sockaddr_in sa;
struct hostent *hp;
struct servent *svc;
int a, s;
char hostname[2048];
unsigned short portnum;
char buf[8192];
unsigned char *token;
int nread;
int nwrite;
char mode;
char queue[23];
char srcfile[255];
struct qdata {
   long typ;          /* 0=1st on queue, n=1st of type n, -n=1st of <n */
    char txt[8192];
};

int qd, qe;             /* forward, reverse queues */

struct xmsg {           /* data from TCP/IP */
    char mode;          /* 1 = GET, 0 = PUT */
    char qname[23];     /* MQ1=inbound MQ2=outbound */
    int length;
    struct qdata mesg;
};

struct xmsg msg;

int *xint;
char *xsg;

main(argc, argv)               /* main */

int argc;
char **argv;

{

unsigned char *p;
int i;
int flag = 0;
int slen = 1000000;      /* socket buffers */

    if(argc < 2){
        printf("Usage: mqclient <host> <queue> <1=GET|0=PUT> [file]\n");
        exit(-1);
    } else {
        strcpy(hostname, argv[1]);
        strcpy(queue, argv[2]);
        mode = argv[3][0];
        if(argc == 5){
            if(mode == '0'){
                strcpy(srcfile, argv[4]);
            } else {
                printf("Wrong mode >%s<\n", argv[3]);
                exit(-1);
            }
        }
    }
```

```
    printf("Client connecting to host %s\n",
hostname);

    if((hp = gethostbyname(hostname)) ==
NULL){

        perror("gethostbyname");

        exit(-1);

    }

    if((svc = getservbyname("ingreslock",
"tcp")) == NULL){

        printf("%s doesn't exist\n");

        exit(-1);

    } else {

        portnum = svc->s_port;

    }

memset(&sa, '\0', sizeof(sa));

    memcpy((char *)&sa.sin_addr, hp->h_addr,
hp->h_length); /* set address */

    sa.sin_family = hp->h_addrtype;

    sa.sin_port = htons((u_short)portnum);

    if((s = socket(hp->h_addrtype,
SOCK_STREAM, 0)) < 0){

        perror("socket");

        exit(-1);

    }

    if(setsockopt(s, SOL_SOCKET, SO_RCVBUF,
(void *)&slen, sizeof(slen)) < 0){

        perror("Client setsockopt");

    }

    if(setsockopt(s, SOL_SOCKET, SO_SNDBUF,
(void *)&slen, sizeof(slen)) < 0){

        perror("Client setsockopt");

    }

    if(connect(s, (struct sockaddr *)&sa,
sizeof(sa)) < 0){

        printf("Unable to connect to %d\n",
portnum);

        perror("Connect");

        close(s);

        exit(-1);

    }

    /* assemble the token */

    strcpy(msg.qname, queue);

    msg.mode = mode;

    msg.length = 0;                /* this
gets set in readmsg() */

    msg.mesg.typ = 9;             /* arbitrary
identification number */

    if(mode == '0'){               /* get that
which we wish to PUT */

        readmsg(&msg, srcfile);

    }

    token = (unsigned char *)&msg;

if((nwrite = send(s, token, sizeof(struct
xmsg), 0)) < sizeof(struct xmsg)){

        perror("Write to socket");

        exit(-1);

    } else {               /* the write
succeeded */

        printf("Client sent %d byte
token:>%s<\n", nwrite, token);

        if(mode == '1'){
/* we asked to read msgs */

            printf("Client waiting to read
queue...\n");

            memset(buf, '\0', sizeof(buf));

            flag = 0;

            while((nread = recv(s, buf,
sizeof(buf), 0)) > 0){

                printf("\nRead %d byte
message:\n", nread);

                for(i = 0; i < nread; i++){

                    printf("%c", buf[i]);

                    if(buf[i] == EOF &&
buf[i+1] == '\0') flag = 1;

                }

                if(flag == 1) break;

                memset(buf, '\0',
sizeof(buf));

            }

            printf("\nReceived messages\n");
```

```
        } else {
/* no reply needed */

            /* do nothing */

        }

    }

    close(s);

}                                 /* main */
```

Now, we need to have our application read in arbitrary data (like double-byte XML), and set it into our data structure for transmission. The following trivial function accomplishes this:

```
/**********************************************
******************************

 * Reads a double-byte XML message from a
file, and appends it to the

 * token array.

**********************************************
*****************************/

readmsg(where, file)                      /*
readmsg */

struct xmsg *where;

char *file;

{


int fd;

int i;


    if((fd = openfile,, O_RDONLY)) == -1){

        printf("Can't open %s\n", file);

        return(-1);

    }

    if((nread = read(fd, buf, sizeof(buf))) >
0){


        /* we assume the message is never
bigger than the buffer */

        where->length = nread;


        printf("Read %d byte message:\n",
where->length);

        memcpy(where->mesg.txt, buf, nread);


        for(i = 0; i < nread; i++){

            printf("%c", buf[i]);

        }

        printf("\n");

    } else {

        perror("Read XML");

        return(-1);

    }

    close(fd);


