FreeBSD® JOURNAL

May/June 2022

Disaster Recovery

Building a Resilient Private Cloud with FreeBSD

LLDB 14—The New Debugger for FreeBSD

Pragmatic IPv6

Post-Mortem Kernel Debugging with netdump(4)

JUNIPER NETWORKS | Driven by Experience™

An ambitious company for ambitious people.

Juniper is changing what's possible in networking. We're going beyond building the networks customers expect—we're building the networks customers deserve. And the world is taking note. But to continue to excel, we have work to do. Change in our industry is accelerating. To power connections and empower change, we need radical thinkers, eternal optimists, and energized personalities. We need people like you.

The Junos Core Kernel team is looking for an ambitious Software Engineer with experience and passion for Kernel/Operating System technologies Successful candidates will support and enhance the FreeBSD operating system. The team is responsible for designing networking/driver domains of our products.

https://juni.pr/FreeBSDEngineering

FreeBSD® JOURNAL

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

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Design & Production • Reuter & Associates

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) 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: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

Welcome to the May/June issue of the *FreeBSD Journal*!

Computers are useful tools that can automate many tasks. Sometimes, however, things can go sideways whether due to natural disasters, hardware failures, software bugs, or misconfigurations. (I bear the scars from at least one of each.) Daniel Bell offers a strategy that can be used to build a system that is resilient in the face of such failures.

While restoring functionality is typically the first order of business during or after a disaster, it is also important to diagnose the causes of the original failure. In the case of software bugs, a debugger is the primary tool to use. Michal Górny and Kamil Rytarowski introduce the newest version of the LLDB debugger from the LLVM project. Mark Johnston discusses kernel debugging over network connections, in particular, how to save crashdumps over a network connection rather than to a local disk.

In a different vein, Hiroki Sato begins a series of articles describing the configuration and use of IPv6 on FreeBSD.

Recently, the editorial boards of the Journal met to plan out issues for the coming year. We had a lively, productive and far-ranging discussion covering topics in the FreeBSD universe. Starting with the September/October 2022 issue, the Journal will be featuring issues focused on Security, Observability and Metrics, Building a FreeBSD Web Server, Embedded, FreeBSD at 30, and Containers & Cloud. We also plan on publishing an issue on FreeBSD 14, timed to coincide with the release of 14.0.

We love to hear from readers. If you have feedback, suggestions for a future article, or are interested in writing for the Journal, please email us at info@freebsdjournal.com.

John Baldwin
Member of the FreeBSD Core Team
and Chair of *FreeBSD Journal* Editorial Board

FreeBSD® JOURNAL

May/June 2022

Disaster Recovery

5 **Building a Resilient Private Cloud with FreeBSD**
By Daniel Bell

15 **LLDB 14—The New Debugger for FreeBSD**
By Michał Górny and Kamil Rytarowski

22 **Pragmatic IPv6** (Part 1)
By Hiroki Sato

31 **Post-Mortem Kernel Debugging with netdump(4)**
By Mark Johnston

3 **Foundation Letter**
By John Baldwin

37 **WIP/CFT: FreeBSD Boot Performance**
By Tom Jones and Mitchell Horne

42 **Practical Ports**
Setting up an NFSv4 Fileserver on Open ZFS
By Benedict Reuschling

47 **We Get Letters**
Disaster Recovery Plan
by Michael W Lucas

50 **Events Calendar**
By Anne Dickison

Building a Resilient Private Cloud with FreeBSD

A case study of a managed service provider infrastructure using tools available in FreeBSD base

BY DANIEL J. BELL

"Oh yeah, definitely. We've had data breaches. I've seen clients lose data." A potential client was asking about my organization's track record of data protection, comparing my little IT team of a half-dozen people with much larger providers, and they weren't happy with my response. Apparently, "We have never had a breach or lost client's data," is a common and rather silly part of the big managed service providers' sales pitches here in Manhattan. I went on to explain that we've helped many of my clients recover from data loss, and everyone sees a breach or a failure sooner or later. Never suffering data loss, as experienced system administrators know, only means you haven't been around long enough to deal with one. The most important thing to do is to be prepared.

Being prepared means having a great infrastructure, which doesn't mean just blindly flinging all our workloads in the cloud. Along with the platitude about never suffering data loss, technical salespeople will often sing a jingle about how their singular dedication to the cloud means immunity to both security and data loss problems. On the contrary, my most recent examples of client data loss were from popular cloud services, and further destruction was only mitigated because of our bare metal infrastructure.

Everyone sees a breach or a failure sooner or later.

For example, one morning I woke up to alarms of heavy disk IO coming from a mid-sized medical office where we use our FreeBSD infrastructure. A company-wide share on a Windows fileserver, a bhyve VM on ZFS, was full of ransomware-encrypted files. As I investigated the servers, my team found the culprit workstations and shut them down to stop the spread. Once everything was contained, the repair itself took only a few minutes, including the time it took for the server to reboot. (See the section on Restoration for example commands.) It felt like a heroic repair, but most of the client's staff hardly noticed. And that's exactly what we were shooting for.

Unfortunately, one of the infected workstations was a shared machine and contained a "cloud drive" that we didn't know about. A medical assistant's personal files there were dutifully synced to the cloud, the bits encrypted forever with a text note on how to pay the ransom. The cloud granted us two perfectly identical directories of garbage. It was a good reminder to everyone involved that one synced copy, no matter how cloudy, is ever enough.

Running our own servers for our dozen clients means that we have a greater level of certainty and flexibility, and it didn't require an enormous server fleet to see both a better defense against data loss and an enormous cost savings versus identical cloud workloads. We use leased and owned servers running FreeBSD in multiple locations in the U.S. and run nearly everything our clients need, including databases, fileservers, remote desktop environments, and complex proprietary web applications. Our host environments are designed to be as simple and replaceable as possible, requiring nothing outside of the FreeBSD base environment to provide an effective and resilient private cloud environment. Using jails for UNIX workloads, bhyve for commercial VMs, and everything organized into ZFS volumes, our network of FreeBSD hosts was easily tuned to be a secure and redundant cloud alternative right out of the box. Here's how we do it.

Preparing Our Infrastructure

Evaluating when [not] to use bare metal

As our client base grew during the early 2010s, we were relying more on the cloud and noticed both ours and our clients' bills creeping up by thousands of dollars per month. A few years ago, I became aware that bhyve, the hypervisor in the FreeBSD base system, ran Windows and just about everything else solidly on most new hardware. My organization could now host almost all of our tenants' workloads on bare metal servers running the FreeBSD base—and FreeBSD has been my weapon of choice since the 90s. Until bhyve was ready for prime time, we still relied heavily on FreeBSD jail hosts and cloud VPSs, but most of our hardware infrastructure used VMware eSXI. Our hardware was aging and becoming tedious to maintain, so it was a perfect time for a change. We evaluated what we could accomplish with a FreeBSD hosting plan, along with some investments in hardware, datacenters, and leased bare metal providers.

It didn't require an enormous server fleet to see both a better defense against data loss and an enormous cost savings versus identical cloud workloads.

The projected cost savings was immediately obvious. We compared the three-year total cost of ownership of a VPS, such as a DigitalOcean Droplet, against *two* equivalent leased or purchased bare metal servers. We estimated that the leased option costs about half as much compared to equal resources in the cloud, and owning the servers would cost less than a quarter of the pure cloud options. In this analysis, we were careful to include liberal estimates for hardware maintenance labor, kilowatt hours, network equipment (FreeBSD routers, of course), and everything else our infrastructure needed to provide at least as much juice as we were getting from the cloud; there really was no contest.

Of course, there are still areas where the mega corporations can effectively compete with small FreeBSD outfits like ours. Regulatory compliance for specific industries, like the U.S. gov-

ernment's FedRAMP assessment program, is too expensive to certify. In workloads requiring datacenters with these certifications, we were stuck running VPSs at major providers that can afford those certifications. Also, some shared services would require a pretty huge infrastructure to create cheaply, such as CDNs, mail delivery, and BIND secondaries (we often use StackPath, AWS SES, and EasyDNS, respectively, for these services). They're much cheaper and less labor intensive to outsource at my organization's scale, and we're happy to work with them.

Designing our network

One of the big benefits of large public cloud providers is that it's easy to create and destroy resources in multiple locations, so everything we self-host needs to be doubled-up and geographically distributed as well. For example, we keep one rack in a datacenter close to my home in New York for convenience, and some in Texas where the kilowatts are cheap. The datacenters we use in California, Florida, and Texas have excellent 24/7 remote hands available and can spin up leased servers, hand us network KVM access, and sell us spare parts when we need them. With this flexibility, we know that we can recover from some of the worst types of hardware failures without owning everything we might need in advance. I've tested these teams, and I was able to get a fresh FreeBSD system running in two hours or faster. And of course, the big cloud guys are still out there in a pinch if we really need them.

We also link all our datacenters, leased servers, and even third-party cloud servers by a management VPN. We recently switched to a WireGuard mesh, which is a snap to configure, easy to scale, and a welcome newcomer to the FreeBSD kernel.

To keep track of everything, we link our documentation database to our private DNS so names of hosts, jails, and VMs are in a predictable unique format, `Function##.Client`. This helps us keep all our management, monitoring, and volume naming easy and intuitive. We store additional information using aliases (CNAME) and text (TXT) records to pull additional information and feed monitoring and maintenance automations. For example, `Function##.Client` will resolve into `Function##.Client.Site`, so we can look up any instance's datacenter code:

```
% host host12.dndrmfln
host12.dndrmfln is an alias for host12.dndrmfln.scr1
host12.dndrmfln.scr1 has address 10.10.10.100
% host fs1.waynecorp
fs1.waynecorp is an alias for fs1.waynecorp.gtm2
fs1.waynecorp.gtm2 has address 10.20.20.200
```

Host configurations

Along with network object names, we try to keep all our hardware and software configurations as consistent as possible between our hosts, which means they'll be quickly replaceable without too much worry. The common grim metaphor is that we want our servers to be more like cattle and less like pets, so we can feel less concerned when one needs to be put out to pasture. There's some difference in specific CPU and memory loadouts, which are based on workload requirements, but as a rule of thumb, we try to optimize performance with SSD mirrors for workload-bearing data drives and HDD raidz2 for backup-focused hosts.

We keep zpool names completely unique for a thin additional veneer of safety. For example, a bare metal server named host12.bts might have pools named boot12, ssd12, or rust12 to

denote boot, flash, and HDD pools, respectively. It's a nice sanity check that has prevented me from copy/paste clobbering the wrong dataset on the wrong server. We always use an SSD boot mirror with the default zpool root structure from the FreeBSD installer. We avoid installing guests on our root pool. For our data zpool, we create volumes representing our major host functions:

- **datapool/jail:** Our jails usually contain a FreeBSD base, but we sometimes use simpler chroot environments or Linux bases.
- **datapool/vm:** bhyve VMs. Each dataset contains configuration notes and bhyve logs, and sub datasets include zvols representing the VM's virtual drives, e.g., `datapool/vm/guest.client/c-drive`. This intuitive structure is compatible with `vm-bhyve`.
- **datapool/Backup:** These are ZFS replications of a backup partner that are updated at least daily. We also may contain `rsync` and `rclone` backups, e.g., from Microsoft OneDrive or other non-ZFS environments, which we snapshot as well.
- **datapool/Archive:** We move a dataset to "Archive" if there are no active replications. For example, if an instance is retired, we'll use `zfs rename` to move it here.

All other configuration is as simple as possible and as identical as possible with the goal that all our VMs and jails will boot quickly on their backup hosts. We limit our usage of packages to monitoring, management, networking, and quality-of-life packages like shells that won't significantly complicate management if a package were to break. For example, we sometimes use the `vm-bhyve` package to simplify execution of our `bhyve` and `bhyvectl` commands, but we have a contingency plan to do without them in a pinch.

The most frustrating wrench in a fast recovery of an instance is a mismatch between network object names, which is easy to break with or without jail/vm managers. For example, we've had several recovery hiccups because `bridge0` connected to a LAN interface on one server and WAN on its recovery host. We used to rename our bridges and epairs to descriptive names such as `lan0` or `vm22a` but found that to be tedious and ultimately unhelpful as our fleet grew. Finally, we settled on a fixed structure for virtual network objects such as:

- Bridge names:
  - bridge0: LAN
  - bridge1: WAN
  - bridge2: Virtual network
  - bridge3: Client VPN
- epair and tap device names: We try to keep these documented but have been most successful using the number to match their last ipv4 octet, e.g., epair201 or tap202. vm-bhyve, the jail `jib` command, and other tools can help with managing these, but we still find it helpful to reliably know which interface belongs to which instance.

Of course, if an off-site backup partner has a different network infrastructure or IPs need to change on recovery, some of this preparation will have to be adapted and documented. We do our best to break and test our recoveries regularly so we can meet our recovery objectives.

Server for durability and redundancy

As the old saying goes, "two is one and one is none," so most of our bits are on four physical machines at any one time. We keep at least two ZFS replicated backups of everything, plus hot or warm application-level backups depending on the type of workload and our recovery point and recovery time objectives. Of course, the warm backups are provisioned with enough resources to operate all of the failovers they're responsible for (or else they could not be consid-

ered particularly balmy). Lastly, we keep an additional copy of everything kept on big old rust buckets with a much more conservative pruning strategy.

Thankfully, we can trust that ZFS replication will ensure everything is backed up with cryptographic certainty, but beyond additional copies we also need to make sure a successful attack on one server can't cascade damage to others. To help protect ourselves, we always use `zfs allow` to run limited-privileged automatic replication processes. We practice the most extreme security measures for tertiary backups to the aforementioned rust-bucket. There is no direct Internet or VPN access allowed to this machine at all; it can only be managed locally at the office.

Maintenance of Our Cloud Environment

Snapshots

Of the myriad excellent ZFS snapshotting utilities, I prefer the simplest ones like `zfs-periodic`. More recently, we've been mimicking the `zfsnap2` readable snapshot naming format, which tells us everything we need to know right from `zfs list: @Timestamp--TimeToLive`. A `zfsnap2` snapshot name for right now with a one-week (suggested) retention policy is just the following:

```
TTL=1w
NOW=`date -j +%Y-%m-%d_%H.%M.%S`
SNAPNOW=$NOW--$TTL
```

For me right now, that's `2022-04-08_16.49.48--1w"`, which is easy to read in a `zfs list`. `zfsnap2` saves us a few keystrokes for snapshotting, and it also has good pruning features based on the TTL values. One of the first things I do when I set up a host is jam these commands right into root's crontab.

```
VOLS="boot02 rust02/vm rust02/jail"
0  0 * * *  echo $VOLS | xargs zfsnap snapshot -ra 1w
10 0 * * 0  echo $VOLS | xargs zfsnap snapshot -ra 1m
20 0 1 * *  echo $VOLS | xargs zfsnap snapshot -ra 1y
30 0 1 1 *  echo $VOLS | xargs zfsnap snapshot -ra forever
0  1 * * *  echo $VOLS | xargs zfsnap destroy -r
```

In this example, the **VOLS** will get daily, weekly, monthly, and annual snapshots with a retention policy of one week, one month, one year, and forever, respectively. On our rust-buckets, we prune more carefully and less frequently for good measure.

Replication

Since our backups are our last line of defense, we try to practice our best security protocols here. We always use pull replication because we want each server to have as small of an attack surface as possible. For example, our dedicated backup hosts have no Internet traffic forwarded to them at all.

