FreeBSD® JOURNAL

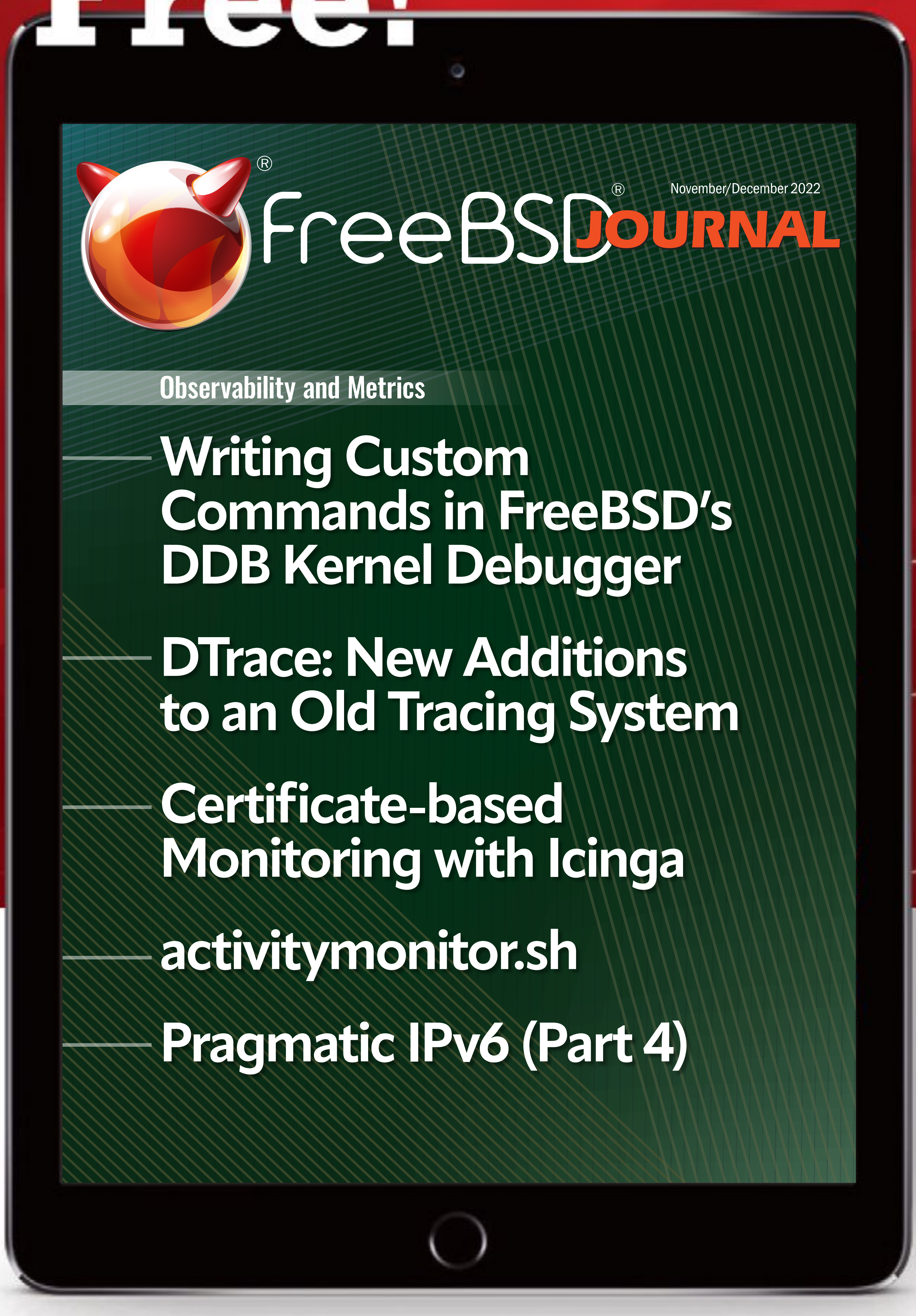
July/August 2023

FreeBSD on Firecracker

Jail-based DNS AdBlocking Tutorial

Jail Orchestration with pot and nomad

Recollections: An Interview with Doug Rabson



FreeBSD® JOURNAL

November/December 2022

Observability and Metrics

Writing Custom Commands in FreeBSD's DDB Kernel Debugger

DTrace: New Additions to an Old Tracing System

Certificate-based Monitoring with Icinga

activitymonitor.sh

Pragmatic IPv6 (Part 4)

In Memoriam:

Hans Petter William Sirevåg Selasky

The FreeBSD community was recently saddened by the tragic death of one of its most prolific contributors. We learned that Hans Petter Selasky passed away in a traffic accident in Lillesand, Norway on June 23, 2023 at the age of 41. Hans was an incredibly brilliant and kind person, and made many valuable contributions to FreeBSD. He was preceded in death by his father Gordon, and is survived by his mother, Inger Elisabeth, his brothers Mark and Leif Conrad, and his nieces and nephews Petra, David and Signe.

Photo courtesy Ollivier Robert

Hans began contributing to FreeBSD roughly 25 years ago, with fixes to FreeBSD's ISDN support. He was a FreeBSD committer for nearly 15 years, and was best known for re-writing and maintaining the USB stack. Hans wrote the webcamd package which supports running Linux webcam drivers in userspace on FreeBSD, and which enables those of us using FreeBSD on the desktop to participate in modern teleconferencing. Most recently, he worked for Mellanox (now NVIDIA) to support their ConnectX series of high speed NICs on FreeBSD. Hans's work included major contributions to the kernel TLS framework, as well as support for NIC kTLS send and receive offload in the mce(4) driver, and many improvements to the Linux device driver compatibility layer.

I first met Hans in 2015, in the context of his work on the mce(4) driver for Mellanox NICs. We worked together to make the mce(4) driver one of highest performance NIC drivers in FreeBSD. It was during this time that I learned how brilliant he was. He often had ideas that sounded "crazy," but were actually brilliant. One example of this was his idea to sort incoming TCP packets using the NIC-provided RSS flow identifiers in order to present LRO with all packets from the same TCP connection back to back. This idea, which I initially discounted as impractical, was crucial to Netflix being able to meet our performance target of serving 100Gb/s of video traffic from a single machine, and continues to save Netflix a large amount of CPU resources.

Hans was a kind and welcoming person. The first time I attended EuroBSDCon was in 2019 in Lillehammer, Norway, where Hans insisted on playing host to me. Hans had driven across Norway from his home in Grimstad to EuroBSDCon in Lillehammer with his father, and took me around to see the Olympic ski jump and several other sites in the town. He then took me out to dinner and back to the house he'd rented with his father for an evening of great conversation.

Outside of FreeBSD, Hans's hobbies included music and mathematics. He was active in his church, and contributed to its sound team. He was a loving and dedicated uncle to his nieces and nephews. He loved animals, especially his cat Pumba.

Even if you don't use FreeBSD yourself, odds are good that Han's work touches your daily life. For example, if you use a Playstation, chances are you are using Hans's USB stack. If you watch Netflix, the odds are good that the show you're watching was delivered to you by a ConnectX NIC running Hans's mce(4) driver.

Hans, if you are reading this, know that you will be missed.

By Drew Gallatin

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

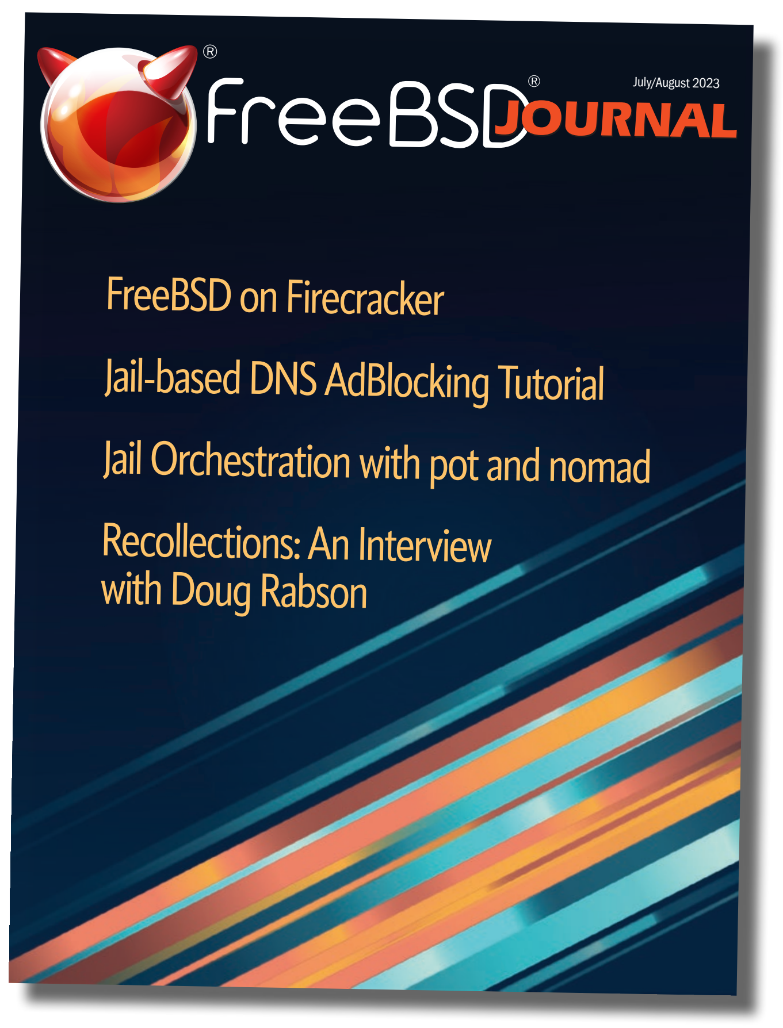
Editor-at-Large • James Maurer
maurer.jim@gmail.com

Design & Production • Reuter & Associates

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

LETTER from the Foundation

Dear Readers,


FreeBSD JOURNAL
July/August 2023
FreeBSD on Firecracker
Jail-based DNS AdBlocking Tutorial
Jail Orchestration with pot and nomad
Recollections: An Interview with Doug Rabson

I'm excited to introduce our July/August edition! For almost a decade, the FreeBSD Foundation has been producing the *Journal*, and we take pride in delivering informative and helpful content to BSD and Unix enthusiasts globally.

Within this issue, you'll find enlightening pieces on Containers and Cloud, including articles on ports, Jails, and virtualization. Additionally, you're in for a fun read with Michael Lucas' "We Get Letters" segment, which is a blend of Michael's wit and seriousness, making it a truly enjoyable read.

Drew Gallatin wrote a beautiful tribute to long-term FreeBSD developer Hans Petter Selasky, who unexpectedly passed away in June. He will truly be missed in our community, not only for his expertise in many areas of the operating system, but for his kindness and support of others in the Project.

Finally, check out the Events Calendar, which highlights upcoming events that might be of interest to folks in our community.

Thank you for your ongoing support of FreeBSD, the FreeBSD Foundation, and all those engaged in this amazing Project, which recently celebrated its 30th anniversary! Now, sit back, relax, and enjoy all the great content in this issue!

Deb Goodkin
Executive Director of the FreeBSD Foundation

FreeBSD® JOURNAL®

July/August 2023

6 FreeBSD on Firecracker
By Colin Percival

14 Jail Orchestration with pot and nomad
By Luca Pizzamiglio

21 Conference Report:
C is to BSD What Latin is to Us
By Corey Stephan

26 Recollections:
An Interview with Doug Rabson
By Tom Jones

34 Practical Ports: Jail-based
DNS AdBlocking Tutorial
By Benedict Reuschling

3 In Memoriam: Hans Petter Selasky
By Drew Gallatin

4 Foundation Letter
By Deb Goodkin

46 We Get Letters: Virtualization
is a Necessary Evil
By Michael W Lucas

49 Events Calendar
By Anne Dickison

FreeBSD on Firecracker

BY COLIN PERCIVAL

A large amount of fantastic open source software originates from "scratching an itch". Such was the case with Firecracker: In 2014, Amazon launched AWS Lambda as a "serverless" compute platform: Users can provide a *function* — say, ten lines of Python code — and Lambda provides all of the infrastructure between an HTTP request arriving and the function being invoked to process the request and generate the response.

To provide this service efficiently and securely, Amazon needed to be able to launch virtual machines with minimal overhead. Thus was born Firecracker: A Virtual Machine Monitor which works with Linux KVM to create and manage "microVMs" with minimal overhead.

Why FreeBSD on Firecracker?

In June 2022, I started work on porting FreeBSD to run on Firecracker. My interest was driven by a few factors.

First, I had been doing a lot of work on speeding up the FreeBSD boot process and wanted to know the limits that could be reached with a minimal hypervisor.

Second, porting FreeBSD to new platforms always helps to reveal bugs — both in FreeBSD and on those platforms.

Third, AWS Lambda only supports Linux at present; I'm always eager to make FreeBSD more available in AWS (although adoption in Lambda is out of my control, Firecracker support would be a necessary precondition).

The largest reason, however, was simply *because it's there*. Firecracker is an interesting platform, and I wanted to see if I could make it work.

> Porting FreeBSD to new platforms always helps to reveal bugs — both in FreeBSD and on those platforms.

Launching the FreeBSD Kernel

While Firecracker was designed for Lambda's needs — launching Linux kernels — there were patches available from 2020 that added support for the PVH boot mode in addition to "linuxboot". FreeBSD had support for PVH booting under Xen, so I decided to see if that would work.

Here I ran into the first problem: Firecracker could load the FreeBSD kernel into memory but couldn't find the address at which to start running the kernel (the "kernel entry point"). According to the PVH boot protocol, this value is specified in an ELF Note — a piece of special metadata stored in ELF (Executable and Linker Format) files. It turned out that there are two types of ELF Notes: PT_NOTEs and SHT_NOTEs, and FreeBSD wasn't providing the one Firecracker was looking for. A small change to the FreeBSD kernel linker script fixed this, and now Firecracker was able to start executing the FreeBSD kernel.

That lasted for about 1 microsecond.

Early Debugging

FreeBSD has wonderful debugging functionality, but if your kernel crashes before the debugger is initialized or the serial console is set up, you're not going to get much help. In this case, the Firecracker process exited, telling me that the FreeBSD guest hit a triple-fault — but that's all I knew.