}                               /* readmsg */
```

About The Author

Mark Sitkowski is a Chartered Engineer and a Corporate Member of the Institution of Electrical Engineers in London. His early career revolved around the writing of analog and digital circuit simulators, and digital signal processing applications. In Australia, he moved to writing financial software for the major banks, and telecommunications software for Telcos, besides conducting training courses on Unix and database applications. Formerly a consultant to Forticode Security, he currently works with Design Simulation Systems on mobile multi-factor authentication systems. Design Simulation Systems Ltd http://www.designsim.com.auxmarks@exemail.com.au

USING FREEBSD AS A FILE SERVER WITH ZFS

Want to know how to get started and configure a working *home server?*

Do not wait for a better moment!

Learn TODAY how to use the current ZFS capabilities to help us build a home file server using FREEBSD 10.3

https://bsdmag.org/course/using-freebsd-as-a-file-server-with-zfs-2/

OpenBSD From a Veteran Linux User Perspective

Dear BSD Readers,

I'm a Computer Engineer and entrepreneur. I've loved computers since I was a kid. My first machine was a 386 with 2 MB of RAM which ran DOS, and since then, I've used about every possible machine and OS that's available. I studied Computer Engineering in Barcelona and then I worked for 8 years analyzing bioinformatics data using machine learning. After leaving that job, I then had some sabbatical time, during which I started some projects and learned to use new tools, languages, and systems. My dream has always been to find an AI startup. While in college, 15 years ago, this was an almost foolish idea. However, thanks to the proliferation of tools and libraries, the commoditization of GPUs, and the focus on AI verticals, the AI field is growing strongly, and there are many market opportunities to apply machine learning to the enterprise. I'm now a founder at Optimus Price, an AI-powered price recommender for e-commerce: https://optimusprice.ai. You can learn more about me at my page: https://cfenollosa.com

Interview with Carlos Fenollosa

Can you tell our readers about yourself and your role nowadays?

I'm a founder at Optimus Price, a startup that develops a SaaS for dynamic price recommendations powered by Artificial Intelligence. I am the CEO which means that, as an engineer, I've had to learn many new skills: business development, sales, management, administration and finances. It's a whole new world, and a very interesting one; businesses are a big pillar of today's world, and I believe that a deep understanding of how they work complements well the technical world and lets you grow as a person.

How you first got involved with programming and the OpenBSD world?

I first used a computer around 1992. It was an obscure brand 386 laptop with 2 MB of RAM that I used to "borrow" from my father when he wasn't looking. My main stack was DR-DOS and later Windows 3.1. We had to use what our parents bought, usually cheap software and games, or software copied from friends. Obviously, we didn't have any Internet, so the only way to learn was by reading library books and computer magazines. I typed numerous QBasic listings by hand and ultimately learned how to write my programs. Now we jump forward in time for more than 20 years. After quitting Windows and using Linux and OSX exclusively for some time, I wanted to learn more about UNIX systems. I experimented with FreeBSD and OpenBSD, and liked both, but I really loved the simplicity of OpenBSD. I found that it had more differences from its cousin Linux than I expected, and that's why I wrote the article "OpenBSD from a veteran Linux user perspective": https://cfenollosa.com/blog/openbsd-from-a-veteran-linux-user-perspective.html

While having a wide field of expertise, please tell our readers on which area you put the most emphasis and why?

I am probably best at designing and developing software products. Solving problems with software is my passion and what I do best, from small scripts to bigger systems.

What was your the best work? Can you tell what was the idea behind it? What was its purpose?

Maybe not my best, but I developed Bashblog, a blogging engine in a 500 line bash script to scratch my own itch. I wanted to write "./bb.sh post", write some Markdown text, and get it converted to a single entry linked from an index page. I published it on Github (https://github.com/cfenollosa/bashblog), and it quickly grew in popularity. Now, it's mostly finished and it has a great community around it. It is definitely my most successful free software project.

What is the most interesting issue you've encountered, and why?

While working at the Barcelona Supercomputing Center, we had all office machines on an SGI cluster to launch long jobs overnight. However, one very specific job failed randomly. Long story short, after about a week of debugging, we discovered that some of the machines had a CPU which didn't support a very specific CPU opcode. Even though the software was a Python script, one of the libraries (numpy) had been compiled on a machine with that opcode, so it segfaulted on older CPUs.

What was the most difficult and challenging event in your life? Could you give us some details?

Probably having to sell my product to people. As a product person and an engineer, I was terrified the first times I met with clients and tried to sell them my product. It turns out that selling is another skill you can learn, and clients appreciate meeting with an engineer and not a salesperson, it gives you more credibility.

What future do you see for FreeBSD and other OSes? Can you tell us about your favorite features in the new releases.

