SEE TEXT ONLY

FROM Microservices to Monoliths

BY DAVE COTTLEHUBER

Over the last 18 months, we've re-platformed twice, hopefully without most people noticing at all. The first major shift was for cost advantage, changing our host provider, and the second was moving to the latest Debian release, which brought with it a flurry of patches and changes—not all of it wanted.

There was a feeling we were beholden to our OS and its patches, rather than to our customers and our business. systemd introduced a cascade of changes requiring changes to all of the services we run, and the changes continue to roll in over time.

Migrating hosting providers enabled us to move our core database servers, running Apache CouchDB (https://couchdb.apache.org/), to much larger and faster machines, using SSD and much more RAM. Building indexes is faster, which allows us to deploy code faster as well. In fact, it's now possible to cache our entire database and indexes in RAM, which has helped the responsiveness of our whole site.

We've always had a fairly microservices-like architecture, and it's stood the test of time. The front-end application is written in Perl using the Catalyst (http://www.catalystframework.org/) framework, and communicates with workers in a variety of programming languages, using RabbitMQ (http://www.rabbitmq.com/) as a message broker between services that run on several different servers. Our two main databases, Apache CouchDB and Kyoto Tycoon (http://fallabs.com/kyototycoon/), have built-in replication, which provides both application-level redundancy and also simplifies operations when doing backups or upgrades.

Despite that, microservices introduced small delays at every step—network latency due to roundtrips, further reliance on stable Internet connections, and extra conversions between JSON and the native programming languages used for each service. End-to-end testing of our application was also very complicated. Something needed to change.

The Best of Both Worlds?

We're not Twitter scale, and we don't need the feature set of Amazon or Google's clouds. We're also mindful of avoiding Cloud Lock-in. Our profitability as a business won't change much if we halve or double the infrastructure we use. We asked, is there a way to have the best of both worlds? The decoupling of microservices, without the latency? The debugging simplicity of a monolith, without the interruptions from continual upgrades? Could we have the operational flexibility of a cluster without the risk of a catastrophic meltdown? Could we have a test environment that ran on a laptop, but that still matched production?

Probably a number of you are mumbling Docker (https://www.docker.com/) under your breath. Some of you are getting sweaty palms and thinking of Linux containers (https://linuxcontainers.org/). But we'd tried them, and it wasn't the answer. We spent more time trying to get the tightly coupled container stack working together than actually shipping code to production, or improving the stability and reliability of our services.

The key failure of the container vision today is that, unless you are Google scale, where you have a significant cost advantage from reduced server footprint, and where you can afford a fleet of container infrastructure engineers to keep up with the evolving landscape, the operational effort simply doesn't stack up to deliver the benefits. It shouldn't be necessary to rewrite how we do logging, monitoring, packaging, and deployment, and to dedicate engineers to maintaining and grooming the container monster, just to simplify shipping code to production.

Plain Old-fashioned Boring Infrastructure

What we needed was some plain old-fashioned boring infrastructure. Loosely coupled at each layer, without introducing latency, and still allowing the flexibility of the container-think movement. Something that had a long support life span if we needed it, but that didn't compromise our ability to keep on the front foot for patches and security of both the OS and the apps that came with it.

In the end, we settled on three core changes:

- move our OS from Debian Linux to FreeBSD
- switch out distributed container-style VM microservices to paired physical servers
- migrate our Perl-based Catalyst core app over to Elixir and Phoenix

Many of you will know of these already, but here's a glimpse into our thinking.

FreeBSD

FreeBSD (https://www.freebsd.org/) is one of the original Free UNIX-like operating systems, and is going stronger than ever. It powers Netflix's mighty streaming servers, and is used in a modified form in Sony Playstations and Apple's iOS and OSX. Most Internet providers use FreeBSD in some form. FreeBSD also has a long support life span for the core OS while at the same time allowing us to use the latest ports and packages—a neat mix of backward compatibility.

And to be honest, FreeBSD already had a leg-up as a couple of us have been using it for a while.

