FreeBSD® JOURNAL

July/August 2024

Storage and Filesystems

NVMe Over Fabrics in FreeBSD

FreeBSD iSCSI Primer

Protecting Data with ZFS Native Encryption

Rolling Your Own Images

Introduction to TCP Large Receive Offload

S&W PUBLISHING LLC
PO BOX 3757 CHAPEL HILL, NC 27515-3757

Editor-at-Large • James Maurer
maurer.jim@gmail.com

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-51 42 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

Welcome to the July/August edition of the *FreeBSD Journal*! As I write this, summer is drawing to a close, and we're transitioning into the fall season. Since my last letter, FreeBSD has welcomed a new core team, and version 14.1 has been officially released. Our release engineer, Colin Percival, has also shared a schedule for upcoming releases, with 13.4 nearing completion and 14.2 expected by the end of the year. In May, during BSDCan, the *Journal*'s editorial board gathered to plan our issues for the coming year. We are thrilled by the vibrant activity within the FreeBSD community and the compelling stories we're privileged to bring you in these pages.

This issue's theme is Storage on FreeBSD. Jason Tubnor and I kick things off with introductions to two network block storage protocols supported by FreeBSD: iSCSI and NVMe over Fabrics. Roller Angel then takes us through ZFS native encryption, exploring how it ensures data security. And in this issue's columns, Christopher Bowman continues his series with insights on using FreeBSD in embedded environments, while Randall Stewart and Michael Tüxen break down TCP LRO. Benedict Reuschling explains how to use Samba for Time Machine backups, and Michael Lucas provides his usual entertaining commentary.

On the conference front, Aymeric Wibo shares his experience at BSDCan 2024. Looking ahead, EuroBSDCon 2024 is just a couple of weeks away in Dublin, Ireland, and the FreeBSD Fall Summit will be held at NetApp's campus in San Jose, California, in November. Members of the editorial board will be attending both events, and we always welcome the opportunity to chat.

We love hearing from readers. If you have feedback, article suggestions, or are interested in contributing, please reach out to us at info@freebsdjournal.com.

John Baldwin
Chair of the *FreeBSD Journal* Editorial Board

July/August 2024

FreeBSD JOURNAL

Storage and Filesystems

WeGetletters
by Michael W Lucas

Dear Least Helpful Technology Columnist,

AI is everywhere. Software companies are adding it to their products. Should I be concerned about my career?

—Worried

Dear Worried,

Proper consideration of your question demands carving away all evasions, mistruths, and outright deceptions. Marketing calls any kind of an algorithm AI. Ask them, and Unix is an AI. The Microsoft Excel SUM function? The peak in AI reliability. Every AI puts people out of work — after all, once upon an aeon *calculator* was a job title and changing a spreadsheet required man-hours of labor. But I'm going to assume that you mean "generative AI," partially because if your job could be done by the SUM function you wouldn't know about this column, but also because it will grant me the opportunity to threaten multimillion-dollar companies.

You have already faced the threat posed by generative AI. While you will never defeat it, that threat guarantees your future employment.

You have already faced the threat posed by generative AI.

Once upon a time there was this person who vexed me so badly, I had to write a book just to complain about them. (Not why I wrote the book, nor why I griped about them therein.) My fierce vituperation was so all-encompassing, they not only changed their name but their gender so they could attempt to rebuild something from the ruins of their reputation. (Totally not why they changed their name. Nor their gender.) Now I'm gonna tell you about working with Delta. (Not that we ever actually worked together. They're just an example person. Libel laws prevent me from explicitly naming Gabriel, though he will hopefully recognize himself. That new emotion you've never experienced before, Gabriel? It's called *shame*.)

You probably have a preferred public discussion platform for technical matters, something like Reddit, the Fediverse, or the penal board web forum. Delta's that person who when they see someone ask a question, they search Google and post the first link it vomits up even though it clearly says "sponsored." When the Detroit-Farawayistan optical fiber goes bad, Delta offers to re-terminate the RJ45s. When someone says, "Have you tried giving the customer what they want?" or "it made too much heat so I unplugged it," that's Delta.

The Deltas of this fallen world give us the valuable opportunity to learn to route around damage.

Generative AI is, by definition, a less competent Delta.

These "generative artificial intelligences" scour the Internet collecting text strings and noting which characters often appear in which order. The programmers heard the phrase "the wisdom of crowds" and thought it wasn't satire. When you enter a string into the system, they produce a string that looks like something that would appear after your string. In other words, if you enter something that looks like a StackExchange question, they provide an answer that looks like something you would get from StackExchange. The average answer on any public technology forum is a poison to the spirit that makes my Perl look glamorous. Not Hollywood glamour. More like Eldritch Faery Queen Glamour that winds up with you chained to your keyboard condemned to write for the entertainment of the Unseelie Court until you become the greatest author on Earth, which would give you lots of practice, but as you no longer receive books from Earth you can't perform the comparison that would conclude your deal. Still, don't do that. The Faery Queen carries one heck of a grudge, especially if you smuggled lockpicks in with you.

Generative AI is, by definition, a less competent Delta.

Yes, you can find good information on the Internet. But it's never in the first search engine result. It's probably not in your first query. Beating useful information out of the Internet is a skill developed through years of negative reinforcement, and one that these generative engines lack. The Internet contains tiny slivers of wisdom entombed amidst vast mounds of festering mediocrity, seasoned with inanity. Generative AI uses several buildings jammed with GPUs, several megawatts of power, and enough clean water to irrigate a small nation to emulate a fresh college grad who really hopes potential employers don't notice his 2.0001 "pity pass" GPA through their challenging Bachelor of Arts in General Studies, or that he's been banned from every library within bicycling distance for Extreme Bigotry. What's not to like, other than authors such as myself joining in one of the innumerable copyright violation lawsuits being assembled against these AI firms?

So no, you don't need to fear generative AI.

You must improve your skill at working around damage.

How did you cope with your Delta? Perhaps you offered a glowing performance review so they could be shunted harmlessly into Human Resources. Or you could have persuaded them to accept the valuable assignment of hexhead screw auditor. Maybe you sent them into the Hall of Backup Tapes with a ball of string so they could find their way back but wanted them to be safe on that most perilous of journeys, and so you tested the string for flammability and discovered to your "dismay" that it went up like flash paper forever marooning them amongst the reels of paper tape. I won't judge, unless the string burned left a trail of ash. You had a problem, you dealt with it in the least illegal manner possible.

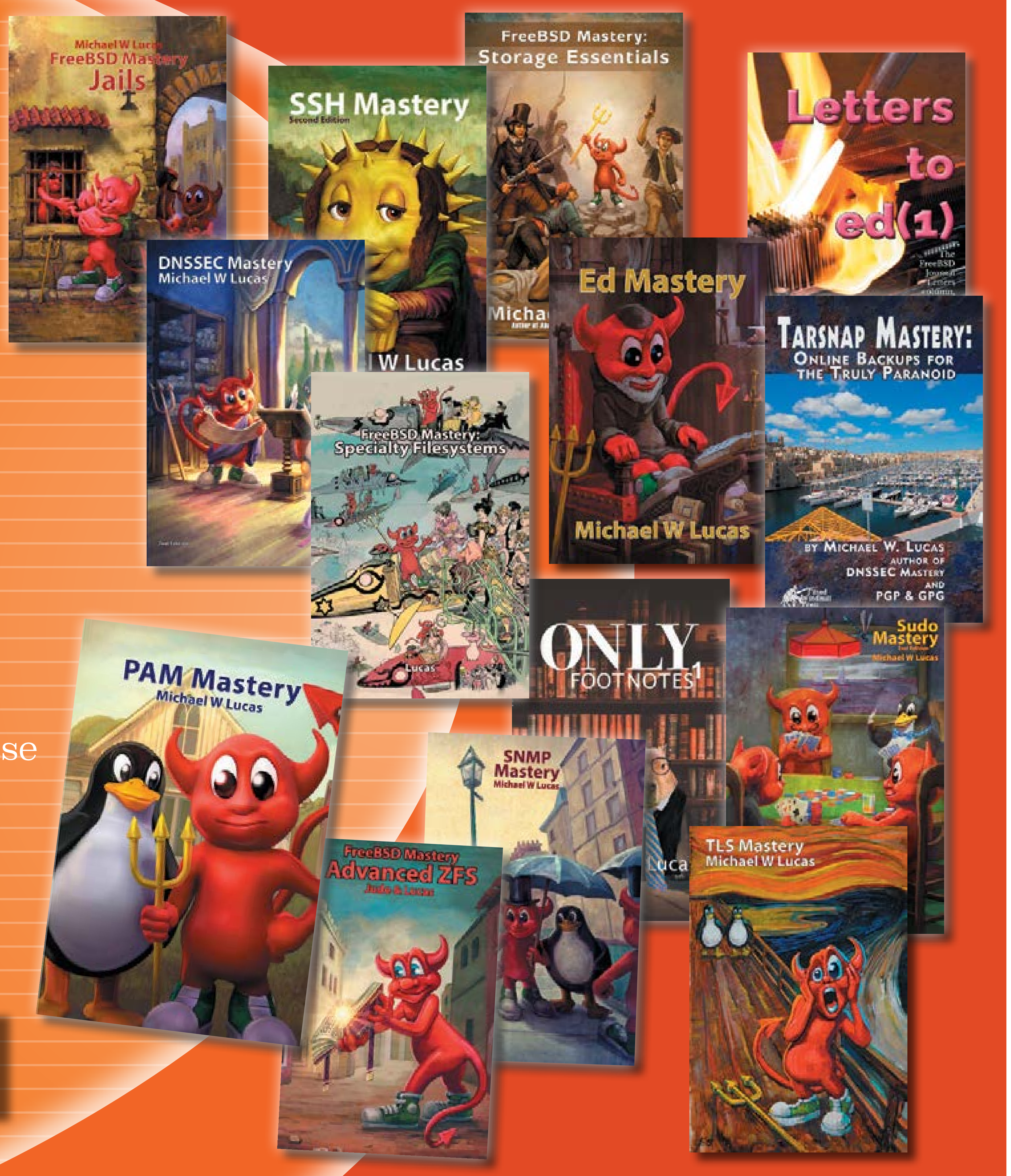
Someone in your organization will catch AI Fever and look to convert your company's worthless payroll into precious payments to AI firms. The simplest way to avoid this is to remember that you already use AI. Somewhere you have a spreadsheet, right? Make sure it adds the numbers for you and boom — AI! It's not a lie. After all, marketing said it was AI and they wouldn't lie. The proper invocation of grep and awk can provide more intelligence than any of these firms.