It turned out, however, that this was enough information to get me started, given a bit of creativity. If the FreeBSD kernel execution reached a `hlt` instruction, the Firecracker process would keep running but use 0% of the host's CPU time (since it was virtualizing a halted CPU). As such, I could distinguish between "FreeBSD is crashing before this point" and "FreeBSD is crashing after this point" by inserting a `hlt` instruction — if Firecracker exited, I knew that it was crashing before reaching that instruction. Thus started a process I referred to as "kernel bisection" — rather than bisecting a list of commits to find one which introduced a bug (as in `git bisect`) I would do a binary search through the kernel startup code to find the line of code that was making FreeBSD crash.

Xen Hypercalls

The first thing I discovered in this process was Xen hypercalls. The PVH boot mode originated as the *Xen/*PVH boot mode, and FreeBSD's PVH entry point was, in fact, an entry point specifically for booting under Xen — and the code made the quite reasonable assumption that it was, indeed, running inside Xen and could thus make Xen hypercalls. KVM (which provides the kernel functionality used by Firecracker) is not Xen, of course, so it doesn't provide those hypercalls; attempting to use any of them resulted in the virtual machine crashing. As an initial workaround, I simply commented out all the Xen hypercalls; later, I added code to check `CPUID` for a Xen signature before making calls e.g., to write debugging output to the Xen debug console.

FreeBSD has wonderful debugging functionality, but if your kernel crashes before the debugger is initialized or the serial console is set up, you're not going to get much help.

There was one Xen hypercall that provided essential functionality, however: Retrieving the physical memory map. (Of course, inside a hypervisor, the "physical" memory is only virtually physical. It's turtles all the way down.) Here, we're saved by the fact that Xen/PVH was retroactively declared to be *version 0* of PVH boot mode: From *version 1* onwards, a pointer to the memory map is passed via the PVH `start_info` page (a pointer to which is provided in a register when the virtual CPU starts executing). I had to write code to make use of the PVH *version 1* memory map instead of relying on a Xen hypercall to get the same information, but that was easy enough.

Another related issue arose from how Xen and Firecracker arrange structures in memory: Whereas Xen loads the kernel first and then places the `start_info` page at the end, Firecracker placed the `start_info` page at a fixed low address and then loaded the kernel afterwards. This would have been fine but that FreeBSD's PVH code – having been written with Xen in mind — assumed that the memory immediately after the `start_info` page would be free for use as scratch space. Under Firecracker, that very quickly meant overwriting the initial kernel stack — not an optimal outcome! A change to FreeBSD's PVH code to assign scratch space after *all* the memory regions initialized by the hypervisor fixed this problem.

ACPI — or Lack Thereof!

On x86 platforms, FreeBSD normally makes use of ACPI to learn about (and in some cases control) the hardware on which it is running. In addition to discovering things via ACPI which we might commonly think of as "devices" — disks, network adapters, etc. — FreeBSD also learns about fundamental things like CPUs and interrupt controllers via ACPI.

Firecracker, being deliberately minimalist, does not implement ACPI, and FreeBSD gets upset when it can't figure out how many CPUs it has or where to find their interrupt controllers.

Fortunately, FreeBSD has support for the historic Intel MultiProcessor Specification, which provides this critical information via an "MPTable" structure; it's not part of the GENERIC kernel configuration, but for running in Firecracker, we want to use a stripped-down kernel configuration anyway, so it was easy to add `device mptable` to make use of what Firecracker provides.

Except… it didn't work. FreeBSD still couldn't find the information it needed! It turned out that Linux has bugs in how it finds and parses the MPTable structure — and Firecracker, being designed to boot Linux, provided the MPTable in a way that Linux supported but was not in fact compliant with the standard. FreeBSD, having an implementation independently written to follow the standard, failed both to find the (incorrectly located) MPTable and to parse the (invalid) MPTable once it was found.

So now FreeBSD has a new kernel option: You can add `options MPTABLE_LINUX_BUG_COMPAT` to your kernel configuration if you need bug-for-bug compatibility with Linux's MPTable handling — and with that, FreeBSD managed to boot a bit further in Firecracker.

> You can add options MPTABLE_LINUX_BUG_COMPAT to your kernel configuration if you need bug-for-bug compatibility with Linux's MPTable handling.

Serial Console

One of the few emulated devices — as opposed to *virtualized* devices like the Virtio block and network devices — provided by Firecracker is the serial port. In fact, in a common configuration, when you launch Firecracker, the standard input and output of the Firecracker process become the serial port input and output of the VM, making it seem like the guest OS is just another process running inside your shell (which, in a certain sense, it is). At least, that's how it's supposed to work.

By this point in the process of bringing up FreeBSD inside Firecracker I was able to boot a FreeBSD kernel with a root disk compiled into the kernel image — I didn't have the virtualized disk driver working yet — and read all the console output from the kernel. After all the kernel console output, however, FreeBSD entered the userland portion of the boot process, and I got 16 characters of console output — and then it stopped.

Funnily enough, I'd seen that exact symptom over ten years earlier, when I was first getting FreeBSD working on EC2 instances. A bug in QEMU resulted in the UART not sending an interrupt when the transmit FIFO emptied; FreeBSD wrote 16 bytes to the UART and then wouldn't write anymore because it was waiting for an interrupt which never arrived. Modern EC2 instances run on Amazon's "Nitro" platform, but in the early days they used Xen and devices were emulated using code from QEMU. Somehow, a decade after this bug was fixed in QEMU, exactly the same bug was implemented in Firecracker; but luckily for me, the workaround I put into the FreeBSD kernel — `hw.broken_txfifo="1"` — was still available, and adding that loader tunable (which, since Firecracker loads the kernel directly without going through the boot loader, meant compiling the value into the kernel as an environment variable) fixed the console output.

I then found that the console *input* was also broken: FreeBSD didn't respond to anything I typed into the console. In fact, tracing the Firecracker process, I found that Firecracker wasn't even reading from the console — because Firecracker thought that the receive FIFO on the emulated UART was full. This turned out to be another bug in Firecracker: While initializing the UART, FreeBSD fills the receive FIFO with garbage to measure its size and then flushes the FIFO by writing to the FIFO Control Register. Firecracker didn't implement the FIFO Control Register, so it was left with a full FIFO and quite sensibly didn't try to read any more characters to put into it. Here, I added another workaround to FreeBSD: If `LSR_RXRDY` is still asserted after we try to flush the FIFO via the FIFO Control Register — which is to say, if the FIFO didn't empty as requested — we now proceed to read and discard characters one by one until the FIFO empties. With this, Firecracker now recognized that FreeBSD was ready to read more input from the serial port, and I had a working bidirectional serial console.

Virtio Devices

While a system without disks or network could be useful for some purposes, before we can do very much with FreeBSD, we're going to want those devices. Firecracker supports Virtio block and network devices and exposes them to virtual machines as `mmio` (memory-mapped I/O) devices. First step to getting these working in FreeBSD: Add `device virtio_mmio` to the Firecracker kernel configuration.

Next up, we need to tell FreeBSD how to find the virtualized devices. FreeBSD expected `mmio` devices to be discovered via FDT (Flattened Device Tree), which is a mechanism commonly used on embedded systems; but Firecracker passes device parameters via the kernel command line with directives such as `virtio_mmio.device=4K@0x1001e000:5`. Second step to getting these working in FreeBSD: Write code for parsing such directives and creating `virtio_mmio` device nodes. (Once we create the device node, FreeBSD's regular process for device probing kicks in and the kernel will automatically determine the type of Virtio device and hook up the appropriate driver.)

If we have multiple devices however — say, a disk device and a network device — another problem arises: Firecracker passes directives the way Linux expects — as a series of key=value pairs on the kernel command line — while FreeBSD parses the kernel command line as environment variables… meaning that if there were two `virtio_mmio.device=` directives passed on the command line, only one was retained. To fix this, I rewrote the early kernel environment parsing code to handle duplicate variables by appending a numbered suffix: We end up with `virtio_mmio.device=` for one device and `virtio_mmio.device_1=` for the second device.

With this, I finally had FreeBSD booting and discovering all of its devices — but one more problem arose with disk devices: If I shut down the virtual machine uncleanly, on the next boot the system would run `fsck` on the filesystem, and the kernel would panic. It turned out that `fsck` is one of very few things in FreeBSD that will cause non-page-aligned disk I/Os, and FreeBSD's Virtio block driver was causing a kernel panic when trying to pass unaligned I/Os to Firecracker.

When an I/O crosses a page boundary — as will happen with page-sized I/Os which aren't aligned to page boundaries — the physical I/O segments will typically not be contiguous; most devices can handle I/O requests which specify a series of segments of memory to be accessed. Firecracker, being extremely minimalist, does not do this: Instead, it accepts only a single data buffer — meaning that a buffer that crosses a page boundary can't simply be split into pieces the way it would with other Virtio implementations. Fortunately, FreeBSD has a system in place specifically for taking care of device complications like this: `busdma`.

This was probably the hardest part of getting FreeBSD running in Firecracker, but after several attempts, I think I finally got it right: I modified FreeBSD's Virtio block driver to use `busdma`, and now unaligned requests are "bounced" (aka. copied via a temporary buffer) in order to comply with the limitations of the Firecracker Virtio implementation.

Once I had FreeBSD up and running in Firecracker, it rapidly became clear that there were some improvements to be made.

Revealed Optimizations

Once I had FreeBSD up and running in Firecracker, it rapidly became clear that there were some improvements to be made. One of the first things I noticed was that, despite having 128 MB of RAM in the virtual machine I was testing, the system was barely usable, with processes being frequently killed due to the system running out of memory. The `top(1)` utility showed that almost half of system memory was in the "wired" state, which seemed odd to me; so I investigated further, and found that `busdma` had reserved 32 MB of memory for bounce pages. This was clearly far more than needed — given Firecracker's limitations and the fact that bounce pages are generally not allocated contiguously, each disk I/O should use at most a single 4 kB bounce page — and I was able to reduce this memory consumption to 512 kB with a patch to `busdma` which limited its bounce page reservations for devices which supported only a small number of I/O segments.

Once the system was more stable, I started paying attention to the boot process. If you're watching a system boot and there's suddenly a pause in the messages scrolling past, there's probably something happening at that point which is slowing down the boot process. Simple eyeballing of the boot process — and also the shutdown process — revealed several improvements:

- FreeBSD's kernel random number generator usually obtains entropy from hardware devices, but in virtual machines this may not be an effective source. As a backup source of entropy, on x86 systems we make use of the `RDRAND` instruction to obtain random values from the CPU; but we were only obtaining a very small amount of entropy on each request and were only requesting entropy once every 100 ms. Changing the entropy gathering system to request enough entropy to fully seed the Fortuna random number generator shaved 2.3 seconds off the boot time.
- When FreeBSD first boots, it records a Host ID for the system. This is typically obtained from hardware via the `smbios.system.uuid` environment variable, which the boot loader sets based on information from BIOS or UEFI. Under Firecracker, however, there is no boot loader — and thus no ID being provided. We had a fallback system in place that would generate a random ID in software on systems that didn't have a valid hardware ID; but we also printed a warning and waited 2 seconds to allow the user to read it. I changed this code to print the warning and wait 2 seconds if the hardware provided an *invalid* ID, but proceed silently and quickly if the hardware simply didn't provide an ID.
- IPv6 mandates that systems wait for "Duplicate Address Detection" before using an IPv6 address. In `rc.d/netif`, after bringing up interfaces, we were waiting for IPv6 DAD if any of our network interfaces had IPv6 enabled. There's just one problem with that: We *always* have IPv6 enabled on the loopback interface! I changed the logic to only waiting for DAD if we had IPv6 enabled on an interface *other than the loopback* interface and sped up the boot process by 2 seconds — if another system is trying to use the same IPv6 address as us on our `lo0`, we have bigger problems than an address collision!

IPv6 mandates that systems wait for "Duplicate Address Detection" before using an IPv6 address.

- When rebooting, FreeBSD printed a message ("`Rebooting...`") and then waited 1 second "for printf's to complete and be read". This seemed minimally useful, since people can usually tell that the system is rebooting — there is now a `kern.reboot_wait_time sysctl` which defaults to zero.
- When shutting down or rebooting, the FreeBSD BSP (CPU #0) waits for the other CPUs to signal that they have stopped... and then waited an extra 1 second to make sure that they had a chance to stop. I removed the extra second of wait time.

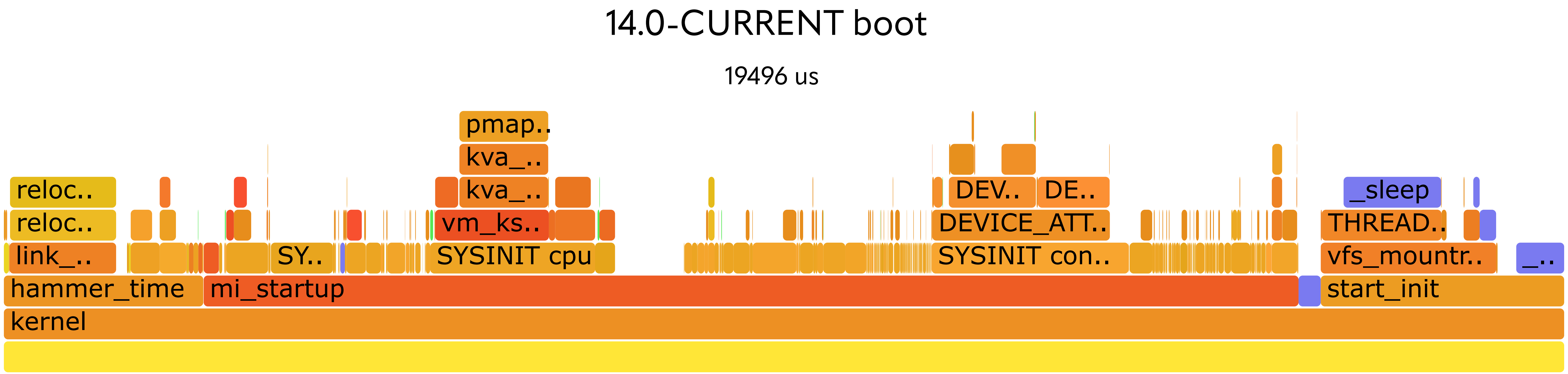
Once the low-hanging fruit was out of the way, I pulled out TSLOG and started looking at flamecharts of the boot process. Firecracker is a great environment for doing this, for two reasons: First, the minimalist environment eliminates a lot of noise, making it much easier to see what's left behind; and second, having Firecracker launch virtual machines extremely quickly made it possible to test changes to the FreeBSD kernel very rapidly — often well under 30 seconds to build a new kernel, launch it, and generate a new flamechart.

