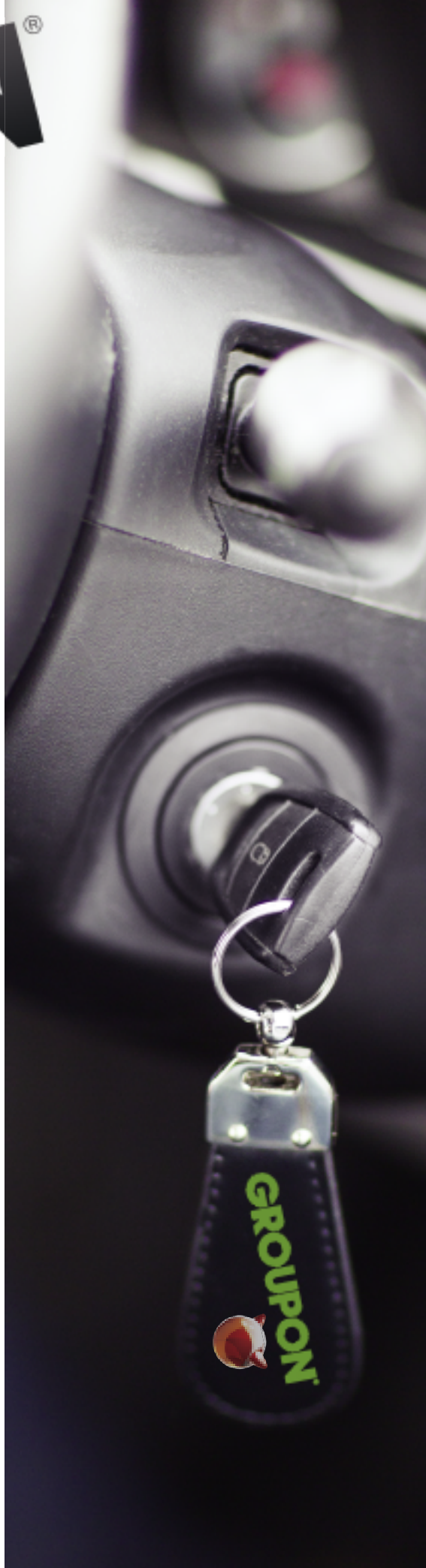
# GROUPON's Deal on FreeBSD

## Familiar look and feel, slightly different trim, now with enhanced results.

**By Sean Chittenden**




**It's 7:27a.m., I'm queued up at a metering light on my way to work, and my phone lights up with a message from one of database team's senior engineers:**

**"Do you have some time for debugging session? We most likely have a BSD issue."**

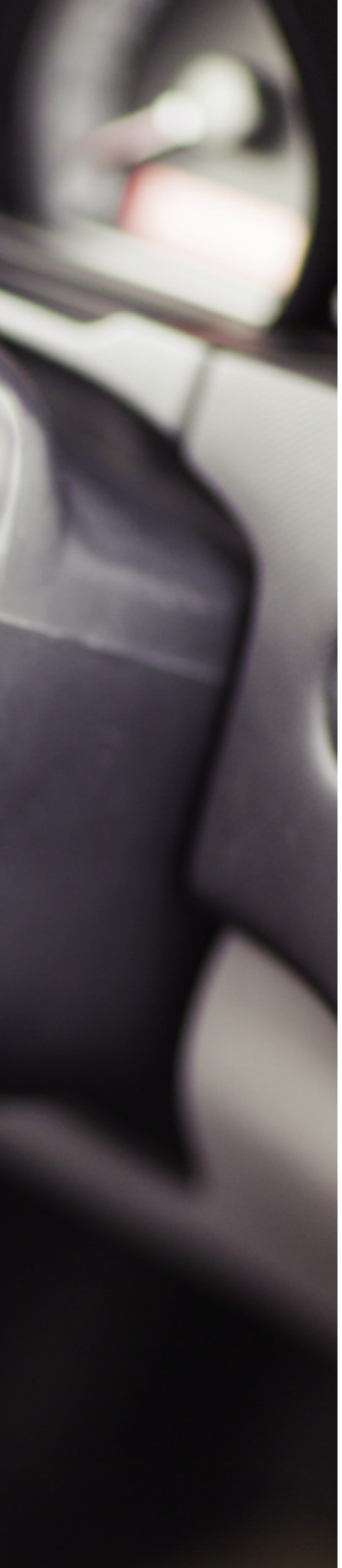
**I grimace and start to think about what he could be stuck on. I'm not exactly sure where their team is in the execution of their current project either, so I can't easily speculate. The thing is, I don't normally hear from this guy unless he's really stuck on something. So hearing "a BSD issue" when I'm not even halfway through my first cup of coffee isn't giving me a warm and fuzzy feeling, and it is definitely not how I want to start my day.**