I think the future will bring a clear divide between five product families: laptops/desktops, servers, mobile, and IoT. The BSDs have a fairly good market share on the server, but they will probably remain a niche in laptops and desktops as they are today. I don't think they will lose market share, though. The big question is the future of mobile and IoT. Mobile is now dominated by Android and iOS, and having been a developer of both, I can see Android evolving heavily or being replaced by something else in the future. I'm not sure if it will be another iteration of Linux or a BSD, but its current architecture and extreme fragmentation both by device and manufacturer has many problems. Regarding IoT, we all have seen what happens when you deploy tiny CPUs with nonexistent security and flawed OSs and services. This is a clear opportunity for the BSDs. The embedded world will evolve to more powerful and power-efficient chips, and an embedded BSD kernel + userland would be very attractive.

Do you have any specific goals for the rest of this year?

I want my company to grow and be able to deliver a great product. If only I could find some time to improve on my github projects...

What's the best advice you can give to the BSD magazine readers?

The BSD magazine audience is probably much more skillful than me in many respects, so maybe I can give some advice regarding product development. It is definitely true that it takes 80% of the development time to finish up the last 20% features to launch a product. However, that is also the time you need to show your still unreleased prototype to your target market, be it a company or just some fellow developers. Otherwise, you will waste your time developing a product that only works for you. This is not wrong per se, but you will miss out on the enjoyment of watching other people use the product you've worked so hard on.

Thank you

Thanks for reading!

OpenBSD From a Veteran Linux User Perspective

For the first time, I installed a BSD box on a machine I control. The experience has been eye-opening, especially since I consider myself an "old-school" Linux admin, and I've felt out of place with the latest changes on system administration. Linux is now easier to use than ever, but the administration has become more difficult. There are many components, most of which are interconnected in modern ways. I'm not against progress, but I needed a bit of recycling. So instead of adapting myself to the new tools, I thought, why not look for modern tools which behave like old ones? This article discusses some of the main differences between OpenBSD and Linux, from a Linux admin perspective. There are some texts on the net discussing the philosophical differences between BSDs and Linux, but not many of them are really hands-on. [This one] is the best, and I recommend you to read it along with this one. Since I am new to OpenBSD, I may get some things wrong. Please email me any corrections. However, my goal is to point out my first impressions. Therefore, if there are any Linux users reading and thinking about making the jump, they can know what to expect.

The "RAMDAC" running joke

First, some background about my Linux experience.

My first computer was a 386 with DOS and Windows 3.1. I had played with Spectrums, Commodores, and IBM PCs (8086). I followed the traditional Windows path: 3.1 -> 95 -> 98 -> ME -> 98 -> 2000. But I always liked computers, and the most visible part of them, besides the hardware, is the OS. I tried to install my first Linux distro on 1999. It was a Red Hat Linux 5.2, if I remember correctly, I got the CD from a magazine because I was still running a dial-up. I was 15 and I thought I knew computers, after all, I had assembled my own, an AMD K6-2 box, from parts.

Red Hat proved me wrong.

Which is your chipset?

Man, I didn't know

Which is the model of your RAMDAC?

What is a RAMDAC?

I need [your monitor modelines]. Don't get them wrong or you will physically damage your CRT

Dude, I'm 15, I can't afford to break anything!

In the end, I didn't break my monitor, but got a black screen which said login:, and didn't know what to do, so I booted back into Windows and played a bit of Warcraft 2.

In that age, we only had one computer. So if you were installing something and needed help, you had to stop, reboot into Windows, dial up the modem, search the Usenet or forums, write down the solution on a piece of paper—no ubiquitous printers—, hope you got the commands right, reboot, start the installation over, reach the point where you previously were, and apply that solution. Not practical at all.

The best help we had were books, and those were expensive and difficult to find in a small town bookstore. For those of us not fortunate enough to buy/find books, we had hobbyist magazines. In Spain, there were a few imported and poorly translated magazines which were expensive, but carried some CDs, the only practical way to get distros.

The first Linux I was able to use was Mandrake 6.0. It had a graphical installer—not that having graphics made any real difference to the final result— but it auto-detected my hardware correctly and booted into X. Yeah! [Old Linux software]! A game called Nethack which had nothing to do with hacking! sysconfig!

Unfortunately, I couldn't connect to the internet because of my Winmodem. Thus, after a few days of tinkering, Mandrake was wiped too.

Months later, I got myself a BeOS CD. It was like Linux, which for me, then, meant it was not Windows. The setup ran totally effortless and it even detected my Winmodem. The internet ran faster than

on Windows. It had a great internet browser, mail and newsgroup clients. Oh, boy! I used BeOS for a long time almost exclusively and only booted Windows to play some games.

A couple of years later, I started Computer Engineering in college. So, I wiped out everything and installed Linux. I got a new machine and a real network connection.

I've run lots of Debians, Red Hats, Mandrakes, Gentoos and Slackwares. We used Solaris and even some VAXes. I ran some servers for student organizations, and finally settled on Debian as "the best" distro: stable, easy to use, no need to compile on our 486, nice hardware detection and with a big community. Finally, I moved to Ubuntu, only because of its LTS releases. Around 2006, I got into Macs, which at first seemed like a nicer Linux, and now I appreciate the hardware+software combo for which I know I won't have to fight with its drivers.