If people insist on using generative AI, grab one of the freely available models and train it on your company's document store. You have decades of badly written emails, proposals, and white papers that can easily compete with the delusional ramblings available on Stack-Exchange. That will let you illustrate that no matter how appalling the average employee is, generative AI is worse.

Suppose the worst happens. A decree comes down from management to deploy generative AI. Solve two problems simultaneously, and have Delta deploy it.

Have a question for Michael?
Send it to letters@freebsdjournal.org

MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *SSH Mastery*, and the brand-new *Run Your Own Mail Server*. His collection of these columns, *Dear Abyss*, is coming to https://mwl.io/ks at any moment.


FreeBSD Mastery Jails
SSH Mastery
FreeBSD Mastery: Storage Essentials
Letters to ed(1)
DNSSEC Mastery
Ed Mastery
Tarsnap Mastery: Online Backups for the Truly Paranoid
FreeBSD Mastery: Specialty Filesystems
Only Footnotes
Sudo Mastery
PAM Mastery
SNMP Mastery
FreeBSD Mastery Advanced ZFS
TLS Mastery
Michael W Lucas

NVMe Over Fabrics in FreeBSD

BY JOHN BALDWIN

NVM Express (NVMe) is a recent standard providing access to non-volatile memory block storage devices such as SSDs. NVMe was originally defined to access non-volatile memory devices via PCI-express. This includes register definitions for the PCI-express controller device, the layout and structure of command submission and completion queues stored in main memory, and sets of commands.

The base NVMe specification defines an **Admin Command Set** used on a single admin submission and completion queue pair associated with each controller. Administrative commands do not handle I/O requests. Instead, these commands are used to create I/O queues, fetch auxiliary data such as error logs, format storage devices, etc. Storage devices in NVMe are called **namespaces** and commands for a specific namespace include a namespace ID. The base specification also defines an **NVM Command Set** used for I/O requests to block-oriented namespaces. The specification is designed for future extensions including additional I/O command sets (e.g. an I/O command set targeting a key-value store). An NVMe controller and its attached namespaces together are called an **NVM subsystem**.

NVMe over Fabrics extends the original specification to enable access to NVM subsystems over a network transport instead of PCI-express similar to using iSCSI to acccess remote block storage devices as SCSI LUNs. NVMe over Fabrics supports multiple transport layers including FibreChannel, RDMA (over both iWARP and ROCE) and TCP. To handle these different transports, Fabrics includes both transport-independent extensions to the base NVMe specification as well as transport-specific bindings.

NVMe over Fabrics extends the original specification to enable access to NVM subsystems over a network transport.

Fabrics defines a new **capsule** abstraction to support NVMe commands and completions. Each capsule contains either an NVMe command or completion. In addition, a capsule may be associated with a data buffer. To support data transfers, the existing PRP entries in NVMe commands are replaced with a single NVMe SGL entry. Fabrics also replaces the shared-memory queues used for PCI-express controllers with logical completion and submission queues. Unlike PCI-express I/O queues, Fabrics queues are always explicitly paired with each submission queue tied to a dedicated completion queue. The details of how capsules and data buffers are transmitted and received on a queue pair are transport-specific, but in abstract terms, command capsules are transferred on submission queues, and completions are transferred on completion queues.

A Fabrics **host** creates an admin queue pair and one or more I/O queue pairs connected to a **controller**. A complete set of queue pairs is called an **association**. A single association may contain multiple transport-specific connections. For example, the TCP transport uses a dedicated connection for each queue pair, so an active TCP association requires at least two TCP connections.

In addition to I/O controllers which provide access to namespaces, Fabrics adds a **discovery** controller type. A discovery controller supports a new discovery log page which describes the set of controllers available in a Fabrics NVM subsystem. The log page may include one or more I/O controllers and/or references to discovery controllers in other subsystems. The log page may include multiple entries for a single controller if a controller can be accessed via multiple transports. Each log page entry contains the type of a controller (I/O or discovery) as well as the transport type and transport-specific address. For the TCP transport the address includes the IP address and TCP port number.

A Fabrics host creates an admin queue pair and one or more I/O queue pairs connected to a controller.

Fabrics hosts and controllers are identified by an NVMe Qualified Name (**NQN**). NQNs are an ASCII string which should start with "nqn.*YYYY-MM.reverse-domain*" followed by an optional trailer. The *reverse-domain* portion of the name should be a valid DNS name in reverse order, and the *YYYY* and *MM* fields should specify a valid year and month when the DNS name was owned by the organization using the prefix. The specification defines a fixed subsystem NQN for Discovery controllers as well as a scheme for constructing a NQN from a UUID. Both the host and subsystem (controller) NQNs must be specified when establishing an association.

FreeBSD 15 includes support for accessing remote namespaces via a host kernel driver as well as support for exporting local storage devices as namespaces to remote hosts. The kernel implementation of Fabrics includes a transport abstraction layer (provided by `nvmf_transport.ko`) to hide most of the transport-specific details from the host and controller modules. This module is auto-loaded as needed. Separate kernel modules provide support for individual transports. These modules must be explicitly loaded to enable use of a transport. Currently, FreeBSD includes support for the TCP transport via `nvmf_tcp.ko`. TCP specific details are documented in nvmf_tcp(4).

Host

The Fabrics host in FreeBSD consists of new nvmecontrol(8) commands and an nvmf(4) kernel driver. The kernel driver exposes remote controllers as nvmeX new-bus devices similar to PCI-express NVMe controllers. Remote namespaces are exposed via nda(4) disk devices via CAM. Unlike the PCI-express nvme(4) driver, the Fabrics host driver does not support the nvd(4) disk driver. All of the new nvmecontrol(8) commands use a host NQN generated from the host's UUID unless an explicit host NQN is given.

Discovery Service

The nvmecontrol(8) **discover** command queries the discovery log page from a discovery controller and displays its contents. Example 1 shows the log page from a Fabrics controller running on a Linux system. For the TCP transport, the service identifier field identifies the TCP port of the remote controller.

Example 1: The Discovery Log Page from a Linux Controller

```
# nvmecontrol discover ubuntu:4420
Discovery
=========
Entry 01
========
 Transport type:       TCP
 Address family:       AF_INET
 Subsystem type:       NVMe
 SQ flow control:      optional
 Secure Channel:       Not specified
 Port ID:              1
 Controller ID:        Dynamic
 Max Admin SQ Size:    32
 Sub NQN:              nvme-test-target
 Transport address:    10.0.0.118
 Service identifier:   4420
 Security Type:        None
```

Connecting To an I/O Controller

The nvmecontrol(8) connect command establishes an association with a remote controller. Once the association is established, it is handed off to the nvmf(4) driver which creates a new **nvmeX** device. The connect command requires both the network address and subsystem NQN of the remote controller. Example 2 connects to the I/O controller listed in Example 1.

Example 2: Connecting to an I/O Controller

```
# kldload nvmf nvmf_tcp
# nvmecontrol connect ubuntu:4420 nvme-test-target
```

Once the association is established, the kernel outputs the text from Figure 1 to the system console and system message buffer. The **nvmeX** device includes the remote subsystem NQN in the device description, and each remote namespace is enumerated as an **ndaX** peripheral.

Figure 1: Console Messages from Connecting

```
nvme0: <Fabrics: nvme-test-target>
nda0:  at nvme0 bus 0 scbus0 target 0 lun 1
nda0:  <Linux 5.15.0-8 843bf4f791f9cdb03d8b>
nda0:  Serial Number 843bf4f791f9cdb03d8b
nda0:  nvme version 1.3
nda0:  1024MB (2097152 512 byte sectors)
```

The **nvme0** device from Figure 1 can be used with other nvmecontrol(8) commands such as **identify** similar to PCI-express controllers. Example 3 shows a subset of the **identify** controller data displayed by nvmecontrol(8). The **nda0** disk device can be used like any other NVMe disk device.

Example 3: Identify a Remote I/O Controller

```
# nvmecontrol identify nvme0
Controller Capabilities/Features
================================
...
Model Number:                 Linux
Firmware Version:             5.15.0-8
...

Fabrics Attributes
==================
I/O Command Capsule Size:     16448 bytes
I/O Response Capsule Size:    16 bytes
In Capsule Data Offset:       0 bytes
Controller Model:             Dynamic
Max SGL Descriptors:          1
Disconnect of I/O Queues:     Not Supported
```

Connecting via Discovery

The nvmecontrol(8) connect-all command fetches the discovery log page from the indicated discovery controller and creates an association for each log page entry. The association from Example 2 could be created by executing `nvmecontrol connect-all ubuntu:4420` instead of fetching the discovery log page and using the connect command.

Disconnecting

The nvmecontrol(8) disconnect command detaches the namespaces from a remote controller and destroys the association. Example 4 disconnects the association created by Example 2. The **disconnect-all** command destroys associations with all remote controllers.

Example 4: Disconnecting From a Remote I/O Controller

```
# nvmecontrol disconnect nvme0
```

Reconnecting

If a connection is interrupted (for example, one or more TCP connections die), the active association is torn down (all queues are disconnected), but the **nvmeX** device is left in a quiesced state. Any pending I/O requests for remote namespaces are left pending as well. In this state, the reconnect command can be used to establish a new association to resume operation with a remote controller. Example 5 reconnects to the controller from Example 2. Note that the reconnect command requires an explicit network address similar to the connect command.

Example 5: Reconnecting to a Remote I/O Controller

```
# nvmecontrol reconnect nvme0 ubuntu:4420
```

Controller

The Fabrics controller on FreeBSD exposes local block devices as NVMe namespaces to remote hosts. The controller support on FreeBSD includes a userland implementation of a discovery controller as well as an in-kernel I/O controller. Similar to the existing iSCSI target in FreeBSD, the in-kernel I/O controller uses CAM's target layer (ctl(4)).

Block devices are created by adding ctl(4) LUNs via ctladm(8). The discovery service and initial handling of I/O controller connections are managed by the nvmfd(8) daemon. The in-kernel I/O controller is provided by the nvmft(4) module. Example 6 adds a ZFS volume named pool/lun∅ as a ctl(4) LUN and starts the nvmfd(8) daemon. Remote hosts can then access the ZFS volume as a NVMe namespace.