For an additional minor security improvement, we never `ssh` to the root user on our backup source. I don't have anything against using root for replication, but it's so easy to compartmentalize ZFS functions with `zfs allow` that we're happy to have that little extra peace of mind. Unfortunately, it's not quite as easy for the `zfs receive` side; the receiving user will need ev-

ery non-default ZFS property permission used, or the replication will throw errors or fail. We found a good compromise by running our first zfs receive operation as root, and then running the regular backup scripts as an unprivileged user. Although there are myriad great replication scripts available for FreeBSD, I wanted to learn the process precisely—and then I just ended up using my own scripts. Here's an example, based on my homegrown replication scripts, for backing up a VM called `drive.client`.

On the host pulling the backups, we'll set up our backup user to replicate to `host12rust/Backups` and make us an ssh-key:

```
pw useradd backup -m
su backup -c 'ssh-keygen -N "" -f ~/.ssh/id_rsa'
cat ~backup/.ssh/id_rsa
zfs create -o compression=zstd host10rust/Backups
zfs allow -u backup receive,mount,mountpoint,create,hold host10rust/Backups
```

On the source host, we make the same backup user and grant it the permissions to send us snapshots:

```
pw useradd backup -m
cat >> ~backup/.ssh/authorized_keys
[PASTE THE KEY HERE]
zfs allow -u backup send,snapshot,hold host05data/jail
zfs allow -u backup send,snapshot,hold host05data/vm
```

We can do the rest of the work from the backup host. There are lots of `zfs send` and `receive` choices, but the only thing we can't live without is `-L` to ensure we get block sizes matching our source. We also like `-c` to send the stream compressed as-is, for lower bandwidth, but we can also omit it to reapply a heavier compression setting on the target volume. We also like forcing `canmount=noauto` to avoid the risk of active, overlapping mounts. This adds a mounting step in recovery, but I think it's worth it. Our first replication, fired as root, looks something like this.

Figure out our latest snapshot:

```
REMOTE="ssh -i ~backup/.ssh/id_rsa backup@host05"
SOURCE="host05data/jail/drive.client"
TARGET="host10rust/backups/drive.client"
SOURCESNAP=`eval $REMOTE zfs list -oname -Htsnap -Screation -d1 $SOURCE | head -1`
eval $REMOTE zfs send -cLR $SOURCE | zfs receive -v -u -x atime -o canmount=noauto
$TARGET
```

Building our script from the above, we can run subsequent replications as our backup.

```
TARGETSNAP=`zfs list -oname -Htsnap -Screation -d1 $TARGET | head -1`
ssh -i ~backup/.ssh/id_rsa backup@host05 zfs send -LcRI $TARGETSNAP $SOURCESNAP |
zfs receive $TARGET
```

Due to shifting schedules or oversights, a snapshot inevitably gets out of sync from time to time and a replication will fail. To figure out the most recent matching snapshot between two datasets, we run a script that does the following:

```
zfs list -oname -Htsnap -Screation -d1 $SOURCE | awk -F@ '{print $2}' > /tmp/
source-snap
zfs list -oname -Htsnap -Screation -d1 $TARGET | awk -F@ '{print $2}' | grep -f /
tmp/target-snap
```

We have much beefier homespun awk replication scripts that take care of all the above for us, including retrying and fixing broken -R replications (e.g., if there are different child snapshots due to a recovery), and monitoring/reporting. The reporting is full of emojis.

Migration and restoration with ZFS

If the virtual interface names are already consistent with the source, all we need to do is check and start the instance. Here's our checklist to make sure a backup partner is ready to roll when duty calls.

✓ The backup server's virtual networks are prepared and ready to adopt the guest, as well as any network management software we need, such as VPNs or `dhcpd`.
✓ Any other guest configuration files are up-to-date and readily available. For jails, we like using the format `/etc/jail.guest_name.conf` so it's usually quickly and painfully obvious when a configuration is missing.
✓ The backup server has the correct amount of resources and sysctl settings, e.g. Linux support.
✓ Our replications are running on the proper schedule: daily, hourly, or quarter-hourly.
✓ We've documented and tested our process and our collaborators know what to do.

If everything is ready to go and tested, the recovery or migration process will be painless:

- If the source hasn't crashed, power it off, take one more snapshot, and replicate it one last time. We have a script written for these stressful moments tuned for fastest possible replication with no intermediate snapshots. See the "Tips and Tricks" section for more details.
- Next, we replicate, move, or clone our snapshot into the production location. It's quickest and easiest to rename the snapshot into the production location for active guests, for example:

```
zfs rename data1pool/Backup/guest.client data1pool/vm/guest.client
zfs mount data1pool/vm/guest.client
```

- Finally, we double-check the guest's configurations launch it.

We often recover data using ZFS clones, which are also a great way to test iterative changes to guests, e.g., testing database upgrades before running them in production. If the clone is going into production, we always zfs promote it when convenient to avoid later dependency problems. In the ransomware recovery example at the beginning of the article, we used a clone to ensure there was no risk of losing any good data after the time of the recovered snapshot:

```
zfs rename data1pool/vm/fs.cli/d-drive data1pool/Backup/fs.cli-d-drive-ransom.
zfs clone data1pool/Recovery/fs.cli-d-drive-ransom@last-night data1pool/vm/fs.cli/
d-drive
[…after hours…]
zfs promote data1pool/vm/fs.cli/d-drive
```

If the old host is accessible, we can rename the migrated dataset into the `pool/Backup` dataset, and flip-flop the backup process.

Tips and Tricks

Migrating instances faster

No matter how prepared I am, there are still situations when I need to perform a rapid replication to a host while under the gun. In these special cases and when it's safe to do so, such as when a trusted switch or VPN is involved, we can drop our encrypted ssh pipe for a little more speed.

Unfortunately, if we use **`zfs send -R`** to pick up child datasets, it will send all child snapshots, which probably isn't the best idea in a pinch. (In the old Oracle ZFS documentation, there was **`zfs send -r`** command, which could be used to only replicate the latest recursive snapshots, but unfortunately this hasn't yet made it into OpenZFS).

Here's a quick one-liner to get a list of the newest snapshots recursively that can be used:

```
VOL="pool/vm/guest.cli"
zfs list -Hroname $VOL |xargs -n1 -I% sh -c "zfs list -Honame -tsnap -Screation %
| head -1"
```

Next, we can user the output of the above to create a network pipes with nc:

```
zfs send -Lcp sourcepool10/vm/guest.cli@2022-02-19_00.19.77--1d | nc -Nl 60042
```

And on the target, we attach to the above pipe:

```
nc -N source 60042 | zfs receive -v localpool/jail/guest.cli
```

And repeat the previous two pipe commands with any child volumes. In our example, we used "nc -N" to close the socket when the stream completes and use the **`zfs send -c`** option to send the replication stream in its currently compressed state. We also sometimes add a compressor to the pipe as described below.

Escaping someone else's cloud (or another hypervisor)

We use a similar technique to copy a remote volume into an image file or zvol. For example, this is great for moving a VM from a cloud provider or another hypervisor into bhyve in a single step, rather than spending more time and space downloading and converting the volume in multiple steps. It's especially handy for evacuating "cloud appliances" that aren't yet available in a raw, `bhyve` friendly format.

To get started, we disable the cloud-init and all provider-specific startup scripts (YMMV if you use cloud-init). Though we've successfully cloned active VPSs in place, corruption is likely if the source volume is mounted. A much better choice is booting from a FreeBSD ISO if the cloud provider allows. If not, then we make a copy of the source volumes and attach them to another VPS. For example, in AWS, we can snapshot target volumes, convert the new snapshots to volumes, and then attach those volumes to a run-

A much better choice is booting from a FreeBSD ISO if the cloud provider allows.

ning host. Note that the examples will need to be modified based on operating system. For example, Linux's stock `dd` has slightly different options, and the sending OS might not have `zstd` (`gzip` and `gzcat` are good alternatives). Be careful not to use a compression level so powerful that CPU time becomes your transfer time bottleneck.

On the source, possibly adjusted for OS differences and device names:

```
dd if=/dev/ada0 bs=1m | zstd - | nc -Nl 60042
```

On our target FreeBSD host:

```
nc -N source 60042 | zstdcat | dd of=ada.img bs=1m status=progress
```

For guests that boot with UEFI, it will be easy enough to move the boot loader file into the right place if `bhyve` doesn't find it automatically. If the VM uses `grub`, it might take a little more elbow grease to make everything line up properly. Here's the `grub-bhyve` command for a CentOS VM I recently released from the clutches of big cloud:

```
echo '(hd0) /dev/zvol/ssd11/vm/pbx.bts/vda' > device.map
grub-bhyve -m device.map -M 8G -r hd0,1 -d /grub2 -g grub.cfg pbx3.bts
[usual bhyve commands]
```

Conclusion

Your disaster recovery plan will come from your team, not from any amount of cloud resources; we have to hire, retain, and train the right talent. In my opinion, we can have it all. Keep the large cloud providers for the lightweight scaling solutions they excel at, and for everything else use a rock-solid foundation of FreeBSD bare metal servers running ZFS, bhyve, and jails. Take your data seriously and save money doing it.

DANIEL J. BELL is the founder of Bell Tech, a small managed service provider that has been operating in New York City for over 20 years. He prioritizes privacy, security, and efficiency by utilizing a unique mix of cutting-edge technologies along with bulletproof, tried-and-true standards.

FreeBSD® JOURNAL

The FreeBSD Journal is Now Free!

Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!

2022 Editorial Calendar

- Software and System Management (January-February)
- ARM64 is Tier 1 (March-April)
- Disaster Recovery (May-June)
- Science, Systems, and FreeBSD (July-August)
- Security (September-October)
- Observability and Metrics (November-December)

Find out more at: freebsdfoundation/journal

The New Debugger for FreeBSD

BY MICHAŁ GÓRNY AND KAMIL RYTAROWSKI

For the last decade, FreeBSD underwent a major effort of replacing the various components of its tool chain with more modern, permissively licensed programs. A part of that effort was replacing the aged GNU GDB debugger with LLDB, the debugger of LLVM project. Moritz Systems took part in that effort by taking up a few projects to modernize and improve the FreeBSD support in LLDB, as well as implement missing features.

LLDB 14 is a culmination of the work done so far. The March LLVM tool chain release includes a number of improvements that bring LLDB closer to a fully featured replacement for GDB. The debugger features FreeBSD support on amd64, arm, arm64, i386 and powerpc. It uses a client-server layout that provides uniform support for both local and remote debugging. In addition to the provided lldb-server, other protocol stubs are supported such as the ones provided by QEMU emulator or the FreeBSD kernel. Multithreaded programs are fully supported, as well as the most common multiprocessing scenarios, with more work underway. In addition to that, FreeBSD kernel debugging support akin to KGDB is provided.

This article details some of the more interesting aspects of LLDB's architecture and its features. However, prior to diving into the specific details, it probably makes sense to start by discussing some of the basic principles on how debuggers are implemented on Unix derivatives such as FreeBSD.

Debugging on Unix Derivatives

Debugging userspace processes on many Unix derivatives, including Linux, FreeBSD and other BSDs is implemented through a combination of ptrace(2) system call and signals. The former is generally used to control the traced process and obtain additional information about its state, while the latter is used to asynchronously report events to the debugger.

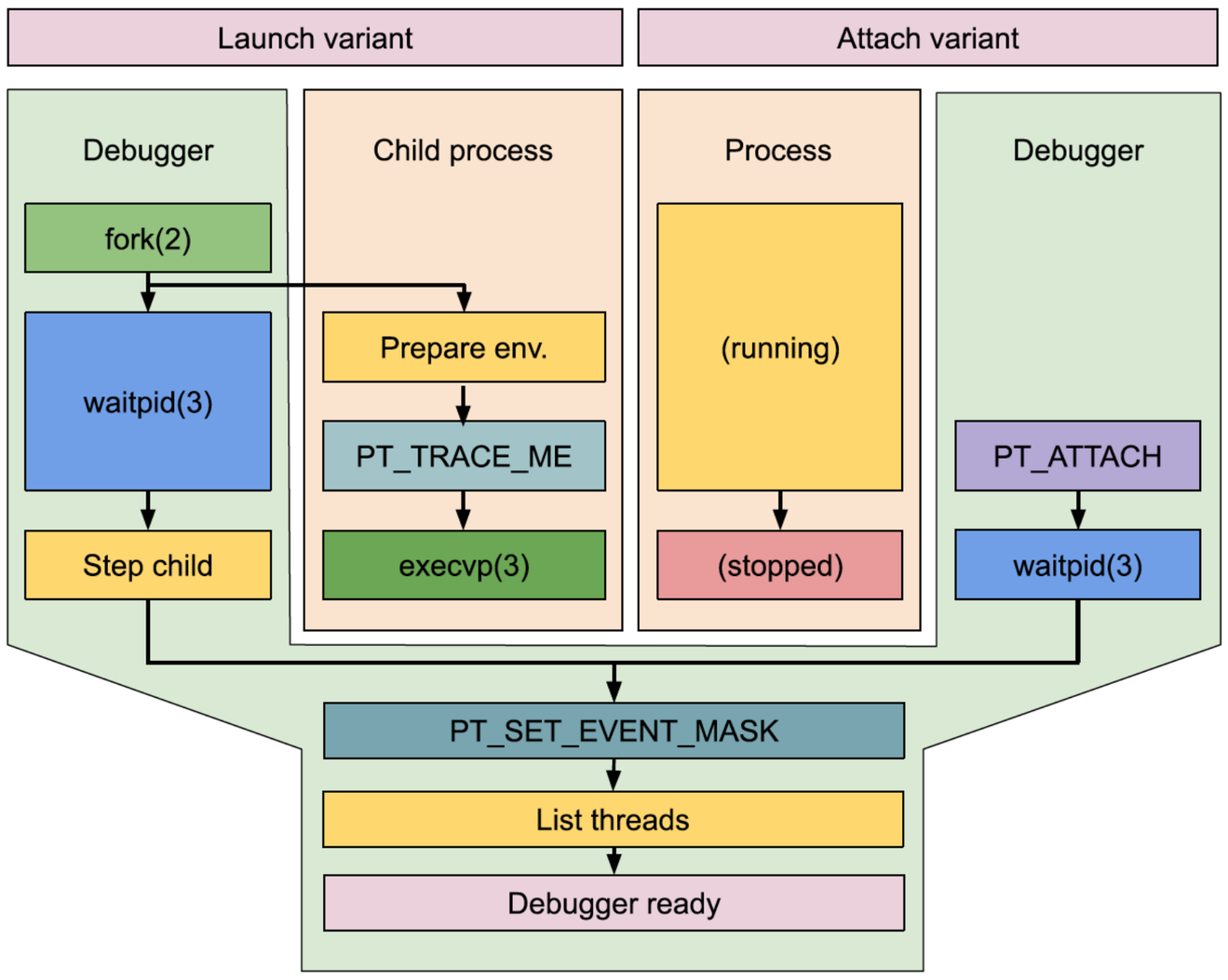


Fig. 1. The initial steps in a debugging session, initiated either via launching a new process or attaching to a running process.

The first step in a debugging session is for the tracer to either attach to a process that is already running, or launch a new program. In the former case, it issues a `PT_ATTACH` request. Once the request succeeds, the traced process is stopped and a signal is delivered to the debugger. In the latter case, the process is a bit more complex as presented on fig. 1.

The `ptrace(2)` API does not provide an explicit request to spawn a new program. Instead, the debugger needs to use the regular system API to do that, e.g. `fork(2) + exec*(2)`. However, just before executing the new program, the child process issues a `PT_TRACE_ME` request. This causes it to start being traced by its parent process (the debugger). At this point, the debugger steps the child process until the exec*(2) call finishes.

