SEE TEXT ONLY

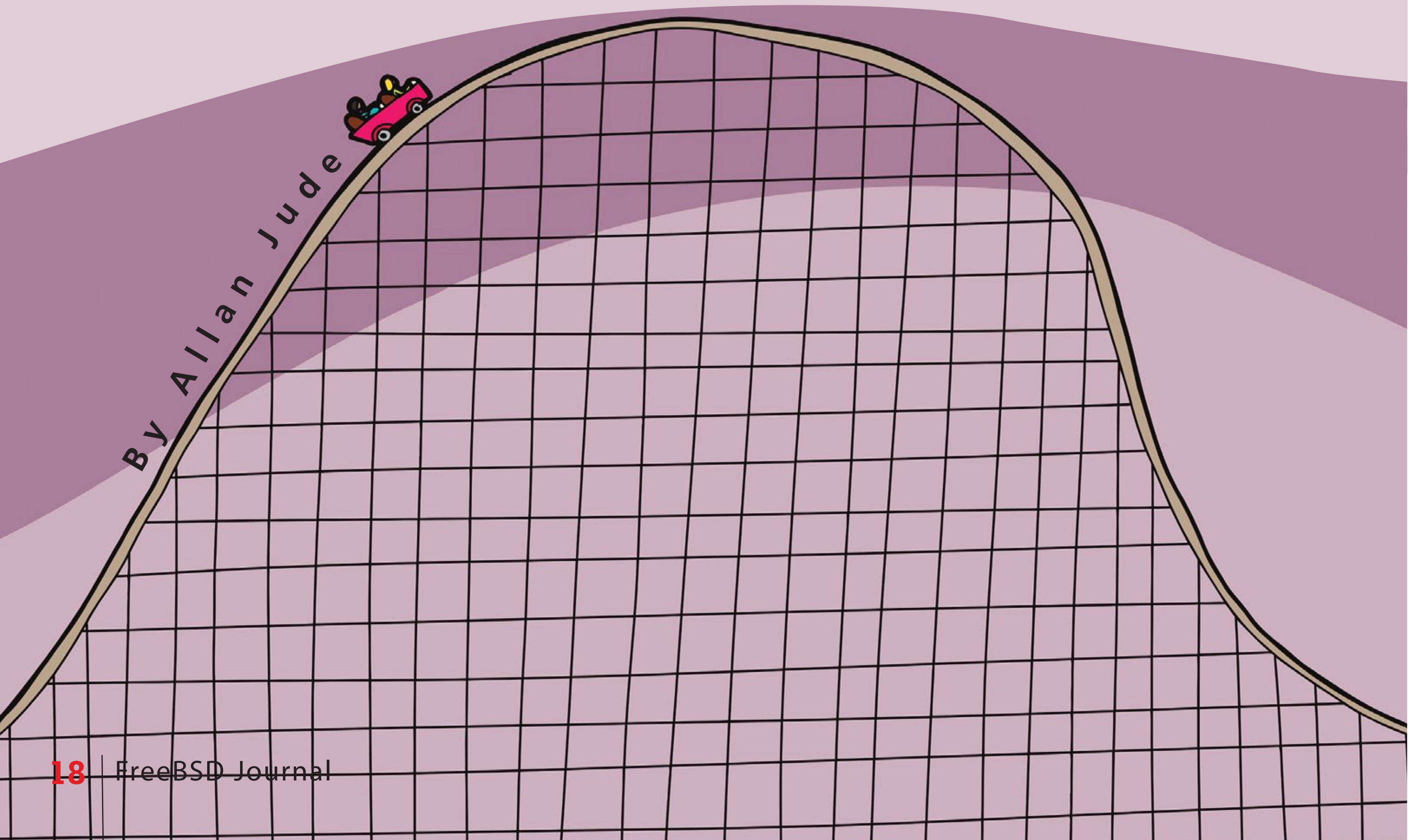
ScaleEngine and FreeBSD

When ScaleEngine was founded, it was based on FreeBSD because that is what the principals were most familiar with. Throughout the past eight years, the decision to use FreeBSD has served us well and provided us with distinct competitive advantages.