In summary, I've seen a lot on UNIX, even more on Linux, and administered a good chunk of them. My servers have always run some sort of Debian. You could say that as I grew older, I also grew tired of fighting with RAMDACs, modelines and Winmodems. Each age brought new "RAMDACs": CD recording, wireless card support, laptop hibernation, webcams, Divx playing, DVD playing, NTFS support etc. Linux always worked on the server but had some quirks in the desktop which made it somewhat unattractive for daily use, even when I run it exclusively on my laptop. Nowadays, Macs offer a UNIX with some peace of mind, and the current status of Linux is good enough. Some of the friends I evangelized long ago—I quit doing that—still use Ubuntu and are happy with it. Linux may never triumph on the desktop (or laptop), but it's good enough for most.

Upgrading a G4 Mac Mini

Now jump to 2015. My home server, a G4 Mac Mini, was already two Debians behind. Some packages weren't ported to *powerpc*. I needed to perform a clean install and upgrade the whole system either way. But this time, I didn't want to use a Linux installation which wants me to reboot every 5 days because of some critical patch. I'm looking at you, Ubuntu.

As you can imagine, my operating system fascination didn't fade out, only my time. I had been closely following the BSDs and using a NetBSD shell account, installed Plan 9 on a virtual machine, and even wrote a toy OS project.

I'm not afraid of compiling stuff—I do it for a living—and may even be open to modifying some code if needed. Why not try something new? Since I had the weird *powerpc* requirement, I ruled out most operating systems. Finally, I decided to play relatively safe and go for a BSD. FreeBSD is the most popular, has more online HOWTOs, and probably more features (ZFS, Jails etc.), though I probably would not be using them. OpenBSD is more hackable, seems to have better documentation, and some cool people I know who use it. I didn't want to quit using a pot to start using a kettle, so I downloaded OpenBSD's `install57.iso` It was impossible to boot the Mini with a USB stick; I'm unsure if it's the firmware's fault or the fact that I was adding the .iso file into the USB instead of the .fs one which didn't seem to be available for macppc.

I found some blank DVDs on a closet, borrowed a computer with a DVD drive—another medium I hadn't touched in years—, and burned the ISO image. The fact that I recorded the first disk with the ISO file on the root folder instead of properly burning the contents into the DVD warned me that this was going to be hard, but fun. Surprisingly, the installation was straightforward. It detected the 10-yr-old hardware, and by following the instructions, I managed to partition the disks and install the boot loader. Eventually, the box was up and running. Well, that wasn't so hard, was it? Now, to restore my old installation. Hm, first of all, `bash` needs to be installed from packages and goes into `/usr/local/bin`. Therefore, I had to modify a lot of scripts which pointed to `#!/usr/bin/env bash`. The `ps` and `tar` commands have slightly different switches which broke other scripts. The base services are different; OpenBSD includes its own HTTP, SMTP and NTP servers. Configuration files are in different places. Here goes my week...

GNU is really not UNIX

A quick note on the GNU/Linux naming discussion, since GNU is entering the equation now. I use the term "Linux" for simplicity. I know that's the kernel name. It also happens to be the popular name, even if not totally correct—according to some. Here is some food for thought; why does the FSF deserve more credit on the name than, say, the Apache Foundation, or the FreeDesktop project, or BSD, for that matter? Why don't we include every key component of the name and call it GNU/FreeDesktop/Apache/OSI/BSD/.../Linux? Including only GNU would be unfair to other big contributors, wouldn't it? So, let's please stop this fight. That being said, the GNU tools and design philosophy make a noticeable difference in administration and userspace, and one can only appreciate it when switching to a different environment. I don't want to overstate it, though. Thanks to POSIX, a Linux admin can run BSD with little extra effort since most of the things are similar. There are, in fact, more similarities than differences. If FreeBSD and OpenBSD are brothers, then Linux is a close cousin. ls is always ls. mkdir is mkdir. But when you're being used to /dev/hda, free -m and cat /proc/cpuinfo you realize that having a different kernel is naturally going to change some of the administration tools. Some say that the GNU tools are bloated and that the BSD toolchain is more "pure UNIX". The reality is that it depends on the specific GNU tool. I've personally found that GNU tools are more complex because they're more powerful, though they are less UNIX-like *(do one thing only and do it well)* and more like complete solutions. That's fine; different, but fine. After all, GNU is not Unix! In recent years, the Linux environment has grown in the GNU toolchain fashion, not the UNIX fashion. One may even say it has grown in the *Windows fashion*: be practical, be accommodating to all, be fast, and be modern. There have always been debates about "bloated and complex code". More recently, systemd. Previously, Apache, sysconfig, iptables/iptools.... The list goes on and on. Wheel out comp.os.linux and look at the flame wars. No software fits all nor should be shamed for its design decisions. In the end, with a few critical exceptions like OpenSSL and the Heartbleed bug, it is just a matter of taste: does the admin prefer simple, pluggable services, or bit monolithic suites? Compatibility or modernness? Familiarity or shiny new things? Standards or NIH? I had been riding the Linux wave for years, until I recently realized that my admin skills needed a total recycling. In a few years, we've gone from /etc/init.d/sshd restart to service sshd restart to systemctl start sshd. That's a bit fast in my opinion. However, I understand it's the price of progress aimed to make computers boot faster and theoretically easier to administer for newcomers. Old admins, on the other hand, have a harder time adapting. Having to choose between recycling into an always changing Linux or a more stable UNIX environment, I chose OpenBSD. Given my history of trying all possible OSs, Let me state again that I'm not against the recent Linux direction. I just wanted to go out and see if there is a different way to do things.