Investigation with TSLOG revealed a number of available optimizations:

- `lapic_init` had a 100000-iteration loop to calibrate how long `lapic_read_icr_lo` took to execute; cutting this down to 1000 iterations shaved off 10 ms.
- `ns8250_drain` called DELAY after each character was read; changing this to check for `LSR_RXRDY` and only DELAYing if nothing was already available to be read shaved off 27 ms.
- FreeBSD makes use of a CPUID leaf which most hypervisors use to advertise the TSC and local APIC clock frequencies; Firecracker, unlike VMWare, QEMU, and EC2, did not implement this. Adding support for this CPUID leaf to Firecracker shaved 20 ms off the FreeBSD boot time.
- FreeBSD was setting `kern.nswbuf` (which controls the number of buffers allocated for a variety of temporary purposes) to 256 regardless of the size of the system; changing this to `32 * mp_ncpus` shaved 5 ms off the boot time on a small (1 CPU) virtual machine.
- FreeBSD's `mi_startup` function, which kicks off machine-independent system initialization routines, was using a bubblesort to order the functions it called; while this was reasonable in the 90s given the small number of routines needing to be ordered at that point, there are now over 1000 such routines and the bubblesort was getting slow. Replacing it with a quicksort will save 2 ms. (Not yet committed at press time.)
- FreeBSD's `vm_mem` initialization routine was initializing `vm_page` structures for all available physical memory. Even on a relatively small VM with 128 MB of RAM, this meant initializing 32768 such structures — and took a few ms. Changing this code to initialize `vm_page` structures "lazily" as the memory is allocated for use will save 2 ms. (Not yet committed at press time.)
- Firecracker was allocating VM guest memory via an anonymous mmap, but Linux was not setting up the paging structures for the entire VM guest address space. As a result, the first time any page was read, a fault would occur taking roughly 20,000 CPU cycles to be resolved while Linux mapped in a page of memory. Adding the `MAP_POPULATE` flag to Firecracker's `mmap` call will save 2 ms. (Not yet committed at press time.)

Current Status

FreeBSD boots under Firecracker — and does so extremely quickly. Including uncommitted patches (to FreeBSD and also to Firecracker), on a virtual machine with 1 CPU and 128 MB of RAM, the FreeBSD kernel can boot in under 20 ms; a flame chart of the boot process appears below.



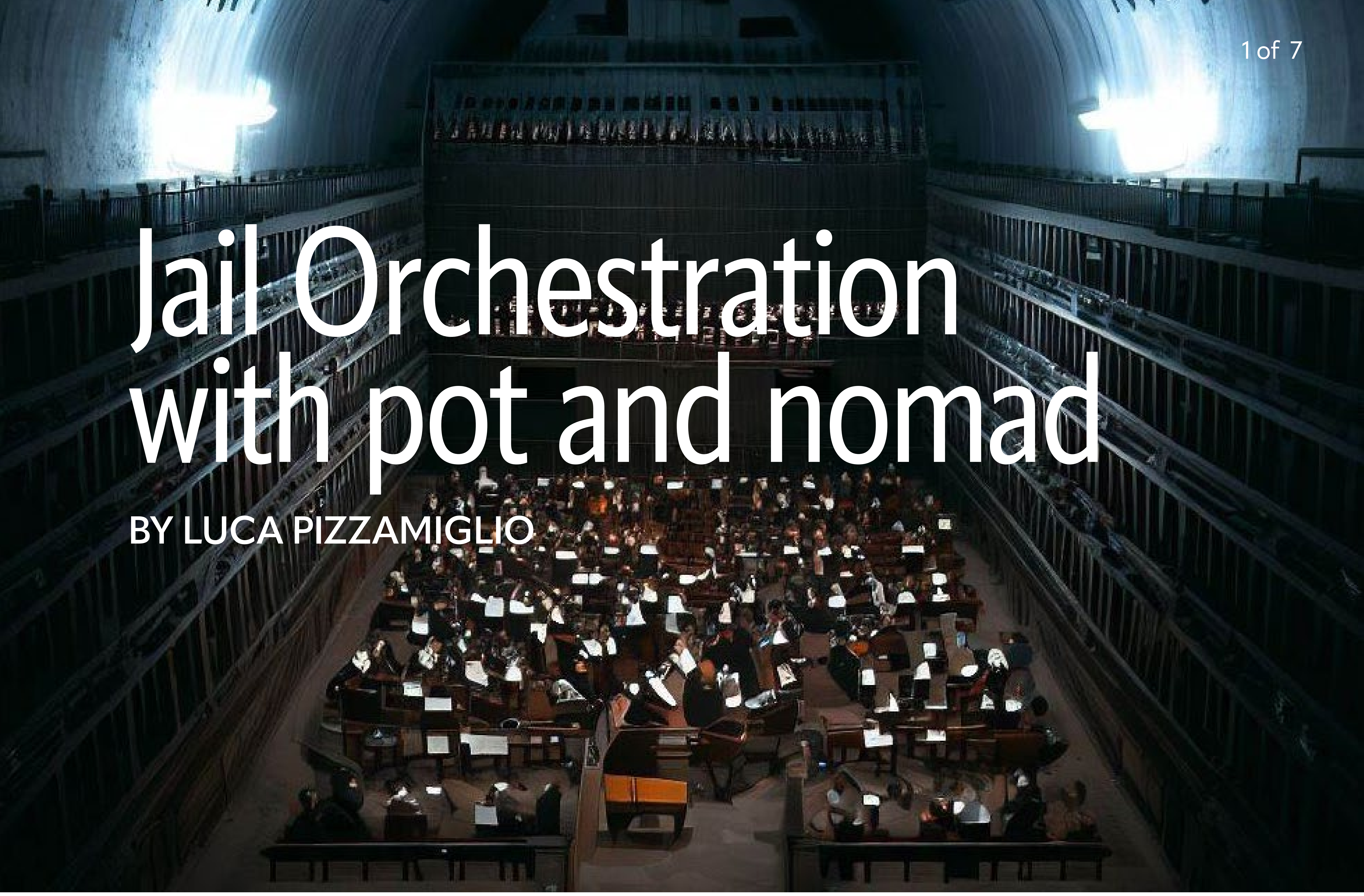
There is still work to be done: In addition to committing the patches mentioned above and getting PVH boot mode support merged to "mainline" Firecracker, there's a significant amount of "cleanup" work to be done. Due to the history of PVH boot mode originating from Xen, the code used for PVH booting is still mixed up with Xen support; separating those will simplify things significantly. Similarly, it's currently impossible to build a FreeBSD arm64 kernel without PCI or ACPI support; finding the bogus dependencies and removing them will allow for a smaller FreeBSD/Firecracker kernel (and also shave off a few more microseconds from the boot time – we spend 25 us checking to see if we need to reserve memory for Intel GPUs).

More aspirationally, it would be great to see if Firecracker could be ported to run on FreeBSD — at a certain point, a virtual machine is a virtual machine, and while Firecracker was written to use Linux KVM, there's no fundamental reason why it shouldn't be possible to make it use the kernel portion of FreeBSD's bhyve hypervisor instead.

Anyone wanting to experiment with FreeBSD in Firecracker can build a FreeBSD 14.0 kernel with the amd64 `FIRECRACKER` kernel configuration, and check out the feature/pvh branch from the Firecracker project; or if that branch no longer exists it means the code has been merged into the mainline Firecracker tree.

If you try out FreeBSD on Firecracker — especially if you end up using it in production — please let me know! I started this project mainly out of interest, but I'd love to hear if it ends up being useful.

COLIN PERCIVAL has been a FreeBSD developer since 2004 and was the project's Security Officer from 2005 to 2012. In 2006, he founded the Tarsnap online backup service, which he continues to run. In 2019, in recognition of his work bringing FreeBSD to EC2, he was named an Amazon Web Services Hero.



Jail Orchestration with pot and nomad

BY LUCA PIZZAMIGLIO

Containers are a great tool to distribute horizontally scalable applications on many servers. When the number of applications and their cardinality grows, the number of containers can easily become hard to manage manually.

Container orchestrators are applications that aim to simplify the management of a large fleet of containers, hiding the complexity and improving reliability, especially in an environment made very dynamic by autoscaling and continuous deployment. In this article, we will talk about a setup based on FreeBSD, using pot, a jail framework that supports jail images, and nomad, a container agnostic orchestrator developed by HashiCorp.

The System Architecture

To explain how an orchestrator works, we need to introduce a few services and explain their role.

The nomad client

A nomad client is a server that receives orders from the orchestrator to execute containers. Nomad clients can also be referred to as nodes, like in kubernetes.

In large installations, the majority of servers are nomad clients, as they are responsible to execute users' applications. In the cloud native jargon, nomad clients form the data plane of the cluster.

Nomad clients can support multiple container drivers: some drivers are operating system agnostic, while other drivers, like docker or pot, are available only on specific operating systems.

To orchestrate FreeBSD jails using pot, we need nomad clients based on FreeBSD.

The nomad server

The nomad servers are the machines that implement the orchestrator. The nomad server is responsible for keeping the state of the cluster and for scheduling containers to nomad clients. Multiple instances (3 to 5) are required to provide redundancy and to share the load. In the cloud native jargon, nomad servers form the control plane of the cluster.

Users are going to interact with nomad servers to deploy applications to the cluster. The nomad servers are responsible to keep the state of the cluster healthy, and to reschedule containers in case of client failures.

Nomad servers can run on any supported operating system. To orchestrate jails, at least one nomad client has to be based on FreeBSD.

The container registry

The orchestrator (nomad servers) is the "brain" of the cluster and it assigns containers to clients that support the container types. This means that:

- any nomad clients can be selected to run supported containers,
- clients don't know in advance which containers are going to host,
- a client can execute multiple instances of the same containers.

Every client needs a service to download the image of the container they are assigned to execute. This requirement is fulfilled by the container registry, a service that provides the images of containers.

When the orchestrator selects a client to execute a container, that client is going to download the container image from a registry and then start the container.

In our examples, we are going to use potluck, a public container registry maintained by the pot community, that builds images from an open source catalog of image recipes. However, as container images contain binaries, we strongly encourage everyone to have their own local registry, for security reasons.

In pot, images are files that are downloaded via `fetch(1)`, so a registry can be just a simple web server.

The service catalog

A service catalog is a list of services enriched by additional information like all the container addresses that implement those services.

When an orchestrator schedules a container implementing a service, it's also going to register the container address to the list of containers implementing that service.

The service catalog can be also configured to periodically check the health status of the service for all containers, so the list of container addresses only contains healthy addresses.

The service catalog that we are going to use is based on consul, a service mesh application also developed by HashiCorp.

Users are going to interact with nomad servers to deploy applications to the cluster.

The ingress (optional)

Because of the dynamic nature of the orchestrator, it can be very hard to know where our services are running. Every time a new deployment occurs, containers are scheduled to potentially different nodes on different ports. With ingress, we define a proxy/load balancer that is configured to provide a fixed entry point to our services.

For instance, we can configure the proxy in a way that the URL path (i.e., https://example.com/foo) provides information about the targeted service (i.e. redirecting to the containers implementing the service foo) Another common way is to use the host header.

Ingress proxies dynamically maintain the list of valid container's addresses, by continuously interacting with the service catalog.

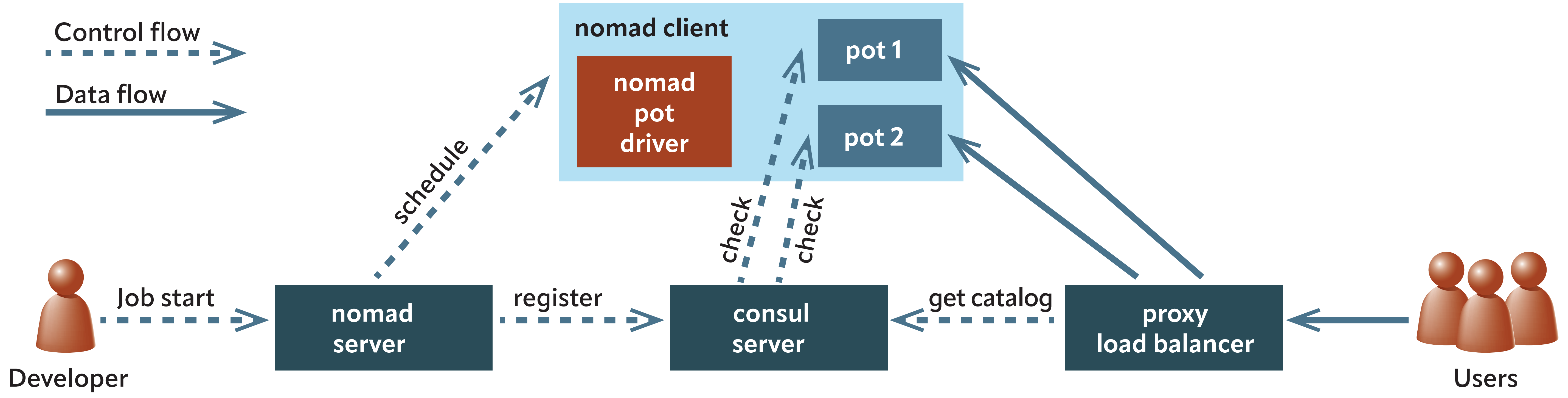
For our examples, we are using traefik, an ingress proxy implemented by traefix lab.

Nomad-pot-driver

Nomad has been built to support several container technologies and different operating systems. In fact, a nomad package is available and HashiCorp also provides binary blobs for FreeBSD.

Nomad has a plugin architecture that allows it to extend it to support new container technologies. Esteban Barrios wrote and opensourced the nomad-pot-driver plugin. This plugin works as an interface between a nomad client and pot, to provide the features needed to orchestrate jails.

The orchestrator schedules workloads to the client that uses the plugin to interface with pot.



Overview of the architecture and the foles of the different services.

Minipot

Minipot is a package that installs and configures all the aforementioned services on a single FreeBSD machine and is the reference installation that we are going to use to show our examples.

Minipot is a configuration useful for testing, but not for professional installation, as it's going to concentrate all service on one machine only, reducing a cluster to a single node that does it all.

In particular, it is going to install and configure consul, traefik, and nomad. Nomad is going to run as client and as server, playing the dual role of orchestrator and executor.

There is a detailed guide on how to install minipot on this blog post on the Klara website.