How ScaleEngine Started

ScaleEngine started its CDN (Content Distribution Network) entirely by accident. Originally, ScaleEngine was an auto-scaling application environment designed to host forums, large web apps, and other high-complexity high-traffic sites. After a short time, we quickly started to push too much bandwidth out of our primary colocation facility. In working around this problem, we quickly grew into a CDN and eventually pivoted our business to focus on that market.

Bandwidth in most colocation, and in Internet transit in general, is measured at "95th percentile." The basic concept is that the amount of traffic pushed to the Internet is measured every 5 minutes throughout the month. At the end of the month, the measurements are sorted, and the top 5% are discarded. The peak amount of usage that remains determines your bill. Usually there are two prices involved. The customer has a commitment level—a minimum amount of bandwidth they buy each month—at a fixed price.



By Allan Jude

Then there is a burst or overage price. The customer is able to use more than the committed amount of bandwidth, but if their 95th percentile exceeds the commitment, they must pay for the additional usage at the higher burst price. This has some benefit to both sides. For the transit providers, it ensures that customers pay for the amount of bandwidth they sustain during peak times, which ensures the transit provider can keep that amount of capacity available and encourages customers to commit to a larger amount of bandwidth to avoid the higher overage rate. For the customers, it offers them the flexibility to use more bandwidth when needed while only having to pay for the committed rate. The customer also has the ability to use the entire capacity of the connection for up to 5% of the month (approximately 36 hours), without any additional cost. The busiest hour of each day or the busiest 90 minutes of each weekday do not count against your final bandwidth bill.

ScaleEngine was in a position where we did not want to increase our bandwidth commitment, but also needed to avoid paying expensive overage fees each month. The majority of the traffic that was consuming the bandwidth was large image files we hosted for various sites, including a screenshot-sharing application. To offset this, we rented a server on the east and west coast of the U.S. Bandwidth for these servers was priced differently, based on total monthly volume, rather than peak usage. We directed traffic for static content—mostly the larger images—to these servers, which received the content via rsync from our primary site. This reduced the bandwidth consumption of our primary colocation to well within our commitment, since the majority of the traffic was now the plain text content of the sites we hosted. Demand for faster-loading images saw us add similar servers in Europe.

Dummynet: Smart Traffic Shaping

From the customer side, the downside to 95th percentile billing is that if your peak time during a month exceeds that free 5%, then you pay for that peak amount of traffic as if you had used that rate for the entire month. This means that there is often an advantage to smoothing out spikes in your usage. This is where Dummynet comes in. Dummynet is the traffic-shaping feature of IPFW, FreeBSD's native firewall. Using a pair of pipes and some queues, we were able to rate-limit replication traffic between our servers such that a large number of new files would no longer create a spike in traffic. Instead of being replicated at full speed, saturating our connection, and creating a large spike, replication would be rate-limited by a pipe and would yield to higher-priority customer traffic in the queue. This flattening of the spike saved thousands of dollars while only causing a minor replication delay.

For more on IPFW and Dummynet, see the May/June 2014 edition of the *FreeBSD Journal.*

Reliability and Flexibility

When you have to manage more than 100 servers, it helps to have a stable and reliable OS. With our server lifecycle we have a mix of FreeBSD versions, 9.3, 10.3, and -CURRENT. Having painless upgrades inside a stable branch (10.2 to 10.3) is a huge benefit, and we can quickly upgrade to the latest version without worrying about fallout in our application stack.