Differences between OpenBSD and Linux

Maybe you're reading this article for its practical value and not for my ramblings, sorry. I thought I had to provide some context. I'm used to googling, RTFMing, and to reading source code to learn what software does. This context is important to judge if you would notice the same differences as I did. Here's a list of things that surprised me the most after completing an OpenBSD install, adapting my old setup to the new environment and running it for a few days.

Simplicity

First of all, everything is much simpler, like the Linux old days. Not necessarily easier, but simpler. More minimalistic. I found this plays well with my mind. OpenBSD follows the UNIX philosophy more closely: lots of small components that do one thing and talk between them by passing text. Because of that, some base components are not as feature-rich, on purpose. Since 99% of the servers don't need the flexibility of Apache, OpenBSD's httpd will work fine, be more secure, and probably faster. For those who need the big boys, just install Apache from the packages. Having a developer-chosen default option for many servers is a time saver. The admin knows it will be well supported and documented, and tightly integrated with other components. The alternative, the Linux way, is to just use what everybody else

uses (Apache), or choose one of the multiple options, always wondering if it's the right one—nginx? lighttpd? thttpd? You know what... *nobody got ever fired for choosing Apache*

Design decisions

Picking up on that thought, the system is very opinionated. When the community decides that some module sucks, they develop a new one from scratch. OpenBSD has its own NTPd, SMTPd and, more recently, HTTPd. They work great. Likewise, the standard shell is pdksh. The [OpenBSD FAQ] states that "*Users familiar with bash are encouraged to try ksh(1) before loading bash on their system -- it does what most people desire of bash.*", which is a bit too bold. ksh does not support history substitution (sudo !!:1) which I use a lot, though I agree that for many users it will be enough. Many people hate bash for some reasons. I am not one of them. Having a super powerful shell has saved me from writing perl scripts for system administration. Bash can always be installed from packages, anyway. This is a big difference from Linux, which is more like a "consensus" operating system. Developers have to keep compatibility and whenever there is a controversial design decision like systemd, dozens of projects [decide to fork]. Not good. Strong opinions, on the other hand, also lead to less support for some, like ext4, ZFS or Linux binary compatibility. For example, ext4 is officially supported read-only but in my case it didn't read some folders properly. FreeBSD plays better on that regard, though they also have more developer manpower. This leads to some use cases, like an OpenBSD desktop, being possible but not the best choice for this OS. Finally, other decisions make little sense. According to disklabel(8), the /usr partition takes about 2G of disk space, not including /usr/src or /usr/obj. This means that there is little space to hold what is essentially the whole system plus ports. I had trouble compiling large ports since /usr ran out of disk space. If a large number of users will be compiling some ports, why not set a larger /usr by default?

Documentation

The man pages are excellent, a delight. Unlike Linux, they are not just a list of switches for the software, but a comprehensive guide to the tool, with lots of examples. They are much, much better—thankfully, because unlike Linux, again, there are not tons of help on public forums. OpenBSD's man pages are so nice that RTFMing, somebody on the internet is not condescending but selfless. Granted, I wouldn't make a UNIX novice run OpenBSD from man pages. But for an experienced admin, they contain exactly the information they need.

Small differences in common tools

Using the BSD toolchain instead of the GNU one means there are small differences between the tools. For example, some ps switches are missing, like the useful -f. The tar options for reading from stdin are also slightly different. When ls is run by root, it automatically appends all hidden files.

df has -h (human) and -k (kilobytes), but no -m for megabytes.

If you've used MacOS you probably know a few of these.

Packages

OpenBSD has packages, like Linux. Unlike it, packages are only available for 3rd party software, not the base system. OpenBSD's base system is more or less what gets installed from the CDs: kernel, shell, coreutils, a small part of X and essential servers (http, ntp, smtp, etc.) Everything else must be installed from packages. The documentation recommends using packages since it is not worth it to compile from ports—the package sources. However, packages don't get security updates. The only way to patch bugs is to compile the ports. Fortunately, there is a simple way to use the best of both worlds: add FETCH_PACKAGES=yes to /etc/mk.confand install software from ports. The system will automatically fetch the package and save the compilation time if there is a current binary available. Another interesting tool is /usr/ports/infrastructure/bin/out-of-date, which checks which ports need an update. So, you can go to /usr/ports/<portname> and make update. This command plays well with previously installed packages. Therefore, you don't have to worry deleting them first.

In summary, after you install the release, if you're interested in getting security updates until next release, the officially recommended path is to follow -stable, use FETCH_PACKAGES and work from ports. This is not very clear in the documentation but the folks at #openbsd helped me figure it out. As a colophon, if you're using x86 or amd64, m:tier provides binary updates for the base system and packages, much like Linux does. Otherwise, if there is any bug in the base system, you'll need to recompile that part yourself. The amount of compiling needed will be determined by the patched component and any related software. Hence, just read the instructions on the patch.