My calendar is jammed full of meetings including a presentation and two deliverables, not to mention the odd escalation that shows up and requires immediate attention. But a "BSD issue" sounds like a rabbit hole of an issue and oddly specific.

I fire back, "I'm about to get on the freeway, can you call me?" He responds, "Can you get back to me when you are in front of `tmux(1)`?"

"Uh oh," I think, "this is probably something legit."

It's been a few months since we've run into any issues, so maybe we're due for an OS bug. Operations is in the process of finalizing its latest iteration of a database containerization platform, so hearing we've stumbled across something doesn't shock me since we're still working



through some details of the current iteration. After all, we were playing with `VIMAGE` at one point so it's not out of the question. Further, with over five petabytes of storage on the most recent deployment, it wouldn't surprise me to find out we've also just run across something run-of-the-mill this morning that comes with having a large deployment.

As I'm rounding the last corner on my way to the office, I get a text message from Chris Schneider, manager of the global database team, "Hey, Bob needs help and thinks he has a BSD issue because he's seeing a `listen(2)` queue overflow kernel message when debugging the new service checker daemon."

I think, "Whew! New code and it's in the application's network code, this should be easy."

And that's how it goes. Every few months as the footprint of FreeBSD grows in the organization, we collectively stumble across "new and exotic error messages" that we add to our repertoire and around which we build processes and understanding.

## Background

A little over a year ago Groupon began developing a next-generation database platform aimed at increasing reliability, reducing the cost of operations, and adding resiliency to some of the anticipated changes in the storage industry, notably the proliferation and widespread adoption of solid-state drives (SSDs). When the database project was initially started, my boss directed me to use SSDs because he wanted to get away from some of the more expensive flash memory technologies we already had in house. While the legacy flash hardware had been adequate for years, its integrated OS drivers and hardware have been operationally problematic and the cost made it prohibitive to reasonably deploy throughout the entire database tier. So off to work we went.

When building out large footprints, the law of large numbers is not just a convenient theorem to cite in a discussion; it needs to be a fundamental design consideration. The words "shouldn't happen" are taboo and a clear sign more research and analysis are required. Like any other part of business, identifying both strategic and operational planning assumptions and managing the risk/benefit is critical. If, however, businesses do manage to incorporate the entirety of potentially known issues or "risks" into their planning assumptions, they're skilled, a bit lucky, and a little clairvoyant. If the planning assumptions don't identify *and* consider all the known risks or create the corresponding contingency solutions to make the risks acceptable, there had better be good justification. In this case, the level of planned consolidation (>15:1 reduction in hardware) and operating efficiency gains that would come from moving off of spinning media and over to SSDs was clearly a benefit; however, the quantity of flash memory cells and the fidelity of the data were identified as risks stemming from the change in storage media. We knew this up front so we had to do something from the onset to manage the identified risks.