Example 6: Exporting a local ZFS Volume

```
# kldload nvmft nvmf_tcp
# ctladm create -b block -o file=/dev/zvol/pool/lun0
LUN created successfully
backend:       block
device type:   0
LUN size:      4294967296 bytes
blocksize      512 bytes
LUN ID:        0
Serial Number: MYSERIAL0000
Device ID:     MYDEVID0000
# nvmfd -F -p 4420 -n nqn.2001-03.com.chelsio:frodo0 -K
```

Each time a remote host connects to the I/O controller, a message is logged by the kernel listing the remote host's NQN (see Figure 2).

Figure 2: Log Messages from New Association

```
nvmft0: associated with
nqn.2014-08.org.nvmexpress:uuid:00000000-0000-0000-0000-ffffffffffff
```

LUNs can be added or removed by ctladm(8) while nvmfd(8) is running. If any remote hosts are connected while a LUN is added or removed, an asynchronous event is reported to the remote hosts. This allows remote hosts to notice that namespaces have been added or removed while connected.

Two new commands have been added to ctladm(8) to manage Fabrics associations. The **nvlist** command lists all active associations from remote hosts. Example 7 shows the output from the **nvlist** command while a single host is connected to the controller from Example 6.

Example 7: Listing Active Associations

```
# ctladm nvlist
  ID Transport        HostNQN                                     SubNQN
   0 TCP
nqn.2014-08.org.nvmexpress:uuid:00000000-0000-0000-0000-ffffffffffff
nqn.2001-03.com.chelsio:frodo0
```

The **nvterminate** command closes one or more associations. Associations for a single connection or NQN can be terminated, or all active associations can be terminated. Example 8 terminates the association from Example 7. After the association is terminated, the kernel logs the messages from Figure 3.

Example 8: Terminating an Association

```
# ctladm nvterminate -c 0
NVMeoF connections terminated
```

Figure 3: Log Message after Terminating

```
nvmft0: disconnecting due to administrative request
nvmft0: association terminated
```

Conclusion

NVMe over Fabrics support will be available in FreeBSD 15.0 including both host and controller support. The development of Fabrics support was sponsored by Chelsio Communications, Inc.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Concord, California with his wife, Kimberly, and three children: Janelle, Evan, and Bella.

FreeBSD iSCSI Primer

BY JASON TUBNOR

We all hear about Network Attached Storage (NAS) being able to provide additional storage for devices on your network. However, the protocols for this storage may not be appropriate for all use cases.

Welcome to the world of Storage Area Network (SAN). Typically, these are found more in enterprises than the home or small business, but that does not mean that they shouldn't be used in such a situation. In fact, you probably have a very good use case if you do lots of virtualization with central storage connected to multiple compute devices or have a need to provide block storage to Windows Workstations used for engineering or graphics design that require more storage than can physically fit within desktop PCs.

Typically, the SAN providers that we hear of in the enterprise space are provided by the likes of Dell EMC, IBM, Hitachi and NetApp to name a few. However, we are spoiled in the FreeBSD space to have a high-performance, iSCSI SAN solution baked right into the base system. Using this, in combination with the powerful ZFS volume manager and file system, we have a flexible, resilient, and fast storage solution that we can make available to network clients. The iSCSI subsystem was implemented and part of the 10.0-RELEASE with numerous performance improvements coming out with the 10.1-RELEASE.

Typically, Storage Area Networks are found more in enterprises than the home or small business, but that does not mean that they shouldn't be used in such a situation.

Internet Small Computer Systems Interface or iSCSI for short is an IP-based protocol for carrying SCSI commands over a TCP/IP ethernet network. It allows the presentation of block device storage to machines distributed across the network. It is flexible enough to even be routed over the internet if required, but the security implications and precautions that need to be considered are out of scope for this text.

In an ideal world, iSCSI should exist within its own Layer 2 physical network where the only communication from compute hosts via dedicated storage interfaces is to that of the storage target. No other general network traffic should exist on this network segment to avoid contention to the storage for this and other compute nodes. In a smaller environment, using a segmented switch with VLANs is an acceptable alternative, however, understand that general network traffic and storage traffic will be competing for interface bandwidth.

The high-level iSCSI terminology is quite simple. There is the initiator (client) and the target (host). The initiator is the active end of the connection whereas the target is passive — it will never try to connect to an initiator.

In the following exercise, we are going to prepare a simple iSCSI configuration on a FreeBSD host to use as a target and provide ZFS **`zvol`** block devices to a FreeBSD and MS Windows initiator.

Our hosts on the network:

```
2001:db8:1::a/64 - FreeBSD ZFS Storage Host (Target)
2001:db8:1::1/64 - FreeBSD Client (Initiator)
2001:db8:1::2/64 - Windows Server 2022 (Initiator)
```

First, we will configure ZFS volumes to present as iSCSI targets for each of the initiators on the storage host. This could also be simply files on a ZFS dataset or UFS partition, but ZFS volumes give you far more control over aspects of the storage, especially ongoing management as data requirements change or in relation to snapshot and replication requirements.

```
zfs create -o volmode=dev -V 50G tank/fblock0
zfs create -o volmode=dev -V 50G tank/wblock0
```

Adjust the **`volblocksize`** attribute when creating the volumes to meet the requirements of the workload that they will be used for. As of 14.1-RELEASE, they will be set to 16K which is a reasonable balance for most workloads.

Create a file **`/etc/ctl.conf`** that is only read/writeable by root. This file will contain secrets, so it is essential that no other group or user has the ability to view or write to this file. Below are the contents that we will add to provide storage points for our initiators:

```
auth-group ag0 {
    chap-mutual "inituser1" "secretpassw0rd" "targetuser1" "topspassw0rd"
    initiator-portal [2001:db8:1::1]
}

auth-group ag1 {
    chap-mutual "inituser2" "hiddenpassw0rd" "targetuser2" "freepassw0rd"
    initiator-portal [2001:db8:1::2]
}

portal-group pg0        {
    discovery-auth-group no-authentication
    listen [2001:db8:1::a]
}

target iqn.2012-06.org.example.iscsi:target1 {
    alias "Target for FreeBSD"
    auth-group ag0
    portal-group pg0
    lun 0 {
        path /dev/zvol/tank/fblock0
        #blocksize 4096
        option naa 0x4ee0ebaf06a1acee
        option pblocksize 4096
        option ublocksize 4096
    }
}

target iqn.2012-06.org.example.iscsi:target2 {
    alias "Target for Windows"
    auth-group ag1
    portal-group pg0
    lun 1 {
        path /dev/zvol/tank/wblock0
        blocksize 4096
        option naa 0x4ee0ebaf06a1acbb
    }
}
```

Let's break down the file to get an understanding of the what and why of each component:

```
auth-group ag0 {
    chap-mutual "inituser1" "secretpassw0rd" "targetuser1" "topspassw0rd"
    initiator-portal [2001:db8:1::1]
}
```

This is the authorization group that can be used across multiple targets. This example binds the **`auth-group`** to a unique initiator with the address 2001:db8:1::1 and requires it to have mutual authentication. You can simply use CHAP authentication as a 'one way' authentication, but it is recommended that, if supported, you use mutual authentication to ensure that both the initiator and the target are correctly authenticating against each other.

This authentication may be sufficient where the initiator and target reside on the same physical network, but it should not be relied upon as a security control. This type of authentication should be seen as a method of ensuring that only the correct storage is allocated to the initiator. Loosely configured iSCSI targets could make the wrong storage available to a target which could have the undesired effect of overwriting data, partition table, or other meta data, so restricting access to a specific target dataset is essential.

```
portal-group pg0        {
    discovery-auth-group no-authentication
    listen [2001:db8:1::a]
}
```

Portal groups set the target environment offered to the initiators. Here we define a portal group to allow initiators to connect to the target via 2001:db8:1::a and so they can discover the target datasets that include this portal group without having to authenticate first. Typically, in a controlled environment, this would be fine to ensure initiators can find what they need to connect to, but would be undesirable in a more hostile or untrusted environment.

```
target iqn.2012-06.org.example.iscsi:target1 {
    alias "Target for FreeBSD"
    auth-group ag0
    portal-group pg0
    lun 0 {
        path /dev/zvol/tank/fblock0
        #blocksize 4096
        option naa 0x4ee0ebaf06a1acee
        option pblocksize 4096
        option ublocksize 4096
    }
}
```

The meat of the configuration is the target. This will include the **`auth-group`** and **`portal-group`** to build up the previously described components for the presentation to the initiator. These can be over-ridden on a per target basis and can be defined without the applicable **`group`** definition.

The iSCSI Qualified Name (IQN) format takes the form **`iqn.yyyy-mm.namingauthority:uniquename`**. This is needed for each target definition.

The alias definition is simply a human readable description for the target.

Each target can have multiple LUNs but here we simply have only one LUN per target.

The LUN context allows you to define the characteristics of the LUN. This is important where ZFS is being used on the presented storage within the initiator. While your ZFS volume on the target has a block size of 16KB, the ZFS pool — when created on the initiator — will complain that the block size of the storage is not 4KB or less. It won't stop you from using it, but a message when you execute **`zpool status`** on the initiator will continually remind you of this. Adjusting the block size attribute to 4096 is not sufficient to remediate the issue. The **`pblocksize`** and **`ublocksize`** options will need to specifically be added for a ZFS use case.

Each target can have multiple LUNs but here we simply have only one LUN per target.

Option naa should be explicitly defined for LUNs. This is either a 64- or 128-bit unique hexadecimal identifier. This is important when you are backing VMWare compute onto an iSCSI target to ensure there is no confusions with LUN assignments.

This is now enough to get you up and running and present storage from your target. Enable **`ctld`** and bring up the daemon:

```
service ctld enable
service ctld start
```

To verify that the storage is presented, you can check that the daemon is listening:

```
# netstat -na | grep 3260
tcp6    0      0 2001:db8:1::a.3260     *.*                     LISTEN
```

Then verify the LUNs are presented using the CAM Target Layer control utility:

```
# ctladm lunlist
(7:1:0/0):<FREEBSD CTLDISK 0001> Fixed Direct Access SPC-5 SCSI device
(7:1:1/1):<FREEBSD CTLDISK 0001> Fixed Direct Access SPC-5 SCSI device
# ctladm devlist
LUN Backend     Size (Blocks)   BS Serial Number     Device ID
  0 block           104857600   512 MYSERIAL0000   MYDEVID0000
  1 block            13107200  4096 MYSERIAL0001   MYDEVID0001
```

Now to configure your FreeBSD initiator. You need to create the **`/etc/iscsi.conf`** configuration file. This also needs to be set as read/write explicitly for root as it contains secrets:

```
fblock0         {
targetaddress   = [2001:db8:1::a];
targetname      = iqn.2012-06.org.example.iscsi:target1;
initiatorname   = iqn.2012-06.org.example.freebsd:nobody;
authmethod      = CHAP;
chapiname       = "inituser1";
chapsecret      = "secretpassw0rd";
tgtChapName     = "targetuser1";
tgtChapSecret   = "topspassw0rd";
}
```

Each of these attributes are:

- **`fblock0`** — This is a human readable identifier; it is not related to anything except grouping each of the following configuration items.
- **`targetaddress`** — The network address of the storage target. This can also be a fully qualified domain name.
- **`targetname`** — This will align with the corresponding target name that was defined in the **`ctl.conf`** file
- **`initiatorname`** — Defining the IQN of the initiator.
- **`authmethod`** — Can simply be defined on FreeBSD as CHAP. Mutal settings will be assumed if **`ChapName`** and **`ChapSecret`** are prefixed with '**`tgt`**'.
- **`chapiname/chapsecret`** — Authentication as defined previously in the **`ctl.conf`** file.
- **`tgtChap[Name,Secret]`** — Authentication that the target needs to complete the authentication handshake with the initiator.