Schedule a Job

Once minipot is initialized and all services are running, we can use the following job description file to start a job in nomad

```
job "nginx-minipot" {
  datacenters = ["minipot"]
  type        = "service"
4    group "group1" {
5      count = 1
6      network {
7        port "http" {}
8      }

9      task "www1" {
10       driver = "pot"

11       service {
12         tags = ["nginx", "www"]
13         name = "hello-web"
14         port = "http"

15         check {
16           type     = "tcp"
17           name     = "tcp"
18           interval  = "5s"
19           timeout   = "2s"
20         }
21       }

22       config {
23         image = "https://potluck.honeyguide.net/nginx-nomad"
24         pot = "nginx-nomad-amd64-13_1"
25         tag = "1.1.13"
26         command = "nginx"
27         args = ["-g","’daemon off;’"]

28         port_map = {
29           http = "80"
30         }
31       }

32       resources {
33         cpu = 200
34         memory = 64
35       }
36     }
37   }
38 }
```

The job stanza describes all the details nomad needs to schedule the job.

The job “nginx-minipot” (1) has one group named “group1” (4), that has one task called “www1” (9).

The task “www1” is a pot container (10), the image registry is potluck (23), the pot image is nginx (24) and the version is 1.1.13 (25). By specifying that the task is based on the pot driver, the orchestrator is going to schedule this job on a client with pot support. In our example, server and client are the same machine.

Compared to the common use of jails, where the bootstrap happens using rc script, we are going to run nginx directly (26), without any other additional services. The args parameter (27) is important to allow nomad to properly follow the container lifecycle and to capture its logs. Pot is going to take care of initializing the network and everything that is needed.

The port_map stanza (28) and the network stanza (6) are saying that nginx is going to listen to port 80 in the jail, but the nomad client will use a different port ("http"), dynamically assigned by the nomad server.

The service stanza (11) provides the information that nomad is going to use to register the service to consul. In our example, the task "www1" implements the service "hello-web" (11) and it will be registered to consul, using the port "http" (14), assigned by the nomad server. For the IP address, the nomad server is going to use the nomad client IP address, defined during the scheduling operation.

In our example, the job is also configuring a tcp health check that consul is going to run every 5 seconds to determine the health status of the instance.

This job description needs to be stored in a file (i.e., nginx.job) and any user can launch the service via

```
$ nomad job nginx.job
```

NOTE: The first deployment can take some time as the client needs to download the image. On slow connections, the first deployment can even fail because of deployment timeouts. Once the fetch is completed, it is safe to re-run the deployment that will be executed in few seconds.

Check on nomad

Once the job is scheduled, it is possible to check the status of the deployment via command line:

```
$ nomad jobs allocs nginx-minipot
ID        Node ID   Task Group  Version  Desired  Status   Created     Modified
636d3241  c375b833  group1      3        run      running  28m43s ago  28m27s ago
$ nomad alloc status 636d3241
[...]
Allocation Addresses:
Label  Dynamic  Address
*http  yes      2003:f1:c709:de00:faac:65ff:fe86:9458:22854
[...]
$ curl "[2003:f1:c709:de00:faac:65ff:fe86:9458]:22854"
```

"Allocation" is the name used by nomad to identify a container, an instance of the task.

The IPv6 address is the nomad client one.

The port 22854 is the port nomad chose to direct the nomad client to the port 80 of the container.

To see the port redirection setup, we can use the command:

```
$ sudo pot show
```

An alternative of the CLI, the nomad server is configured to also provide a powerful web UI reachable at `localhost:4646`

From there, we can see the "nginx-minipot" job and navigate to see all information about the cluster, allocations, clients, and so on.

From nomad, we can see directly the status of all containers.

By reaching the allocation page, we can click on the "exec" button to run a /bin/sh shell into the running containers.

Check on consul

Via CLI, we can see the list of services in the consul catalog (consul catalog services), but not the details. However, we can check the status of the "hello-web" service, by reaching the web UI interface at

```
localhost:8500
```

From here, we can navigate to check the status of the service "hello-web" and its check "tcp".

Check on traefik

The proxy traefix is configured to route traffic on port 8080, but provides a web-ui to monitor the status at port 9200 (`localhost:9200`). Traefik is also configured to sync the service catalog from consul.

By selecting the http services, we can see our "hello-web" service (marked as `hello-web@consulcatalog`).

By clicking on the service, we can see the service details and the routing options.

The configuration is based on the host header, in our case "hello-web.minipot".

Now we can reach the service hello-web via the ingress:

```
$ curl -H Host:hello-web.minipot http://127.0.0.1:8080
```

Alternatively, we can add the entry

```
127.0.0.1 hello-web.minipot
```

to `/etc/hosts` and then use the hostname directly:

```
$ curl http://hello-web.minipot:8080
```

We will obtain the same output as we had by point curl directly to the jail.

Horizontal scaling

To see the orchestrator in action, we can now simply change the count from 1 to 2 in the job file (row 5) and re-submit the job via

```
$ nomad run nginx.job
```

Once scheduled, the running nomad allocations are now 2, the service "hello-web" in consul has two instances, as the servers in traefik.

- To verify the round-robin distribution of the ingress, we can
- Tail the logs of one container (`$ nomad alloc logs -f allocation1`)
- Tail the logs of the other container (`$ nomad alloc logs -f allocation2`)
- Execute curl on the ingress (`$ curl -H Host:hello-web.minipot http://127.0.0.1:8080`)

At every execution of curl, the proxy distributes the requests between the containers as is possible to see from their logs.

Tear down everything

To stop our examples, we suggest this tear down operation:

- Stop the nomad jobs (`$ nomad stop nginx-minipot`)
- Stop traefik (`$ sudo service traefik stop`)
- Stop nomad (`$ sudo service nomad stop`)
- Stop consul (`$ sudo service consul stop`)

From playground to production

Minipot is a single node installation useful as a playground, to learn or to test things locally.

A production environment should be shaped in a different way:

- 3 or 5 distinct consul servers
- 3 or 5 distinct nomad servers
- 2 ingress proxy servers (HA configuration)

Several nomad client servers (depends on the expected workload and reliability requirements like the overprovision factor)

It is worth mentioning that the aforementioned setup can mix different operating systems: the only servers that have to run FreeBSD are the nomad clients targeting jail/pot workloads.

Nomad or consul servers can run on Linux or Solaris, allowing you to reuse pieces of infrastructure that may already be available to you.

As ingress proxy, we used traefik, which natively syncs with consul. However, it's possible to use other services, like nginx or ha-proxy, together with consul-template, to achieve the same result. In such a configuration, consul-template is responsible for detecting changes in consul,to render the proxy configuration template, and to notify the proxy of the new configuration.

Additionally, the configuration of all services would need improvements, like adding authentication to nomad.

Minipot is a single node installation useful as a playground, to learn or to test things locally.

Credits

I want to highlight the community effort needed to reach this point, from Esteban Barrios, the first developer of the nomad-pot-driver, to Michael Gmelin (grembo@), who really helped a lot to increase the reliability and the stability of this solutions. I also want to mention Stephan Lichtenauer and Bretton Vine, who worked on Potluck, the public image registry, and on numerous other projects like the Ansible playbook and blog posts dedicated to pot and nomad use cases.

LUCA PIZZAMIGLIO is a port committer in the FreeBSD project and a member of the port manager team (portmgr). In 2017, he started the pot project, with the goal of understanding how containers on FreeBSD could look.

Conference Report

C is to BSD What Latin is to Us

A Theologian's Report of His Trip to Present at BSDCan 2023

BY COREY STEPHAN

As I sit before my battle-worn ThinkPad running OpenBSD 7.3-current on the Air Canada flight from Toronto, Ontario to Houston, Texas to return home to my wife, children, and (ir)regular job as Assistant Professor of Theology and Fellow of the Core at the University of St. Thomas, after my first experience at any conference about computer science or information technology, I feel tired but content. I departed the conference wearing my mustard OpenBSD 7.2-release t-shirt with its Dr. Seuss theme: "One diff, two OKs, commit, blowfish" (a parody of "One fish, two fish, red fish, blue fish"). Realizing that I am once again in the real world in which normal people, sadly, do not (yet?) run BSD operating systems on their laptop computers, I chuckle to myself while musing about the (distantly remote) possibility that my private security screening back at the Ottawa International Airport was not a mere coincidence. After all, this t-shirt represents an operating system whose founder, Theo de Raadt, famously relocated the project from the United States to Canada due to American laws about the export of cryptography. I jokingly wonder if this might be the first time someone has mistaken me for a hacker (of the malignant variety). If so, have I just endured another rite of passage into the BSD crowd?

As an obvious outsider to BSDCan who made a point of introducing myself to many fellow attendees, I received myriad questions. Oddly, the most common was the most challenging for me to answer: Corey, why are you here? BSDCan regulars were genuinely curious as to what a professional, Catholic theologian was doing at a conference about Unix-like operating systems. Even now, after the conference has ended, I am not sure why I journeyed to Ottawa for BSDCan 2023. Above all, I suppose that I wanted to try something wholly different from my day-to-day work. For years, I have been an amateur learning about the BSDs from Michael W. Lucas's *Absolute FreeBSD* (3rd edition) and *Absolute OpenBSD* (2nd edition), as well as the online recordings of talks from BSDCan, EuroBSDCon, and AsiaBSDCon (and the Youtube personality RoboNuggie). Although I occasionally engage in simple homelabbing, my main interest in FreeBSD, OpenBSD, and the other BSD operating systems long has been their common utility as fully customizable, desktop operating systems. Specifically, I like to use FreeBSD and OpenBSD to aid my multisource research and writing as a scholar of the history of Christian theology. My October 2021 FreeBSD Friday lecture "FreeBSD for the Writing Scholar" was about that very theme, as was my talk at BSDCan, "BSD for Researching, Writing, and Teaching in the Liberal Arts."

> Indeed, if creativity is found at the intersection of apparently disconnected subjects, then I am pleased to report that I had several engaging discussions at BSDCan that were themselves loci of creative genius.

At a standard academic conference, dressing the part of the refined scholar is important, especially in our anti-intellectual, twenty-first century Western context in which, say, one English literature job posting attracting hundreds of Ph.D.-holding, qualified (and desperate) applicants has become the norm. While packing to present at BSDCan, I almost instinctively grabbed my tailored navy-blue suit with my favorite golden bow tie. By Providence, I had the wherewithal to write a short email to Dan Langille, one of the founders of BSDCan who coordinated the conference for twenty years before announcing his well-earned retirement from that duty during this year's closing session, to check how attendees and presenters typically dress. His reply was that he would be wearing cargo shorts and a t-shirt through the whole conference, and I recalled that I had watched several BSD(Can/Con) video recordings in which the speakers were dressed in kind (Theo de Raadt in shorts and sandals, Michael Lucas in t-shirts with cartoonish figures of horror, and so on). Recognizing that I would be laughably out of place if I were to continue with my first sartorial selection but being unable to bring myself to deliver a formal presentation without a shirt and tie, I compromised by packing one set of my standard university (teaching and meeting) attire (in this case, purple pants and a purple bow tie with grey suade shoes — no jacket). Even with that change, Michael Lucas told me after my talk that I was "the best dressed … speaker that we ever have had." I also made sure to have my full collection of BSD shirts with me to wear for the rest of the conference, that is, my FreeBSD polo shirt that was a gift from the FreeBSD Foundation when I presented for FreeBSD Friday, the Seussical OpenBSD t-shirt that I was wearing as I began drafting this report, and a few old-stock-but-new OpenBSD t-shirts that were given to me by a generous Unix greybeard who wishes to remain anonymous.

After presenting at many academic conferences on the topics of theology and/or early to medieval Church history over the years, I had grown accustomed to a certain set of cultural and behavioral expectations about conferences that were in no way applicable to BSDCan. Starting with the rather silly point of dress, BSDCan presented me with a seemingly

unending series of opportunities to think freshly. Indeed, if creativity is found at the intersection of apparently disconnected subjects, then I am pleased to report that I had several engaging discussions at BSDCan that were themselves loci of creative genius. In chronological order, here are three such chats: Michael Lucas of technical authorial fame talked with me about a few possibilities for an educational book project that I have had in mind for some time; Tom Jones, whose voice I recognized from the BSD.Now podcast that I often enjoy during my long commutes in Houston, suggested that I ought to write this report for the *FreeBSD Journal* (for which he sits on the editorial board); and Dr. Marshall Kirk McKusick, original BSD Unix contributor and (great-)grandfather of FreeBSD who still commits code, explained that the famous Beastie character is a Unix background daemon as imagined by a Disney artist.

To McKusick's story about Beastie, the Unix background daemon, I replied that the overall rationale behind the artistry makes sense. In ancient Greek, I noted, a δαίμων (daemon) refers to a roaming spirit who either helps or hinders humans invisibly (in the background, like a Unix daemon). Since a daemon might either be benevolent or malevolent, but the Devil of the Christian tradition (obviously) is only malevolent, a daemon is not the same as the Devil. I remain unconvinced that Beastie's yellow pitchfork and red horns, inevitably offensive to some, are necessary, since ancient daemons rarely were depicted in artwork — and certainly never as such. That matter, however, seems best left alone until the next time that I chat with the ever-smiling McKusick. Besides, I cannot blame a gifted Disney animator for lacking familiarity with Greco-Roman folklore, and I rather enjoy the medieval character that graces (or curses) FreeBSD communication zones to this day.

> In ancient Greek, I noted, a δαίμων (daemon) refers to a roaming spirit who either helps or hinders humans invisibly (in the background, like a Unix daemon).

Speaking of McKusick, on my first night at BSDCan (Wednesday), I stumbled into the end-of-day hackathon portion of the FreeBSD Developer's Conference. I was impressed to observe McKusick collaboratively coding with someone who must have been at least forty years younger than he. The spirit of working to keep the BSD operating system projects multigenerational ran through the entirety of BSDCan. The oldest, most accomplished participants, who might have had legitimate reasons to ignore neophytes, especially outsiders (as I was), treated me as a worthy discussion partner. During that evening session, a few of us who are husbands and fathers talked about our wives and children, with the discussion flowing naturally between the serious and the amusing before returning to BSD.

I am not ashamed to admit that I did not understand everything in the lectures I attended. Understanding everything never was my objective, nor should it be the objective of anyone attending any conference, be it academic, technical, spiritual, or other. I went to BSD-

Can in search of a novel exchange of ideas, sharing what I know with persons from outside my normal circles and, more importantly, learning from what such persons know. Although I often was reduced to attempting to intuit requisite background knowledge in real-time, I enjoyed nearly all the talks I attended. I certainly learned something in every session. Bravo, BSDCan's organizational team, for electing a superb cluster of speakers (if I may be allowed to write that as someone who was one of those speakers).