In Operations some of our primary planning assumptions include: a percentage of our servers die, hard drives fail, RAM develops single-bit parity errors, and datacenters go dark. We understand and plan for all these situations—spare servers, use of hot-swap hard

drive bays, procure ECC RAM, and geographic redundancy initiatives—and much, much more. The introduction of SSDs (versus spinning rust) introduced a substantive change to the operating model because the flash cell-based SSDs have a dramatically elevated bit error rate (BER). We knowingly understood this change in storage media would result in a material increase in the rate at which data would corrupt—rot—at rest. In the case of bit rot, the law of large numbers dictates it's not a matter of "if," but "when," and "how often." This created the need to update our team's planning assumptions and risk mitigation strategies.

The performance gains of moving from spinning rust to SSDs are significant. Seek times on 10K RPM disks are on average measured between 2 and 5 milliseconds, but the 99th percentile latencies can be measured in the tens or hundreds of milliseconds. Moving from milliseconds ("ms," one thousandth of a second) to microseconds ("us," one millionth of a second) represents a three order-of-magnitude improvement and this performance gain is now able to be rolled out to all teams and applications, not just the Tier-1 databases. And sure, the performance gains are nice for benchmarks, but for engineering teams, performance gain translates into a real-world efficiency gain which shrinks development schedules. Inefficient queries that were frequently cache-miss and would take 50–100ms now take 60–200us. CPU usage rises to desirable levels and engineering teams don't have to worry as much about performance efforts. In effect, use of SSDs allows us to trade OpEx for CapEx by reducing the time required to ship many products, but what's the trade-off and what does an organization do about it?

A year before Carnegie Mellon University and Facebook pointed out in their fantastic study published at the ACM SIGMETRICS '15 conference in June of 2015, "A Large-Scale Study of Flash Memory Failures in the Field," we were toiling away with designs for the database platform that would compensate for our observed bit rot on SSDs. At the time, we were seeing errors, but we didn't know how widespread the problem would be, just that we expect it to happen and therefore we had had to compensate for this inevitability. When the CMU/FB flash memory paper came out, their findings were very much in line with what we had observed; however, their testing methodology was much more extensive and the results poignant. To sum-

marize a few highlights regarding their findings:

- six different major SSD vendors
- uncorrectable bit error rate (UBER) is reasonably common
- the most reliable storage platform saw at least one UBER on 4.2% of its installation over a 12-month period of time
- the least reliable platform saw at least one UBER on 34.1% of its installation over the same duration and workload

Clearly it's still early days for SSDs and things will improve over time, but the variance in UBER across vendors is worth noting. For one workload, we selected a vendor's platform and experienced a >15% UBER in under a four-month period of time and expect that to settle out around 30–40% over a similar 12-month duration. One nonscientific observation we have made is that drives which throw UBERs will throw lots of errors and that the distribution of UBERs is not even across the fleet (i.e. some drives throw lots of errors, others throw very few, or no errors – we have not looked into why as of yet).

Maybe you are thinking to yourself, "So what if a bit flips? It's one bit. A gigabyte is 8-billion bits and one bit `""` shouldn't actually affect anything." Out of 8-trillion bits, one bit is vanishingly insignificant, yet if a flipped bit happens to be in `/usr/local/bin/vim` and it refuses to start, is it still insignificant? What if the bit flipped in the middle of a database table, how would you know? The `vim(1)` example is not hyperbole, as it was the first program to experience a detected UBER. The error occurred three months after one of our test machines was provisioned, it's `mtime` was three months in the past, and one day *BAM*. That's basically how bit rot happens: it silently, quietly, and probabilistically kills a bit somewhere in the middle of the night, and does so without any fanfare or helpful error messages. Try playing around with striped disks instead of mirrors in a test environment for a few months. On a disk of full of 1's and 0's, suddenly one of them flips. It's possible someone else on this planet is more unlucky than you and you won't have any errors, but that level of denial won't change reality. Sweet dreams, everyone.