To enable use, simply issue:

```
service iscsid enable
service iscsictl enable
service iscsid start
service iscsictl start
```

This should then attach the target's storage presentation to the initiator:

```
# iscsictl -L
Target name                           Target portal    State
iqn.2012-06.org.example.iscsi:target1 [2001:db8:1::a]  Connected: da0
```

While we are here, let's look at the simple switches that will be used most frequently with the **`iscsictl`** command:

- **`L`** This lists targets mounted to the initiator and where they are connected.
- **`Aa`** Attach all targets defined in the iscsi.conf file
- **`Ra`** Remove all targets that are connected to the initiator

```
# gpart create -s GPT da0
da0 created
# gpart add -t freebsd-zfs -a 1M da0
da0p1 added
# zpool create tank da0p1
# zpool list tank
NAME   SIZE  ALLOC   FREE  CKPOINT  EXPANDSZ   FRAG    CAP  DEDUP    HEALTH  ALTROOT
tank  49.5G   360K  49.5G        -         -     0%     0%  1.00x    ONLINE        -
# zpool status tank
  pool: tank
 state: ONLINE
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          da0p1     ONLINE       0     0     0

errors: No known data errors
```

The manual pages for the configuration, daemons, and control tools are exceptionally well written and can be referenced to get a better understanding of what else is available.

This only touches the surface of the full power that is available with the iSCSI implementation within FreeBSD, but it gives you an idea and practical examples of how it can be harnessed to provide flexible, remote storage options for your computing infrastructure.

JASON TUBNOR has over 28 years of IT industry experience in a vast range of disciplines and is currently the ICT Senior Security Lead at Latrobe Community Health Service (Victoria, Australia). Discovering Linux and Open Source in the mid 1990s, then being introduced to OpenBSD in 2000, Jason has used these tools to solve various problems in organizations that cover different industries. Jason is also a co-host on the BSDNow Podcast.

Contents

Protecting Data with ZFS Native Encryption

BY ROLLER ANGEL

ZFS has native support for encrypting datasets which allows you to easily protect data with industry-standard cipher suites. The major benefit to encrypting a dataset on a disk vs full-disk encryption of the disk is that a dataset can be unmounted when not in use, while full-disk encryption requires the disk to be powered down to get the data encrypted while it is at rest. Keep in mind that ZFS native encryption has the concept of loading and unloading keys. Simply unmounting the encrypted dataset is not enough. You must also unload the key associated with that particular dataset. If the key is still loaded, the dataset can be mounted and the data will be available. Unloading the key will make the mount operation fail. Loading the key is a prerequisite to mounting the dataset. Nested child datasets inherit the encryption key of their parent — but they don't have to. Different encryption keys and cipher suites may be used even if the parent dataset uses different encryption settings. Finally, changing keys is as easy as issuing the `zfs change-key` command on the dataset.

Those are the basic concepts to get started.

Turning on the encryption parameter for a newly created dataset and setting a key format is enough to get started. If an encryption cipher suite isn't specified, the default of `aes-256-gcm` is used. The default is subject to change as new cipher suites get added in the future. The encryption property of an existing dataset is read-only, modifying the property of an unencrypted dataset to turn on encryption isn't allowed. To specify the encryption properties, you need to know what options are available. I suggest reading up on the available options in the `zfsprops` man page. Do this by typing the command `man zfsprops`. I also recommend reading the man page `zfs-load-key`. For our first encrypted dataset, we will start with the default cipher suite, the passphrase key format, and create a dataset called `secrets`. I'm using a FreeBSD jail machine I created in my lab for this article called alice. All the jails in my lab exist on the zpool called lab. I've made a zpool available to the jail with the name zroot. From inside the jail I must use the entire path `lab/alice/zroot` as the name of my zpool in order to create a dataset within. For contrast, on my laptop I can simply use the name of my zpool and directly create a dataset there. Listed below are the commands to create an encrypted dataset on both the alice jail as well as on my laptop. Like any ZFS dataset, setting a mount point is a

Loading the key is a prerequisite to mounting the dataset.

good idea, just keep in mind ZFS is a layering filesystem so don't use an existing path as the new dataset mount point.

alice jail:

```
zfs create -o encryption=on -o keyformat=passphrase -o mountpoint=/secrets
lab/alice/zroot/secrets
```

my laptop:

```
zfs create -o encryption=on -o keyformat=passphrase -o mountpoint=/secrets zroot/secrets
```

After running the **`zfs create`** command, I'm prompted to enter in a sufficiently long passphrase. Now, I have a mounted dataset with encryption enabled where I can store data that I need to protect. While the dataset is mounted, I can use it like any other unencrypted dataset. When I'm done adding the secret data, I can unmount the **`secrets`** dataset and unload the key together in one command by typing **`zfs unmount -u lab/alice/zroot/secrets`**. To decrypt and mount the data again, I run the command **`zfs mount -l lab/alice/zroot/secrets`**. This will ask me for my passphrase, load the key, then mount the dataset. Omitting the **`-u`** flag in the unmount command will only unmount the dataset, leaving the key loaded. The dataset can still be mounted without prompting for the passphrase with **`zfs mount lab/alice/zroot/secrets`**. To unload the key after the dataset has been unmounted, I run **`zfs unload-key lab/alice/zroot/secrets`**. Now the previous mount command will fail because the key isn't loaded and I didn't provide the **`-l`** flag to ask ZFS to load the key before mounting. To load a key and allow the dataset to be mounted, I run **`zfs load-key lab/alice/zroot/secrets`**. I'm prompted for my passphrase and the previous mount command will now succeed because the key is loaded. To check whether a key is loaded or not, view the properties of the dataset. Some useful properties to look at are displayed when I run **`zfs list -o name,mountpoint,encryption,keylocation,keyformat,keystatus,encryptionroot lab/alice/zroot/secrets`**. The **`KEYSTATUS`** column shows **`available`** meaning the key is loaded. To see all of the dataset properties I can use **`zfs get all lab/alice/zroot/secrets`**.

While the dataset is mounted, I can use it like any other unencrypted dataset.

Next I create a dataset nested below the **`secrets`** dataset with a different cipher suite and key format. This time, I'll use a key file instead of a passphrase. To create a dataset that uses a key file, I first need to generate the key and store it in a file. I do this by typing the command **`dd if=/dev/urandom bs=32 count=1 of=/media/more-secrets.key`**. I used the **`bs=32`** because the key file is required to be 32 bytes long. Also, the output is going to the **`/media`** path because I have mounted a portable USB drive there and have used **`dd`** to generate the key file and store it directly onto the drive. This is so there is no trace of the key file on my machine when I unload the key and I unmount and remove the USB drive. I recommend storing this key file on more than one USB drive as a safeguard in case a USB drive is damaged. Now that the key file has been generated I can create the nested dataset with the AES-256-CCM cipher suite by running **`zfs create -o encryption=aes-256-ccm -o keyformat=raw -o keylocation=file:///media/more-secrets.key -o mountpoint=/secrets/more-secrets lab/alice/zroot/secrets/more-secrets`**. Viewing the properties of this new dataset I can see the `ENCROOT` column is set to `lab/alice/zroot/secrets/more-secrets`. I can use the same methods to unmount and unload the key as I used on the `secrets` dataset. As I get more datasets and keys, I may want to consider unloading all the keys with `zfs unload-key -a` or I can unload a subset of keys by unloading just the keys for the `secrets` dataset and all descendant datasets with `zfs unload-key -r lab/alice/zroot/secrets`. Conversely, If I want to load a subset of keys for the secrets dataset and all descendant datasets, I run `zfs load-key -r lab/alice/zroot/secrets`.

If I decide later on that the data in this `more-secrets` dataset doesn't need to have a separate key file, and instead, I want it to inherit the settings from its parent dataset `secrets` (switching from custom generated key file to the passphrase configured earlier) all I need to do is run `zfs change-key -i lab/alice/zroot/secrets/more-secrets`. View the properties again and notice that `ENCROOT`, `KEYLOCATION`, and `KEYFORMAT` have all changed. The encryption suite, however, doesn't change because the encryption suite can only be set on the creation of a dataset. Since `more-secrets` is contained within `secrets` it will unmount as part unmouting the `secrets` dataset. Although mounting the `secrets` dataset will not also mount `more-secrets`. That will need to be mounted separately, but the key will only need to be loaded once since they both share the same key. To switch back to using the key file, I run `zfs change-key -o keyformat=raw -o keylocation=file:///media/more-secrets.key lab/alice/zroot/secrets/more-secrets`. If I want to permanently destroy the data in `more-secrets`, I simply unmount the dataset, unload the key, and destroy the key file and any backup copies of the key file I have made. Now, the data is not able to be recovered. I can then run `zfs destroy lab/alice/zroot/secrets/more-secrets` to remove the dataset.

ZFS native encryption allows data to be easily protected with encryption.

One final note I'd like to share is regarding backups of encrypted data. As you have seen, ZFS native encryption allows data to be easily protected with encryption. Snapshots of encrypted datasets can be received on an untrusted backup server in their encrypted form. Without the key, the remote backup server won't be able to mount the dataset. Use the `--raw` flag of the `zfs send` command to help accomplish this. For more details, I suggest reading the man page `zfs-send` to get an idea of how it works and then get a copy of the book *ZFS Mastery: Advanced ZFS* to really dive deep into the specifics and to learn a myriad of techniques to further hone your ZFS skills.

I hope you enjoyed this how-to article and that you begin protecting your sensitive data using the native encryption offered by the amazing ZFS filesystem.