The three big advantages of FreeBSD for us are zfs (http://open-zfs.org/wiki/Main_Page), jails (https://www.freebsd.org/doc/handbook/jails.html), and ports (https://www.freebsd.org/ports/references.html).

zfs is arguably the leading filesystem of all time, with great flexibility, and power including inbuilt high-speed compression, data checksums (no bitrot or silent data corruption), snapshots for replication and backup. It also supports boot environments (http://callfortesting.org/bhyve-boot-environments/), which is a clever snapshot-based way of managing upgrades safely. This makes testing and ultimately deploying a new version of FreeBSD, or our apps and services deployed upon it, largely risk-free, and very, very simple.

FreeBSD jails are about a decade old, and similar to Linux containers conceptually. However, there are no venture-backed companies fighting over turf in the hope of achieving a VMWare-like monopoly, and the software is well integrated

into the operating system and community tooling. The performance of jailed applications is effectively the same as that of running in the main kernel, both from a network and a filesystem standpoint.

The ports tree is a significant feature that all BSD-derived operating systems share a massive repository (subversion or git as you please) of every piece of open-source software you could possibly imagine. As our core business is providing simplified domain purchase and management through custom software, we are often in the position of needing a specific version of a tool or application, or needing custom patches deployed immediately while we wait for the upstream application owners to merge a patch, or for it to trickle down into the OS distribution's package manager. With the ports tree, we have a custom private repo for our own packages, and the ability to carry and patches or to hold back specific versions for our own needs, in a very simple and straightforward fashion. Where we've needed new packages, or to get changes committed, it's proved extremely simple to do so. This makes maintaining our own infrastructure very simple indeed—installing a handful of packages gets us up and running much faster than in the past.

There is a fourth advantage for us, however. The FreeBSD community is close-knit, with a reasonably consistent culture about doing things right. In practice, this means there's very little gap between issues we identify or knowledge we are missing, and the developers and community creating it. The documentation on FreeBSD itself is part of this culture of doing things right, and the integration between the OS itself and the tools it ships with are the result, and we are already looking forwards to contributing further to the community.

Overall, as a result of moving to FreeBSD we hope to spend significantly less effort managing our infrastructure, and more time invested in improving our services and business.

A Pair of Servers

FreeBSD's jails allow us to run microservices on a single box. Arguably this is no different from Linux containers in practice; however, we are not fighting to accommodate a stream of changes that add no value to our business along the way, and we are not forced to change out our operations tools and processes. The value to our customers is not in having the latest container tech running; it's in having simple and reliable services for the infrequent times they need to acquire or manage their domains throughout the year.

In each region, we are deploying a pair of physical servers, each one providing the same services and applications. Within each pair, we are using DNS round-robin name resolution and CARP, a low-level IP availability solution built into FreeBSD, to provide load balancing and failover at a network level between our boxes.

The next layer up in the paired server stack is the awesome haproxy (http://www.haproxy.org/) load balancer, which we use to ensure that we direct our users to the closest and best-performing application server. haproxy also allows us to dynamically remove and add back-end services from the pool, whether during deployment or maintenance, and communicates across the cluster to maintain a transparent view of services for our customers.

This consolidation brings disparate virtual machines back onto the same server, while still using FreeBSD jails to maintain the microservices-like separation. Luckily, none of our apps have required major changes to make them run on FreeBSD—UNIX standardization has been a huge benefit here. When this is complete, we'll have significantly reduced our app latency by removing the network round-trips that we have today.

The key difference here from the Docker-style container architecture is that there is very little coordination or dependency between these layers.

CARP is fundamentally a network protocol, and we could easily disable it should there be any issues, or choose some other facility. haproxy could be replaced by nginx (http://nginx.org/) which we already use today for slightly different functionality. zfs provides an incredible filesystem, available within jails and to the core operating system, as a solid and reliable platform for our services and your data. Logging, monitoring, and upgrades are all done using the same decoupled tools using well-known UNIX standards in place for decades—and there's nothing wrong with preserving that simplicity where it suits us.