Configuration files

The base system config files are properly centralized in /etc, but not the ports. The porting quality is excellent, better than any Linux distro. Every port is adapted to the OpenBSD system and made sure it behaves correctly. However, some maintainers decide that all the port files need to be contained in some folder, like *transmission-daemon*, which stores its config into /var/transmission/.config/transmission-daemon/settings.json. It's a bit crazy to store a system-wide daemon config file into /var which, according to man hier, contains *Multi-purpose log, temporary, transient, and spool files*. Apparently some daemons are chrooted by default, and there is a global "catch-all" README folder on /usr/local/share/doc/pkg-readmes which contains specific info about packages. transmission-daemon had no related info, so maybe I'll contact the maintainer.

Chroot

Speaking of roots, nearly all daemons in the base system are chrooted and privstep by default. The base system has a lot of hardening by default, which is one of the main reasons why OpenBSD has almost no remote holes on the default installation. Since chrooting software in Linux can be cumbersome, it's very convenient to get it done for you, so thanks!

Experienced community

I feel like the learning curve is a feature, not a bug, intended to keep newcomers away. OpenBSD is unapologetically elitist. Honestly, I don't mind that. I've been administering systems for more than a decade and not all environments are for everybody.

OpenBSD *can afford* to be elitist because it is a small system, with a clear direction, the documentation is crystal clear, and it doesn't make vague promises.

make build

As you can see, there is a big *con* to using OpenBSD coming from a Linux world, the process for patching security issues. On Linux, I was used to run a single command and let any part of the system (base or 3rd party) update itself. With OpenBSD, it takes a lot more effort and time, especially in my old machine. This process leaves the admin only one realistic option: follow the -stable branch, which is basically the same code as the CD release with small patches, and recompile stuff regularly. Otherwise, the installed system will be exposed to potential security holes until the next release. I feel that this needs to be more prominent in the OpenBSD docs, especially on the Migrating to OpenBSD section: *if you want an updated and stable system you'll need to recompile stuff constantly, there is no equivalent to apt-get upgrade*. To get a secure production system with OpenBSD, the officially recommended path is to:

- Install the CD release
- Download the source code
- Recompile the kernel (recommended by "following -stable")
- Recompile userland
- Download ports tree

Add FETCH_PACKAGES=yes to /etc/mk.conf to let ports fetch packages, if available, and install software using the ports syntax. Recompile when there is a security issue which affects your setup, though you may skip some compiling if using m:tier. Of course, this is a feature, not a bug, but it's the biggest admin change from old Linux users. That's a lot of effort compared to apt-get update && apt-get upgrade. Honestly, had I known it, I would've more strongly considered keeping my Debian installation. I

read all the online documentation before installing OpenBSD, and I felt like this point wasn't really clear. Since you can safely use -stable ports/packages with a -release base system, steps 3-4 can be avoided or shortened if you don't want to update your base to -stable. That's what I would recommend to former Linux users, but take this newbie's advice with a grain of salt. In any case, for low-performing machines like mine, maybe the "recommended" path to follow -stable and rebuild the source for every errata is not the best one. For small fixes, it may be better to apply the patch and follow its instructions. Apparently in faster machines, it's more convenient to recompile the base system since it takes just a few minutes. Had I been using x86 or amd64, I'd have totally gone for m:tier. So you can dismiss this section if that's your case. To be totally fair, it's rare for OpenBSD to have remote holes on the CD release. Thus, one could be relatively safe by only upgrading from release to release. But the truth is that there is no simple way to binary patch for critical updates unless using the third-party m:tier on one of the supported architectures. With that it mind, to summarize, consider the following options:

Use a -release base and -stable ports (with `FETCH_PACKAGES=YES`), cherry-picking patches from base and updating ports by `make update`. This may be the recommended path for low-performing machines.

Use a -stable base, too. You can then update the whole system with a handful of commands and won't need to follow patch instructions.

Use -release and update from m:tier

Keep using -release until a next -release comes, unless there is an unlikely remote hole that forces you to recompile the base. This may be the best option for newbies if the only person using the box is the admin, so there is no way to suffer local attacks.

Conclusion

From a user perspective, all of this is transparent; OpenBSD has a UNIX terminal or Xwindows session and everything works as expected. But a Linux admin will need to adapt to these new tools and allocate some more time for administration.

OpenBSD has pros and cons. Personally, my main pros are the excellent documentation, its minimalism and the choice of default daemons. The only con is the need to recompile to patch errata. If I had just one wish for OpenBSD, it would be a more straightforward updating system for security errata.

Now, the dreaded question. *Is it worth it?*