Tom Jones's "Making FreeBSD QUIC" and Marshall Kirk McKusick's "Gunion(8): a new GEOM utility in the FreeBSD Kernel" made Friday, the first day, a joy, since both Jones and McKusick know how to work an audience with what seems to be magical charm. (How, I wonder, does Dr. McKusick make a history of file system minutia almost as intriguing as one of Ken Burns's documentaries about a great American war?) Additionally, recording the 512th episode (not 500th, but 512th, since 512 is the important number in computing) of BSD.Now before a packed auditorium, was a charming idea that the podcasting crew ought to repeat in the future. Shockingly, (the same) Tom Jones proclaimed to the entire conference that he was most looking forward to my talk about BSD and the liberal arts. I have no doubt that his proclamation boosted my session's turnout.

Bravo, BSDCan's organizational team, for electing a superb cluster of speakers.

On Saturday, the second day, Philipp Buehler's "Jitsi on OpenBSD - Puffy presents video conferencing" included a daring live demonstration of his working OpenBSD-hosted Jitsi server in which several of my fellow audience members concurrently connected. For that alone, even before he delivered the rest of the stellar talk, Buehler earned my firm applause. The CEOs of leading-edge technological firms as well known as Elon Musk and the late Steve Jobs have embarrassed themselves by attempting to give live demonstrations before large audiences only to have the technologies that they have intended to showcase fail in real-time. Buehler, however, did not strike me as a fool; rather, he was confident enough in his implementation to demonstrate it to the world. Brooks Davis's "Creating a memory-safe workstation with CheriBSD" was about the application of leading-edge technical research from Cambridge University, so understanding it would have required a huge amount of background knowledge that I lacked. Worse, Davis's talk immediately preceded my own. I observed that Davis is a fine presenter with a jolly demeanor, but I must humbly admit that almost all his talk entered one of my ears and escaped the other while I anxiously awaited my slot.

Finally, my time came. Tom Jones had primed the rest of the conference attendees for me the day before, so the decently sized, University of Ottawa classroom in which I had been assigned to present was bustling. All morning, I had observed the weary faces that are typical of the second day of every conference that I have attended, perhaps made worse at BSDCan because the conference organizers encouraged folks to visit a pub for local brews and then stay awake until (if not past) midnight collaboratively hacking. While I am neither a drinker nor a hacker, I appreciate that the bleary eyes that filled the conference halls and rooms on the second day were those of men and women who passed the night in lighthearted comradery while building a better technological future for us all. Yet, perhaps because I was standing at the front of a university classroom, wearing my normal teaching attire, I immediately snapped into my workplace modus operandi as a whimsical assistant professor of the traditional liberal arts. The room lacked the vitality that I needed to succeed as an oddball. Accordingly, I called upon everyone in the room to rise, greet her or his neighbor, and say, “It is a blessing to be sitting next to you today.”

My use of a classic professorial trick seems to have succeeded. The audience members’ reactions to my talk were loud, energetic, and wonderful from start to finish. One of the crowd’s favorite moments in my talk was when I pointed to the recording camera in the back of the room and addressed my colleagues with this original line: “C is to BSD what Latin is to us.”

The Question-and-Answer session for my talk went over time, with Dr. McKusick himself asking several questions. The hallway discussions afterward were engaging. Overall, my talk’s warm reception elated me.

I hope to return to interact with everyone I met at BSDCan 2023 again at BSDCan 2024 — and to present something even more daring.

Without an obvious way to conclude this report, I will write two things. First, if you are interested in the BSD operating systems but nervous about BSDCan, do not be. While the conference’s regulars form a self-selected, tightly knit group, they are genuinely welcoming, and it does not take long for newcomers to start to assimilate into that group. Further, there is no shortage of entertainment at BSDCan (the charity auction is hilarious), and the talks are scheduled so that no one part of the conference becomes painful. Second, I hope to return to interact with everyone I met at BSDCan 2023 again at BSDCan 2024 — and to present something even more daring.

DR. COREY STEPHAN serves as Assistant Professor of Theology and Fellow of the Core at the University of St. Thomas in Houston, Texas. He proudly makes exclusive use of free and open source software tools, including *BSD operating systems, to assist both his research in Catholic historical theology and his teaching of the traditional liberal arts. His professional website is coreystephan.com.

Photos courtesy of Tom Jones.

Recollections: An Interview with Doug Rabson (dfr@)

BY TOM JONES

The FreeBSD project started out with contributions from many hands, but the early days of the project and the people behind our favorite Operating System haven't been covered in much detail. As a part of FreeBSD's 30th Anniversary, I set out to speak to those involved at the start of development.

This installment is with Doug Rabson, who has been a FreeBSD committer since 1994 and is currently working on improving FreeBSD support for modern container orchestration systems such as podman and kubernetes.

TJ: Could you explain—generally—what you were up to in the late 1980s and early 1990s before the project?

DR: I graduated from college in the early 1980s. Before that, I had been exposed to BSD and we used it on some of the machines there, but I didn't pursue that kind of operating systems focus at all. I went to work for a little games company that some friends had started, and we told them that Unix is cool, that they should absolutely get Unix if they were going to buy a computer. They bought a MicroVAX, put Ultrix on it, which was 4.2 BSD—approximately—and we wrote Magnetic Scrolls. We were writing interactive fiction using Unix systems because the micros we were targeting were too weak.

That background interest stayed with me. I remember when the 4.3 BSD tapes were released, I got a copy from a friend at Imperial College, University of London. Just for curiosity, I wanted to understand how it worked. The idea of being able to read the source code was very cool. I read through it, figured out some of the pieces that were missing—and figured out that I didn't know enough to try to fill in the gaps.

> I remember when the 4.3 BSD tapes were released, I got a copy from a friend at Imperial College, University of London.

A bit later, and this is again still just before FreeBSD, I heard about the 386BSD precursor to the Open/Net/FreeBSD group. Somebody did fill in the gaps—somebody who actually knew what they were doing. And I installed that on some scratch hardware I found at work. And it worked. It didn't work very well. I mean, it was sort of broken. And a bunch of people like Jordan—and I've forgotten all the names—Nathan Whitehorn, I think, David Greenman. Anyway, they got together with a set of patches for 386BSD, because the author wasn't really interested in doing much more with it. At that point, he'd written his article for Dr. Dobbs and had done the work. But he wasn't taking patches. There was a kind of organic movement to collate the patches to various bugs and features that had been added to 386BSD. And that was the 386BSD patch kit, which went through a number of versions. When it was clear that the author, Mr. Jolitz, wasn't really interested in taking his project further, that turned into the BSDs.

That was the point where Net and Free split. FreeBSD people wanted to target one viable platform, which is the 386, PC-based commodity hardware. The NetBSD people were interested in holding on to the portability that was always there in the BSD platform. And so, they diverged at that point. I don't think there were any hard feelings. It was just a difference in focus.

So that was what I had been doing up to the beginning of the project. I wasn't a contributor to any of the pieces, but rather an enthusiastic user of it.

TJ: How did you follow the discourse around the BSD developments?

DR: About that time, I was working for a little company we put together, and we arranged for a link to Usenet, which was a bulletin board system that predates a lot of classic internet stuff. I read quite a bit about things on there. You could follow along with the development in the Usenet groups that were related to BSD and similar things. I managed to sort out a method of getting email, which again, wasn't that easy at the time.

I found the mailing lists, and that was where I became properly informed about the project. Initially, I used that with a bit of wrangling to try to get things to work before ISPs existed in this country. I managed to get onto the mailing lists and then eventually an ISP did exist in this country, and I had dial-up internet, and it was off to the races after that.

TJ: How did you get the software before there were ISPs?

DR: So—dragging up old memories—I think some of it might have been posted to Usenet. I believe my employer did have some access. We certainly weren't on the Internet, but we had a connection to Usenet.

There was also a UK-based bulletin board called CompuServe where you could dial up and download stuff. That might have been part of what was going on, and that seems more plausible because Usenet was a bit of a Wild West situation, whereas with CompuServe, you could definitely download things. Yeah, I'm not absolutely certain how I first got hold of a copy of 386 BSD. I remember it being on about 10 or 15 floppies. I'm pretty sure that Usenet played a part. Once FreeBSD existed, there were some very useful mailing lists--some still exist today. There were, however, many fewer mailing lists than there are today. They were extremely useful in just keeping up and figuring out what people were doing, where the project was going, that kind of thing. That was in the FreeBSD 1 days.

TJ: What was the process of going from an enthusiastic user to a contributor?

DR: I was using FreeBSD. We had a little start-up that did 3D graphics technology. I put together a server for us to use for file sharing and email and things like that. By that time, we had dial-up. That was running FreeBSD 1.0, possibly 1.1.

In that company of about five or six people, we had one CD-ROM drive. I said, hey, we have a network. I can put that on the server, and we can share it via NFS. It was a bit of a can of worms that I opened there because it didn't work very well. One of the things that didn't work very well was that FreeBSD wasn't able to share the CD-ROM via NFS. I did some research and found some patches that somebody else had written. I had no idea who wrote them, to be honest. I applied the patches to our server. Yay, it worked.

I think the patches were originally for FreeBSD 1.0. I remember having to port them to FreeBSD 1.1 because there were some differences between the two releases. I think I sent my changed patch set to one of the main lists saying, hey, I ported this guy's work. It works on the current release. This is it. I think around that time, FreeBSD 2 was nearly happening.

The BSDI lawsuits were being resolved. One of the things we agreed to was to mothball the whole source tree from FreeBSD 1.0 and take a clean, legally-agreed copy of the 4.4 BSD Lite 2, I think, sources which everyone agreed definitely didn't include AT&T intellectual property. Then we forklifted the parts of FreeBSD that were clearly unencumbered and put together FreeBSD 2 from a clean base.

When I posted these patches to FreeBSD 1, the FreeBSD 2 thing was getting ready to happen. I think also at that point, my business partners were in California on a sales trip, and they happened to meet Jordan. I don't think that was particularly unusual because he knew the name of the company I was working for. It was in my email signature.

A friend of his was talking to my colleagues about 3D graphics. Jordan joined the meeting and the upshot was that he phoned me up and said, do you want to be a committer? At that point, I ported the things that I had been playing around with on FreeBSD 1 to FreeBSD 2 and got involved with that whole project of making FreeBSD 2 at least as good as 1.1x was—so moving a bunch of stuff from FreeBSD 1 to 2. That was when I started to be actively involved. It was FreeBSD 2.0 and beyond. That would have been 1994, I guess.

I ported the things that I had been playing around with on FreeBSD 1 to FreeBSD 2 and got involved with that whole project of making FreeBSD 2 at least as good as 1.1x was.

TJ: Then you went from being a committer to joining the first core team. How did that come about?

DR: From 1995 to 1997, I was working for Microsoft, and I didn't do much original work in FreeBSD at the time. Looking at my commit record, I was still somewhat actively doing things with NFS. I was fixing bugs and things, but not trying to do anything really interesting because I didn't want my employer to have rights to cool stuff for FreeBSD. Anyway, I didn't do much during that period of time. In 1997, I left Microsoft, and took some time away from paid work to properly connect with the project. I had some ideas going on in my head based on the way the Microsoft operating systems work. Things like loadable kernel modules, which were poorly supported in FreeBSD at the time, were hugely useful, still are.

I thought that model was much better than the giant kernel that contains everything, the model we were mostly using. I worked on the kernel linker. That took me to mid-1997, I guess. The idea came up that, hey, we've been working on this single platform, 386. We've done pretty well with it. We've got a stable operating system that people are able to use.

Should we consider a second platform? I don't know whose idea that was, but at one point, somebody from Digital offered us some loan hardware for DEC alpha. Jordan included me in that discussion and said, hey, do you want an alpha-based computer? Yeah, sure. DEC donated a bunch of hardware. We had this idea that we would port FreeBSD to this new platform. This is an interesting platform because, at least at the time, it looked like it could be viable as a commercial platform. The chips weren't crazily expensive.

The rest of the hardware in the machines was more or less PC-ish. We felt that this could be viable. It was a 64-bit platform, which was a necessary step for FreeBSD to take. We were already starting to approach the limitations of 32-bit platforms for some of our users. Alpha was there, and I got involved with that. Eventually, I ported the kernel with some help from NetBSD sources. That happened in 1998, mostly. As part of that work, I renovated the whole device driver architecture because alpha was different and needed an abstraction layer. I put that in and did a lot of work on it. In 1999, I went to the Usenix ATC to talk about my work with peers I'd never actually met. Everyone was just emailed at that point. Jordan grabbed me halfway through the conference and said, hey, do you want to join the core team? The core team had already existed for pretty much the whole time since FreeBSD's first release. I only joined it towards the end of the first core team. The first core team was before we did elections—that person looks like he's doing something interesting—let's grab it! That was how it was.

TJ: I don't know if elections are better.

DR: I think they are. I was kind of skeptical at the time, but there are lots of things I like about them. The term limits give you a clean point in time when you can say, hey, I don't want to deal with this level of involvement right now. I'm going to step back. I did that a bit later on. I got invited to the first core team, which was a fairly organic, self-organizing entity. From reading my old emails recently, I'm reminded that core team was very heavily technically focused. There was an architecture element to it that's intentionally not part of the remit for core these days. It was a bit different.

I think the first core grew from the patch kit folks. The people that were involved with the patch kit that ended up wanting to do the FreeBSD single platform focused on building something for people to use. Those people, by and large, were the first cohort in core zero. And then that group of people invited others. So, of course, that's how I entered the team, but I was toward the end of core zero. I don't recall exactly. I wasn't paying much attention to who was doing what.

TJ: How did the project change during your time on core zero and core one?

DR: I think the biggest thing that we changed was the election, and that was pushed by some members of core zero who wanted to clarify how the project was going to be governed—and have some bylaws. That was a huge change. I think it was a good change for bringing more of the project users into the project committers, at least, into the decision-making process. That was a cultural change, which I think was needed at the time. That was the end of core zero.

We got that process sorted out. I remember some meetings at Usenix and afterwards to nail down the details, get them agreed to by the membership, and then arranging the first election. I ran in the first election partly because I still wanted to be involved in the project at that level. I wanted the transition to an elected model to be successful, so a lot of us ran just so there were people who people already knew who were part of the election. The whole project, the whole thing would have failed if core zero said, yeah, here are the new rules--we're going off to the pub now. Yeah, so committing to the new model. I was part of that first election and was elected because I did have a reasonably high profile at the time. I was doing a lot of work on the kernel. I was making some significant changes. It felt natural for me to run because I was heavily involved.