Assuming you didn't pick the platform which had just over a third of its drives throw UBERs, what is more concerning was the incidence of UBERs increased in proportion to the utilization of the SSD. At the time of this publication and according to the CMU/FB flash memory paper, the probability of an SSD from one of the reliable vendors experiencing at least one UBER over the course of a 12-month duty cycle will stabilize between one in 10 or one in 25. Depending on the drive density in a server, that could be >1 UBER per server per year if you load up 24 SSDs per server, or just over one in 10 servers if you only put 2x SSDs per server. Are we feeling squeamish about using SSDs in production yet? If not, I encourage you to find the CMU/FB flash memory paper and ruminate accordingly because those error rates are too high to simply ignore.

Let's put it another way: if bit rot represents a needle in a haystack, is one needle in every other bale of hay acceptable? How about one needle in every two hay bales? What's the probability of a flipped bit causing a problem? How about a dozen new needles every quarter added to every hay bale? Would anyone notice or care? Does your organization's appetite for risk change if we change the haystack metaphor from a sewing needle to a used hypodermic needle from an adjacent clinic? Thanks, but no thanks. Ignorance is not bliss and we want certainty at the storage layer for our databases, not chance and luck.

> One of the other interesting findings from the CMU/FB flash memory paper was contained in section 4, notably the period of early failures on a 720GB SSD starts to wane around 3TB of written data. Taking RAID into consideration and only 8x SSDs per host that would mean the current generation of SSDs should have a burn-in procedure write out just over 24TB worth of data before putting the host into production. As the technology becomes more robust, I'm not sure this advice will remain accurate.

To put some context on the operating circumstances, Groupon's core business is to provide a commerce experience which delights our customers. The business is such that we require operational efficiency, reliability, and correctness. In the previous 12 months ending in Q1 2015, Groupon had $6.3B in gross billings. In Q1 of 2015 alone we sold 54M units (units reflect vouchers and products sold before cancellations and refunds), and have cumulatively sold over 800M units as of Q1 2015. At the end of Q1, we averaged over 160M monthly unique visitors, and better than 80% of our customers return to the site to make future purchases (ForeSee Groupon Customer Satisfaction Study, March 2015—commissioned by Groupon).

A significant portion of that business was going to land on this new platform so correctness wasn't something we could be uncertain about. We needed to plan for the future and accept that when handling large amounts of data, whatever could theoretically happen, will happen, and will happen at a predictable rate. To guarantee the fidelity of information on

petabytes of data across arbitrary applications that may or may not have the ability to checksum their own data, we needed some way of mitigating the identified risks from SSDs.

The solution and outcome should be obvious at this point, but the unexpected benefits along the way—both technical and organizational—have been numerous and remarkable.

## FreeBSD to the Rescue

There are many technical problems in the world, but many of the actual hard problems aren't technical so much as they are organizational. The primary problem we needed to overcome was the ability to run arbitrary applications that had no capabilities to checksum their own data at rest, and this technical problem required a technical solution. In 2014 Groupon was a Linux-dominant shop. After much cantankerous wrangling, discussion, review, and testing of alternatives, we settled on FreeBSD with ZFS as the technical solution to the problem.

In the world of mature operating systems with a great track record in production, it's hard to beat FreeBSD. Yet after we settled on and committed to FreeBSD as the solution to our imminent and burgeoning bit error rate ("bit rot") problem, the next question was, "But will it blend?" We had secured a means of solving our bit rot problem, but we weren't sure how the rest of the organization would adapt to an OS that wasn't Linux.

Technically what does it mean to change or grow support for an additional operating system? It turns out, not that much if you're set up to handle multiple distributions of Linux. "So this project is going to support current growth, integrate previous acquisitions, plan for future application and product launches, AND change operating systems for the database tier?" "Yup, because the change isn't that significant." Sure, it requires some investment up front, but the analogy is closer to owning a Mercedes and buying a BMW and learning to drive the new car effectively. The shifter is in a different place, the radio is different, but both are fantastic pieces of engineering, require gas, and get you to your destination comfortably and in style.