Honestly, I wasted too much time. Some of it was to be expected, since I needed to learn a different environment. Had I been 10 years younger, this wouldn't have been a problem, but my time is more limited now. The fact that I needed to compile things on an old machine probably didn't help. Keep that in mind when considering a BSD for an old, weird architecture. After the initial investment, I want to see if maintenance is easier and release upgrades are smoother than with Debian. Manually upgrading things is a pain in the neck, but all other factors lead me to think that OpenBSD is a great server OS. Maybe I was expecting something else from the docs I read. It is probably my fault, though. Anyway, I want to contribute to the available documentation by writing this document so that other Linux admins can make a more informed decision. On the other hand, my geeky side is content. OpenBSD rocks. It is a different—a real—UNIX and I've really come to appreciate simple code and software. As an admin, having minimalistic, default servers is a blessing. Again, should you try OpenBSD? The answer is yes, though be careful if you're either in a rush or have specific software requirements. The first days are a bit hard, and recompiling on a slow machine takes time.

If you like UNIX, it will open your eyes to the fact that there is more than one way to do things, and that system administration can still be simple while modern.

Revised with contributions from TJ and Mike. Thanks!

Interview with Felix Weinrank

Felix Weinrank is a computer scientist from Germany. He is currently a Ph.D student in the Department of Electrical Engineering and Computer Science at Münster University of Applied Sciences. His research interests include the SCTP transport protocol, low-latency Internet communication and network emulation.

Can you tell our readers about yourself and your role nowadays?

I am a Ph.D student in the Department of Electrical Engineering and Computer Science at Münster University of Applied Sciences.

Currently, I am involved in two research projects which are related to transport protocols. The first project, in cooperation with the University of Duisburg-Essen and funded by the German Research Foundation, seeks to improve the interaction of media and non-media streams in WebRTC peer-to-peer communication.

The second project, in cooperation with several European partners from universities and industrial companies, offers application developers a new and unified API for network communication. You will find more details at www.neat-project.org.

How did you get involved with computer science?

I was about seven or eight years old when I became fascinated by my father's computer. I played around with it whenever possible – and I often broke it.
So, he gave me an old second-hand Intel 486 and from then on, I was responsible to fix it. I learned a lot about the technical background and was always curious how the things work.

While having a wide field of expertise, please tell our readers on which area you put the most emphasis and why?

I put most of my emphasis on improving the SCTP protocol.

It is very flexible with respect to the development of new extensions and can be used for many use cases. Since SCTP is part of the WebRTC stack, where it is used as the transport protocol for Data-Channels, it is widely deployed and integrated into almost every browser.

What tools do you use the most often and why?

First of all, the Atom editor. I like the balance between the lightweight design and the endless ways to extend it.

Since I am usually programming code in C, there are a lot of tools involved. Git, Clang, Valgrind and LLDB, to only name some of them.

In addition to writing real networking code, we are using the OMNeT++ discrete event simulator in combination with the INET framework. This allows us to implement and test new features before we integrate them into the operating systems.

What was the most difficult and challenging issue you've done so far? Could you give us some details?

The current tasks my colleagues from Essen and I are working on. While real-time media streams aim to have low delay, non-media streams always aim to use all the available bandwidth. These two behaviors are controversial and we are working on mechanisms to combine them in an optimal way.

Do you have any specific goals for the rest of this year?

To finish some research papers and successfully publish them.

Thank you.

COLUMN

The gig economy giant, Uber, has had its operating licence suspended by Transport for London. Apart from concerns over the way the company operates, a more sinister reference was made to Greyball software which effectively tricked law enforcement and those Uber didn't wish to deal with. Where should the line be drawn between good practice and deception?

Rob Somerville has been passionate about technology since his early teens. A keen advocate of open systems since the mid-eighties, he has worked in many corporate sectors including finance, automotive, air- lines, government and media in a variety of roles from technical support, system administrator, developer, systems integrator and IT manager. He has moved on from CP/M and nixie tubes but keeps a soldering iron handy just in case.

For many years I have worked at the edge of the envelope where the bad guys reside. As a webmaster, developer and security specialist, I have absolutely no time for those that choose to reside on the dark side and profit from the misery of others, be that intellectual, financial or just pure ego and one-upmanship. Undoubtedly, all serious IT professionals have messed around at some stage pulling IT related pranks on colleagues and peers (my personal favourite was editing command.com with a hex editor, and changing the error messages around) but with no long term damage, and a wry smile and a laugh on both sides. So, I'll be the first to admit when I come across a particularly cunning and devious piece of malware my first reaction (after using a few choice words and picturing a particularly gruesome scenario involving the perpetrator) is to respect the innovation, creativity and the "in your face" sheer audacity.