After attaching or launching, the traced process is stopped. The debugger performs additional setup, e.g. through setting the reported event mask or querying additional information about the debugged process. The remainder of the debugging session consists of the debugger issuing `ptrace(2)` requests to control the debugged process and query additional information about it, and the kernel delivering process-related events via signals. `SIGTRAP` has a special role here, as it is used to indicate the majority of events specific to the debugging process, e.g. breakpoint and watchpoint hits.

The Architecture of LLDB

LLDB utilizes plugins in order to abstract away large parts of its code base. At the moment of writing, there are 26 plugin categories existing in the LLDB source tree. The plugins provide support for different platforms, ABIs, programming languages, file formats and so on. While the plugin architecture is not considered complete yet (most notably, dynamic loading of plugins is not supported at the time of writing), it enforces the necessary encapsulation to prevent the code from becoming unmaintainable.

At the core of the plugin system, there is one category crucial to LLDB's debugging functionality: process plugins. These plugins implement all the routines needed to launch a process or attach to one already running, and debug it. In the modern versions of LLDB, the process

plugin category holds two kinds of modules: the actual client plugins built on `Process` class, and lldb-server backends built on `NativeProcessProtocol` class.

Historically, every operating system supported by LLDB had its own client process plugin. Prior to LLVM 13, this was also the case for FreeBSD. When running on this platform, the LLDB client would load the respective process plugin and use it to control the debugged program. The debugger's UI and `ptrace(2)` invocations would both be done from a single process.

A more modern approach used by LLDB is to move the actual debugging process abstraction into the `lldb-server(1)` executable. The platform support is moved into a dedicated lldb-server backend. The client uses the `gdb-remote` plugin to either spawn a new lldb-server instance, or connect to another debug server using the GDB Remote Serial Protocol. This server does not have to be lldb-server — it could be the gdbserver from GDB or one of the implementations provided by e.g. QEMU, Valgrind or the FreeBSD kernel.

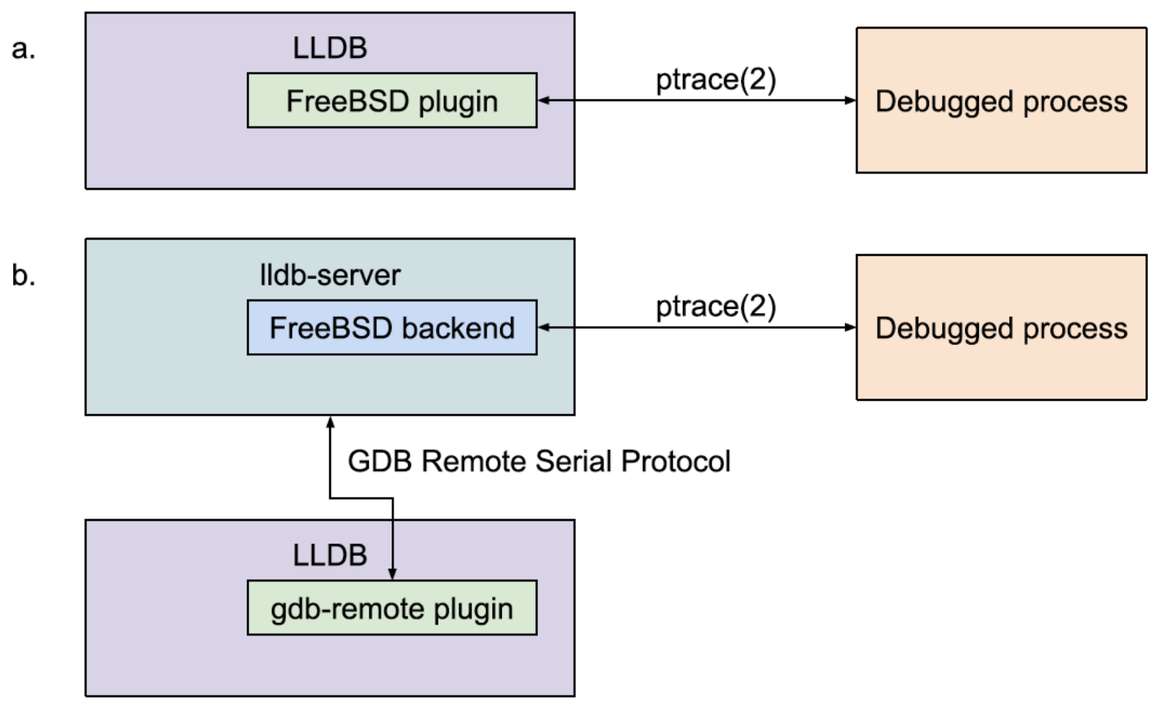


Fig. 2. Traditionally, the FreeBSD plugin was loaded into LLDB, and the debugged process was traced directly from the LLDB executable (subfig. a.). The more modern approach is to move this logic into lldb-server, and have LLDB communicate with it using the GDB Remote Serial Protocol (subfig. b.).

This change marks the evolution of LLDB from a standalone debugger to a client-server model that is capable of cross-platform remote debugging. This layout is also used when debugging locally, providing isolation between the debugger's UI (i.e. the LLDB client) and the server issuing the actual `ptrace(2)` calls.

As of LLDB 14, the vast majority of officially supported platforms use the client-server model and the `gdb-remote` process plugin. The other process plugins primarily implement support for a variety of core dumps formats.

Local and Remote Cross-platform Debugging

The LLVM toolchain is designed as a cross-compiler from ground up. This also applies to LLDB, as it features support for cross-debugging across different architectures and operating systems. However, since the majority of the debugging scenarios requires running the debugged program, LLDB needs to overcome the limitations of the operating system kernel.

A variety of operating systems feature the ability to run executables built for different ABIs that are compatible with the current CPU, most commonly 32-bit i386 executables on a 64-bit amd64 kernel. When this is the case, it is valuable for the kernel to support such programs via the `ptrace(2)` API and for a debugger to be able to use it correctly. The problem can be classified into three scenarios:

1. Native debugging — the kernel, the debugger and the debugged executable architectures are the same.

2. Debugging non-native programs — the kernel and the debugger architectures are the same but the executable architecture is different.
3. Running a non-native debugger — the kernel and the debugger architectures are different.

Table 1. Example mapping of architectures to the debugging scenarios.

Case	Kernel	Debugger	Executable
1	amd64	amd64	amd64
	i386	i386	i386
2	amd64	amd64	i386
3	amd64	i386	i386

The cases 1. and 3. are the same from the debugger's standpoint. Both the debugger and the traced executable are built for the same architecture. The debugger needs to explicitly feature support for the executables of this architecture, and its `ptrace(2)` API. Case 3. additionally requires the kernel support for non-native `ptrace(2)` API.

The second case is perhaps the most interesting. The debugger is built for the native system architecture, and therefore uses the native `ptrace(2)` API. However, this API needs to be adjusted for the non-native executable format. For example, if an i386 executable is run on amd64, the `PT_GETREGS` request uses 64-bit register dump format and it is desirable that the debugger translates between it and the format used natively on i386.

While local cross-debugging is limited by kernel features, remote debugging is much more powerful. In this scenario, lldb-server, gdbserver or any other server implementing a compatible protocol can be spawned on one machine, and the LLDB client can be used to connect to it from another. The two machines don't have to be running the same architecture or even the same operating system.

In fact, it gets even better. Remote debugging is not limited to tracing userspace applications. It can be used to connect to the GDB stub found in FreeBSD kernel over the serial port, and inspect the kernel's state. It can be used to connect to GDB stub implemented in QEMU in order to control the virtual machine's CPU and memory.

LLDB 14 features a more compatible implementation of the GDB Remote Serial Protocol than prior versions. Over the years, LLDB has evolved from using a custom variation of the protocol that was suitable only for communication between LLDB and lldb-server, to one that is compatible with many other debugging servers.

One interesting example of this evolution are register definitions. The earliest versions of lldb-server were transmitting them in JSON-based format that fitted the LLDB's internal layout best. Afterwards, XML-based format was added for compatibility with GDB. Finally, fallback definitions were added to the client for a variety of architectures in order to support stubs that did not transmit target definitions at all.

Debugging Multithreaded Processes

Every modern debugger needs to be able to handle multithreaded programs. In general, the support for multithreading consists of:

- receiving thread creation and termination events
- being able to distinguish other events applicable to a specific thread (e.g. per-thread signals, breakpoints)
- being able to control running and stopping individual threads

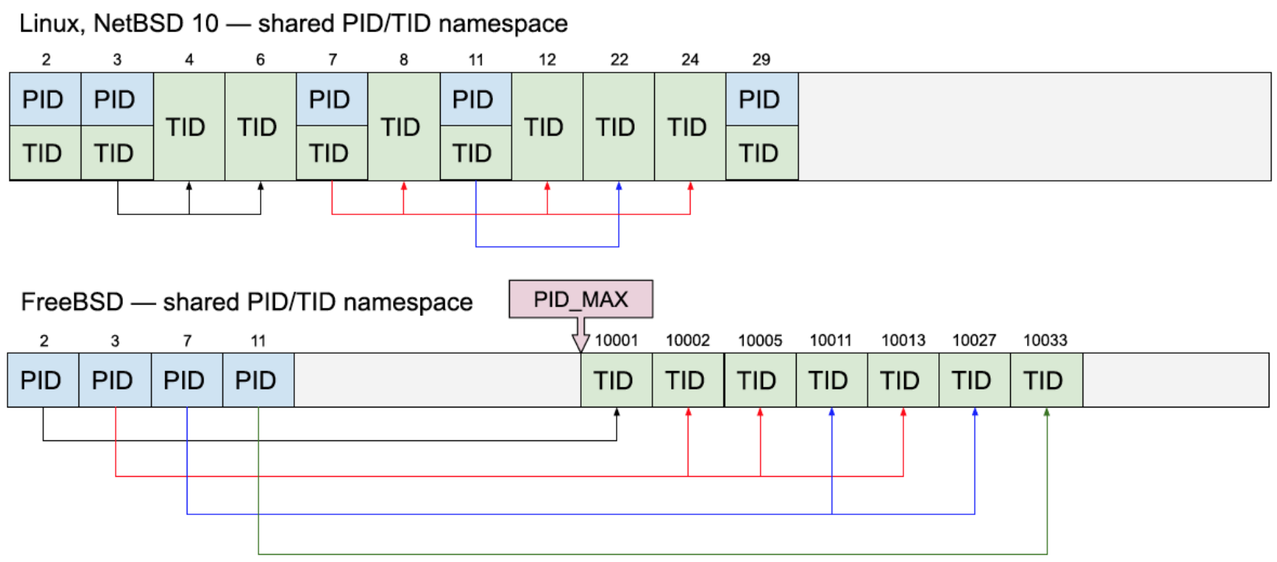


Fig. 3. On Linux and NetBSD 10, the primary thread shares identifier with the process identifier. On FreeBSD, process and thread identifiers use disjoint ranges.

The exact mechanism of handling multiple threads of a debugged program varies depending on the operating system. Modern kernels use a single namespace for process and thread identifiers. TIDs are globally unique. On Linux (and the future NetBSD 10 release) the primary thread has the same identifier as the corresponding process identifier. On FreeBSD process and threads identifiers use disjoint ranges.

This layout also implies how per-thread requests are handled via `ptrace(2)` API. On Linux and FreeBSD, TID can be passed in place of PID to perform a per-thread action. For historical reasons, on NetBSD the thread identifier needs to be passed separately along with the process identifier.

Thread list change events are generally enabled via setting an appropriate event mask. The kernel reports these events via issuing a `SIGTRAP` signal with appropriate data. However, it should be noted that the debugger needs to account for the events being received out of order, that is e.g. a breakpoint hit from a new thread arriving before the thread creation event.

On the current versions of FreeBSD and NetBSD, the tracing API could be called process scoped. When a thread-specific event occurs, the whole process is paused. The debugger receives a signal and needs to use `ptrace(2)` to obtain additional signal information, particularly the identifier of the corresponding thread. There are also requests to control whether a particular thread will remain paused, continue running or enter single-stepping when the process is resumed.

On the other hand, the Linux API treats threads in greater isolation. The debugger needs to trace every thread separately. Signals are reported for a specific thread. When a single thread stops, other threads of the process remain running.

Debugging Multiple Processes

The debugger's support for multiple processes can cover a variety of use cases, from programs forking themselves in order to run multiple operations simultaneously, to running external programs or complete pipelines. At the time of writing, LLDB has two features for debugging multiple processes: support for multiple targets and handling of fork events. Furthermore, there is an ongoing work to introduce full multiprocess support.

In LLDB terminology, a target represents a single debugged process. Appropriately, in order to be able to trace multiple processes simultaneously, LLDB needs to create and track multiple targets. In the most common case of using gdb-remote plugin to trace native processes, every process is traced by a separate lldb-server instance, and every target maintains a separate connection to its respective server. The limitation of this approach is that every target needs to be created separately, e.g. via launching the executable or attaching to a process that is already running.

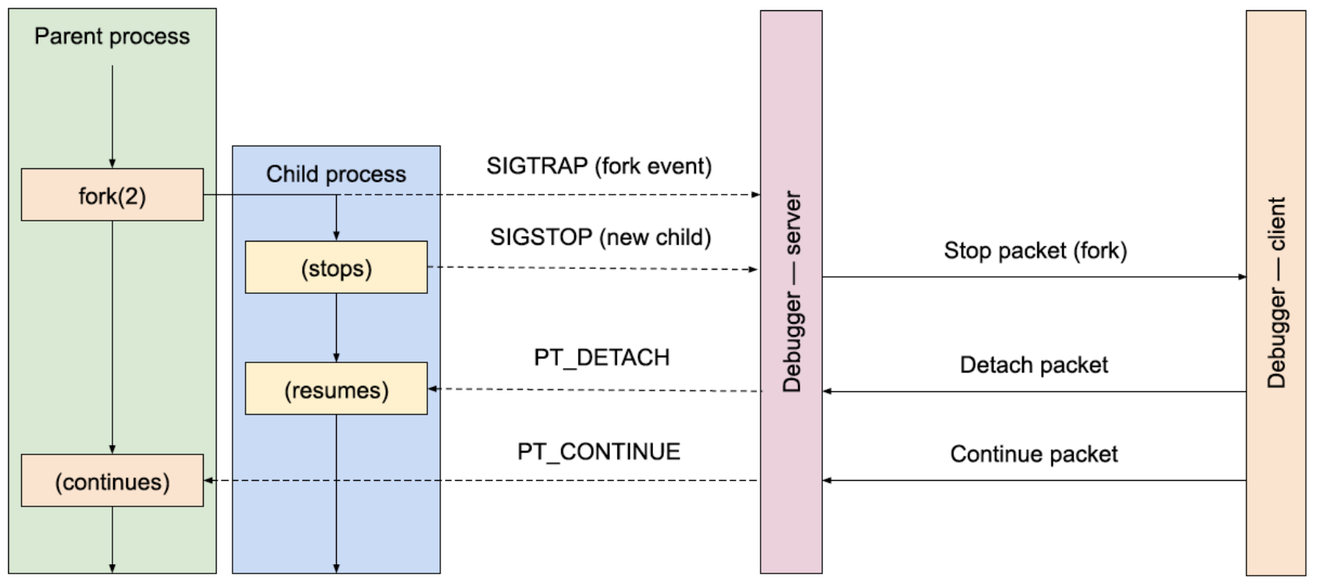


Fig. 4. When a process forks, the debugger receives `SIGTRAP` from the parent process and `SIGSTOP` from the child process. The server reports stop due to fork to the client, and the client requests detaching one of the processes (via `PT_DETACH`) and resuming the other one (via `PT_CONTINUE`).