## No `fsck`'s Given

DTrace support, along with the ability to easily roll a custom kernel and use of `poudriere(1)` were some of the initial big ticket items that showed up in the pro column when making the decision. Without much fanfare, however, the lack of file system checks `fsck(1)` showed up as a strength, too, and now it's one of the bigger items that no one misses or even talks about anymore. Imagine running a `fsck(1)` on a near line backup server with 288TB of storage. A traditional block-oriented file system would have taken days or weeks to complete. With ZFS? Zippo. Literally, no time. Such activities and concerns are now anachronistic in our modern environments and no one has waxed poetic about their absence. Though at the time ZFS was introduced, it was a question that frequently came up because file system checks are one of the tools administrators can use if they get into trouble. Taking away a utility as ingrained as `fsck(1)` highlighted one of the challenges of organizational change: personal anxiety from either learning something new, or trusting something new in production needs to be considered.

During the lead up to committing to this direction, it was clear there was some apprehension with some of the database administration team. FreeBSD isn't Linux, therefore it's different, but how different? Supporting a new OS isn't necessarily about making sure all programs compile and run. Is FreeBSD different in a material way? Is the transition going to be an overwhelming experience? What is the level of effort required for each admin to become familiar with the new OS? When will the uncertainty or doubts subside?

The answer in our case was to conduct a series of 30-minute, online tutorials and open mic question-and-answer sessions. In order to meet the scheduling needs of a worldwide team, we scheduled video conference calls every few days at 6:30 a.m. PDT to meet and go over some aspect of FreeBSD. To keep the sessions open for candid conversation and focused on the learning needs of each person, we broke out and grouped attendance based on skill level and identified personality traits that kept the open mic Q&A productive and bidirectional for everyone.

The first session was as simple as logging in via `ssh(1)`, logging in to MySQL and PostgreSQL, running a few basic observability commands such as `top(8)`, `iostat(8)`, `vmstat(8)`, and a few new commands such as `systat(8)` and `gstat(8)`. The overwhelming response was, "This isn't different from what we're used to." To finish, we revisited `` `top -m io` `` as a "one-more-thing" moment, and that yielded a number of, "OH! You mean I don't have to XYZ anymore?" The unknown had become exciting and now helpful to everyone's day-to-day activities.

Sessions after that moved quickly and covered things like the base system versus `ports(7)`, using `pkg(1)`, compiling a port by hand, and the difference between `/etc` and `/usr/local/etc`. Shortly after, we went in to some of the more murky waters like `pf(4)` and `ipfw(8)`, and the history for why there is more than one firewall in FreeBSD. During a few sessions a little DTrace-based observability trickled out here and there in order to show some latency quantiles, counts, stack traces, etc. The intention wasn't to teach DTrace now so much as give perspective and instill the understanding that "if all else fails, there's always DTrace." And if it does come to using DTrace in production to identify a problem, chances are you're not having the least stressful day of your life and other people will be around helping out.

It wasn't until around the fourth session that we introduced ZFS. In that seminar we covered snapshots, cloning, `zpool scrub`, and rolling back. To punctuate one of the more profound uses of ZFS, we had also replicated a small 1TB database in advance. During the Q&A, we took a ZFS snapshot without shutting the process down, trashed 600GB worth of data with a mix of faux-inept `UPDATE`s, `DELETE`s, and `DROP TABLE`s. We then shutdown the database and rolled back to a previous snapshot in about 30 seconds. When we brought the process back up, it had about 5 minutes worth of replication lag to catch up on. Compared to other snapshot technologies which incur a performance hit when taking them, ZFS was like magic.

***TIP:*** Use DTrace during a `'zfs snapshot'` to measure VFS write latencies. Wow people with write latencies measured in single-digit microseconds (not milliseconds) (**see Box 1**).