ROLLER ANGEL spends most of his time helping people learn how to accomplish their goals using technology. He's an avid FreeBSD Systems Administrator and Pythonista who enjoys learning amazing things that can be done with Open Source technology — especially FreeBSD and Python — to solve issues. He's a firm believer that people can learn anything they wish to set their minds to. Roller is always seeking creative solutions to problems and enjoys a good challenge. He's driven and motivated to learn, explore new ideas, and to keep his skills sharp. He enjoys participating in the research community and sharing his ideas.

Contents

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

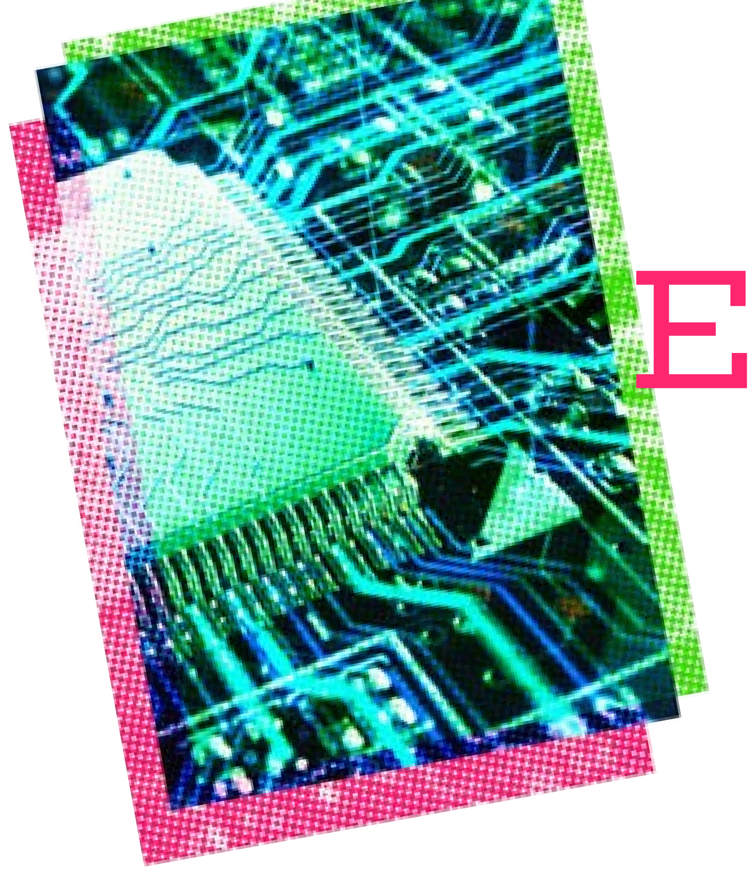
Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™
FOUNDATION



Embedded
FreeBSD

Rolling Your Own Images

BY CHRISTOPHER R. BOWMAN

In the last column, I talked a little about the board I've been using, the Digilent ARTYZ7. In this one, I'm going to talk about rolling your own images. At some point you're going to want an image slightly different from the one you have, and so you'll want to build from source and create an image to write to an SD card.

Let's start by understanding exactly how the ARTYZ7 boots. The Zynq-7000 SoC Technical Reference Manual is a gold mine of technical information, and in Chapter 6, it talks about how the chip and thus all Zynq boards boot. There are a ton of technical details, but based on the way the jumpers are configured by default, the ARTYZ7 boots from the SD card. You may have downloaded and poked around the image I provided in the previous column. If you did, you'll see that it's formatted in MBR mode with a FAT partition first and a second MBR partition with a FreeBSD slice that has UFS on it. The processor will look for an MBR and look for the FAT16 or FAT32 partition. It will look for a file **`boot.bin`** on the FAT partition. If it finds one, it will load that into memory and begin executing that. If you wanted to program to the bare metal, you could write your application and call it **`boot.bin`**. That's a little much for me. Instead, when booting FreeBSD a first stage boot loader is used, and like many embedded boards we use Das U-boot. U-boot for short is an open-source community-maintained boot loader. In our case U-boot is used twice. First, U-boot is compiled as a minimal First Stage Boot Loader (FSBL) which sets up the hardware and then looks for a second stage boot loader which is also U-boot but compiled with a much richer functionality. In our usage, the second stage U-boot lives in the file named **`U-boot.img`** on the FAT partition. This U-boot second stage loader loads the lua loader from the file **`EFI/BOOT/bootarm.efi`** on the FAT partition. The lua loader will then load the kernel, among other things, into memory and execute it.

At some point you're going to want an image slightly different from the one you have.

So, if we're going to build an image, we're going to need to create the partitions and get U-boot installed as well as FreeBSD on an SD card.

Building U-boot is relatively straightforward. While there are many ports for U-boot for various boards, there isn't one for the ARTYZ7. I've created one which you can find HERE. While I haven't managed to get this into the FreeBSD ports tree, you can simply drop this into the **/usr/ports/sysutils** directory in a recent ports tree. Chapter 4.5 of the FreeBSD handbook has really excellent instructions for installing ports via git and building a port. Once you've added the port to your ports tree a simple **make** in **sysinstall/u-boot-artyz7** should cause U-boot to be downloaded and built automatically. After **make install** the files **boot.bin** and **U-boot.img** should appear in **/usr/local/share/U-boot/U-boot-artyz7**. Note: I used to be able to use high "**-j**" values with **make** to use multiple cores to build but that seem to fail on the most recent ports tree (2024Q3)

Building from source is also well documented in the FreeBSD handbook. In Chapter 26.6. Updating FreeBSD from Source you'll find a lot of information on downloading the FreeBSD sources and building from them. If you have the FreeBSD sources install in **/usr/src** you can simply go there and run:

```
# make buildworld
# make buildkernel KERNCONF=ARTYZ7
```

While this should work just fine on the ARTYZ7 boards, after all it has a full FreeBSD install, you should be prepared for it to take a long, um **long** time. As PC hardware has become so incredibly powerful and inexpensive, I use an AMD64 system to host all my files, development environment, and do all my building. FreeBSD has built-in support for cross compiling and building. On my PC I use the following to get my PC to build from sources for my ARM based ARTYZ7 board:

```
# make buildworld TARGET=arm TARGET_ARCH=armv7 -j32
# make buildkernel KERNCONF=ARTYZ7 TARGET=arm \
    TARGET_ARCH=armv7 -j32
```

This will build a cross compiler from AMD64 to ARMv7 and use that compiler to build everything.

For a kernel config file, I've copied the ZEDBOARD config in **src/sys/arm/conf/ZEDBOARD** to **src/sys/arm/conf/ARTYZ7** and I've changed the name in there to match. The **-j32** switch causes the compile processes to use up to 32 processes while doing the builds. On an AMD 5950x PC with a fast SSD it takes about 10 minutes to build world and another 60 seconds or so to build the kernel. I couldn't imagine how many days it would take on the ARTYZ7.

Once you build everything from source the process can diverge in a couple of ways:

1. You can mount an SD card on your host development system and install directly onto the SD card from the host.
2. You can use the **mdconfig** command to create a file-backed memory device. This allows you to treat a file as if it were a block device. You can use all the standard FreeBSD tools to partition the device and mount the partitions in the filesystem. From there you can install as if it were a physical device and, at the end, you're left with a file that is suitable for use with **dd** to copy to an SD card or provide to others
3. The final method, and the one I'm using and will discuss here, is to install to a directory on my host PC and then use **mkimg** and **makefs** to build an image from the host directory which is again suitable for copying to an SD card using **dd**.

Let's look a little more closely at method 3. I typically create an msdos directory and a ufs directory to represent the two partitions I'm going to need on my SD card:

```
# mkdir msdos ufs
```

Next, I do an install to the **ufs** directory:

```
# make installworld installkernel TARGET=arm \
    TARGET_ARCH=armv7 -j32 DESTDIR=ufs
```

You'll also need to run the distribution target which creates all the default configuration files in **/etc**. When doing a source upgrade of a working box, you wouldn't normally run this, as it would overwrite your configuration files, but when building a system from scratch, you do want default configuration files:

```
# make distribution TARGET=arm \
    TARGET_ARCH=armv7 DESTDIR=ufs -j32
```

At this point, I have a complete install in **ufs** and I can go in and do any customization I want to my image. For instance, I can customize **ufs/etc/rc.conf** to automatically bring up the ethernet interface, **cgem0** using DHCP. I can install **ssh** keys into **ufs/etc/ssh** so that the system has the same **ssh** keys all the time instead of having new ones generated on first boot. Since I'm using a host system to do all my building and host all my files, I like to configure **/etc/fstab** to nfs mount my home directory.

I've found it very handy to create a user account so I can log in immediately:

```
# echo 'xxxx' | pw -R ${ufs} useradd -n crb -m -u 1001 \
    -d /homes/crb -g crb -G 1001,wheel,operator\
    -c "Christopher R. Bowman" -s /bin/tcsh -H 0
```

The **-R** option to pw causes pw to edit the password files in the **ufs/etc** instead of my host system. The **-H0** option allows me to use **echo** to pipe my password to pw without having to type it in interactively (you'll need to use the encoded password entry from your host system here instead of **xxxx**). You may also find it handy to modify the root account so that it has a password instead of no password.

Now that I've got the **ufs** directory customized as I want it to appear in my image, let's turn our attention to the FAT partition.

I need to install **boot.bin** and **U-boot.img**:

```
# cp /usr/local/share/U-boot/U-boot-artyz7/boot.bin msdos
# cp /usr/local/share/U-boot/U-boot-artyz7/U-boot.img msdos
```

boot.bin loads **U-boot.img** which emulates EFI firmware for the FreeBSD loader. EFI systems look for a file on a FAT partition called **EFI/BOOT/bootarm.efi** so we'll copy the FreeBSD lua loader there:

```
# mkdir -p msdos/ EFI/BOOT
# cp ufs/boot/loader_lua.efi msdos/EFI/BOOT/bootarm.efi
```

Note I copied the ARM version from our build not the host version which would be AMD64 code.