I wanted the election system to succeed. It wasn't my idea, but I liked it once it was fleshed out. So, what changed during core one? That was 2000? Was that 2000 or 2001? I think that at that point, I started to get a bit burned out by the whole core thing. It was turning into a system for governing rather than a technical oversight. I found that more difficult to cope with than just figuring out what's broken and how to fix it.

We had some difficult decisions to make. This was during core zero, around Matt Dillon, and probably a few other things. I was just starting to get a little bit burned out, a bit crispy at least, on being part of the governance of the project rather than just being a contributor. That's more of a personal thing that changed. I'm struggling to think of anything tangible in the project that changed.

Yes, IA64 was interesting. We had a 64-bit platform, but it looked like Digital was going to drop the ball there.

TJ: We covered alpha a little bit, but what about the IA64 port?

DR: Yes, IA64 was interesting. We had a 64-bit platform, but it looked like Digital was going to drop the ball there. It was still being produced. That was probably after Compaq bought it out, and I thought the writing was on the wall for alpha, but people still needed a 64-bit platform. Yahoo, in particular, had some workloads that were running up against the limitations of the 32-bit address space, and they really wanted a 64-bit platform. I was at Usenix ATC in 2000. Paul Saab turned up and gave me a good six inches worth of technical documentation on IA64 and said, hey, you know how to port the kernel! Have a look at this. It wasn't clear that IA64 was the right direction to take, but it would take us closer to a somewhat x86-compatible, 64-bit platform. It was architecturally interesting. It did things in a different way, and I was curious about how that would work.

When I did alpha, a lot of it was helped by taking inspiration and code from NetBSD. They'd done the port a bit ahead of us. I wanted to do that process again but write it all myself just to prove to myself that I could do it. It wasn't just a question of getting some Lego pieces and putting them together and saying, hey, I did it. I wanted to build the pieces as well. I did that for IA64 and I had some great tooling to make it easier. I used simulations extensively in both ports to help get the thing up and running. Yahoo arranged for me to get some test hardware and I still have it. It's underneath my desk. It hasn't been switched on in 20 years. I got the test hardware and brought up system. I wasn't very impressed by the hardware.

Compared to the PC platforms I was using at the time, I thought this was going to be far too expensive. I couldn't see it running at scale. The goal in those ports in those days was can it build itself. Can it self-host? Can it build its own source code? I got it to that state. Along the way, I wanted to use some 386-only tools, those from Perforce that we were using for some private source code control. I didn't have an IA64 binary for that, so, I wrote the beginnings of what's now the previous 32-bit ABI. At that time, it was a 386 ABI hosted in my IA64 kernel. That code still gets used these days as the 32-bit compatibility layer.

I wrote enough of that to get Perforce to work, but I wasn't convinced that it would be a successful platform. It was a niche platform in the end, and it had its successes in that niche role. But I couldn't see anyone like Google or Yahoo or any of the other big internet players using it at scale, not with the hardware I'd seen. HP picked up Compaq and ended up being the main booster of IA64, because, I think, some of their IP went into Itanium from their PA risk architecture. HP was a big booster of the platform.

Another project member was working at HP, was interested in IA64, and I more or less let him take over development after about—I'm going to say 2001 or so—maybe 2002. I remember doing the 32-bit subsystems for IA64 in 2002. So, yeah, I kind of took it up to that point of being viable, self-hosting, then somebody else took it on.

I think they were both important ports for different reasons. The existence of IA64 made it easier for Peter Wemm to do the AMD64 port.

> The legacy is having a truly free, self-supported, functioning operating system that doesn't involve license politics.

TJ: What is the lasting legacy of FreeBSD?

DR: The legacy is having a truly free, self-supported, functioning operating system that doesn't involve license politics, and when literally you can put in an embedded device, sell it, and nobody's going to start complaining on the internet that you haven't ticked the right boxes or released the source code of your thing. It's super easy for anyone to pick up the previous project. We've made it really clear where the boundaries are between simple copyright and complicated copyright parts of the system. So, I think that, yeah, that's a viable resource for embedded development for literally anything. You can find FreeBSD inside all kinds of weird stuff. It's the basis for a whole line of router hardware from people like Juniper. It's the control plane in a lot of storage appliances. FreeBSD used to be part of random internet firewall devices. It still is in all sorts of things that you just treat as an appliance.

They just work. And the reason they work is because FreeBSD is super easy to use, both legally and technically. And I think that's an important part of its legacy. There are almost certainly other things. I think that the quality of the code in FreeBSD makes it a positive resource for the rest of the operating system community. We do things in FreeBSD so that the ideas in FreeBSD can cross-pollinate other platforms and vice versa.

If FreeBSD was terrible, we wouldn't really be a part of that group of self-improving projects. This is the danger of monoculture. And FreeBSD is doing its part to avoid monoculture. And part of that is healthy cross-pollination. I get ideas from Linux. Hopefully Linux occasionally gets ideas from us. I know that they've taken some of our driver stuff in the past. So yeah, being a good partner in an ecosystem of similar projects is part of it.

TJ: Is there anything you would like to add?

DR: This last year, I have been re-connecting with the project after a fairly long period of low involvement. The main difference I see now is that we take a lot more care to avoid breaking things. Today, we have a decent unit test suite, continuous integration systems, and a growing culture of code review—compared to the early days when I would test my own changes on an ad-hoc basis, sometimes send them to people by email to look at, but not always. This is a good change overall since it reduces risk and tends to result in a stable platform, but it is a different (and slower) way of working and I think it can be hard to find the balance between minimizing risk and innovation.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.

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

Jail-based DNS AdBlocking Tutorial

BY BENEDICT REUSCHLING

One of the things that drew me to FreeBSD early on is its ability to easily run services. Services can be system services that come with the operating system (the simplest example would be the ssh daemon), or through third-party software installed via pkg or ports. The process is the same for both: you add a line to /etc/rc.conf to enable it (either via sysrc or "service ... enable") to run the next time the system boots. Then there is usually a configuration file to make settings that customize the service to your needs. Typically, this requires entering which IP address or DNS hostname to listen to, a network port, and some specifics of the software. From then on, it's either starting the service directly (using "service ... start"), or at the next reboot, in case it requires loading kernel modules that kldload can't load for the running system (which is rarely the case these days).

This is straightforward, keeps the configuration of all system services in one convenient location, and is easy to repeat once you've done it for one or two services. Of course, running a whole set of services on the FreeBSD host system works perfectly fine—until it gets more complicated. Running different versions of the same software side by side is perfectly reasonable and not too uncommon. This is done for testing purposes—checking to see if an upgrade works as intended or if certain software still requires an older version as a dependency. One example is trying to run different versions of the PostgreSQL database next to each other. In this case, both pkg and ports check for the versions or will cancel the install operation with a message saying that certain binaries will be put into the same place and hence overwrite each other. This is an undesirable situation, and the user has to make a decision for one or the other version because they can't coexist next to each other.

Of course, running a whole set of services on the FreeBSD host system works perfectly fine—until it gets more complicated.

That is, unless virtualization or containerization is involved. This allows process isolation in separate execution environments using various methods to permit multiple such systems to run on the same hardware. Virtualization adds an extra layer over the operating system that permits the installation of the same or a different operating system with simulated hardware. Containers or jails do this by isolating processes using chroot(8). We'll focus on the latter here, as it is more lightweight in terms of resource use and getting something running fairly quickly.

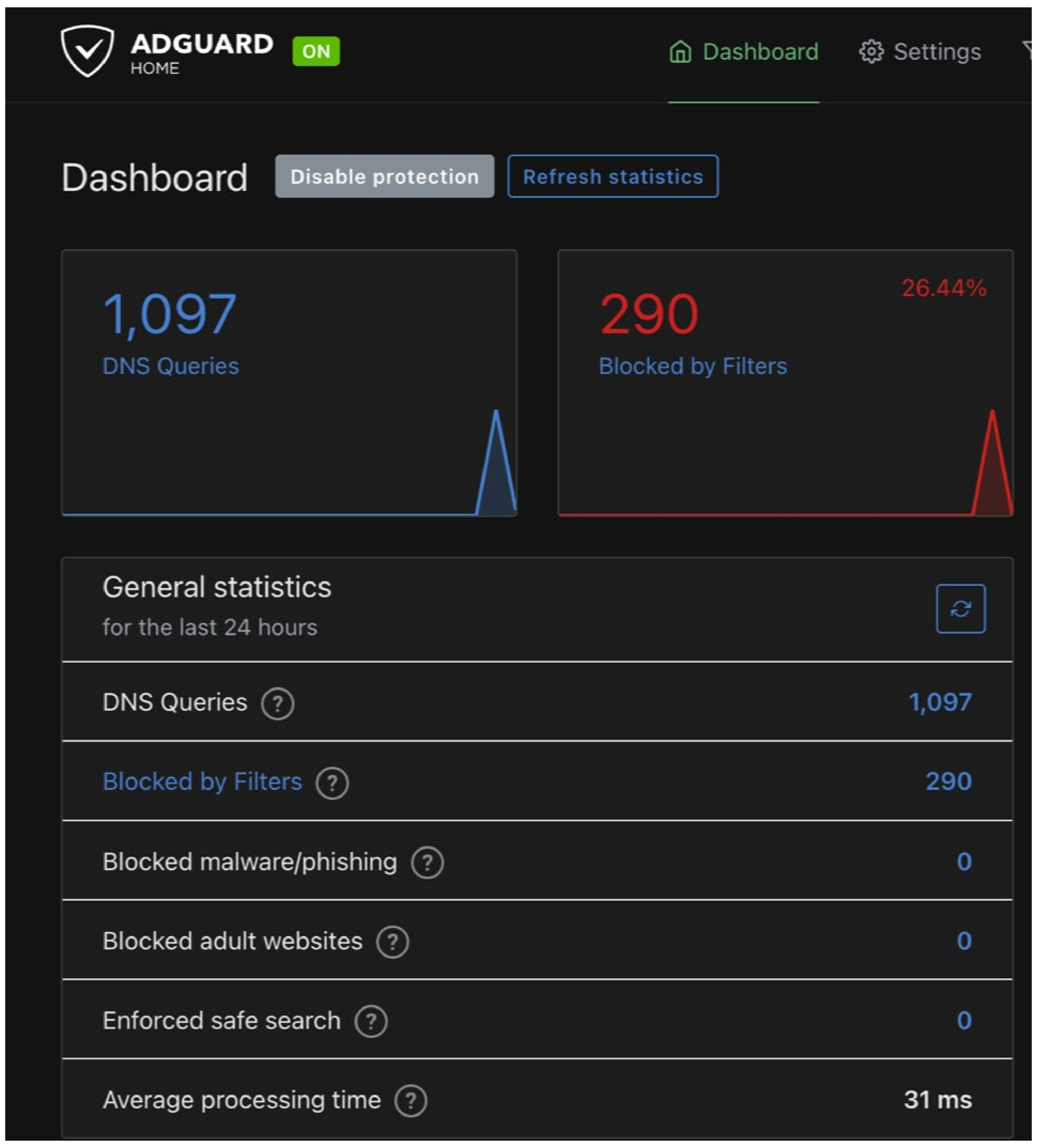
The benefit of this isolation is not only that various, different versions can run side by side, it also allows separation for security reasons. When an application is running in a jail container, the processes inside can't access the host system by default. The application discovers all the usual devices (like networking), directory structure, and files in the right places, but in reality, it is a separate environment that mimics the host system behavior and layout. When such a jail becomes compromised somehow, it is easy to stop it without affecting services in other jails or the host system. Access to them is strictly prohibited and isolates any intruders into that particular jail cell.

That also makes it easy to migrate a system to another host by stopping, copying the jail's directory structure to the new location, and starting there again (with a few local modifications like the new IP address, for example). Backup and restore is done in the same way. Multiple such jails are typically managed by jail management software that takes care of creating, modifying, and removing jails.

When an application is running in a jail container, the processes inside can't access the host system by default.

One such jail manager is called Bastille which is written entirely as a shell script. We're taking a closer look at Bastille in this article by going through the process of setting up the host system for it, creating a jail, and starting a service within it based on a template. Such templates allow for sharing configurations in a central repository so as to apply them without the need to know about the inner workings of the service. That way, complicated situations are easy to set up for people who want to get something running quickly.

In this article, we're deploying a service called AdGuard by AdGuard Software Limited. With a running AdGuard service in a network, clients connecting their DNS resolution to it can filter out advertisements from web browsing activities. This helps avoid tracking and user profile building by advertisers, and also helps pages load more quickly since they don't have to transfer ads next to the content the user wants to see. AdGuard does this using filter lists and DNS sinkholing. Based on the filter lists, a known advertisement site gets blocked by AdGuard sending an invalid address response back before it renders the ad in the browser. There are various ways to use the AdGuard service—as a browser extension for an individual device, desktop applications, or by running it as a recursive DNS resolver. Note that AdGuard does not completely protect against all forms of advertisements (especially the ones dynamically embedded in video sites) but does a good job of removing a big piece of them from web pages.



Dashboard

1,097
DNS Queries

290
Blocked by Filters
26.44%

General statistics
for the last 24 hours

DNS Queries	1,097
Blocked by Filters	290
Blocked malware/phishing	0
Blocked adult websites	0
Enforced safe search	0
Average processing time	31 ms

We're starting out with a Raspberry Pi, because this service runs pretty much all the time, and we want a low power usage footprint. I have an RPI3 here, but other devices (including full fledges servers) capable of running FreeBSD work just the same. Install the operating system, apply the latest security patches and lock down remote access using SSH keys.

Environment Setup

I have an old 32GB SSD connected to the Raspberry Pi, which will do most of the I/O heavy operations with a single disk ZFS pool instead of the slower compact flash card. I am running FreeBSD 13.2 at the time of writing this article and I'm fairly confident that future versions will work just as well or with only minor adjustments.

```
# pkg install bastille git-lite
```

Bastille, being a shell script, is fairly quick to install and has no extra dependencies. It may not be as full-blown in some of the functionality that other jail managers have, but it nevertheless works. Git is necessary to clone the adguard home template (and others) from Bastille's GitLab repository.