The other feature is handling of `fork(2)`, `vfork(2)`, `posix_spawn(3)` and their equivalents. Similarly to the support for multiple threads, this is enabled via setting an appropriate event mask. When the process is forked, the debugger is signaled and starts tracing both the parent and the child processes. However, at this point LLDB does not feature full support for tracing both processes, and instead detaches one of them. LLDB can be configured to either continue tracing the parent process, or detach it and trace the newly forked child instead.

The purpose of the ongoing multiprocess effort is to combine both these features to provide the full support for debugging process trees. A key feature is to be able to start tracing the child process immediately without having to resume it or its parent. There are two primary possible implementations: using GDB-style multiprocess extensions to support multiplexing multiple traced processes within a single GDB Remote Serial Protocol connection, or using a separate connection for every new process.

Non-live Process Debugging Targets

While admittedly the primary use of LLDB on FreeBSD is to debug userspace processes, there are other kind of 'process' plugins: notably plugins handling core dumps and the FreeBSD kernel debugging. Similarly to the gdb-remote plugin, these modules are loaded directly into LLDB client and do not utilize the client-server architecture.

Core dumps on modern Unix derivatives are recorded using the ELF file format, much like executables on these platforms. Appropriately, they are handled by an elf-core plugin in LLDB. However, this plugin is not suitable for handling FreeBSD kernel core dumps since these dumps use physical memory layout rather than the virtual layout used by regular processes.

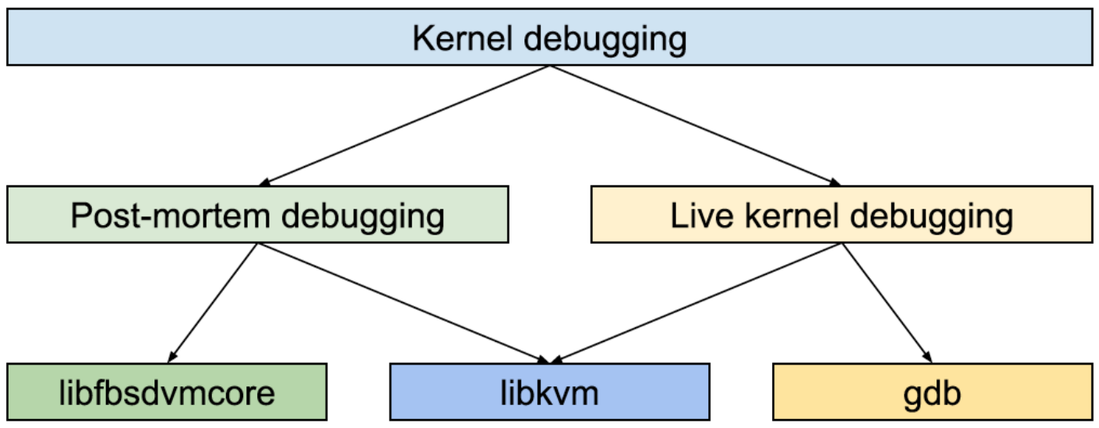


Fig. 5. Common methods of debugging the FreeBSD kernel

FreeBSD kernel debugging is a wider topic. There is a variety of methods provided for working either with the live kernel (i.e. the kernel of a running system) or with a kernel core dump. Firstly, the kernel itself features a built-in debugger, `kdb`. Secondly, it features a GDB stub that can be used to attach a remote debugger using the GDB Remote Serial Protocol, over a serial

port. The recent versions of LLDB include improved serial port support for precisely this reason. Thirdly, it is possible to gain access to kernel memory through `/dev/mem` special device. Finally, it is possible to use a kernel core dump.

LLDB 14 introduced a new `FreeBSDKernel` plugin that provides support for the last two scenarios. It enables LLDB to open and process kernel core dump files correctly, both in the newer minidump format and in the older 'full memory dump' format. This is done with the help of one of the two libraries: either libkvm that is provided by the FreeBSD base system, or libfbsdvmcore that has been created as a portable alternative for cross-platform debugging. This makes it possible to use LLDB as a replacement for the KGDB tool.

Summary

Over the last years, LLDB has made major progress as a debugger. While it still does not feature 100% feature parity with GDB, it improves with every new release. Furthermore, it goes beyond aiming to be 'just' a replacement for GDB. Built on top of LLVM tool chain, it features extensive expression parsing support built on top of Clang, JIT suport, Python and Lua scripting support. Plugin-based design and good test coverage makes extending it a pleasure.

The progress made on the LLDB front has made it possible to finally retire the aged GDB from FreeBSD and replace it with LLDB that fits the project much better. Furthermore, thanks to fruitful cooperation between the projects, it was possible to integrate KGDB functionality directly into LLDB, removing the need for a separately maintained frontend. These achievements were possible in large part due to the support of the FreeBSD Foundation.

There is a lot of interesting work happening in LLDB all the time. A part of it is a recently started project on implementing full support for multiprocessing — effectively enabling users to efficiently debug complete trees of forked or spawned processes. Another interesting future project is RISC-V architecture support.

KAMIL RYTAROWSKI and **MICHAŁ GÓRNY** are open source enthusiasts and contributors for over 10 years. They founded Moritz Systems, a company focused on BSD development. They authored the modern LLDB plugin for NetBSD, and have been improving LLDB's support for FreeBSD since September 2020.

Pragmatic IPv6 (Part 1)

BY HIROKI SATO

Have you ever tried to use IPv6 on your FreeBSD box? While IPv4 is still the golden standard of today's Internet world, IPv6 is considered as a promising protocol which is expected to eventually replace IPv4. The first RFC of IPv6 was published in 1995, and FreeBSD has supported it for over 20 years by default. The implementation is mature, not an experimental one anymore.

Why is IPv6 less popular? Some people believe that IPv4 is enough for their purpose and do not want to change. That is a valid reason. However, I believe one of the reasons preventing IPv6 from being popular is system administrators' lack of experience. Similar to the history of IPv4, implementations and best current practices of IPv6 have also been changed over the years. This makes finding books, articles, or literature containing practical information on IPv6 challenging. As long as the pros and cons of using IPv6 are unclear, no skilled sysadmin wants to use this new technology.

This series of articles introduces (or re-introduces) IPv6 to you by showing various examples you can try on your FreeBSD box, not only the theoretical aspects. IPv6 is not a complete replacement of IPv4 and works fine with the existing IPv4 network. Even if you are already using IPv6, you should find something new or valuable.

Introduction

Before diving into the details, I would like to start this with basic concepts that you have to understand. They might be boring if you are already using IPv6, but this article assumes that the readers are familiar with IPv4 network management but not with IPv6 as yet. Definitions of the following technical terms will be explained as the very first introductory topic:

- IPv6 address format and text representation
- Subnet prefix and interface identifier
- Address scope and zone

IPv6 Address

This might be the most visible difference from IPv4. IPv6 has a 128-bit long address[1] while IPv4 has 32-bit long one. The line (1) in Figure 1 shows an example. This IPv6 address is represented in 8 fields separated by colons. You can see 4 digits in hexadecimal in each field because it has a 16-bit value.

It is a complete expression of a 128-bit value. However, there is a recommended way for text

representation of an IPv6 address[2]. The rules are as follows:

- Successive “0”s and/or leading “0”s must be omitted,
- The first longest successive “0”s longer than 16-bit must be replaced with “::”,
- Shorten as much as possible.

```
(1)  2001:0db8:0000:0000:0001:0000:0000:4444
(2)  2001:0db8:0000:0000:0001:0000:0000:4444
(3)  2001:db8:0:0:1:0:0:4444
(4)  2001:db8::1:0:0:4444
```

Figure 1: An IPv6 address in a hexadecimal format (16-bit field x 8)

(2) in figure 1 is one after the successive zeros are removed, and (3) is one after the first longest zeros are removed. (4) is the final form and what you will see in outputs from IPv6-aware software, including FreeBSD.

This convention of the address text representation is just for consistency and simplicity. You can use all representations from (1) to (4) as your input. It would be best to design your software to use (4) in the outputs. An address in a log file is a good example. It should be able to use “grep” in the format of (4).

Subnet Prefix and IID

The next keyword is "subnet prefix" and “interface identifier (IID).” Let’s revisit an IPv4 address before that. Figure 2 shows the structure of an IPv6 and IPv4 address. In IPv4, an address is represented in “dotted octect” format. There are two addresses: host address and network address. The host address identifies a "node" or an end host. The network address identifies a network segment, a domain where nodes can communicate with each other without a router. A network address is calculated by using a host address and a netmask as shown in Figure 2. If you have `192.168.2.1` as the host address and `255.255.255.0` as the netmask, then the network address is `192.168.2.0`. Machines with the same network address belong to the same network segment and can communicate directly.

```
(1)  2001:0db8:0000:0000:0001:0000:0000:4444
     subnet prefix (n bit)   interface identifier (128 – n) bit
(2)  2001:db8::1:0:0:4444/64
                             prefix length
(3)  192.168.2.1             255.255.255.0
     host address            netmask
(4)  network address = AND(host address, netmask) = 192.168.2.0
(5)  192.168.2.1/24
     subnet mask length
```

Figure 2: Subnet prefix and interface indentifier

An IPv4 netmask is designed as a 32-bit value, and it is not necessary for the “1” values in the mask to be contiguous. `255.255.255.0` is (1111 1111 1111 1111 1111 1111 0000 0000) in binary. So the leading “1” is contiguous. Most of the netmasks you see today should be contiguous because it is the standard way of routing domain engineering for the Internet. In the very early days “network class” was used and three classes A, B, and C, were defined by netmasks —"255.0.0.0," "255.255.0.0," and "255.255.255.0"— respectively. The host address determined one of these classes automatically, so you needed no netmask. This was changed at

some point, and CIDR (Classless Inter-Domain Routing) was introduced[3]. In CIDR strategy, the netmask is independent of the host address, and the leading "1"s in the mask are usually contiguous. So the netmask can be represented by using the leading "1"s length. This length is called "subnet mask length" and is often represented at the tail of an IPv4 address with a slash when you want to show the network address information at the same time (see also the last line of Figure 2).

Some modern systems no longer support the non-CIDR netmask. FreeBSD still supports it. You can try the following to see what happens. Have you ever seen an output of netstat(1) like this?

```
# ifconfig bge0 inet 192.168.2.1 netmask 255.128.255.0
# ifconfig bge0 | grep netmask
    inet 192.168.2.1 netmask 0xff80ff00 broadcast 192.255.0.255
# netstat -nrf inet | grep bge0
    192.128.0.0&0xff80ff00 link#1 U bge0
```

Let's go back to IPv6. IPv6 has almost the same concept as "host address" and "network address." A 128-bit address is divided into two parts; one is "subnet prefix," and another is "IID (interface identifier)." The subnet prefix corresponds to "network address" in IPv4. It is a 128-bit address generated by a subnet prefix and "0"s in the lower digits. You can consider a 128-bit netmask, but it is always represented by the leading "1"s length similar to IPv4 CIDR subnet mask length. In IPv6, it is called "prefix length."

You can assign an IPv6 address to an interface on your box. You need a subnet prefix, an I ID, and prefix length to do that. A prefix length for an IPv6 node (e.g., your machine) will be 64 if there is no specific reason. In the core specification of IPv6, the prefix length is variable. However, most IPv6-related protocols assume an IID is 64-bit or longer. If you do not specify the prefix length, 64 will be used as the default value. An IPv6 address assigned to your FreeBSD box is usually split into a 64-bit subnet prefix and a 64-bit IID. An interface identifier identifies the node in the same network segment.

When you get Internet access from your ISP, typically, you will receive an IPv4 address. If your ISP supports IPv6, it provides an IPv6 "prefix." They typically provide prefixes with `/48`, `/56`, `/60`, or `/64` for you. This means you can use 16 bits for your LANs in the case of a `/48` prefix since the prefix length of addresses for your machines is always `/64`.

Let's Try IPv6 on FreeBSD

Are you getting tired of abstract things? Let's go to IPv6 world on your FreeBSD box. As mentioned in the preface, FreeBSD has supported IPv6 by default for a long time. The GENERIC kernel has no runtime knob to enable or disable IPv6. All you need to use IPv6 is simply to add an IPv6 address to one of the interfaces.

Manual Configuration by using ifconfig(8)

To understand IPv6 step-by-step, let's get started configuring it by hand. `bge0` is assumed as a primary NIC on your box in the following examples. You can try them on your working NIC and IPv4 network. It will not break them at all.

Try command lines shown in Figure 3. The first `ifconfig bge0` shows the current status of `bge0`. It should show `nd6 options` line with `IFDISABLED` option. This option means "IPv6 is disabled on this interface".

The second `ifconfig bge0 inet6 -ifdisabled` command removes this option. You can double-check the status by another `ifconfig bge0` command.

By third `ifconfig` command, you should see `inet6` line. This is an IPv6 address configured to `bge0`. The `fe80::d63d:7eff:fe78:fc64%bge0` part is the address, and `prefixlen 64` part shows the prefix length. The "`%bge0`" part looks a bit odd, but it is a part of the address and will be explained later.

You can "ping" this address. The `ping(8)` command for IPv6 is named as `ping6(8)`. Note that `ping(8)` and `ping6(8)` are merged into a single command on and after FreeBSD 13.x, while `ping6(8)` is still available in the newer releases.

If an `sshd(8)` daemon is running on your machine, you can even connect using the IPv6 address. Try the last command in Figure 3. Note that if `AddressFamily` is limited to `inet` in your `sshd_config(8)` configuration file, it fails. See the output of `sockstat -6 | grep sshd` to check whether `sshd(8)` listens to the IPv6 socket.

From users' point of view, TCP/IP applications with IPv6 support like `ssh(1)` work in the same way as IPv4 except for the address format.

```
% ifconfig bge0
bge0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
    options=c019b <RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_HWCSUM,TSO4,
    VLAN_HWTSO,LINKSTATE>
    ether d4:3d:7e:78:fc:64
    inet 192.168.100.104 netmask 0xffffff00 broadcast 192.168.100.255
    media: Ethernet autoselect (1000baseT <full-duplex>)
    status: active
    nd6 options=29<PERFORMNUD,IFDISABLED,AUTO LINKLOCAL>

# ifconfig bge0 inet6 -ifdisabled

% ifconfig bge0
bge0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
    options=c019b <RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_HWCSUM,TSO4,
    VLAN_HWTSO,LINKSTATE>
    ether d4:3d:7e:78:fc:64
    inet 192.168.100.104 netmask 0xffffff00 broadcast 192.168.100.255
    inet6 fe80::d63d:7eff:fe78:fc64%bge0 prefixlen 64 scopeid 0x1
    media: Ethernet autoselect (1000baseT <full-duplex>)
    status: active
    nd6 options=21<PERFORMNUD,AUTO LINKLOCAL>

% ping fe80::d63d:7eff:fe78:fc64%bge0
PING6(56=40+8+8 bytes) fe80::d63d:7eff:fe78:fc64%bge0 --> fe80::d63d:7eff:fe78:f-
c64%bge0
16 bytes from fe80::d63d:7eff:fe78:fc64%bge0, icmp_seq=0 hlim=64 time=0.181 ms
16 bytes from fe80::d63d:7eff:fe78:fc64%bge0, icmp_seq=1 hlim=64 time=0.107 ms
16 bytes from fe80::d63d:7eff:fe78:fc64%bge0, icmp_seq=2 hlim=64 time=0.094 ms
^C
```

```
--- fe80::d63d:7eff:fe78:fc64%bge0 ping6 statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/std-dev = 0.094/0.127/0.181/0.038 ms

% ssh -v fe80::d63d:7eff:fe78:fc64%bge0
OpenSSH_7.9p1, OpenSSL 1.1.1k-freebsd 24 Aug 2021
debug1: Reading configuration data /home/hrs/.ssh/config
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Connecting to fe80::d63d:7eff:fe78:fc64%bge0 [fe80::d63d:7eff:fe78:
fc64%bge0] port 22.
debug1: Connection established.
....
```

Figure 3: A command line example to add an IPv6 address to bge0 manually

IPv6 Address Autoconfiguration

An IPv6 address was configured, and it was usable. However, where does this address come from? In IPv4, you should have configured an address by specifying it literally in an `ifconfig`(8) command line. What we did was just `ifconfig bge0 inet6 -ifdisabled`. What happened?