Not bad for spinning rust. Did someone forget to mention ZFS can also be really, really fast?

Everyone's personal rate of adoption and rate at which they address the anxieties from using something new is different, but there's clearly a trajectory for anxiety where it moves from the dread or excitement of learning something new (frequently both), to some feeling of unease due to a lack of knowledge or experience, acceptance and embracing the new, and eventually ends when the day-to-day activities that were semi-unsettling become natural and comfortable. Changing technology is not entirely about the technology, it's about the process of working with people and building the necessary familiarity required to scale a technology in an organization.

***TIP:*** Do benchmarking and other high-glitz testing to prove systems are not fragile while working to buttress confidence and combat anxieties. For instance, take a random write workload from a Linux host with a Fusion IO card and move it to FreeBSD, ZFS, and give the zpool only 16x spinning disks. Suggestion: repoint a stream of >100K ganglia metrics being written to RRD files (**see Box 2 next page**).

Within a week most everyone's personal Rosetta Stone for translating between FreeBSD and Linux was 50% complete, and in another two weeks it was closer to 80%. Everyone became pretty confident their muscle memory would work equally well under pressure on FreeBSD as well as on Linux. As it turned out, FreeBSD is a fantastic crossover platform for picking up ZFS and DTrace without having to mentally remap basic commands like `top(1)` to `prstat(1)`. The lack of friction for the basics is one of the things that made this transition possible. The oddball question about file name limits for includes in `pf.conf(5)`, or other elements of the stack get escalated up, but they're usually a by-product of making aggressive use of the various functionality in FreeBSD.

Coincidentally the start of this project coincided with a rash of OpenSSL vulnerabilities, so the

1

```
# dtrace -s vfs-io-postgres.d
Latencies (ns)
postgres Write

       value       ------------- Distribution -------------  count
        1024 |                                               0
        2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1325
        4096 |@@@@@@                                         267
        8192 |@@                                             72
       16384 |                                               0
       32768 |                                               0
       65536 |                                               19
      131072 |                                               2
      262144 |                                               0
```

```
# zpool iostat tank 1
                capacity     operations    bandwidth
pool    alloc   free   read  write   read  write
-----   -----   -----  -----  -----  -----  -----
tank     958G  9.94T      0   210K   1022   330M
tank     958G  9.94T      1   207K  4.99K   326M
tank     958G  9.94T     32  30.5K  79.9K  46.9M
tank     958G  9.94T     22  9.62K   202K  15.9M
tank     958G  9.94T     15  10.2K   169K  16.5M
tank     958G  9.94T     36  10.5K   198K  14.9M
tank     958G  9.94T      6  10.8K  39.4K  17.4M
tank     958G  9.94T     12   189K   209K   298M
tank     958G  9.94T      1   210K  7.96K   340M
tank     958G  9.94T     10   218K  23.0K   355M
tank     958G  9.94T      2   224K  4.49K   359M
tank     958G  9.94T      6   228K  12.5K   367M
tank     958G  9.94T      7   140K  53.4K   225M
tank     958G  9.94T      9  26.9K  40.9K  44.0M
tank     958G  9.94T      0  9.43K      0  13.9M
tank     958G  9.94T      0  9.69K      0  16.3M
tank     958G  9.94T      1  74.0K  3.49K   120M
tank     958G  9.94T      6   226K  17.0K   366M
tank     958G  9.94T      0   225K      0   385M
tank     958G  9.94T      0   176K      0   515M
tank     958G  9.94T      0  84.7K      0   382M
tank     958G  9.94T      0  39.6K      0   163M
```

top "just in case." Statements like "Whoa, we couldn't do that before" were routine for a period of time; then everything settled down to a steady state of productivity and enhanced use of the platform. We actually found a good, sanctioned use of ZFS dedupe (versioning historical packages), using `carp(4)` on top of `vlan(4)` tagged LACP `lagg(4)` interfaces became "the way," etc. With `lz4` compression we had more than doubled our effective storage capacity per server (an important thing when putting large numbers of SSDs into production), yet months after the initial testing, deployment, and adoption, we still hadn't seen the boogie man of bit rot.