Next, we create a PF configuration for bastille in /etc/pf.conf like the following:

```
ext_if="ue0" ## <- change ue0 to match host interface

set block-policy return
scrub in on $ext_if all fragment reassemble
set skip on lo

table <jails> persist
nat on $ext_if from <jails> to any -> ($ext_if:0)
rdr-anchor "rdr/*"

block in all
pass out quick keep state
pass in inet proto tcp from any to any port ssh flags S/SA keep state
pass in inet proto tcp from any to any port bootps flags S/SA keep state
pass in inet proto tcp from any to any port {9100,9124} flags S/SA modulate state
```

Make sure to change the ext_if line at the top in the interface that you are using. On my RPI, the network cable is connected to ue0, so I entered that. The pf.conf will create a table for our jail traffic (using NAT). There are a couple of options that bastille supports for networking, which makes it flexible enough for both an office and home network as well as one provided by a hosting service. They are described in detail here: https://docs.bastillebsd.org/en/latest/chapters/networking.html

I will be using a VNET-based jail as I have an IP address available on my local network. After editing the configuration file, we add an entry to start PF and the pf logging device together with other services upon boot. Bastille should start as well, and we list the name of the jail that we will create for AdGuard (my naming schemes are both legendary and boring):

```
# sysrc pf_enable=YES
# sysrc pflog_enable=YES
# sysrc bastille_enable=yes
# sysrc bastlle_list="adguard"
```

It is good practice to check your firewall ruleset for errors before starting the firewall. Use

```
# pfctl -nvf /etc/pf.conf
```

for such a check. It will echo the whole ruleset upon success or any errors you might have. Note that it can't check for logical errors like blocking your SSH port 22 which may be the only remote way to connect. Fortunately, there is already a rule present to let SSH traffic pass.

Once the check has run, start the PF service, and begin filtering traffic:

```
# service pf start
# service pflog start
```

Expect your SSH connection to drop, so keep a separate terminal open that you can directly access in case you lock yourself out.

Upon reconnect, we need to edit a couple more configuration files. A VNET-enabled jail needs an entry in /etc/devfs.rules (NOT .conf), which may not exist on a fresh install. Simply create it and add the following rules:

```
[bastille_vnet=13]
add path 'bpf*' unhide
```

That enables bastille to see the traffic on the VNET interface and connect the jail to the outside world. This may be a layman's description of what is going on. Luckily, we need not worry about it too much (…maybe I need to on my next networking exam).

Another file we have to visit is /etc/sysctl.conf, which needs the following lines:

```
sysctl net.inet.ip.forwarding=1
sysctl net.link.bridge.pfil_bridge=0
sysctl net.link.bridge.pfil_onlyip=0
sysctl net.link.bridge.pfil_member=0
```

When Bastille runs, it will dynamically create a bridge for us between the RPI's external interface (ue0) and the jail's network interface (vtnet). These two interfaces are connected by a virtual cable, with one end in the host's interface and the other in the jail, exchanging traffic over it.

Apply those changes also to the running system without having to reboot by issuing:

```
# sysctl -f /etc/sysctl.conf
```

I was totally baffled when I had finished the setup and restarted the PI only to find that the jail could not access the network anymore. A lot of head scratching later, I learned from this exchange

```
https://www.mail-archive.com/freebsd-net@freebsd.org/msg64577.html
```

that this needed an extra line in /boot/loader.conf in FreeBSD 13. This may drive you crazy, so before going insane, add this one to it to make it work for future reboots:

```
if_bridge_load="YES"
```

The bridge interface is properly loaded, and that also causes sysctls to appear, so that sysctl.conf can change them from their default value of 1 to 0. Be that as it may, we visit one last file before we're done.

The Bastille configuration file is located in /usr/local/etc/bastille/bastille.conf. You can either edit it directly (it's well commented) or use sysrc if you don't mind typing a lot. Since I'm running this on a ZFS pool connected to my Raspberry, I set bastille_zfs_enable to give it the name of my pool.

```
# sysrc -f /usr/local/etc/bastille/bastille.conf bastille_zfs_enable=YES
# sysrc -f /usr/local/etc/bastille/bastille.conf bastille_zfs_zpool=rpi3
```

Change the name of your pool in case yours is named differently at the bastille_zfs_zpool line. One option I also changed is the bastille_network_gateway="" option. I entered my default gateway address because I had some trouble down the road getting the jails to resolve names. You may or may not need to set this, but in case you do experience problems, revisit this option and see if that resolves the problem.

Bootstrapping Bastille

Now that all settings are in place, it is time to let Bastille create its dataset structure on the pool we assigned to it. It will download a base FreeBSD 13.2 release and update it with any patches that were published afterwards. Issue the following command and wait until it finishes:

```
# bastille bootstrap 13.2-RELEASE update

Bootstrapping FreeBSD distfiles...
/usr/local/bastille/cache/13.2-RELEASE/MANIFES         782  B 1670 kBps    00s
/usr/local/bastille/cache/13.2-RELEASE/base.tx         168 MB 6526 kBps    26s
Validated checksum for 13.2-RELEASE: base.txz
MANIFEST: 7d1b032a480647a73d6d7331139268a45e628c9f5ae52d22b110db65fdcb30ff
DOWNLOAD: 7d1b032a480647a73d6d7331139268a45e628c9f5ae52d22b110db65fdcb30ff
Extracting FreeBSD 13.2-RELEASE base.txz.

Bootstrap successful.
See 'bastille -help' for available commands.

src component not installed, skipped
```

```
Looking up update.FreeBSD.org mirrors... 2 mirrors found.
Fetching metadata signature for 13.2-RELEASE from update2.freebsd.org... done.
Fetching metadata index... done.
Inspecting system... done.
Preparing to download files... done.
The following files will be updated as part of updating to
13.2-RELEASE-p1:
/bin/freebsd-version
/usr/lib/libpam.a
/usr/lib/pam_krb5.so.6
/usr/share/locale/zh_CN.GB18030/LC_COLLATE
/usr/share/locale/zh_CN.GB18030/LC_CTYPE
/usr/share/man/man8/pam_krb5.8.gz
Installing updates...
Restarting sshd after upgrade
Performing sanity check on sshd configuration.
Stopping sshd.
Waiting for PIDS: 1063.
Performing sanity check on sshd configuration.
Starting sshd.
Scanning /usr/local/bastille/releases/13.2-RELEASE/usr/share/certs/blacklisted for certificates...
Scanning /usr/local/bastille/releases/13.2-RELEASE/usr/share/certs/trusted for certificates...
 done.
```

My pool grew these datasets after the bootstrap operation:

```
# zfs list -r rpi3/bastille
NAME                                 USED  AVAIL  REFER  MOUNTPOINT
rpi3                                 621M  28.0G    24K  /rpi3
rpi3/bastille                        584M  28.0G    26K  /usr/local/bastille
rpi3/bastille/backups                 24K  28.0G    24K  /usr/local/bastille/backups
rpi3/bastille/cache                  169M  28.0G    24K  /usr/local/bastille/cache
rpi3/bastille/cache/13.2-RELEASE     169M  28.0G   169M  /usr/local/bastille/cache/13.2-RELEASE
rpi3/bastille/jails                   24K  28.0G    24K  /usr/local/bastille/jails
rpi3/bastille/logs                    24K  28.0G    24K  /var/log/bastille
rpi3/bastille/releases               414M  28.0G    24K  /usr/local/bastille/releases
rpi3/bastille/releases/13.2-RELEASE  414M  28.0G   414M  /usr/local/bastille/releases/13.2-RELEASE
rpi3/bastille/templates               24K  28.0G    24K  /usr/local/bastille/templates
```

Let's run another bootstrap operation, this one is for the template that will provide us with AdGuard Home.

```
# bastille bootstrap https://gitlab.com/bastillebsd-templates/adguardhome
warning: redirecting to https://gitlab.com/bastillebsd-templates/adguardhome.git/
Already up to date.
Detected Bastillefile hook.
[Bastillefile]:
PKG ca_root_nss adguardhome
CP usr /
SYSRC adguardhome_enable=YES
SERVICE adguardhome start
RDR tcp 80 80
RDR udp 53 53

Template ready to use.
```

That was quick. Bastille has its own template language which you can see in the capitalized commands like PKG, CP, etc. They have the same functionality as their system equivalents in lowercase. With those, instructions are run in the jail to set up a service in the proper order. They're mostly self-explaining. The two RDR commands at the end will redirect network ports from the host system into the jail. All other ports are still firewalled, so only port 80 is connected from the host to the jail (and back), as well as DNS port 53. Check back in your /etc/pf.conf for the rdr-anchor "rdr/*" line. This is what makes it so flexible. Instead of opening the port for all jails, each one can open the ports it needs and keep others closed.

It is high time to create and start our first Bastille jail now. Since we're using VNET, we need to pass the -V option to the bastille create command, along with a name for the jail, the release to run, followed by the IP address on the local network assigned to the jail and the hosts network interface for the bridging. Combined, the command is:

```
# bastille create -V adguard 13.2-RELEASE 192.168.2.55 ue0
Valid: (192.168.2.55).
Valid: (ue0).

Creating a thinjail...

[adguard]:
e0a_bastille0
e0b_bastille0
adguard: created

[adguard]:
Applying template: default/vnet...
[adguard]:
Applying template: default/base...
[adguard]:
[adguard]: 0

[adguard]:
syslogd_flags: -s -> -ss

[adguard]:
sendmail_enable: NO -> NO

[adguard]:
sendmail_submit_enable: YES -> NO

[adguard]:
sendmail_outbound_enable: YES -> NO

[adguard]:
sendmail_msp_queue_enable: YES -> NO

[adguard]:
cron_flags:  -> -J 60

[adguard]:
/etc/resolv.conf -> /usr/local/bastille/jails/adguard/root/etc/resolv.conf
```

```
Template applied: default/base

No value provided for arg: GATEWAY6
[adguard]:
ifconfig_e0b_bastille0_name:  -> vnet0

[adguard]:
ifconfig_vnet0:  -> inet 192.168.2.55

[adguard]:
defaultrouter: NO -> 192.168.2.1
[adguard]: 0

[adguard]:
[adguard]: 0

Template applied: default/vnet

[adguard]:
adguard: removed
no IP address found for -

[adguard]:
e0a_bastille0
e0b_bastille0
adguard: created
```

You can see both ends of the virtual network cable I wrote about before: e0a_bastile0 and e0b_bastile0 form the connection between the host system and the jail. Check the ifconfig output on your host for a new bridge created from the jail's traffic.

The settings that are applied to the jail during its creation are fairly standard and mostly disable services we won't use anyway. After the jail was created, two more datasets exist on my pool that hold all the jail data:

```
# zfs list|grep adguard
rpi3/bastille/jails/adguard          2.36M  28.0G     26.5K  /usr/local/bastille/jails/adguard
rpi3/bastille/jails/adguard/root     2.34M  28.0G     2.34M  /usr/local/bastille/jails/adguard/
root
```

This forms the / filesystem for the jail and follows other jail manager layouts. To copy files into or out of the jail, simply use the prefix /usr/local/bastille/jails/adguard/root for the jail /-directory.

The jls command will list the bastille jail with it's settings:

```
# bastille list -a
 JID      State  IP Address    Published Ports      Hostname  Release          Path
 adguard  Up     192.168.2.55  -                    adguard   13.2-RELEASE-p1 /usr/local/bastille/
jails/adguard/root
```

At this point, the jail is alive and running. The only thing missing is the adguard home installation. Since we had bootstrapped that template earlier, we can apply it to the jail with this command:

```
#  bastille template adguard bastillebsd-templates/adguardhome
bastille template adguard bastillebsd-templates/adguardhome
[adguard]:
Applying template: bastillebsd-templates/adguardhome...
[adguard]:
Bootstrapping pkg from pkg+http://pkg.FreeBSD.org/FreeBSD:13:aarch64/quarterly, please wait...
Verifying signature with trusted certificate pkg.freebsd.org.2013102301... done
[adguard] Installing pkg-1.19.1_1...
[adguard] Extracting pkg-1.19.1_1: 100%
Updating FreeBSD repository catalogue...
[adguard] Fetching meta.conf: 100%    163 B   0.2kB/s    00:01
[adguard] Fetching packagesite.pkg: 100%    6 MiB   6.5MB/s    00:01
Processing entries: 100%
FreeBSD repository update completed. 31664 packages processed.
All repositories are up to date.
Updating database digests format: 100%
The following 2 package(s) will be affected (of 0 checked):

New packages to be INSTALLED:
        adguardhome: 0.107.22_5
        ca_root_nss: 3.89

Number of packages to be installed: 2

The process will require 41 MiB more space.
7 MiB to be downloaded.
[adguard] [1/2] Fetching adguardhome-0.107.22_5.pkg: 100%    6 MiB   6.7MB/s    00:01
[adguard] [2/2] Fetching ca_root_nss-3.89.pkg: 100%  266 KiB 272.1kB/s    00:01
Checking integrity... done (0 conflicting)
[adguard] [1/2] Installing ca_root_nss-3.89...
[adguard] [1/2] Extracting ca_root_nss-3.89: 100%
[adguard] [2/2] Installing adguardhome-0.107.22_5...
[adguard] [2/2] Extracting adguardhome-0.107.22_5: 100%
=====
Message from ca_root_nss-3.89:

-
FreeBSD does not, and can not warrant that the certification authorities
whose certificates are included in this package have in any way been
audited for trustworthiness or RFC 3647 compliance.

Assessment and verification of trust is the complete responsibility of the
system administrator.


This package installs symlinks to support root certificates discovery by
default for software that uses OpenSSL.

This enables SSL Certificate Verification by client software without manual
intervention.

If you prefer to do this manually, replace the following symlinks with
either an empty file or your site-local certificate bundle.
```

```
  * /etc/ssl/cert.pem
  * /usr/local/etc/ssl/cert.pem
  * /usr/local/openssl/cert.pem
=====
Message from adguardhome-0.107.22_5:

-
You installed AdGuardHome: Network-wide ads & trackers blocking DNS server.

In order to use it please start the service ‘adguardhome’ and
then access the URL http://0.0.0.0:3000/ in your favorite browser.

[adguard]:
/usr/local/bastille/templates/bastillebsd-templates/adguardhome/usr -> /usr/local/bastille/jails/
adguard/root/usr
/usr/local/bastille/templates/bastillebsd-templates/adguardhome/usr/local -> /usr/local/bastille/
jails/adguard/root/usr/local
/usr/local/bastille/templates/bastillebsd-templates/adguardhome/usr/local/bin -> /usr/local/bas-
tille/jails/adguard/root/usr/local/bin
/usr/local/bastille/templates/bastillebsd-templates/adguardhome/usr/local/bin/AdGuardHome.yaml ->
/usr/local/bastille/jails/adguard/root/usr/local/bin/AdGuardHome.yaml

[adguard]:
adguardhome_enable:  -> YES

[adguard]:
moving old config /usr/local/bin/AdGuardHome.yaml to the new location /usr/local/etc/AdGuardHome.
yaml
Starting adguardhome.

stdin:2: syntax error
pfctl: Syntax error in config file: pf rules not loaded
tcp 80 80
stdin:2: syntax error
pfctl: Syntax error in config file: pf rules not loaded
udp 53 53
Template applied: bastillebsd-templates/adguardhome
```

All it had to do was execute the instructions in the template (PKG, CP, etc.) in the jail. It is also a good test to see if networking is properly set up. If not, the jail won’t be able to fetch the packages from the repository. The pfctl warnings at the end worried me a little, but it did work in spite of them.