And so it is with the Silicon Valley incubators, innovators and disrupter's. While the investors and venture capitalists look for the next and best opportunity (the cunning of the malware), few sit back and look objectively at the subtle and not so subtle changes that are going on behind the scenes. Google faced this same dilemma, by setting information free the whole concept of property rights and intellectual ownership were thrown into question. It has battled through though, and the general perception of the organisation is now one of the benevolent dictator, with a huge insurmountable market share. Too big to fail, Google teeters between innovation and establishment, almost suffering from an identity crisis that the market interprets as edgy.

Uber on the other hand, while an innovator, has required to morph into an entirely different giant than Google. Whereas little of Google's current territory was occupied, Uber has had to crush the opposition, and like some form polymorphic virus change the classic business model so that the competition cannot challenge it legally or competitively. Whereas Google is much more an ethereal company, only contacted via our keyboards and LCD screens, you sit in an Uber and chat with the driver. Be it Google or Uber, however, the hidden hand of

technology is at work behind the scenes. Whereas Google's focus is based on openness and sharing, Uber's is based on sharing and efficiency. Few people pay Google money. Every Uber user must. So, the model has to be subtly different, despite the common goal of world domination and at the same time, busting through preconceived ideas and moral frameworks. While Google can occupy like a settler in a new land, Uber must push aside and crush the opposition. For both sides, realize it is one or the other, old school or new school.

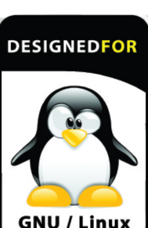
Hence, a very nasty taste is left in one's mouth when encountering such commercial tactics as Greyball. From a purely brand and customer support angle it flies in the face of good practice. After all, if you can win over a dissatisfied customer not only will you have secured a degree of loyalty that money can't buy, but you will have built that ethereal, invisible quality that "the brand" encapsulates. Think of unicorn, rainbows and chocolate. But no, not only does Uber want to choose who its customers are, but also what level of weighing justice performs with her scales. It is the ultimate in passive aggressive tactics, contempt disguised as nonchalance.

I'm an old-fashioned kind of guy, and I believe wholeheartedly in the rule of law, provided that the law is applied equitably to everyone, including corporations and individuals. Tax questions aside, I don't think Google would have the bare faced cheek to stick the middle finger up to the established order in the same way that Uber has achieved with Greyball. Though, it is interesting that after Dara Khosrowshahi, the Uber CEO, had apologized in writing, the Mayor of London welcomed the re-opening of dialogue with Transport for London. Money talks, and being accused of not being a fit and proper operator will obviously cost the company dearly if it loses London, which will worsen if other major cities decide to follow suit. Maybe, Uber has just walked into the hallowed arena, where they are too big to fail, least of all in the perspective of adding cultural value to the capital of England. In a sense, they have outmaneuvered Google by being more evil yet at the same time closely following in the path their geographical peer has hewn in the rock.

At the end of the day though, the whole moral and ethical question boils down to progress. As individuals, consumers and technologists, our decision must be a holistic one – how much progress are we willing to accept in hardware, society and our understanding of right and wrong? How long can an organisation that has such power and influence remain impartial until their true motives are revealed? Google, after all promoted itself on the basis of "Do no evil". I don't think that it is an overestimation to say that particular rhetoric is increasingly being questioned, and the historical fact remains that large, innovative tech companies eventually swallow the kool-aid of the military-industrial complex and end up in the fetal position in the quiet obscurity of the corner. Think of IBM and HP. At one point, these companies, like Google and Uber are currently, were innovators. While there might be the occasional flash of inspiration, they have sunk into the swamp of safety, conformity, establishment and the bedrock of the pension fund. Risk adverse, grey haired and creaking a bit at the joints. And I have no doubt that both Google and Uber will eventually follow that path as well, given sufficient time.

Rack-mount networking server

Designed for BSD and Linux Systems



Designed. Certified. Supported

Up to **5.5Gbit/s** routing power!

KEY FEATURES

- 6 NICs w/ Intel igb(4) driver w/ bypass
- Hand-picked server chipsets
- Netmap Ready (FreeBSD & pfSense)
- Up to 14 Gigabit expansion ports
- Up to 4x10GbE SFP+ expansion

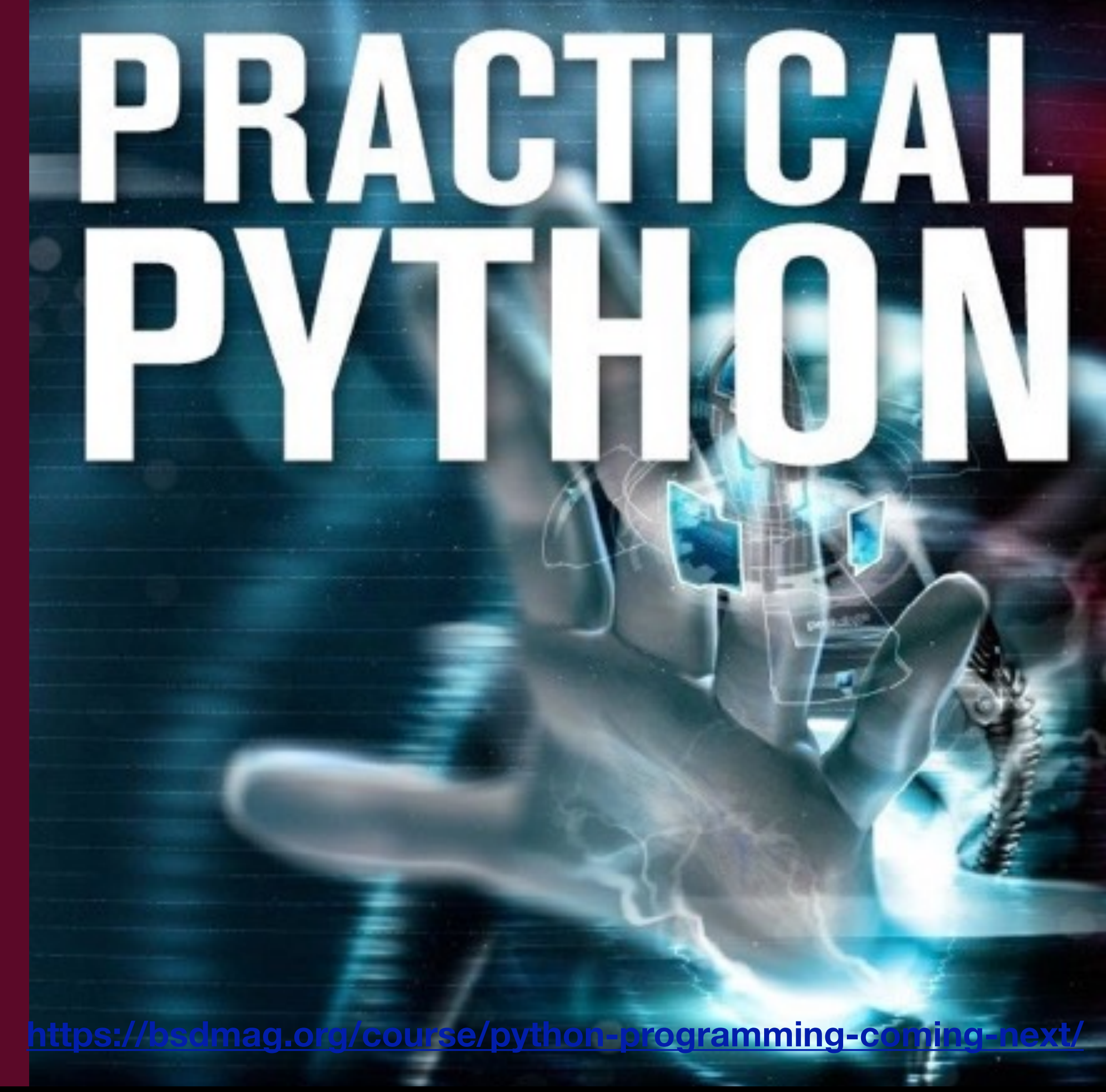
PERFECT FOR

- BGP & OSPF routing
- Firewall & UTM Security Appliances
- Intrusion Detection & WAF
- CDN & Web Cache / Proxy
- E-mail Server & SMTP Filtering

contactus@serveru.us | www.serveru.us
8001 NW 64th St. Miami, LF 33166 | +1 (305) 421-9956

Become a Python Programmer today and start learning one of most-wanted skills of 2017!

This is the most comprehensive course for the Python programming language. If you want to start programming or if you want to learn about the advanced features of Python, this course is for you!



https://bsdmag.org/course/python-programming-coming-next/

In this course we will teach you :

- understanding python and how to install it
- understanding python virtual environments
- using the interpreter and running a python script
- advanced text editors for coding
- understanding basic python data types
- knowing what classes are and how to use them
- understanding exceptions and know how to handle them
- knowing how to get data from a publicly accessible API
- understanding the concept of duck typing
- understanding how files work
- understanding the csv python module to read/write csv files
- anding how to hit API's to get information
- understanding how data should be structure to feed plot module
- using python plotting library