FreeBSD supports IPv6 SLAAC (StateLess Address AutoConfiguration)[4]. This is a neat feature of IPv6 that configure an IID automatically. Remember that an IPv6 address has a prefix and an IID. The prefix is provided by the network operator (you or someone else), but the IID is what you have to choose. It must be unique on the same network.

SLAAC fills the IID part by using the MAC address[5]. It is not exactly the same as the MAC address, but a unique IID is generated based on it. You can see whether SLAAC is enabled or not by checking "nd6 options" line in the output of `ifconfig`(8). If it has "`AUTO_LINKLOCAL`," an IPv6 address is automatically generated and configured.

What about the prefix part? If you configured no IPv6 network or your ISP offered no IPv6 prefix, your LAN has no IPv6 prefix. However, the `bge0` interface has an address `fe80::d63d:7eff:fe78:fc64%bge0`. Figure 4 (1) was the output of `ifconfig`(8). It means an address shown in (2) in a full 128-bit format. The prefix is `fe80::` in the shortened representation.

The `fe80::` subnet prefix is one of the reserved prefixes for "link-local" communication. You can use this prefix freely on any network interface. Addresses within this prefix are limited to communication between directly-connected machines to the link. In IPv4, 169.254.0.0/16 is reserved for the same purpose, but it is optional and not widely used for actual communications[5]. In IPv6, the link-local subnet prefix is essential in the core protocol. This is why this address is automatically configured just by enabling IPv6 on the interface.

So you can consider that an IPv6-capable interface always has at least one address with the link-local subnet prefix. You can use this address for standard TCP/IP applications while it is also used in various vital communications.

```
(1)   inet6 fe80::d63d:7eff:fe78:fc64%bge0 prefixlen 64
(2)   fe80:0000:0000:0000:d63d:7eff:fe78:fc64%bge0/64
      prefix                    IID
```

Figure 4: An automatically-configured IPv6 address

Address Type, Scope, and Zone

The last mysterious part of the auto-configured address is `%bge0`. If you try `ping6(8)` without this part, `ping6(8)` fails. What is this?

An IPv6 address has its "scope." The scope is a topological span within which the address may be used as a unique identifier[7]. Although several scopes were proposed in IPv6 RFCs in the past, only two scopes named "global" and "link-local" are practical today. The global scope means routable on the Internet. The link-local scope is valid only on a single, specific link.

The keyword "zone" is related to "scope." Figure 5 shows an example network with two or more NICs. A zone is an extent which a scoped address must be unique. If you have two NICs and they are connected to the same network, you will have an `fe80::` prefix address on each interface automatically. In this case, these two must be unique, and this extent is called "a link-local zone." Each link-local zone is identified by using a zone identifier. On FreeBSD, the zone identifier is the scope identifier or the interface name of the NIC. You can see "scopeid" keyword in the output of `ifconfig(8)`. The value just after the keyword is the scope identifier.

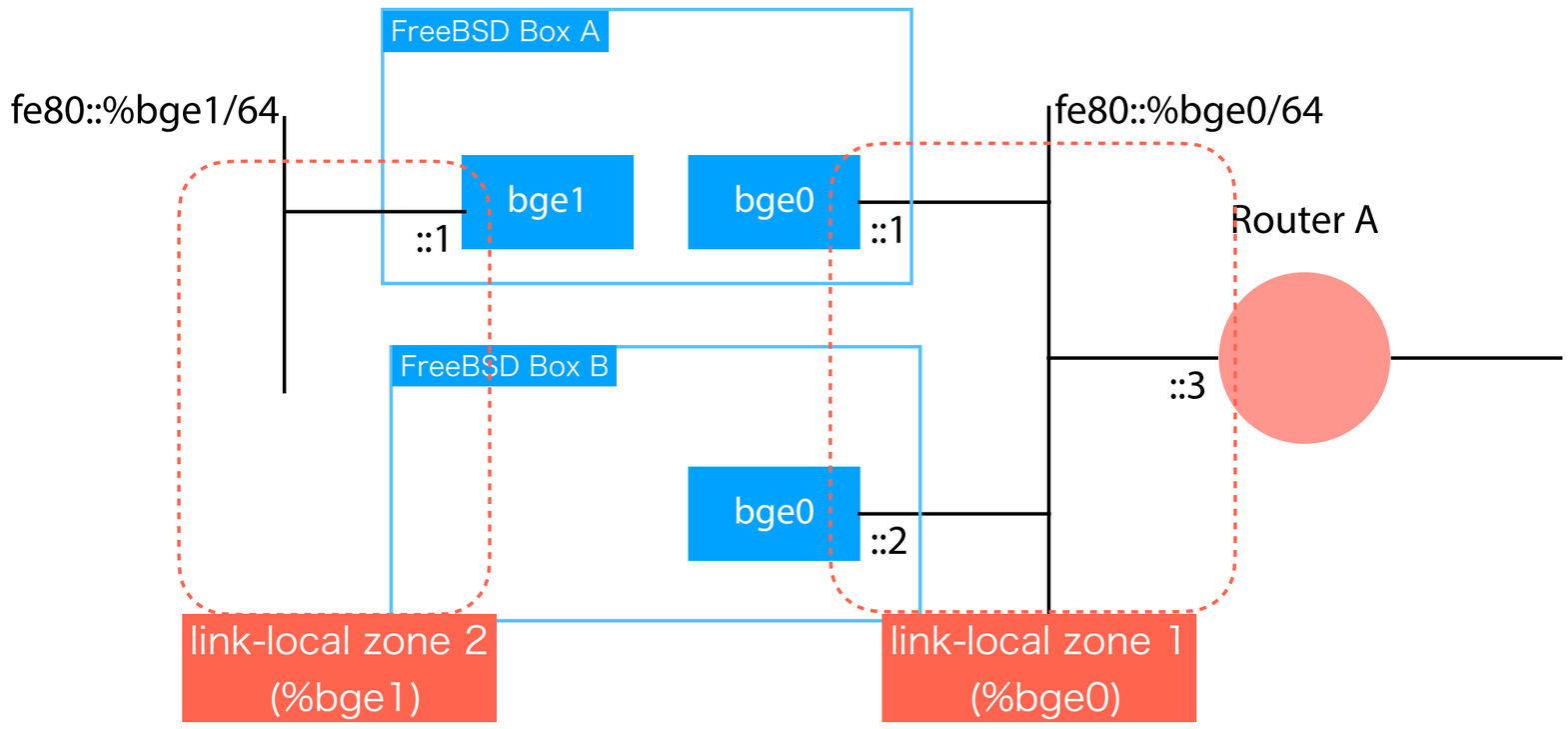


Figure 5: Relationship between zones and network segments

Let's go back to `%bge0`. This is the zone identifier of this address. Link-local scoped addresses are unique within the zone but may not be unique on the same machine. So you have to specify which zone must be used. `%bge0` part is required to specify the zone. The scope identifier of `bge0` is 1, so you can write it as %1 instead. A link-local address with no zone identifier is considered invalid[7].

And an IPv6 address has two types; "unicast" and "multicast." This is independent of the address scope. If it is unicast, the address is used for regular 1:1 communication. If multicast, the communication can have multiple receivers.

The prefix automatically determines the scope and type:

- if the first 8 bits are "1111 1111" (ff00::), it is a multicast prefix,
- if the first 10 bits are "1111 1110 10" (fe80::), it is a link-local scope and unicast prefix (called as LLA, linklocal address),
- the others are global scope, unicast prefixes (called GUA, global unicast address).

Useful Multicast Addresses

Multicast addresses are not popular in standard IPv4 deployment. In IPv6, multicast is used everywhere. More details will be covered in later issues, but I would like to introduce the following two addresses:

- `ff02::1`—link-local scope, all-nodes multicast address
- `ff02::2`—link-local scope, all-routers multicast address

Every IPv6 node "joins" the all-nodes multicast address. This means that all machines connected to the specified zone will respond to communication to this address. Let's try the command shown in Figure 6. If no other IPv6-capable device is connected to `bge0`, you will see no response except for your machine itself. If you have some, you will see multiple ICMPv6 echo replies. If you use `ff02::2`, routers on the same link will respond. If you have Apple macOS or iOS devices, you can see them because they enable IPv6 by default.

These two link-local multicast addresses are helpful for diagnostics. Remember that you can find an IPv6 neighbor by using `ff02::1`, and you can find an IPv6 router by using `ff02::2`.

```
% ping6 ff02::1%bge0
PING6(56=40+8+8 bytes) fe80::d63d:7eff:fe78:fc64%bge0 --> ff02::1%bge0
16 bytes from fe80::d63d:7eff:fe78:fc64%bge0 , icmp_seq=0 hlim=64 time=0.073 ms
16 bytes from fe80::21b:78ff:fe39:84f6%bge0 , icmp_seq=0 hlim=64 time=0.194 ms(DUP!)
16 bytes from fe80::225:90ff:fe13:503a%bge0 , icmp_seq=0 hlim=64 time=0.276 ms(DUP!)
16 bytes from fe80::a6ba:dbff:fee0:b190%bge0 , icmp_seq=0 hlim=64 time=0.351 ms(DUP!)
16 bytes from fe80::202:a5ff:fee9:4104%bge0 , icmp_seq=0 hlim=64 time=0.427 ms(DUP!)
16 bytes from fe80::202:a5ff:fee9:c87d%bge0 , icmp_seq=0 hlim=64 time=0.511 ms(DUP!)
16 bytes from fe80::8c:7aff:fe24:1c64%bge0 , icmp_seq=0 hlim=64 time=0.585 ms(DUP!)
16 bytes from fe80::202:a5ff:fee9:c3c9%bge0 , icmp_seq=0 hlim=64 time=0.658 ms(DUP!)
16 bytes from fe80::202:a5ff:fee9:3952%bge0 , icmp_seq=0 hlim=64 time=0.730 ms(DUP!)
16 bytes from fe80::202:a5ff:fee9:2e5e%bge0 , icmp_seq=0 hlim=64 time=0.802 ms(DUP!)
:
:
```

Figure 6: A ping6 command with ff02::1

IPv6 Node Configuration

An IPv6 node must have the following configuration:

- At least one link-local address on each NIC,
- a lookback address.

The loopback address is `::1`. In IPv4, `127.0.0.1` is used and configured on `lo0`. The `::1` is also automatically configured on `lo0`.

You can configure more IPv6 addresses on a single interface. In IPv6, many addresses are used on the same interface for various purpose. For example, you can add another link-local address `fe80::1/64` by using `ifconfig(8)`:

```
% ifconfig bge0 inet6 fe80::1/64
```

Unlike IPv4, this does not remove or replace the already configured IPv6 addresses. Check the output of `ifconfig(8)` and try `ping6(8)` or `ssh(1)` to use the new address. If you want to remove a specific address, you can use `-alias` flag like this:

```
% ifconfig bge0 inet6 fe80::1/64 -alias
```

What address/prefix can I use?

You might be confused about what IPv6 address you can actually use. In the IPv4 network, you probably know "private address space" is always available for your local network[9]. `10/8`, `172.16/16`, and `192.168/24` are widely used. So what about IPv6?

As mentioned in the previous sections, you can use `fe80::/64` prefix for link-local communication. You can generate addresses by using this prefix freely as long as there is no duplication. Most TCP/IP applications work with them[10]. Note that you need to add `%zoneid` part in the address if you use a link-local unicast address.

If you have a global IPv6 prefix from your ISP, you can use it in addition to the link-local prefix. If you configure the default router for IPv6, you can do IPv6 Internet communication. The global prefix is not a replacement for the link-local one; the link-local addresses are essential for the core IPv6 protocol. Do not forget that you need them even if you have global ones.

So is there no private address replacement in IPv6? Yes and no. ULA (Unique Local Address) is a reserved address space for a similar purpose[11]. However, deployment of ULA needs some more careful consideration than IPv4.

More details about IPv6 deployment scenarios, including clarification about the questions above, will be covered in the next issue.

Configuration in rc.conf

Lastly, let's see how to configure IPv6 addresses in `/etc/rc.conf`. For IPv4, `ifconfig_bge0` line is used to configure the `bge0` interface. For IPv6, `ifconfig_bge0_ipv6` is used to indicate that `bge0` is IPv6-capable. The simpliest version of the configuration is as follows. In this configuration, only one link-local address is automatically configured:

```
ifconfig_bge0="inet 192.168.0.10/24"
ifconfig_bge0_ipv6="inet6 auto_linklocal"
```

If you want to add another link-local address manually, you can add "inet6" line into `ifconfig_bge0_ipv6`. This configuration adds two link-local addresses, one by SLAAC and another by the `ifconfig_bge0_ipv6` line:

```
ifconfig_bge0="inet 192.168.0.10/24"
ifconfig_bge0_ipv6="inet6 fe80::1/64"
```

More addresses can be added in the same way as IPv4. The following example adds a global unicast address `2001:db8::1/64` by using `ifconfig_bge0_alias0` line:

```
ifconfig_bge0="inet 192.168.0.10/24"
ifconfig_bge0_ipv6="inet6 fe80::1/64"
ifconfig_bge0_alias0="inet6 2001:db8::1/64"
```

Summary

This column introduced IPv6 basics and configuration examples on FreeBSD. Once you understand the address structure and configuration by using ifconfig(8), you can use automatically-configured link-local addresses. While they are not globally routable on Internet, they are still helpful for local network communication or diagnostics.

In the next issue, deployment details using IPv6 global prefix and more configuration examples of FreeBSD and thirdparty software on IPv6-enabled network.

Footnotes

[1] RFC4291: "IP Version 6 Addressing Architecture"

[2] It is summarized in RFC 5952: "A Recommendation for IPv6 Address Text Representation"

[3] RFC 1519: "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy"

[4]RFC 4862: "IPv6 Stateless Address Autoconfiguration"

[5]The IID is not always generated by using the MAC address, but it is one of the most popular implementations. RFC 7217 and RFC 8064 have discussed several issues and a new standard. This topic will be covered in later issues of this column.

[6]RFC 3927: "Dynamic Configuration of IPv4 Link-Local Addresses". Microsoft calls this as "Automatic Private IP Addressing (APIPA)".

[7]RFC 4007: "IPv6 Scoped Address Architecture"

[8]The zone identifier “bge0” is actually interface-local, not link-local. If you have bge0 and bge1 which are connected to the same network segment, both `%bge0` and `%bge1` means the same link-local zone. IPv6 network stack on FreeBSD internally assumes interface-local and link-local are the same.
[9]RFC 1918: “Address Allocation for Private Internet”
[10]Unfortunaltely, a few applications are incompatible with LLA. They will be explained in later issues.
[11]RFC 4193: “Unique Local IPv6 Unicast Addresses”

HIROKI SATO is an assistant professor at Tokyo Institute of Technology. His research topics include transistor-level integrated circuit design, analog signal processing, embedded systems, computer network, and software technology in general. He was one of the FreeBSD core team members from 2006 to 2022, has been a FreeBSD Foundation board member since 2008, and has hosted AsiaBSDCon, an international conference on BSD-derived operating systems in Asia, since 2007.