2

ability to build and deploy updated packages quickly became one of the more welcoming features. In a little over 40 minutes a full rebuild of every package in use, with our own custom patches, was made available for installation worldwide. Twiddle a few knobs in a half-dozen files, and you could integrate custom in-house software, each package built in an isolated clean room `jail(8)`, complete with a good chain of trust all the way back to the original package

## Waiting in the Dark

The vision of the database project was this: build a database platform robust to bit rot that the database team could administrate. The second half of that objective had been achieved, but it wasn't until a few months after the first set of systems were provisioned that we had our first catch. Ever since, every few weeks we see ZFS

3

```
$ zpool status tank | head -n 3
  pool: tank
 state: ONLINE
  scan: scrub repaired 4.50K in 53h44m with 0 errors on Tue May 26 21:36:26 2015
```

author. This was an off-the-shelf win that came by simply deploying `poudriere(1)`, something we completed in a matter of days. Ever since, we only spend a few hours every month maintaining a full release and this agility has been noticeable. As one engineer put it, "`poudriere(1)` is a thing of beauty, something to be admired." The package building infrastructure sets FreeBSD apart compared to other maintained platforms.

Culturally we'd rounded the corner with the flow and ebb of anxiety, people had adjusted and were now flourishing. General concern for FreeBSD in production had waned and we no longer felt compelled to travel places with a laptop "just in case." Statements like "Whoa, we couldn't do that before" were routine for a period of time; then everything settled down to a steady state of productivity and enhanced use of the platform. We actually found a good, sanctioned use of ZFS dedupe (versioning historical packages), using `carp(4)` on top of `vlan(4)` tagged LACP `lagg(4)` interfaces became "the way," etc. With `lz4` compression we had more than doubled our effective storage capacity per server (an important thing when putting large numbers of SSDs into production), yet months after the initial testing, deployment, and adoption, we still hadn't seen the boogie man of bit rot.

detecting and repairing errors (**see Box 3**). Our first "catch" happened months and months earlier and every time someone notices, we get to say, "So that just happened," and move along with life without caring. 4.5K repaired? Probably a single bit flipped. 45M repaired? A vendor's SSD controller probably repeatedly crashed between scrubs and wrote out a reasonable amount of bogus data. We can do all of this because ZFS can detect corrupted bits and self-heal for well-constructed zpools where enough redundancy exists. Automatically. At runtime. Transparent to the application.

We do scrub zpools, and some take ages to

```
pool: tank
state: ONLINE
 scan: scrub in progress since Mon Jun 14 04:30:54 2015
       6.14T scanned out of 108T at 16.8M/s, (scan is slow, no estimated time)
       0 repaired, 5.71% done
```

4

complete. Big bricks take weeks to complete (**see Box 4**). And others with SSDs still take a moderate amount of time to run (**see Box 5**).

There is no mission-accomplished moment for this effort. It's just a vigilant, constant process of iteration, reaction, optimizing for the future, and increasing the rate at which we can iterate and improve the current situation. But does it mean we've been successful in mitigating the technical risks? Yes. A resounding yes. The rate of iteration started slowly as all of the tooling took hold, but now that we're on to our third, arguably fourth iteration, the duration between iterations is getting smaller and the improvements per iteration are increasing. On the technical side we're using `poudriere(1)` to great effect, deploying databases using `iocage(8)`, moving to DNS-based failover mechanisms which will scale better than prior layer 2 efforts using `carp(4)`.