Elixir and Phoenix

Since almost the beginning of iwantmyname, the programming language Erlang/OTP (http://www.erlang.org/) has been at the heart of things. It's the programming language that Apache CouchDB is developed in, as well

as RabbitMQ, our message broker, and our core search (https://iwantmyname.com/?domain=) application is also written in Erlang. Its robustness has shown time and time again as it transparently deals with issues such as transient connection failures to our API partners, and it has generally required significantly less maintenance effort than our other services. As a concurrent functional language with soft-real-time characteristics, it is ideally suited to building websites and services that make heavy use of asynchronous internal and external APIs.

Our front-end app, written in Perl's Catalyst (http://www.catalystframework.org/) framework, was leading edge when we first started using it, but it's become more and more of a hindrance in evolving to a more robust, mobile first system. Perl's forking worker model means that we use a significant amount of RAM across our infrastructure just to ensure we can handle what is definitely not "web scale" user and network load.

After experiencing several years of solid Erlang reliability, we picked Elixir (https://elixir-lang.org/), a new functional programming language that runs on the same Erlang VM, and the Phoenix (http://phoenixframework.org/) web framework written in the same language, to upgrade and eventually replace our front-end application.

Erlang's BEAM virtual machine provides Elixir, and Phoenix, with a screamingly fast, robust, and reliable concurrent web framework, without the memory overhead of a forking model, that is able to handle many concurrent connections transparently. Both Erlang and FreeBSD are used heavily by WhatsApp, and they shot to worldwide notice (https://duckduckgo.com/html/?q=erlang+whatsapp) when Facebook acquired the low-staffed app.

Live Debugging

Sometimes stuff breaks in production for no apparent reason, and we need to know why—in a hurry. In recent months, we traced and debugged transient Internet outages, upstream API changes, timeouts and failures, unanticipated third-party library concurrency model changes, and much much more—much pain and frustration!

Debugging was a painful process, usually involving reading all the log files, using low-level Linux tools like strace, all the while inserting print statements and redeploying furiously while trying to understand the underlying issues from the resulting flood of information.

Aside from being able to roll back safely any changes using boot environments (http://callfortesting.org/bhyve-boot-environments/) and packages, our new stack provides some incredible introspective live debugging capabilities, which makes it easier for us to deal with problems in real-time and without downtime, and most importantly without needing to change compiler settings or edit our production code on the fly. These advantages alone would be reason enough to move. The Erlang VM provides a native erlang tracing (http://erlang.org/doc/man/dbg.html) library, and the community has extended this with erlang dtrace (http://erlang.org/doc/apps/runtime_tools/DTRACE.html) support, and the delightful recon (https://ferd.github.io/recon/recon_trace.html). FreeBSD itself supports natively DTrace (http://dtrace.org/blogs/), the powerhouse introspection tool first developed for Sun Solaris, and since ported to several other platforms.

We've dubbed this setup the "FRECK Stack"—FreeBSD, RabbitMQ, Elixir, CouchDB, Kyoto Tycoon. Yes, we had some closely related abbreviations in mind, but we managed to control our childish mirth.

Looking Ahead

At the end of the day, iwantmyname (https://iwantmyname.com/) is a domain registrar, and when it comes to domains, a "plain old boring infrastructure" is exactly what we—and you—should want.

With FRECKS, we're now more stable and more secure than we've ever been—allowing us to truly focus our brain muscles on overdue UI and UX upgrades. More changes to our iwantmyname platform are coming, but behind the scenes; it'll hopefully be smooth sailing out front—touch wood! ●

DAVE COTTLEHUBER enjoys building and supporting distributed systems, especially when he can use both Elixir and FreeBSD. He shaves yaks at https://iwantmyname.com/, a domain reseller based in New Zealand. He lives in Vienna, Austria, with his wife and three boys, next to a lake. In his free time, he drinks Zwickl, eats schnitzel, wears lederhosen, and contributes to a number of open-source projects.