Post-Mortem Kernel Debugging with netdump(4)

BY MARK JOHNSTON

FreeBSD kernel panics are a hopefully rare occurrence, but they do happen from time to time. You may have been unlucky and hit a kernel bug on a production system, or perhaps you are developing a kernel patch and uncovered a bug while testing. In such situations, a reboot will bring the system back online, but the contents of RAM will be lost, making it impossible to find the root cause of the panic.

FreeBSD supports both live debugging and post-mortem debugging of kernel panics. While live debugging is often simpler, its nature means that the panicked system cannot be rebooted until the developer has finished debugging. This is often impractical, so post-mortem debugging via core dumps is a common debugging activity. For a long time, the FreeBSD kernel has had the ability to save a core dump, often called a "kernel dump," following a panic; once a kernel dump has been saved, the panicked system can be rebooted and brought back online, and the dump can then be used to diagnose the problem.

FreeBSD supports both live debugging and post-mortem debugging of kernel panics.

When a userspace program crashes and dumps core, the operating system saves its state in a regular file somewhere in the file system. When the kernel crashes, though, life is not as straightforward: the kernel itself is responsible for mediating access to its file systems, and following a panic the kernel is by definition in an inconsistent state, so writing data to a file is a fraught endeavor. A kernel panic is bad enough, but it'd be much worse if the kernel went on to corrupt its own file systems!

FreeBSD's traditional solution to this problem is to write the kernel dump to a raw disk partition, often the same one used for swap space. Doing so is much simpler than modifying a file system, and since swapped-out data does not persist between reboots, there is little risk of overwriting important data.

Configuring kernel dumps is easy: in `/etc/rc.conf`, set the dumpdev variable to the name of the disk device to which kernel dumps should be saved, or set it to the string `"AUTO"` if the swap partition is to be used. Under the hood, this mechanism uses `dumpon(8)` to tell the kernel which disk device to use. When booting up following a panic, FreeBSD will automatically run

`savecore(8)`, which reads the saved kernel dump and places it in `/var/crash` for later use.

Disk-based kernel dumps work well so long as the system has a spare partition where they can be saved. This is not always the case, though: systems might boot disklessly and not have any persistent storage at all, or, as is common with embedded devices, there might not be any disk space to spare. In these situations one historically had to resort to live debugging, or a one-off hack like using a USB thumb drive to store the kernel dump. However, as of FreeBSD 12.0 there's a better way!

Introduction to `netdump`

`netdump(4)` is a relatively new feature which lets a panicked FreeBSD transmit a kernel dump over a network before rebooting. In short, it uses a custom UDP-based protocol to transmit the contents of RAM to a server, implemented by `netdumpd(8)` (`ftp/netdumpd` in the FreeBSD ports system). This lets one get a dump from a panicked kernel without requiring any local storage on the system.

It should be stated up front that `netdump` does not perform any encryption or authentication, so the contents of kernel memory are transmitted directly over the network. Since kernel memory typically contains secret information, it is important to use `netdump` only on trusted networks.

`netdump` has a long history: it started life circa 2000 as FreeBSD 4 patch by Darrell Anderson at Duke University, and was ported forward over the years by developers at several FreeBSD-using companies. It was finally committed to the FreeBSD src repository in 2018 and first became available in FreeBSD 12.0.

Internally, `netdump` is built on top of `debugnet`, a standalone IPv4/UDP implementation which is specialized to be usable in a panicked kernel. In particular, `debugnet`'s UDP stack runs in a single thread, does not perform any heap memory allocations, and does not block (e.g., to wait for an interrupt or a mutex). These constraints come from a need to minimize the complexity of kernel code which executes after a panic: since the kernel has already crashed, `netdump` must avoid making the situation worse while it does its job.

Because `debugnet` transmits and receives packets, it needs to be able to talk to network interface controller (NIC) hardware. Thus, individual NIC drivers require modification in order to be used by `netdump`. Typically this modification consists of adding a "polling" mode to the driver's packet transmission and receive paths. In practice the required modifications are straightforward to implement and typically involve adding less than 100 lines of C code to a given driver. Many widely used drivers implement `debugnet` support today, including all of the Intel drivers (in fact, all drivers implemented using the `iflib` framework), modern Mellanox drivers, the VirtIO network driver, and several drivers for GigE NICs often found in desktop systems or server management ports; a full list is given in the `netdump(4)` manual page.

Finally, `debugnet` hooks into the kernel's packet buffer allocator. This is because driver code will continue to use the standard `mbuf(9)` allocator interface to allocate buffers after a panic, but `netdump` needs to avoid relying on the standard allocator. During system initialization, `debugnet` pre-allocates and reserves memory for use after the kernel panics, thus ensuring that mbuf allocations will be successful and won't perturb the state of the kernel more than necessary.

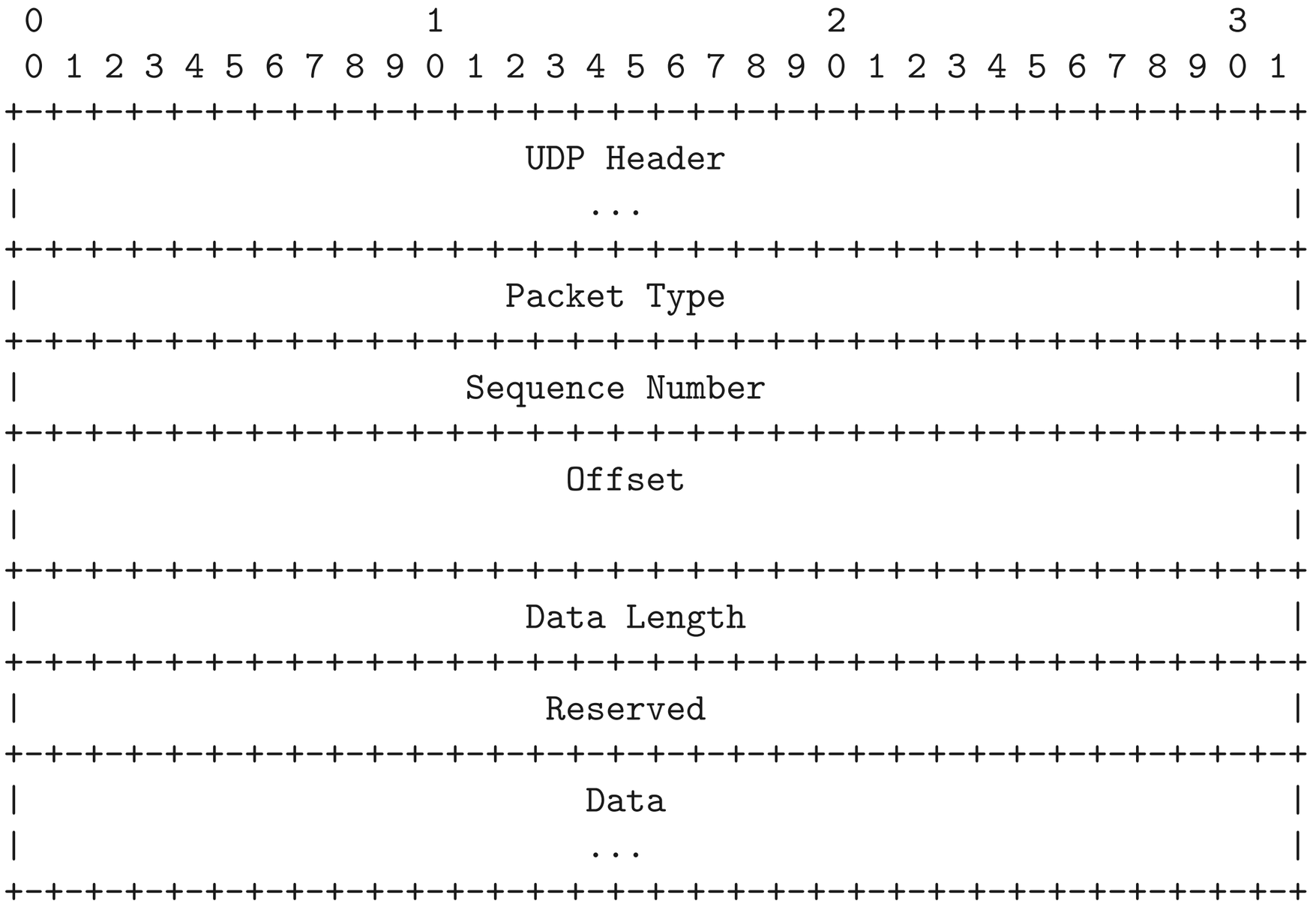
The `debugnet` Protocol

The `debugnet` protocol, in keeping with the requirements of `netdump`, is very simple and specialized to its task. It is implemented on top of UDP and currently uses only IPv4; IPv6 could be supported as well, but so far this has not been implemented.

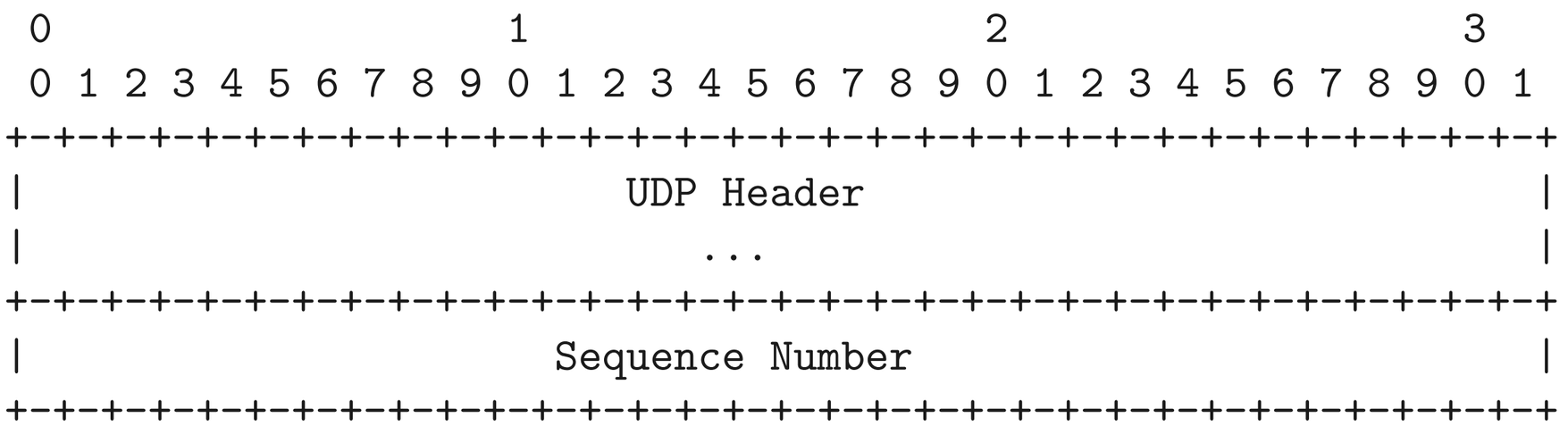
`netdump`s are initiated by the panicking system, which acts as a client, with the server imple-

mented by `netdumpd`. The `debugnet` protocol has two packet types: client messages, and acknowledgements:

Message:



Acknowledgement:



When initiating a `netdump`, the client first has to discover the MAC address of the next-hop router. To do so, its configuration includes a "gateway" IP, for which `debugnet` broadcasts ARP requests.

Once the router address is known, the client first sends a message with type `NETDUMP_HERALD` (1) to the server on port 20023. This establishes a session with the server, which binds to an ephemeral port and sends an acknowledgement to the client at port 20024. All subsequent packets sent by the client go to this ephemeral port. All client messages are acknowledged by the server.

Once the session is fully established, the client begins transmitting kernel dump data. Messages containing this data have type `NETDUMP_VMCORE` (3). Each message gets a unique sequence number and specifies the offset and length of the data relative to the beginning of the kernel dump file. Upon receipt of a `NETDUMP_VMCORE` message, the server writes the data at

the corresponding offset in the dump, and then transmits an acknowledgement. The client will typically transmit a burst of chunks of data and wait for acknowledgements to arrive for all of them before continuing.

Once all of the kernel dump data has been transmitted and acknowledged, the client provides some metadata describing the panic in a `NETDUMP_KDH` (4) message, and then completes the session with a `NETDUMP_FINISHED` (2) message. At this point, the kernel dump is available on the server's file system and can be used for debugging.

Configuring `netdump`

Armed with some knowledge of how **`netdump`** works under the hood, we can explore its configuration. There are, effectively, four configuration variables that **`netdump`** needs to work:

1. the client IP address
2. the server IP address
3. the gateway IP address
4. the interface to use (e.g., `em0`)

Just as with traditional disk-based kernel dumps, **`netdump`** can be configured with **`dumpon`**(8). For example, with a client at 10.0.1.157 on `vtnet0`, the server at 10.0.1.236, and a gateway at 10.0.1.1, one can configure **`netdump`** like so:

```
# dumpon -c 10.0.1.157 -s 10.0.1.236 -g 10.0.1.1 vtnet0
```

Then, on the server, **`netdumpd`** can be run as a foreground program

```
$ netdumpd -d . -D -P ./netdumpd.pid
netdumpd: default: listening on all interfaces
Waiting for clients.
```

This will cause kernel dumps to be saved in the current directory, as specified by the `-d` flag. To test the setup, we can manually trigger a panic and tell the kernel to dump core:

```
# sysctl debug.kdb.panic=1
debug.kdb.panic: 0panic: kdb_sysctl_panic
cpuid = 1
time = 1655412790
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe007c573af0
vpanic() at vpanic+0x151/frame 0xfffffe007c573b40
panic() at panic+0x43/frame 0xfffffe007c573ba0
kdb_sysctl_panic() at kdb_sysctl_panic+0x61/frame 0xfffffe007c573bd0
sysctl_root_handler_locked() at sysctl_root_handler_locked+0x9c/frame
0xfffffe007c573c20
sysctl_root() at sysctl_root+0x213/frame 0xfffffe007c573ca0
userland_sysctl() at userland_sysctl+0x187/frame 0xfffffe007c573d50
sys___sysctl() at sys___sysctl+0x5c/frame 0xfffffe007c573e00
amd64_syscall() at amd64_syscall+0x12e/frame 0xfffffe007c573f30
fast_syscall_common() at fast_syscall_common+0xf8/frame 0xfffffe007c573f30
--- syscall (202, FreeBSD ELF64, sys___sysctl), rip = 0x8011a773a, rsp =
```

```
0x7fffffffd938, rbp = 0x7fffffffd970 ---
KDB: enter: panic
[ thread pid 784 tid 100098 ]
Stopped at      kdb_enter+0x32: movq    $0,0x1279963(%rip)
db> dump
debugnet: overwriting mbuf zone pointers
debugnet_connect: searching for gateway MAC...
netdumping to 10.0.1.236 (02:9a:88:79:b5:0a)
Dumping 257 out of 4057 MB:..7%..13%..25%..32%..44%..56%..63%..75%..81%..94%
netdump finished.
debugnet: restoring mbuf zone pointers

Dump complete
```

On the server, we should see something like the following:

```
New dump from client devvm [10.0.1.157] (to ./vmcore.devvm.0)
.................(KDH from devvm [10.0.1.157])
Completed dump from client devvm [10.0.1.157]
```

Now we have a kernel dump in the directory specified by the `-d` flag!