Now, we've got our two directories **msdos** and **ufs** which contain the files we want on the SD card we just need to create the image file. This is a 4-step process:

```
makefs -t msdos \
  -o fat_type=16 \
  -o sectors_per_cluster=1 \
  -o volume_label=EFISYS \
  -s 32m \
  efi.part msdos

makefs -B little \
  -o label=rootfs \
  -o version=2 \
  -o softupdates=1 \
  -s 3g \
  rootfs.ufs ufs

mkimg -s bsd \
  -p freebsd-ufs:=rootfs.ufs \
  -p freebsd-swap::1G \
  -o freebsd.part

mkimg -s mbr -f raw -a 1\
  -p fat16b:=efi.part \
  -p freebsd:=freebsd.part \
  -o selfbuilt.img
```

The first command builds a file system image file (**`efi.part`**) from the **`msdos`** directory. The second command builds a file system image file (**`rootfs.ufs`**) from the **`ufs`** directory. The third command takes the **`rootfs.ufs`** file and assembles it into a FreeBSD slice with a 1 gigabyte swap partition. The final command bundles our **`efi.part`** and **`freebsd.part`** files into a single image (**`selfbuilt.img`**).

If I insert my SD card into my host I get a **`/dev/da0`** device and a simple **`dd`** will copy the image onto the SD card:

```
# dd if=selfbuilt.img of=/dev/da0 bs=1m status=progress
```

At this point all that's left to do is insert the SD card into the ARTYZ7 board, hit the reset button and watch the glorious boot process. Because of the configuration I did in the ufs partition, I'm able to **`ssh`** into my board as soon as it's done booting and my home directory is nfs mounted. At this point it's time for world domination! Or a beer — it's always time for beer.

CHRISTOPHER R. BOWMAN first used BSD back in 1989 on a VAX 11/785 while working 2 floors below ground level at the Johns Hopkins University Applied Physics Laboratory. He later used FreeBSD in the mid 90's to design his first 2 Micron CMOS chip at the University of Maryland. He's been a FreeBSD user ever since and is interested in hardware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.

Adventures in TCP/IP

Introduction to TCP Large Receive Offload

BY RANDALL STEWART AND MICHAEL TÜXEN

TCP Large Receive Offload (TCP LRO) is a protocol-specific method to minimize the CPU resources used for receiving TCP segments. It is also implementation specific, and this article describes its implementation in the FreeBSD kernel. At any given time, TCP is often used for unidirectional communication, although TCP provides a bidirectional channel. This is the case, for example, if the application protocol using TCP as its transport protocol is of the request/response type like HTTP.

TCP LRO can reduce the CPU resources required in a number of ways including:

- Combining acknowledgments so that a single large stretch acknowledgment is delivered to the TCP stack instead of multiple smaller acknowledgments. This applies to the case where the TCP endpoint is mostly sending user data.
- Combining multiple inbound data segments into one, big, larger piece of data. This helps if the TCP endpoint is mostly receiving user data.
- Bypassing parts of IP stack processing. Therefore, it is useful for TCP LRO to intercept the packets at the network interface layer.

At any given time, TCP is often used for unidirectional communication.

All of these methods are focused on cutting down the number of times the TCP stack gets called and/or minimizing the number of cache misses that the CPU will have to take by compressing all of the processing into one, or a series of packets, processed together. For most all FreeBSD drivers, a single software TCP LRO process is used, though some specific hardware and its drivers do support hardware TCP LRO. This article will discuss only the software TCP LRO in FreeBSD.

Evolution of TCP LRO

The initial implementation of TCP LRO was implemented by Andrew Gallatin in 2006 and was specific to the mxge(4) driver. It was then made generic to all drivers by Jack Vogel in 2008. It had only two focuses:

1. Collecting and merging together small inbound data segments to present a larger, single, inbound data segment to TCP, or,
2. Collecting a number of acknowledgments and presenting one, single, larger acknowledgment to the TCP stack.

Both methods were implemented to cut down on the number of times the TCP receive path was called to save CPU resources. Its implementation was very careful to only handle consecutive segments and ones without TCP options (the only allowed TCP option was the timestamp option). The initial implementation remained pretty much untouched in FreeBSD for almost a decade, except for the addition of IPv6 support by Bjoern A. Zeeb in 2012.

The Addition of Sorting

By 2016 the TCP LRO code was starting to show its age, with the ever faster NICs being deployed on both clients and servers, more and more packets were arriving on each driver interrupt. The initial implementation only allowed for eight different connections to have data collected and compressed. This worked fine in workloads with only a few connections, but was less effective for workloads with a large number of connections, since a driver was sending in far more packets from different connections on each interrupt. With so many more packets from multiple connections arriving in an interrupt, the chances of a single connection seeing packets with small enough interleaving to fit in the eight connection limit grew less and less to the point where TCP LRO was rarely effective, especially for the server case.

By 2016 the TCP LRO code was starting to show its age.

This is when Hans Petter Selasky had a brilliant idea, he added an optional path for a driver to call that would sort the inbound packets before submitting them to TCP LRO. This meant that all packets arriving from each connection could be processed together. Which then meant that you maximized TCP LRO's effectiveness on each interrupt. This change vastly improved TCP LRO performance while still allowing older drivers to remain unchanged.

Packet Queuing

As TCP LRO became more effective, other problems with this more efficient path began to show up including:

a. TCP's congestion control prefers to see every acknowledgment, since an acknowledgment advances its congestion window. Compressing acknowledgments can hamper the congestion control algorithm.
b. Modern TCP stacks often would like to have precise Round Trip Time (RTT) information, compressing multiple acknowledgments can hide this information from TCP.
c. Implementations of TCP ECN needed to see the IP header bits so that ECN signaling from the network can be monitored and reacted to, compressing data or acknowledgments effectively hides this information.
d. If a TCP stack is pacing packets (We will discuss pacing packets in a future column.), then processing a series of acknowledgments when the stack is prohibited from sending out packets increases overhead. This is because the acknowledgment can't send

and yet results in a number of cache misses on the TCP stack during its processing, which will then have to be repeated when the stack is allowed to output.

This set of problems brought about another optimization where the TCP stack enables the TCP LRO code to directly queue packets to it for processing when it next awakens. This then allows all the data in the IP and TCP headers to be processed at a single time when the stack can send out data and reveals all the information (including the timing due to receive timestamps being added either in hardware by the NIC, or in software in the TCP LRO code) that TCP wants to see.

Compressed Acknowledgments

This new queuing mechanism worked well but also caused an additional set of cache misses when a series of acknowledgments arrived. This is because each packet in queue to the stack results in a cache miss when it is processed. In the old compressed scheme, information was lost but superior optimization was performed, since only one cache miss would occur for some number of arriving acknowledgments.

This brought about another TCP LRO optimization. When consecutive acknowledgments arrive, the TCP LRO code can now compress them into a special packet that holds an array of the arriving packet information. This compression technique allows all the previously lost data (including arrival times) to be presented to the TCP stack in the array structure so that only one cache miss is taken to access the special packet. Note that a TCP stack must signal the TCP LRO code that it supports this special type of processing.

Each packet in queue to the stack results in a cache miss when it is processed.

Inner and Outer Headers

The last set of optimizations to TCP LRO have to do with the way inbound IP packets are examined. Originally, only Ethernet frames containing a TCP segment using IPv4 or IPv6 were supported. To support other encapsulations of TCP segments, for example VXLAN which makes it possible to encapsulate an Ethernet frame into a UDP packet, the packet parsing was generalized to support an inner and outer header. This way, packets with UDP as an outer header and TCP as an inner header can be processed by TCP LRO. This assumes that the NIC can do the checksum offloading for both protocols.

Management of TCP LRO

If a NIC driver supports TCP LRO, it can be enabled or disabled using the `lro` or `-lro` parameter of `ifconfig`.

A NIC driver must contain a `struct lro_ctrl`, which contains in addition to other fields a pointer to:

- An array of pairs consisting of a pointer to `struct mbuf` and a sequence number. The number of these pairs is `lro_mbuf_max`.
- A number of `struct lro_entry`. The number of these entries is `lro_cnt`.

The `struct lro_entry` is used to store the information about one aggregated set of received TCP segments. If such an entry is not used, it is contained in the `lro_free` list. When it is used, it is contained in the `lro_active` list and also accessible via the hashtable `lro_hash`.

These two lists and the hash table are also contained in **`struct lro_ctrl`**.

There are two ways for a NIC driver to initialize the TCP LRO specific data. The classical way is to call the **`tcp_lro_init()`** function. The number **`lro_cnt`** of **`struct lro_entry`** which should be allocated, is specified by the loader tunable **`net.inet.tcp.lro.entries`**. When using the classical way of initialization, the array of pairs has no entries. The modern way is to use the function **`tcp_lro_init_args()`** which allows the caller to specify the **`lro_cnt`** and **`lro_mbuf_max`**. This means that the array of pairs might also be allocated.

No matter which way was used for initializing the **`struct lro_ctrl`**, calling the function **`tcp_lro_free()`** frees all allocated resources.

Passing TCP Segments to TCP LRO

The NIC driver has a classical and a modern way of trying to pass a TCP segment to TCP LRO. If passing the TCP segment over to TCP LRO fails, the NIC driver must continue the normal processing of the TCP segment. One reason for TCP LRO to fail is if the NIC was not able to verify the checksums on the received IP packet.

There are two ways for a NIC driver to initialize the TCP LRO specific data.

To use the classical way to pass the TCP segment to TCP LRO, the NIC driver calls **`tcp_lro_rx()`**. Basically this starts the processing done by **`tcp_lro_rx_common()`**, which is described in the next subsection. The modern way to pass TCP segments to TCP LRO, which also requires the modern way of initialization, is to call **`tcp_lro_queue_mbuf()`**. This function just computes a sequence number for the TCP segment and stores it in combination with the TCP segment in the next free entry of the array of pairs. If the array becomes full by this operation, **`tcp_lro_flush_all()`** is called which is also described in the next subsection.

No matter whether the classic or modern way of passing TCP segments to TCP LRO is used, the time when the TCP segment is passed to TCP LRO is saved if there is no hardware receive time from the NIC available.

Processing TCP Segments in TCP LRO

When the modern way of passing TCP segments to TCP LRO is used, one additional initial step is done. **`tcp_lro_flush_all()`** sorts all entries in the array of pairs based on the sequence number field. This results in all TCP segments for the same TCP connection being most likely located next to each other in the array and in the sequence they were received. Then **`tcp_lro_rx_common()`** is called for all the entries in the array. From now on, the processing of the TCP segments is the same, no matter whether the classic or modern way of passing them to TCP LRO is used.

`tcp_lro_rx_common()` parses the TCP segment and uses that information to lookup the corresponding entry of type **`struct lro_entry`** in the hashtable. If such an entry is found, the TCP segment will be added to the packet chain of TCP segments. If no entry is found, a new one is created and the TCP segment is added to the entry. Note that when the TCP LRO code runs out of free entries then an older entry is flushed which then frees up that structure to be reused for the new allocation.