And that "BSD issue" from earlier? It turned out to have been an issue with the service check script, kinda. FreeBSD doesn't require you to troll through `` `netstat -s` `` to find the overflow counters; instead, the kernel emits a helpful error message that informs administrators the `listen(2)` queue of un-`accept(2)`'ed filedescriptors had overflown. During development Bob had stuck a 60 second `sleep(3)` call in the request handler to test something out, which is fine, except the load balancer was hammering away on the service checker daemon every second, and with a `listen(2)` backlog of eight, it

5

```
% zpool status tank
  pool: tank
 state: ONLINE
  scan: scrub in progress since Tue Jun 16 15:52:26 2015
        1.58G scanned out of 1.17T at 5.56M/s, 61h9m to go
        0 repaired, 0.13% done
```

took exactly eight seconds before the kernel started blasting out informational messages. The fix was easy and took 20 minutes to add: mixin a multithreaded socket handler and shuffle around a few method calls to allow multiple threads to participate in `accept(2)`'ing new connections off of the `socket(2)`. Sixty minutes from open to resolved. An easy problem that didn't actually require escalation, but a subtle difference and a reminder of the consequence of a dominant monoculture mindset.

Introducing technical changes to an organization requires successfully navigating the necessary personnel change elements. Taking into consideration the needs of individuals who will be long term participants or owners of the resulting technical change should be a primary objective of the entire change process. People going through this change process will transition through a series of phases before becoming fully aligned with the change. This is especially true when not all beneficiaries were participants in the research or decision making process (this frequently happens in large organizations). The model we worked from was centered around addressing the announcement and introduction, personal anxieties, building technical and organizational acceptance, followed by achieving steady productivity and operational gains. Consider bringing in external training to fill gaps in understanding (many of which may transcend the particulars of a specific OS, we did that for a broader audience and it worked out well - thank you Rich and Dru), set up small-group seminars, and even share "war stories." Once the organizational changes were considered and taken into account, the fruits of the technical change were compelling for us and the extra tool in the toolbox has proven to be valuable and has mitigated the risks identified in our planning assumptions (**see Box 6**). Being able to do performance monitoring for IO in the microsecond level and having automatic self-healing from bit rot? Yes, please. Oh, and sleep well. We do. ●

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option bufsize=8m
#pragma D option switchrate=10hz
#pragma D option dynvarsize=16m

/* See /usr/src/sys/kern/uipc_mqueue.c for vop_read_args.
 * Also see sys/uio.h.
 */

dtrace:::BEGIN
{
        i = 60;
}

profile:::tick-1sec
/i > 0/
{
        i--;
}

profile:::tick-1sec
/i == 0/
{
        exit(0);
}

vfs::vop_read:entry, vfs::vop_write:entry
{
        self->ts[stackdepth] = timestamp;
        this->size = args[1]->a_uio->uio_resid;
        this->name = probefunc == "vop_read" ? "read" : "write";
        @iosize1[execname, this->name] = quantize(this->size);
}

vfs::vop_read:return, vfs::vop_write:return
/this->ts = self->ts[stackdepth]/
{
        this->name = probefunc == "vop_read" ? "read" : "write";
        @lat1[execname, this->name] = quantize(timestamp - this->ts);
        self->ts[stackdepth] = 0;
}

profile:::tick-15sec
{
        printf("--- Tick 15 ------------------------------\n\n");
        printf("Latencies (ns)\n\n");
        printa("%s %s latency (ns)\n%@d\n", @lat1);
        printf("IO sizes (bytes)\n\n");
        printa("%s %s bytes\n%@d\n", @iosize1);
        printf("------------------------------------------\n\n");
        trunc(@lat1);
        trunc(@iosize1);
}
```

6

SEAN CHITTENDEN is an Architect for Groupon Production Operations (seanc@groupon.com). He is a long-time participant of the FreeBSD (seanc@FreeBSD.org) and PostgreSQL communities, a 15+ year veteran of large scale web infrastructure including databases, networking, and storage. In a prior life he owned and ran a datacenter and technology reseller whose clients included Facebook and Yammer before becoming a ghost in the machine among Silicon Valley startups.