In this example, the client and server are on the same link. The gateway parameter is thus redundant and can be omitted:

```
# dumpon -c 10.0.1.157 -s 10.0.1.236 vtnet0
```

Configuring `netdump` using `/etc/rc.conf` is a bit trickier. If the relevant IP addresses are static, then they can be passed using the `dumpon_flags rc.conf` variable. If not, one may instead use a hook in the system's DHCP client to invoke `dumpon` once the client address is known. The `dumpon.8` manual page provides an example of how to do this with `dhclient`(8).

Starting in FreeBSD 14.0 and 13.2, `debugnet` will be able to infer a client address in most situations, simplifying configuration.

`netdump` On The Fly

One limitation of `netdump` was the need to configure it in advance of a panic. Starting in FreeBSD 13.0, it is possible to configure `netdump` *after* a panic, from DDB (the in-kernel debugger). This is done using DDB's `netdump` command:

```
# sysctl debug.kdb.panic=1
...
Stopped at      kdb_enter+0x32: movq    $0,0x1279963(%rip)
db> netdump -s 10.0.1.236
debugnet: overwriting mbuf zone pointers
debugnet_connect: searching for server MAC...
netdumping to 10.0.1.236 (02:9a:88:79:b5:0a)
Dumping 258 out of 4057 MB:..7%..13%..25%..31%..44%..56%..62%..75%..81%..93%
netdump finished.
```

```
debugnet: restoring mbuf zone pointers

Dump complete
```

Next Steps

A kernel dump on its own is not very useful: debuggers require that a core dump be paired with an exact copy of the kernel and its debug info. When packaging up a core dump to send to a developer, be sure to include the matching kernel. By default, kernel debug info is split into separate files under `/usr/lib/debug`. Thus, it is usually safest to include the following:

1. the kernel dump file (usually `vmcore.<something>`)
2. the contents of `/boot/kernel/`
3. the contents of `/usr/lib/debug/boot/kernel/`

`netdumpd` sports a `-i` flag which can be used to specify a script to run after a `netdump` completes. This can be used to perform post-processing of kernel dumps. A discussion of kernel debugging itself is outside the scope of this article, but a past article provides lots of information.

`netdump` can be very useful for debugging the kernel in certain environments, but has some limitations. A few mentioned already include the lack of confidentiality, missing IPv6 support, and fixed port numbers. If you find yourself running into such limitations (or bugs, for that matter!), please be sure to report the problem in the FreeBSD project's bug tracker or on the project mailing lists.

MARK JOHNSTON is a software developer and FreeBSD src committer living in Toronto, Ontario, Canada. He currently works for the FreeBSD Foundation and is interested in most aspects of operating system development. When not sitting in front of a computer he enjoys playing in a city dodgeball league with friends.

WIP/CFT: FreeBSD Boot Performance

BY TOM JONES AND MITCHELL HORNE

For a lot of us, the time a FreeBSD system takes to start up is mostly decided by how long the machine takes to go from power through the system firmware and into loader. On server hardware, the first stages to initialize the system and get to loader can take minutes. In this environment, it is hard to worry much about a couple of seconds in the FreeBSD boot process.

In a cloud environment, where you are charged by the second, time spent in the boot process is time wasted, it is time you are billed for from which you don't get any value back. Faster boot times mean that your platform can be used in dynamically scaled environments more easily. If your system takes minutes to boot versus tens or single digit seconds to boot, you must guess more about future load, rather than being able to spin up and down hosts as demand requires them.

Cloud environments are where the first set of tools to improve the performance of FreeBSD boot were added. In 2018, Colin Percival presented "Profiling the FreeBSD kernel boot: From `hammer_time to start_init`" at AsiaBSDCon. Colin's work added a timestamped event log—called TSLOG—to the kernel, which can be used to track the time the kernel spends in each subsystem during boot [https://papers.freebsd.org/2018/bsdcan/percival-profiling_the_freebsd_kernel_boot/].

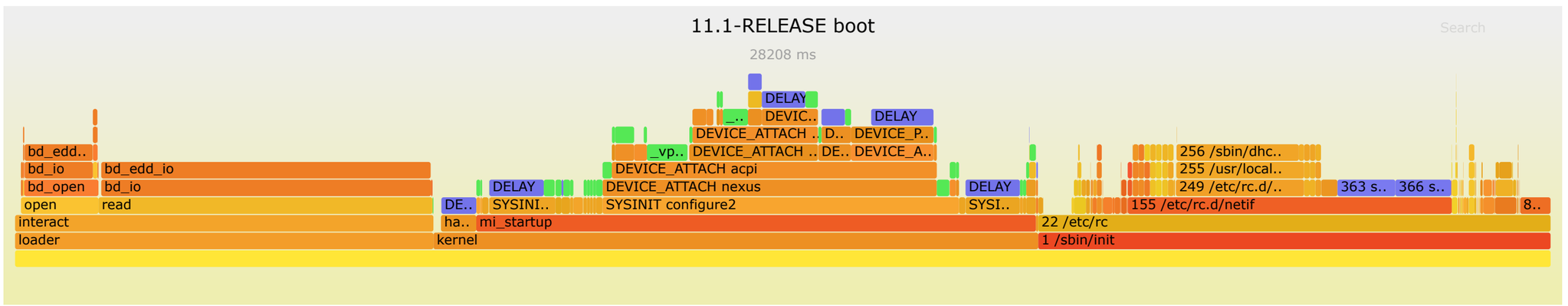
The TSLOG framework traces events that are compiled into kernels with the TSLOG option. Events are implemented with macros which compile to nothing when the option isn't present. TSLOG events go into a buffer and accumulate until the buffer is full and then they are silently dropped, this allows TSLOG to capture early events in preference to later ones in the system.

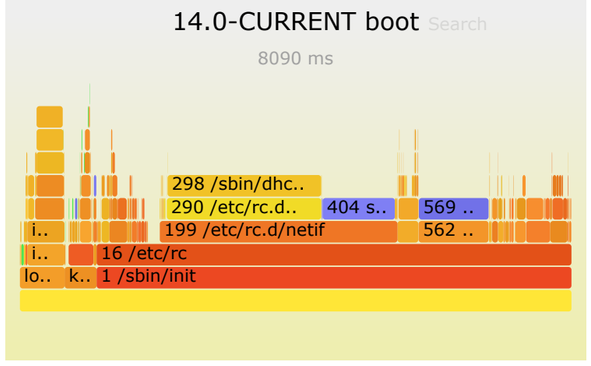
On server hardware, the first stages to initialize the system and get to loader can take minutes.

With TSLOG, the boot time can be tracked and analyzed, to analyze the output log from the system. Colin uses timeshare plots called FlameCharts that work well for this, FlameCharts are like FlameGraphs [https://www.brendangregg.com/flamegraphs.html], but are sorted in chronological order rather than alphabetically.

In each flame graph, the horizontal time represents how much of the boot process is spent in each area, broken down vertically by sub system.

WIP/CFT: FreeBSD Boot Performance





Figures 1 and 2 show how Colin and others have used TSLOG to find slow periods in the boot process. Fgure 1 shows approximately where we started in 2017 and Figure 2 shows the time they have managed to trim out of the boot process in last 5 years, with most of the of the hard work happening recently.

BootTrace

TSLOG is a great way to discover where the kernel was spending time during start up. In early 2022, Mitchell Horne began upstreaming a framework from NetApp called BootTrace. BootTrace allows us to perform the same sort of tracing as TSLOG, but with a couple of major enhancements that give us even more coverage. BootTrace provides the enhanced ability to trace userspace processes, allowing us to cover the rc subsystem, and we can trace the shutdown process.

Shutdown process is difficult to trace with TSLOG because when it is done, the host is gone. In some ways, shutdown (or reboot) is just as important as startup, time spent putting the system to bed helps with system consistency, but the host really isn't doing work during shutdown. BootTrace works around this limitation and gives us a great total view into how the system starts and stops.

How to Contribute

We now have a wealth of tooling to look at the performance of the FreeBSD boot and shutdown process, but developers can only look at the systems and application workloads to which they have access.

Colin is focused on running FreeBSD in Amazon Web Services. There are many, many more cloud providers around and each is likely to have areas that can be tuned to get the fast boot and shutdown times out of a FreeBSD system.

The best way to contribute to improving FreeBSD performance in these areas is to run the tooling we have and share the resulting FlameCharts. Colin is eager to have boot FlameCharts for more and non-cloud systems. You can use the FlameCharts to figure out where the 'hot spots' (or maybe they are 'cold spots') are in the boot and shutdown processes. Once these have been identified to the FreeBSD project, developers can figure out how to improve performance in each case.

TSLOG lead to large improvements in the FreeBSD boot process. FreeBSD boot has gone from ~30 seconds in 2017 on 11.1-RELEASE when Colin started and is down to ~9 seconds in

WIP/CFT: FreeBSD Boot Performance

March 2022 on 14-CURRENT. Some of these gains are in the form of second-sized improvements, but plenty of others came as 100ms reductions in the time subsystems spent initializing.

There is still a way to go to get FreeBSD down to clear Linux, which can boot in ~1 second on EC2. The way to get there is by testing with the boot profiling tools we now have and highlighting areas for improvement to developers in the project.

Tracing Boot Time with TSLOG

TSLOG is reasonably straightforward to run if you have experience building and using your own FreeBSD kernels. You need to build a custom kernel with an 'options TSLOG' and then run the script Colin has provided in the freebsd-boot-profiling repository [https://docs.freebsd.org/en/books/handbook/kernelconfig/].

The latest version of these steps should be on the Boot Time FreeBSD wiki page [https://wiki.freebsd.org/BootTime]. With the FlameChart from Colin's script in hand, you can now follow the much more difficult process of identifying areas in the boot process that take longer than they should.

As of writing, `rtsold` takes up a large portion of the userspace boot time on my systems. This is an important daemon for doing IPv6 autoconfiguration, but it might also be a good starting point for improving your systems boot performance if you use DHCP6 for your network configuration.

Tracing Boot and Shutdown Time with BootTrace

With FreeBSD 14-CURRENT the new BootTrace framework from NetApp is present in the FreeBSD kernel. BootTrace is built into the kernel but is disabled by default.

BootTrace allows the tracing of boot, run time and shutdown time events which it stores in three separate logs. On a FreeBSD 14-CURRENT system from March 2022 you can enable tracing by setting the `kern.boottrace.enabled sysctl` to 1 in `/boot/loader.conf`.

Once enabled, the system will output the boot and run time logs via the `kern.bootrace.log sysctl`.

```
 CPU     msecs      delta process          event          PID     CPUtime IBlks OBlks
   0    177873          0 kernel       sysinit 0x2100001   0        0.00     0     0
   0    177873          0 kernel       sysinit 0x2110000   0        0.00     0     0
   0    177873          0 kernel       sysinit 0x2140000   0        0.00     0     0
   0    177873          0 kernel       sysinit 0x2160000   0        0.00     0     0
  15    182874          0 kernel       sysinit 0xf100000   0        0.00     0     0
...
  15    182874          0 kernel       sysinit 0xfffffff   0        0.00     0     0
  15    182875          1 swapper      mi_startup done     0        0.00     0     0
   9    182880          5 init         init(8) starting... 1        0.00     0     0
   9    182880          0 init         /etc/rc starting... 1        0.00     0     0
  14    202622      19742 init         /etc/rc finished    1        0.61   909    23
Total measured time: 24749 msecs

CPU      msecs      delta process          event          PID     CPUtime IBlks OBlks
  14    202622          0 init         multi-user start    1        0.61   909    23
Total measured time: 0 msecs
```

WIP/CFT: FreeBSD Boot Performance

Shutdown tracing is possible by setting the `kern.bootrace.shutdown_trace sysctl` to 1 and shutting down the system. Shutdown tracing is quite a difficult problem. Until boot performance tracing at the end of the shutdown process, you don't have a system from which to pull the information. To work around this, BootTrace logs the shutdown log to the systems console. To recover the log, you will need to have a console that records message sent to it (such as serial).

The log from my test system looks like this:

```
CPU    msecs     delta process         event                                PID  CPUtime IBlks OBlks
 11  8089055         0 init            single-user from multi-user            1     0.57   914    45
 11  8128611     39556 init            halt & poweroff from multi-user        1     1.38  1274   142
 15  8129849      1238 init            kernel shutdown (clean) started        1     1.69  1274   248
  0  8129849         0 init            system halting...                      1     1.69  1274   248
  0  8129849         0 init            system powering off...                 1     1.69  1274   248
  0  8132523      2674 init            shutdown pre sync complete             1     1.69  1274   248
  0  8132523         0 init            bufshutdown begin                      1     1.69  1274   248
  0  8132524         1 init            shutdown sync complete                 1     1.69  1274   248
  0  8132615        91 init            shutdown unmounted all filesystems     1     1.69  1274   248
  0  8132715       100 init            bufshutdown end                        1     1.69  1274   248
  0  8132715         0 init            shutdown post sync complete            1     1.69  1274   248
  0  8132715         0 init            shutdown final begin                   1     1.69  1274   248
```

BootTrace offers three write only `sysctls`, `boottrace`, `runtrace` and `shuttrace`. When written to an event will be logged as '`${procname}: name`'. These `sysctls` make it easy to add BootTrace logging to your own applications.

Dealing with the Results You Get

Once you have identified an area in the boot process in your environment, you then need to determine if it can be improved and, if possible, suggest how. From TSLOG results, you should look for items in the FlameChart that take up a substantial portion of the boot time first and dig down into those in the FlameChart svg.

More subsystems and userspace components could benefit from BootTrace events. Adding these into your workloads' start up and shutdown scripts can provide insight into where issues are and you might uncover issues that impact FreeBSD users generally.

Some subsystems build in long delays to allow other network hosts to synchronize. These sorts of delays might be prime candidates for trimming down the boot process.

FreeBSD developers welcome good bug reports via the bug tracker, if you can provide hints or patches that will fix boot performance issues, then all the better.

The FreeBSD boot process is never going to be finished. Over time, as services and hardware change, it is going to vary. A lot of the most significant boot performance benefits came from reducing interactions with emulated legacy hardware. On systems with many cores, loads of time was spent writing characters to the emulated VGA console. Over time, hardware will age out of use and faster or slower equipment will follow. With your help we can keep FreeBSD competitive in the cloud and stop wasting time starting up machines.

TOM JONES wants FreeBSD-based projects to get the attention they deserve. He lives in the North East of Scotland and offers FreeBSD consulting.

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

PRACTICAL PORTS

Setting up an NFSv4 Fileserver on OpenZFS

BY BENEDICT REUSCHLING

We recently experienced a small disaster at work that got me re-thinking our current file-serving strategy. We deploy distributed applications like MongoDB, Hadoop, Spark and others via Ansible. The binaries to install these don't come from a software repository but are downloaded from the vendors website because of the up-front registration to get the Enterprise edition with extra features. The archives that contain these binaries are quite large (even compressed) and copying them from the Ansible controller to the target machines over the network usually takes some time.

A while ago, we figured there could be a faster way to do this: put all those binaries into a network share (coming from Ceph in this case), mount them on the target machine, and then point Ansible to them. For Ansible, this looks like the files are "local" to the target machine, so it installs right from the share. Even in those instances where we still need to copy the files to a local directory (in case of Hadoop or Spark, which is nothing more than an archive to extract), it's much faster than transferring them from the Ansible controller first.

When running the script again, things went wrong…

One of the last actions in the deployment playbook is to unmount the share. The share gets mounted as read-only so that no accidents can happen, and we simply don't need write access during the deployment. Even if the share does not unmount properly and may still be around when our students start using the server, they cannot delete any important files because of the missing w-bits.