The NIC driver or the TCP LRO code itself can trigger a flush operation, which will result in processing the information in the entries of type **`struct lro_entry`** such that it is suitable to be processed by the TCP stack as described in the next subsection.

Passing Information from TCP LRO to the TCP Stack

If alternate TCP stacks like the TCP RACK or the TCP BBR stack are used, the High Precision Timer System (HPTS) is employed. If only the FreeBSD base TCP stack is used, this is not the case.

If the HPTS is not loaded in the FreeBSD kernel, the following will happen in case a flush operation is triggered: TCP LRO will combine the packet chain for an entry of type `struct lro_entry` into a single large TCP segment by concatenating all the user data of the individual TCP segments. Of course, this only works if there are no gaps or overlaps. If that happens, TCP LRO might only combine smaller parts. The information about the acknowledged data will also be combined and this large, generated TCP segment will be injected into the interface layer. This results in less packets needing to be processed, but results also in the loss of the information when the individual TCP segments were received, as well as any IP-level ECN bits. Depending on the congestion control or loss recovery, this can have a negative impact.

If the HPTS system is loaded, a flush operation results in a lookup of the TCP endpoint. This information is used to determine if the TCP stack used by the TCP endpoint supports mbuf-queueing. If it does not, the same processing as for the FreeBSD base stack is performed. If the TCP stack supports mbuf-queueing, but not compressed ACKs, the packet chain of the entry is copied over to the TCP endpoint and the TCP endpoint might be triggered to process that packet chain. This is what is done when the TCP BBR stack is used which supports mbuf-queueing but not compressed ACKs. If the TCP RACK stack is used, which also supports compressed ACKs, multiple ACKs, which have been received in sequence, can be stored in a special data structure, which allows passing them in a more memory-efficient way. Please note that when mbuf-queueing and compressed ACKs are used, the information from when the individual packets were received is preserved and passed to the TCP endpoint.

Future Evolution

Accurate ECN for TCP is a TCP feature currently being specified by the Internet Engineering Task Force (IETF) and support for it is under development for FreeBSD. In addition to using two new TCP options, it changes the use of two existing TCP flags and makes use of one additional flag. This requires changes to the TCP LRO code to still allow the aggregation of incoming TCP segments for TCP connections supporting Accurate ECN.

The VXLAN support can also be improved to make use of mbuf-queueing.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.

PRACTICAL PORTS

Samba-based Time Machine Backups

BY BENEDICT REUSCHLING

"I wish I'd saved the bandwidth for doing backups for something useful, as I won't ever need them" — said no one ever. In case of disasters — of which there are plenty, backups are an important part of IT resilience. There are failed drives, stolen laptops, broken firmware drivers that render data unreadable, and much more. Having backups and doing them on a regular basis is vital for continuing business operations. And testing them is just as important. But what happens if the backup solution is no longer supported and will not work on newer systems? How are new systems backed up and how do they integrate with the existing solution?

Having backups and doing them on a regular basis is vital for continuing business operations.

I wrote an article about setting up a FreeBSD Apple Time Machine in the March/April 2022 issue of this Journal. I ran that setup for a long time without issues--neither on the backup side nor that I had to restore from it. As time moved on, Apple changed the underlying Apple Filing Protocol (AFP) underneath. In newer versions of macOS (to which I diligently upgrade for security and feature enhancements), the protocol underwent some changes that made my original instructions not work anymore. As described above, its current setup still works, but for new time machine systems that I backup to, it won't work anymore. Apple integrated Server Message Block (SMB, better known as Samba) into the protocol in macOS 10.9 (Mavericks). The migration completed with macOS 11 (Big Sur), which removed the AFP server part and made SMB the new standard, which Time Machine uses internally.

I noticed this when I set up a new time machine backup system. Many moons ago, I had poured my instructions from the 2022 article into an Ansible playbook for easier deployment. The playbook still works, but results in newer macOS versions not being able to mount the exported drive as a storage location for Time Machine. Lucky me, I found that other people had had the same issue and had solved it. The instructions were not as cohesive has I'd hoped, though. The following setup combines different sources — reddit, personal blogs, the FreeBSD forums, and the Samba documentation. I have tested it on two different machines, fleshing out some instructions, and adding missing commands to ensure it works as intended. I kept the original "based on ZFS", because that's what I trust my important data to. You can leave this part out and run it on a different filesystem if you want. However, don't blame me if that does not work out so well!

Requirements

For this setup, you need a machine (FreeBSD in my case) for storing the backups. It should have a decent network connection and fast storage. The storage also needs to be redundant in some way, think RAID1 and levels above it. Capacity-wise, it depends on two factors: how many people will backup their data to it and how much data they have. The Time Machine configuration dialog allows you to set a disk quota and encryption, both of which are good options. ZFS employs quotas and reservations, so I set those on the filesystem level to the same value. Time Machine automatically removes older backups when the available storage space won't fit new backup data. The more storage you have, the longer the backup history that you can potentially restore from.

The encryption part is also nice to have, especially since we're sending the data over networks that may not be encrypted or are part of a VPN. On the receiving system, an encrypted ZFS dataset could be added for the data at rest. However, keep in mind that when the backup target needs to restart, the encrypted dataset is not mounted unless someone enters the passphrase for mounting the dataset. You could configure ZFS to get the passphrase from a file somewhere, but I leave that as an exercise to you in securing access to such a file that holds a passphrase in clear text.

On the sending side, any macOS system will be able to mount and configure the drive, protected by individual user credentials. This allows multiple people to backup up to the same Time Machine. Having beefy bandwidth for this server becomes clear when considering that concurrent backups might run at the same time. I learned that when configuring more than one time machine backup location on a Mac, the system is smart enough to not back up to both at the same time — preventing the system I/O from grinding to a halt doing lengthy backups to two (or even more) locations.

Configuring the Backup Server

A base installation of a FreeBSD release of choice (ideally still supported) with the latest security and errata patches installed goes without explicitly writing about it. We chose not to use a jail here, although nothing is stopping us from doing so. First, install the Samba package:

```
# pkg install samba419
```

Next, we configure our backup storage for ZFS. In my case, I have a dedicated pool aptly named backup and mounted at the same location (`/backup`). I create a separate dataset for Time Machine and set a quota and reservation, since I store other data on it and want to reserve some space for those files as well.

```
# zfs create -o quota=1.5T -o reservation=1.5T backup/timemachine
```

I know I will have two users (Tammy and Tim, Alice and Bob are on holiday) backing up their Macs to that destination. I value both of them equally, so both will get the same space reservations and quotas set. Remember that a quota and reservation on a dataset applies to all datasets below it as well. The 1.5 TB will also apply to their datasets, limiting them already. But it's fine for them to run with 500GB each, so I set a **`refquota`** and **`refreservation`** to only apply to that individual dataset.

```
# zfs create -o refquota=500g -o refreservation=500g backup/timemachine/tammy
# zfs create -o refquota=500g -o refreservation=500g backup/timemachine/tim
```

The two of them will never be able to log into my backup server (they don't care anyway), but they still need to have a user on the system to mount the storage for Time Machine. I ran **`adduser`** for both of them, giving them no home directory (**`/var/empty`**) and also no shell access (**`/usr/sbin/nologin`**).

Run **`chmod`** and **`chown`** commands on their dataset mountpoints to protect those directories from prying eyes.

```
chmod -R 0700 /backup/timemachine/tammy
chmod -R 0700 /backup/timemachine/tim
chown -R tim /backup/timemachine/tim
chown -R tammy /backup/timemachine/tammy
```

The last thing to do is setting a password on the Samba side for those two users. This is the password needed for the prompt on macOS when mounting the time machine backup volume.

```
# smbpasswd -a tim
# smbpasswd -a tammy
```

Avahi Configuration

That's all for required software — straightforward and easy enough. Next we need to create two configuration files. One for the time machine service and the other one for the Samba configuration. The time machine service runs with Avahi — no not the lemurs from Madagascar, the software. This zeroconf network implementation enables programs to publish and discover services (like our time machine) in the local network. The configuration file is XML based, resides in **`/usr/local/etc/avahi/services/timemachine.service`** (create the file when it does not exist yet) and looks like this:

```
<?xml version="1.0" standalone='no'?>
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
 <name replace-wildcards="yes">%h</name>
 <service>
   <type>_smb._tcp</type>
   <port>445</port>
 </service>
 <service>
   <type>_device-info._tcp</type>
   <port>0</port>
   <txt-record>model=RackMac</txt-record>
 </service>
 <service>
   <type>_adisk._tcp</type>
   <txt-record>sys=waMa=0,adVF=0x100</txt-record>
   <txt-record>dk0=adVN=FreeBSD TimeMachine,adVF=0x82</txt-record>
 </service>
</service-group>
```

The file defines which ports to listen on for Samba services (445), how the icon looks for the mounted drive (`RackMac`) and what the display name for it should be (`FreeBSD Time-Machine`). It's fine to change the latter to give it a more descriptive name. I did not change any other parts of this file.

Samba Configuration

Samba is the open source implementation of the SMB protocol and has been around for 32 years. In the beginning, its main purpose was for compatibility between Unix systems and Windows. Active Directory integration, Domain Controller and other functionalities were added as Windows grew those appendixes. Since there is not a single Windows box involved in this setup (you can hear the sound of a sigh of relieve here), we use Samba's file sharing functionality as part of AFP here.

The samba configuration file is under `/usr/local/etc/smb4.conf` and contains this:

```
[global]
workgroup = WORKGROUP
security = user
passdb backend = tdbsam
fruit:aapl = yes
fruit:model = MacSamba
fruit:advertise_fullsync = true
fruit:metadata = stream
fruit:veto_appledouble = no
fruit:nfs_aces = no
fruit:wipe_intentionally_left_blank_rfork = yes
fruit:delete_empty_adfiles = yes

[TimeMachine]
path = /backup/timemachine/%U
valid users = %U
browseable = yes
writeable = yes
vfs objects = catia fruit streams_xattr zfsacl
fruit:time machine = yes
create mask = 0600
directory mask = 0700
```

Two sections (`global` and `TimeMachine`) define the necessary options for the backup destination. The lines prefixed with `fruit:` establish compatibility with macOS. Individual documentation for these lines is in the Samba documentation (see link in References at the end of the article). Change the path line in the `[TimeMachine]` section to the one created earlier on FreeBSD. The %U part is a placeholder for individual user names (tammy and tim in our case) backing up their files. That way, when adding another user later, we do not need to change this line at all. The create and directory masks ensure proper permissions so that the files don't get intermixed and users can't see or change each others backups.