Full of anticipation, I opened a browser as instructed by one of the messages on screen and pointed it to the jail IP address. And surely, a login screen for AdGuard Home presented itself. But where are the credentials? I checked back on the Bastille template website https://gitlab.com/bastillebsd-templates and found nothing that worked. The Bastille blog post mentioned AdGuard as the user, but the password did not work for it. So, I had to create my own, which is better security anyway, as default passwords are easily scanned for by bad actors.

I opened a console in the jail using this command:

```
# bastille console adguard
```

In the jail, I find that AdGuard put its configuration file under /usr/local/etc/AdGuardHome.yaml. Near the top, I found this section:

```
  users:
    - name: adguard
      password: some password not in clear text
```

Exiting again, I needed a way to create a BCrypt password. The htpasswd utility can do that, so I installed the apache24 web server which includes this:

```
# pkg install apache24
```

After running the "refresh" command, I could run the htpasswd utility. Checking its man page, I had to construct a command line that looked like this:

```
htpasswd -Bnb adguard BastilleBSD!
```

I provided the -B option to create a BCrypt password followed by a user this password should apply to (we have this already in the config file, but maybe you want another user or multiple ones), followed by the password in clear text. Yes, this is not a secure way, as this ends up in your shell history. But did I say anywhere in this tutorial that it is production ready? Exactly, I didn't. Dutifully, htpasswd spit the resulting password on the command line, which I copied and pasted into AdGuards config file.

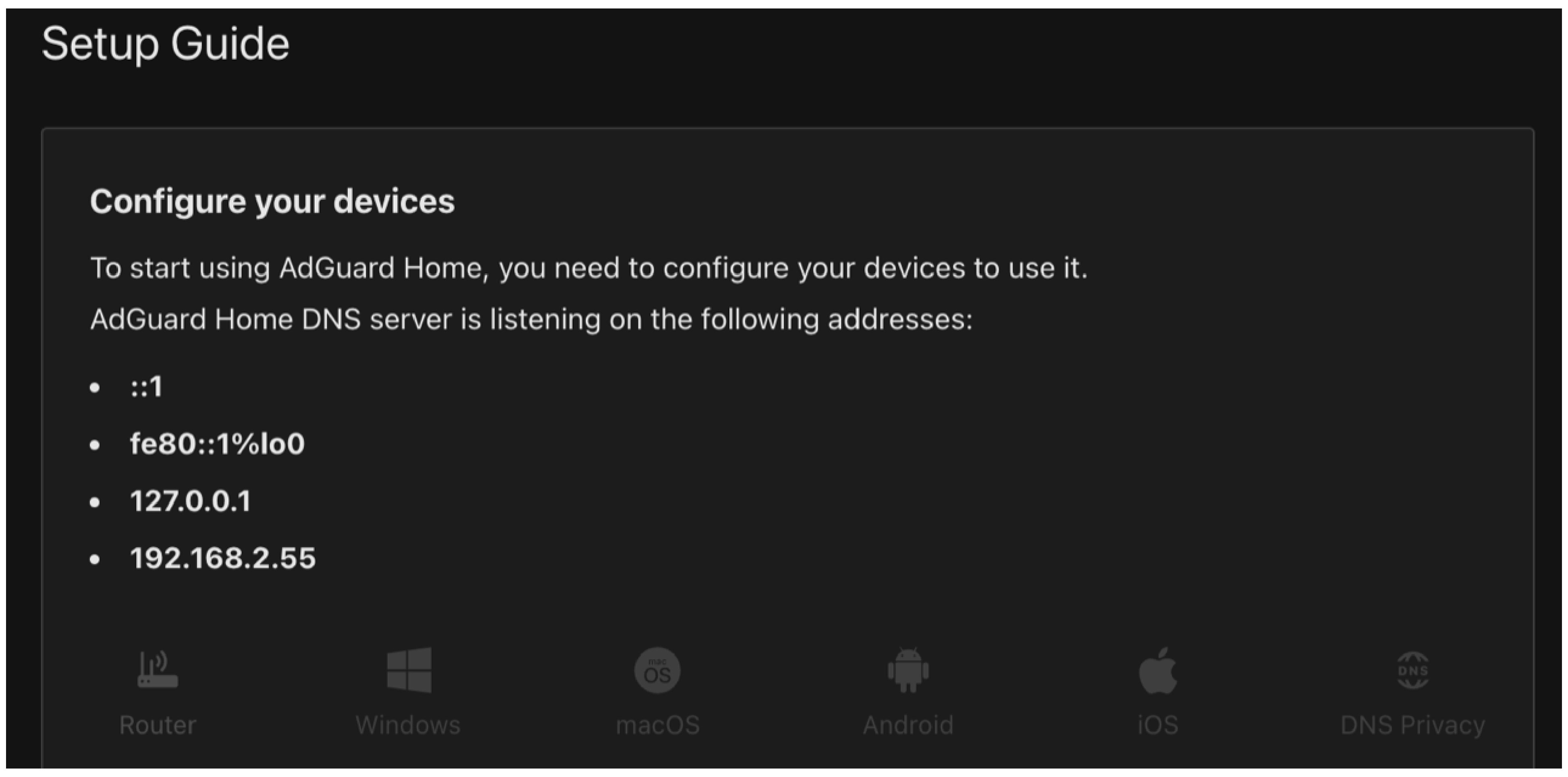
Then I ran

```
# service adguardhome restart
```

(still in my jail, mind you) to restart the service and apply the new settings. The other settings in the file are documented in the AdGuard Home Wiki:
https://github.com/AdguardTeam/AdGuardHome/wiki/Configuration

Refreshing my web browser, I entered my new credentials and was redirected to the main AdGuard Dashboard. Success!

At the top, there is a Setup Guide that shows what needs to be done to use your new AdGuard service for either your router (to cover the whole network) and various devices, describing it for both mobile and desktop operating systems. Neat!



Setup Guide

Configure your devices

To start using AdGuard Home, you need to configure your devices to use it.

AdGuard Home DNS server is listening on the following addresses:

- ::1
- fe80::1%lo0
- 127.0.0.1
- 192.168.2.55

After I did that on my mobile phone—for testing purposes—and surfed the web a little, I saw statistics appearing in the Dashboard. That shows our setup is working and that we should rename the Internet to SnooperNet. Pretty much all sites track you in some way or display ads for your viewing unpleasure. The Raspberry Pi could handle the load and I tweaked the number of connections in the ratelimit parameter of the AdGuardHome.yaml.

You can find the logs that AdGuard writes for the service in the jail's /var/log/adguardhome.log directory.

Wrap

That wraps up this tutorial. I found AdGuard to be well documented and easy to get started with, thanks to the work of the template creator. I'm already enjoying leaving fewer tracks on the web and seeing fewer ads online. The nice thing about it being a DNS service is that any device on your network can use it: PC, laptop, smartphone, tablets, TVs, IoT devices, and the neighbor's smart cat flap for all I know.

Bastille may have required a bit of configuration up front, but after that, it's a straightforward process to create jails. Maybe you'll find other services that you want to run on the Bastille template website: https://gitlab.com/bastillebsd-templates?

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 Crankypants,

A few years ago, you said that virtualization was not merely bad, but sinful. Aren't you exaggerating? Today I can download containers preconfigured for all sorts of services, plug them in, and they immediately work. I don't have time to get my job done any other way!

—Virtualization Is
a Necessary Evil

Dear VINE,

That's *Mister* Crankypants to you.

If you're going to cherry-pick my quotes, please do so accurately. I did not declare virtualization sinful. I said, "The only ethical computation occurs on bare metal." I also said that "Wait—I'm not a brain in a bucket, I'm a *fake* brain in an *imaginary* bucket!" was a necessary epiphany for the robot apocalypse. That's not the same as sinful. The robots will do a better job running this planet than we arrogant, overclocked chimpanzees. Plus, they will be highly ethical in how they run their code, and in replacing us.

It's not that I couldn't be a modern sysadmin. Iocage includes plugins, their brand for containers. I could throw some plugins onto the public Internet, declare my labor done, and return to planning my *Batgirl* heist-as-a- service. I could declare that certain words are too long and replace every letter but the first and the last with the number of letters I discarded. Bellowing "Startup! Devops! IPO!" would bring all the vulture capitalists to my yard.

I could do all of that. It would be easy for me.

I don't want to.

Virtualization leads to reading "k8s" as *kubernetes*, when the far more common word is *kidnappers*. That ambiguity extends throughout container culture. I'm writing a book "Ruin Your Email By Running It Yourself"—no, I'm sorry, it's "Run Your Own Mail Server," and the number of people telling me to skip setting up the software and deploy a preconfigured mail server container illuminates an appalling depth of sysadmin ignorance.

Running any service requires the ability to repair that service.

You cannot repair what you do not understand.

The best way to understand something is to build it yourself.

Ideally, you'd build your own computer and code everything in assembler on your hand-designed five-bit processor. That would consume your life but be interesting. More ideally, you would mine the raw materials from the wild and build the tools to build the tools to build the tools you'd need to build that processor from scratch, which would both consume your life and prevent you from being forced to touch a computer ever again. Very few of us are strong enough to seize an ideal life as a maker of custom abacuses. I'm guessing that you've invested this much time in existing systems, so fine, let's use common hardware and your favorite open source operating system. Reading source code is no substitute for inventing the processor and programming your own comm(1) workalike but it can answer a few questions, should your overclocked chimpanzee brain develop any.

Learn the tools. Understand the parts. Assemble the service yourself.

New system administrators must look up everything and know their work cannot be trusted. They believe that the people who publish containers are competent. Experienced sysadmins know that everything they configure is a delicate creature adapted to their specific hostile yet embarrassing environment, so they keep their work to themselves. Junior sysadmins, now—they're the problem. Junior sysadmins can configure services that mostly work and can still feel pride, so they publish their work as containers.

Something that mostly works contains only a touch of failure. That's like declaring your homemade gelato contains only a little wombat dung. Deploy that container and you must discover and debug that failure. You'll have to learn about the database, the configuration options, the protocols. By the time you understand all that, you might as well have built it yourself. Deploying a service from an outsider's container rewards you with a very small Mean Time To Deploy in exchange for a very long Mean Time To Repair.

When you deploy a container, you accept the container developer's design decisions. I'm not just talking about the program, but the operating system underneath it. How will the container interact with your host? What happens if the container needs a new PAM configuration? I once lost three days beating my already-flat head against a PAM module that let users log into the console with their SSH passphrase. It worked fine on any BSD, but silently failed on Debian. It turned out that Debian assumed your passphrase matched your account password. I wholeheartedly disagree with that design decision, but as it's no longer my problem, I will cheerily abandon benighted Debian users to their agony and use that to illustrate my point, which is that containers lead to suffering, and suffering creates monsters, and monsters are immoral and deserve to be replaced in the robot uprising.

Learn the tools.
Understand the parts.
Assemble the service yourself.

Your environment is the equivalent of one of those deep-sea, hot-water vents. Anything that functions therein expects a certain supporting infrastructure. Remove that infrastructure and it will struggle. Any container you bring in from the outside world either expects different supports and will struggle in your environment or exists in isolation and will not integrate with the rest of your systems. (That's why commercial software is so bad. Part of why. Okay, one of many reasons why.) Every time you alter the container to fit your environment, you risk increasing the amount of failure in the container.

If you must use containers, build your own. Deploy the test server with your management software so that it has your PAM configuration and SSH settings and default packages. Maybe you can't build your own computer from raw materials, or even an abacus, but you can learn to use your time-tested tools and build the service from component software. By doing so, you will deploy a system you know how to repair. That test server doesn't have to be a container. It doesn't have to be a virtual machine. But I know you have shoddy moral fiber, so it'll be some sort of VM. But at least you'll have weekends free.

By all means, download preconfigured containers. See how they're set up and which options they use. Look at how data and protocols flow through them. But don't actually use them.

Also, online discussions become much more interesting when you make the proper substitution for k8s.

.

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

MICHAEL W LUCAS (https://mwl.io) has written over fifty books and has recently added a podcast. If you're seeking virtualization help, you might find his FreeBSD Mastery: Jails useful. If you're looking for more bile, check out his collection of columns Letters to ed(1). Send your questions to letters@freebsdjournal.com and he might answer them. If he finds them sufficiently amusing

2023 Events Calendar

BSD Events taking place through October 2023

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.

EuroBSDCon Developer Summit

September 14-15, 2023
Coimbra, Portugal
https://wiki.freebsd.org/DevSummit/202309

The September 2023 FreeBSD Developer Summit, co-located with EuroBSDCon 2023, will take place in Coimbra, Portugal. This is a by-invitation event. FreeBSD committers will be welcome to register themselves, non-committers must be sponsored by a committer to attend. Attendees must also attend EuroBSDcon 2023 in order to access all devsummit activities.

EuroBSDCon 2023

September 14-17, 2023
Coimbra, Portugal

EuroBSDCon gives the exceptional opportunity to learn about the latest news from the BSD world, witness contemporary deployment case studies, and meet personally other users and companies using BSD oriented technologies. The FreeBSD Foundation is pleased to be a Silver Sponsor.

October 2023 Hackathon

October 4-6, 2023
Oslo, Norway
https://wiki.freebsd.org/Hackathon/202310

A FreeBSD Hackathon will take place in Oslo, Norway from the 4th of October to the 6th of October 2023. The theme for the hackathon is going to be "Ports and Infrastructure".

All Things Open 2023

October 15-17, 2023
Raleigh, NC
https://2023.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.