All was fine and good until one fateful day when one of our student helpers working on a new version of the NoSQL databases needed to put a newer version of said software on the share. It was easy enough to mount the share read-write (they are allowed to do that as part of their job) and put the new binaries next to the others. When running the script again, things went wrong: the playbook ran and faithfully did its work. What I get later is an email from the student telling me that the script mounted the share (still read-write) and did some cleanup

action afterwards. However, what it did not check for is whether the share was cleanly unmounted from the system. Which did not happen in this case. So, the cleanup job happily ran over the still mounted Ceph share and thoroughly cleaned all the files in there. Oops!

The email from the student asked whether I still had an older snapshot from the Ceph share to restore the files. I did not, as the share was provided by our IT department. When inquiring there directly, it turned out that they did not backup that share at all. A new backup system was put in place recently but was not ready yet to do the backups. There were also no other backups available, even though the Ceph share was replicated among three separate buildings on campus. No luck there.

I was rather calm about this for several reasons: first, this could have easily happened to me, second, the files could be restored and nothing on the share was irreplaceable. Third, this provided me with an opportunity to make things more robust in case this happened again in the future, which is the best takeaway from events like this.

I updated the playbooks to do an extra check after the unmount task to determine whether the share was actually unmounted. If not, it would do a forced unmount, and if that also failed, then the playbook execution would stop at that point. There are usually good reasons for a network share to not unmount properly (typically when there are some files still accessed), but not continuing would be better than risking another accidental data erase.

Since I did not have full control over the Ceph share and my network share needs are not that big, I decided to run my own. Instead of Ceph, I chose FreeBSD's ZFS with its integrated NFSv4 sharing. That way, I could run regular snapshots which don't grow too much when there are no changes on a mostly read fileshare. Also, instead of relying on simply mounting the share read-only, I could set the ZFS property of the same name to "`on`." With `readonly=on`, not even the root user would be able to remove files on a dataset with that property. Plus, I could get the regular data-integrity checks that ZFS does and maybe some space savings from compressing that dataset.

Implementing the Solution

Here are my notes setting up the NFSv4 server on a FreeBSD system. First, I went to `/etc/rc.conf` and added the following lines:

```
nfs_server_enable="YES"
nfsv4_server_enable="YES"
nfsuserd_enable="YES"
hostid_enable="YES"
rpcbind_enable="YES"
mountd_enable="YES"
rpc_lockd_enable="YES"
rpc_statd_enable="YES"
```

This enables the NFS server in general, ensures proper permissions using `nfsuserd`, and does correct locking for the RPC calls that NFS makes. Next, I checked that `/etc/exports` contained only the following line:

```
V4: /
```

This is telling the NFS server on FreeBSD that it should use NFS version 4, but that the actual definition of what paths are shared will be determined by ZFS. I've read on the web that this file could even be empty now, but it does not hurt to have it in there either.

I created the ZFS dataset that will do the file sharing like any other:

```
zfs create -o atime=off zroot/fileshare
zfs set mountpoint=/fileshare zroot/fileshare
```

The nice thing about ZFS is its inheritance. If I decide to create a sub-dataset below fileshare that should also serve NFS, I don't have to separately configure it, as it already inherits all properties from the parent (except the mountpoint). If I don't want to share the sub-dataset, then I can just as easily disable the sharing by setting the `sharenfs` property to off (which is the default).

Let's first copy some files over to the NFS share and then set the `readonly` property to "on" (that's what we came here to do in the first place):

```
cp /some/important/files /fileshare
zfs set readonly=on zroot/fileshare
```

Clever ZFS users could take this one step further by taking a snapshot of the share, mounting that into the system and then sharing that over the network. This also ensures that the files can't be changed, as ZFS snapshots are read-only in nature. But I'll leave that as an exercise to you for another day.

In the `sharenfs` property, I can define all the parameters that I would normally need to put into a separate NFS config file. That way, the information about how to share this dataset over NFS stays with it even when it gets sent to a different pool. This all-in-one-place nature of ZFS is making configuration much simpler, as there are fewer places to look for errors.

In my case, I only wanted to share the NFS on a certain subnet. You can also list hostnames or IP-addresses separated by commas instead. That way, you limit who can mount the share to a number of hosts for some extra security.

```
zfs set sharenfs="-network 192.168.0.0 -mask 255.255.255.0
-maproot=user,-alldirs" zroot/fileshare
```

The `maproot=user` part defines that if a user accesses the share and there are files with the user's permissions, then the server maps them to the same local permissions, even though they may be different on the server. For example, Joe may have a local `uid/gid` of 2000, while on the NFS server, the users all begin starting at 3000. The NFS server will have Joe's files set with `uid/gid` as 3000, but when Joe accessed the share, he will see his familiar 2000 of the local system to not get confused. The `-alldirs` option allows mounting at any directory within `/fileshare`. Find out more about these and other options by reading exports(5).

That's all for the server part. We need to start all the services listed in `/etc/rc.conf` to start sharing the mounted dataset:

```
service nfsd start
service mountd start
service nfsuserd start
```

Some of these services should be automatically started with the NFS server, but carefully check the status output from each of these services to see that they are running. The output of

```
sockstat -4l
```

as well as

```
rpcinfo
```

and

```
nfsstat
```

help you determine any problems when the share won't mount for some reason.

Another way to see the currently shared datasets is running

```
cat /etc/zfs/exports
```

to see the whole list.

Next, we look at the client. I use FreeBSD and Ubuntu Linux systems to mount the share and describe what each needs to access it. Starting with the FreeBSD client, it needs only a few lines in `/etc/rc.conf`:

```
nfsuserd_enable="yes"
hostid_enable=YES
nfscbd_enable=YES
```

The NFS user daemon takes care of the mapping of user ids and groups from the server as described above. The `hostid` uniquely identifies this system to the NFS server and the NFS callback daemon handles callback requests from the server. The man page for it assures me that mounts will work without it, but it won't hurt to activate it right away so as not to scratch my head about it later.

Starting those services right away, we can look at what the NFS server (called `myfiler`) is offering to us:

```
showmount -e myfiler
```

This should give us a list of exported shares that we can mount. From the command-line, the mount command is invoked as follows:

```
mount -t nfs -o nfsv4 myfiler:/fileshare /media
```

If you like to have this share mounted each time your system starts, add it to `/etc/fstab` like this:

```
myfiler:/fileshare          /media          nfs       rw,tcp,noatime,nfsv4 0 0
```

The `noatime` and `rw` options are not strictly necessary as we took take care of them from the ZFS side earlier, but the `nfsv4` must be there to let the system know that it is talking to version 4 of NFS.

At this point, you should be able to mount the share, see the files in there together with the correct user and group IDs.

Over on a Ubuntu Linux system, we first need to install the NFS server bits as they are not part of the base system:

```
apt install nfs-common
```

Use the package distribution of your particular distribution, the rest of the setup should be the same from here on. It turns out that this is all that's needed. Mounting the share on the command line is done via:

```
mount -t nfs -onfsvers=4 myfiler:/fileshare /media
```

Of course, mounting can happen to any other local directory that exists, not just to /media. I just use it because it exists and is usually empty. Mounting over an existing directory will hide its contents until the next time the NFS share is unmounted again. Ensure that you don't do this on any vital directories that this system needs to run properly. Whatever directory you picked, in case you also want to have the share mounted each time the Linux system boots, put this line into `/etc/fstab`:

```
myfiler:/fileshare          /media          nfs       rw,nfsvers=4     0   0
```

That's all. The server that I put in place will regularly copy the contents from the NFS share to the Ceph to have an extra backup. But I worry less now that ZFS is backing my files: regular snapshots and the `readonly` property should avoid future mistakes like the one described above.

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.

WeGet letters

by Michael W Lucas

Dear Most Honest Advice Columnist in Technology,

I've been told that I must write my employer's disaster recovery plan. Seriously, this whole exercise is bogus. We don't have the resources to do actual disaster recovery, nor will the cheapskates I work for pay for any preparedness. I gotta come up with something though. What's the quickest, easiest way to ditch this problem so I can get back to my real job?

—Falling Behind on Doom WADs

Dear Experienced Sysadmin,

The main problem with cynicism in IT professionals is that it is inadequate. We think that by becoming cynical we brace ourselves to cope with the worst this industry can do to us, but cynicism is not a Boolean. One is neither "cynical" or "naïve." Cynicism is an evil overlord's dungeon. The more you explore it, the deeper you learn you can go, but you can never quite delve the worst depths. Cynicism can always be deepened and sharpened.

Your objection is not rooted in needing a disaster recovery plan. Anyone who works with computers knows the innate treachery of all hardware and software. Your real problem is that you have been assigned work that will never be used. Your organization has not devoted any resources to disaster recovery. It's a checkbox on the Checkbox Compliance Chief's list. The company believes that they need to fill the checkbox, not actually plan for the disaster. Your question does nothing but illuminate your lamentable shortage of cynicism--when the solution is obvious:

Be the disaster you want to see in the world.

Bad things happen. Everybody knows this intellectually. People don't internalize how vulnerable they are until they experience a short sharp shock of ruin. By providing seamless and robust computing experiences, you are depriving your organization of necessary inoculations of panic and despair. People do not believe in the need for disaster recovery until they have experienced disasters.

In the days before Wi-Fi, I was responsible for internal technology at a consulting firm. One of my duties was "security." If you are pretending to run a secure environment, every piece of equipment should require some sort of authentication for configuration. We had many printers without passwords and a large contingent of people who did not want to bother with passwords on them. I could have spent months or years arguing about the importance of passwords, or I could accept consensus and wait. One day, the company owners had a whole bunch of visitors in the office for a critical meeting. Several were connected to the office network so they

could get Internet. Ten minutes into the presentation, some heinous prankster connected to an over–welcoming printer and swapped its IP address and default gateway. Moments later, I was notified that the entire company network was broken. Thanks to my vast expertise with a packet sniffer, I was able to identify and fix the problem within minutes. Sadly, I was unable to determine who performed that malicious prank at such an inconvenient moment, but the CTO mandated my preferred password policy thirty minutes later. My recommendation for setting up an isolated visitor network in the big conference room was also immediately accepted, which made implementing future disasters more difficult but not impossible. A sysadmin should always rise to a challenge, after all.

A disaster recovery policy need not be onerous. Look at your organization's functions. Which are critical, and which should they have stopped doing years ago? The most vital function, of course, is payroll. Being assigned the duty of disaster recovery planner gives you the power to be nosy. Dig into your organization to verify that no matter what, you will be paid. Of all the disaster recovery tasks, this is always the easiest to get cooperation on. Everybody is only there for the money, after all. Even folks who claim to be entirely "mission driven" become surly when the money doesn't show. Your payroll person has done their best, but I have never seen an accountant who truly understands technology's eagerness for perfidy. You will find problems. Write up your recommendations for this most vital of tasks. This will get you credibility, because you've demonstrated that you understand the organization's most important role in people's lives.

From there, expand. Present each part of your plan as you create it. If you get pushback, tell people "that's fine" and change your plan to read *in a disaster, this function will not be restored*. Hey, it's a plan. It's written down. It's even honest. What more could anyone ask? Sometime later--not too soon, you don't want people noticing any sort of pattern or trend--a small disaster might make them change their minds and you will already have a plan prepared.

No plan can be considered complete until it is tested.

No plan can be considered complete until it is tested. Ideally, test each part of your plan as you write it. There is no need for anyone to know all the flaws in the first draught of your plan. It would only worry them. Verify that you can restore backups on decommissioned servers over junk switches. Make sure that you have enough spinning rust in the scrap pile to hold the important databases as well as the stupid ones that the CEO insists remain active because of this one personally traumatizing incident fifteen years before. Once you've successfully tested everything, schedule a disaster recovery test that you tell people about. It will still have problems, of course, but nobody would believe it was a real test if it ran perfectly.

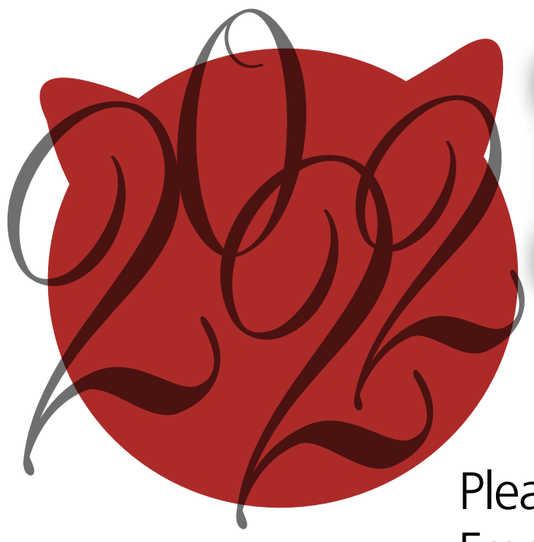
The nice thing about disaster recovery plans is that they are always used. Even if your organization's headquarters is not immolated by fire-breathing hippopotami before your next performance review, one day your fancy all–SSD storage array will implode into a naked singularity, and you'll be forced to retrieve the derelict spinning rust array from the scrapheap and try to make it stagger along under load. The DBAs won't be happy, but DBAs are never happy so ignore their whining. Network gear will fail unexpectedly, because even though you've configured everything to spew its errors to syslog, nobody reads the logs. Fortunately, HP still honors their famous lifetime warranty on their 10/100 switches, so if you make sure they still work beforehand you will have a ready-steady replacement.

You can't do much about the greatest disaster of all, people. While you can list solutions in your plan, implementing any of them would violate the law. You might choose to carry out such plans anyway but do remember that documentation is considered evidence of premeditation.

Yes, DOOM is fun, but real disasters can be more fun. Disaster recovery plans do not have to be pointless wastes of your time. Hone your cynicism, seize control of the inevitable, and use disasters to improve your life.

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

Many people consider **MICHAEL W LUCAS** an expert in disasters, but only because he is so often found at the center of them. He is the author of *Absolute FreeBSD*, the *FreeBSD Mastery* books, as well as books documenting the debacles of PAM, SNMP, TLS, DNSSEC, and more. Get a full list at https://mwl.io. His most recent book is *Letters to Ed(1)*, a collection of the first three years of this column. Spending your hard-earned money on any of them would be another disaster.



Events Calendar

BSD Events taking place through November 2022

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.

SCALE 19x

July 28-31, 2022
Los Angeles, CA
https://www.socallinuxexpo.org/scale/19x

SCaLE is the largest community-run open-source and free software conference in North America.

- Check out the FreeBSD Workshop with Roller Angel on Friday, July 29
- Visit the Foundation Booth #502 July 29-31
- Use code BSD for 50% off registration

EuroBSDCon FreeBSD Developer Summit

September 15-16, 2022
Vienna, Austria
https://wiki.freebsd.org/DevSummit/202209

Join us for talks and discussion groups on day 1 followed by a hackathon on day 2.
The CFP is open.

EuroBSDCon 2022

September 15-18, 2022
Vienna, Austria
https://2022.eurobsdcon.org

This yearly conference gives the exceptional opportunity to learn about the latest news from the BSD world.

All Things Open

October 30 - November 2, 2022
Raleigh, NC
https://2022.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 media partner for this year's All Things Open.

FreeBSD Office Hours

https://wiki.freebsd.org/OfficeHours
Join members of the FreeBSD community for FreeBSD Office Hours. From general Q&A to topic-based demos and tutorials, Office Hours is a great way to get answers to your FreeBSD-related questions.

Past episodes can be found at the FreeBSD YouTube Channel.
https://www.youtube.com/c/FreeBSDProject.