Starting up

The remaining steps boil down to enabling and starting the **dbus** (avahi) and **samba** services.

```
# service dbus enable
# service dbus start
# service samba_server enable
# service samba_server start
```

On the macOS side (the backup clients), go to the finder and press CMD-K (Shortcut for "Connect to Server"). Enter **smb://server.ip.or.dns**. If all goes well, enter the username and password. This is the one for tim or tammy that we entered in the **smbpasswd** dialog earlier. If that succeeds, the share gets mounted into the system. Next, head to the time machine configuration dialog and add a new time machine volume. Make sure to visit the options button in there and check the box for the encrypted backup. You can only set this once before the initial backup and not afterward. You can also limit the amount of disk space the backups should consume, but that is optional, since we did it on the ZFS level already. Next, the long initial first backup can take place. After that finishes, time machine will backup up the Mac to this location by automatically mounting and unmounting the share when reachable on the network.

Summary

That's all. The samba configuration is easy enough to do and users should be able adapt it to their own needs. I found the solution just as reliable as the old one. I've adjusted my Ansible playbooks to use the new Samba-based setup. I still enjoy the fire-and-forget approach to backup up my Mac, knowing that I can pull individual files or the whole installation back when I need to, with the most current files that I had worked with.

References and Sources

- Samba Documentation
- Reddit Post
- FreeBSD Forums Post
- Dan Langille's blog

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.

BSDCan 2024

BY AYMERIC WIBO

A couple months ago, I had the opportunity to attend and speak at this year's BSDCan in Ottawa, one of the three big yearly BSD conferences (the other two being AsiaBSDCon and EuroBSDCon). This was my first time in North America — I hail from Belgium, the land of waffles, beer, and taking 652 days to form a government — so I started off my trip with a couple weeks of touring around the Northeast Corridor before flying out from Boston to Ottawa for BSDCan.

My scheduled talk was on my Google Summer of Code (GSoC) work on porting the BATMAN implementation in the Linux kernel (`batman-adv`) to FreeBSD, and, more generally, its use cases in projects like Freifunk, and the LinuxKPI system on FreeBSD for porting Linux kernel drivers.

BSDCan is the definitive BSD event in North America and welcomes BSD Unix developers, administrators, and users of every level of experience.

Upon arrival at the quite quaint Ottawa International Airport, I was whisked away by bus and Ottawa's light rail to the University of Ottawa where the conference is held. I checked in to the speaker accommodations, the 90U dorms, which were quite nice. The students clearly have it quite good here! I was sharing a room with Kirk McKusick who hadn't yet arrived, so I headed over to the Father & Sons tavern, the de facto hangout spot for BSDCan attendees, to try and get to know some of them. I was arriving at the tail end of the Goat BoF (Birds of a Feather) so there were quite a few people there. I had a couple pints and some hearty poutine and then headed back to the dorms to meet up with Kirk, who I hadn't seen since last EuroBSDCon in Coimbra, Portugal.

On the first two days of BSDCan, tutorials are held in parallel with the FreeBSD DevSummit. I attended the DevSummit, which is where the FreeBSD developers and other guests get together to discuss the goings-on and future of the project. There were also a couple interesting talks, such as Mitchell Horne's presentation of RISC-V hardware and his work supporting it in FreeBSD, or Alex Pshenichkin's quite interesting talk on Antithesis' deterministic hypervisor (the “Determinator”, based on FreeBSD’s bhyve) for software testing and the different considerations in running deterministic virtual machines that aren’t immediately obvious at all.

During the DevSummit, there’s also a moment set aside for “Next Release Planning”. This is where everyone pitches in and discusses the features to enter (or be axed from) the next release of FreeBSD (15.0 in this case). These features are categorized into completed in-tree features, completed features that have yet to be upstreamed, features that are being worked on, features that really need to be worked on, and features that would be nice-to-haves but aren’t priorities. My very small contribution to this list was S0ix idle support which is necessary for sleeping on newer CPU’s (including AMD Framework laptops), for which work was started but subsequently abandoned a few years ago. I see focusing on consumer hardware support as an important way to get FreeBSD into the hands of more people, and for them not to be turned away by X or Y feature not working on their shiny new laptop, and I’m glad to see that this sentiment seems to be echoed by others in the project.

The next two days were the main conference days. I attended Kirk’s talk on secrets to FreeBSD’s success and Shawn’s State of the Hardened Union (with whom I talked a couple times during the conference with regards to BATMAN and open wireless networks) before presenting my own. It was the first time I’d given a proper talk, so I didn’t really know what to expect or if I’d be comfortable speaking about a technical subject in front of an audience. In sum, I’m quite happy, but I did speed-run my talk a bit, and feedback I received after the talk did affirm that I was going much too fast. Something for me to work on for next time ;) I do think I managed to get my main points across, though, and I was happy with the questions I received afterwards.

During the DevSummit, there’s also a moment set aside for “Next Release Planning”.

On the last day, I attended Warner Losh’s talk on allowing people to contribute to FreeBSD through GitHub, which I think is very important for new contributors as it lowers the barrier to entry vs. learning how Phabricator works, especially for greenhorn developers who have learned their craft exclusively through tools like GitHub. In fact, when I first tried to submit my first FreeBSD contribution, it was through GitHub, before understanding that was not the preferred way to do it. The last talk I attended was on Sheng-Yi Hong’s work on kernel debugging with LLDB, which is something that I’m personally looking forward to. Sheng-Yi Hong was a fellow GSoC student during the year I was working on BATMAN, so it was great to meet him in the flesh and chat with him. He’s a cheerful and upbeat guy, and I’m looking forward to meeting him again at future conferences!

After the closing session and the ritual auction was the social event at Sens House, which is one of the things I enjoy most about these conferences. I feel like I almost get more value out of the social aspect of conferences than the talks themselves; it’s the opportunity to meet the people you only know through their handle or their patch notes. I have made many friends and been exposed to a lot of different use cases and perspectives regarding FreeBSD and other software through these interactions that I likely wouldn't have had otherwise.

The next day, after an insanely copious breakfast at Father & Sons, Kirk and I packed up and headed to the airport as we had a flight to Chicago, where we were meeting up with Eric Allman (who sadly couldn't make it to BSDCan this year) at O'Hare.

Shortly after, we learned of the very unfortunate news that, after the last day of the conference, Michael Karels had passed away. Mike was a very important and beloved figure in the development of the 4.4BSD-Lite release, which all modern BSD's trace back to. I'm very grateful to have had the opportunity to meet him at BSDCan, and I wish his family and friends all the best in these difficult times.

After a day of sight-seeing in the Windy City, we took the 3-day California Zephyr train to Berkeley, where I stayed for a few weeks before it was time for me to fly back home, and that was the end of my BSDCan adventure.

Overall, the conference and its general organization were quite excellent, from the catering to the AV to the surrounding events. I'd like to extend a huge thank you to the people who broke their backs behind the scenes to get this all done and done extremely smoothly and professionally at that. BSDCan covered my travel expenses and handled everything related to accommodation on their side, which really helped to alleviate some of the major points of stress of a trip so far away from home. The talks were engaging, I got to meet a bunch of wonderful and interesting people, and the beer at Father & Sons was cold.

If you have a subject you'd like to talk about, I really recommend submitting a paper. I'm looking forward to attending again next year!

AYMERIC WIBO is a CS student at UCLouvain in Belgium and has been using and developing projects based on FreeBSD since high school. His primary interests lie in graphics and networking..

Write For Us!

Contact Jim Maurer with your article ideas.

(maurer.jim@gmail.com)

2024 Events Calendar

BSD Events taking place through November 2024

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.

September 2024 FreeBSD Developer Summit

September 19-20, 2024
Dublin, Ireland
https://wiki.freebsd.org/DevSummit/202409

The September 2024 FreeBSD Developer Summit will be colocated with EuroBSDCon 2024, which takes place in Dublin, Ireland. This is a by-invitation event. FreeBSD committers will be welcome to register themselves using this wiki; non-committers must be sponsored by a committer to attend. Attendees must also attend EuroBSDcon 2024 to access all devsummit activities.

EuroBSDCon 2024

September 19-22, 2024
Dublin, Ireland
https://2024.eurobsdcon.org/

EuroBSDCon is the International annual technical conference held in a different European country each year. It focuses on gathering users and developers working on and with 4.4BSD (Berkeley Software Distribution) based operating systems family and related projects. The FreeBSD Foundation is pleased to again be a Silver Sponsor.

All Things Open

October 27-29, 2024
Raleigh, NC
https://2024.allthingsopen.org/

All Things Open is the largest open source/open tech/open web conference on the East Coast and one of the largest in the United States. It regularly hosts some of the most well-known experts in the world, as well as nearly every major technology company. FreeBSD is proud to be a non-profit partner for this year's All Things Open.

OpenZFS User and Developer Summit 2024

October 26-29, 2024
Portland, OR
https://openzfs.org/wiki/OpenZFS_Developer_Summit

The twelfth annual OpenZFS Developer Summit and first annual OpenZFS User Summit will be held in Portland, Oregon, USA Oct 26-29 (Sat-Tue), 2024.
User Summit Themes include:

- Storage LAN and WAN Networking for OpenZFS
- Mixed Operating System OpenZFS Environments
- Machine-readable and writable OpenZFS including JSON, SNMP, REST
- Security, Encryption, and MPAA TPN Compliance
- Channel Program Workflows
- Extreme OpenZFS including All-Flash, DRAID, Special Allocation Classes
- Candidate theme: Continuous Replication Brainstorming
- Candidate theme: User-space OpenZFS
- Candidate theme: OpenZFS DMU Object Storage

The Foundation is pleased to be a Bronze Sponsor.

Fall 2024 FreeBSD Summit

November 7-8, 2024
San Jose, CA
https://freebsdfoundation.org/news-and-events/event-calendar/fall-2024-freebsd-summit/

The FreeBSD Summit is an annual event to bring the community together to learn, network, and drive FreeBSD use. Each year, the event gathers FreeBSD users, including decision makers, software engineers, and individual contributors and users, to share best practices and successes in their use of FreeBSD. The FreeBSD Summit also provides the unique opportunity to discuss issues with the developer community in person.

Please note: For the Fall 2024 event, the name is changing from "Vendor Summit" to "FreeBSD Summit" to better represent the audience and prepare for its growth in future years.

Registration will open in late September.

Thank you to our Venue Sponsor, NetApp and our Gold Sponsor, the FreeBSD Foundation!