FreeBSD's approach to third-party libraries and applications has been an important part of our ability to deploy new features. The FreeBSD ports system is a rolling release, meaning that new applications and updated versions of those applications are added daily. The ports system also allows us to not be at the mercy of the vendor-provided compile-time options. We use `poudriere(8)` to compile our own versions of the set of packages we use, with the options that suit us. However, if we need something else, especially just temporarily, we can fall back to the packages provided by FreeBSD. The fact that the same ports tree works on all three supported branches of FreeBSD means we can have the latest version of nginx on every server, be it 9.3 or the latest -CURRENT. Being able to get the latest version of nginx rather than what was the latest version when FreeBSD 9.0 was released, means we can rapidly adopt new upstream features. The FreeBSD ports system provides another important feature: multiple versions of the same application. We can choose between PHP 5.6 and PHP 7.0, or nginx and nginx-devel. A quarterly stable branch of the ports tree is available for those who wish to avoid the daily churn.

How FreeBSD and ZFS Made the Difference

When we started selling CDN services directly rather than as part of a hosting service, the architecture needed an overhaul. We had a number of issues that needed addressing, but FreeBSD provided a solution to each one of them.

We migrated away from rsync, which was too slow once the number of files grew beyond a few tens of thousands, and we needed to be smarter about which content we cached on each different edge server to avoid wasting precious storage space. We also needed to avoid wasting bandwidth replicating data that was rarely requested after it was a few weeks old. Again, the goal was to reduce the bandwidth pressure from our primary storage servers, to avoid having to buy additional expensive Internet transit. We switched to the nginx caching module, which saves files to the edge server the first time they are requested. This worked well at first, using hash-based directories and filenames to break the large number of files into manageably sized directories. Eventually we encountered performance problems with UFS, the default FreeBSD filesystem. The "dirhash," a cache of directory metadata, was limited by default to only a few megabytes of memory. Even when the size of the cache was expanded and the time-to-live increased, walking through the directories was painfully slow, which adversely affected cold startup times. Switching to ZFS changed everything.

ZFS has a much smarter caching system. Most filesystems use a standard LRU (Least Recently Used) cache, which, when full, removes the item used the longest time ago to make room for a new item. ZFS, instead, uses the ARC (Adaptive Replacement Cache), which consists of four lists. The first, the MRU (Most Recently Used), is very similar to LRU. There is also a separate MFU (Most Frequently Used) cache for files used most often. This provides an important optimization: when walking through the entire directory structure of cached objects, which can cause the entire LRU/MRU to be cycled, all of the items that were in the cache are removed and replaced by the entries being walked. Now the cache is full of items we likely will not use again, and we have to wait for the cache to recover to provide a reasonable performance boost. With the addition of the MFU, the files we use most frequently are not purged from the cache by the directory walk. Now we also had control over how much of the filesystem cache could be used for metadata. Unlike UFS, ZFS will only purge items from the metadata cache if it is full or if the memory is needed elsewhere. ZFS limits the metadata to 25% of the total cache size by default, but this can be adjusted if needed. In one of our use cases, hosting the thumbnail images for a very popular related content plugin, we were storing more than 30 million small files. The files were stored on an array of SSDs, so throughput was not an issue. ZFS allowed us to cache only the metadata in ram so that the entire directory structure would reside in ram, and only reading actual data blocks would result in reads going to the SSDs. This greatly lowered latency on cold files, compared to caching a mix of data and metadata for hot files and nothing for cold ones.

ZFS also brought with it a number of other options and features we could use and provide to our customers. One of the first lessons we quickly learned was to create a dataset for each customer. When we had just one dataset with a directory for each customer's video files, with a full regimen of snapshots, if a customer canceled their account we couldn't recoup the space without removing the snapshots. This would cause us to lose the history for every other customer in the process. Once we transitioned to a dataset per customer, we could remove snapshots or the entire filesystem for a customer and get the space back immediately. This also allowed us to create reservations and quotas for customers, to ensure they always had enough space, or to limit the amount of space they used.

ZFS also brought with it the final nail in rsync's coffin, block-level replication. Now we can replicate entire filesystems to our edge servers as an atomic operation. Doing an incremental update only takes the time required to transfer the actual blocks that have been added or updated; there is no "scan" time or directory walking involved. When dealing with 30 million small files, the time savings over rsync are astronomical.

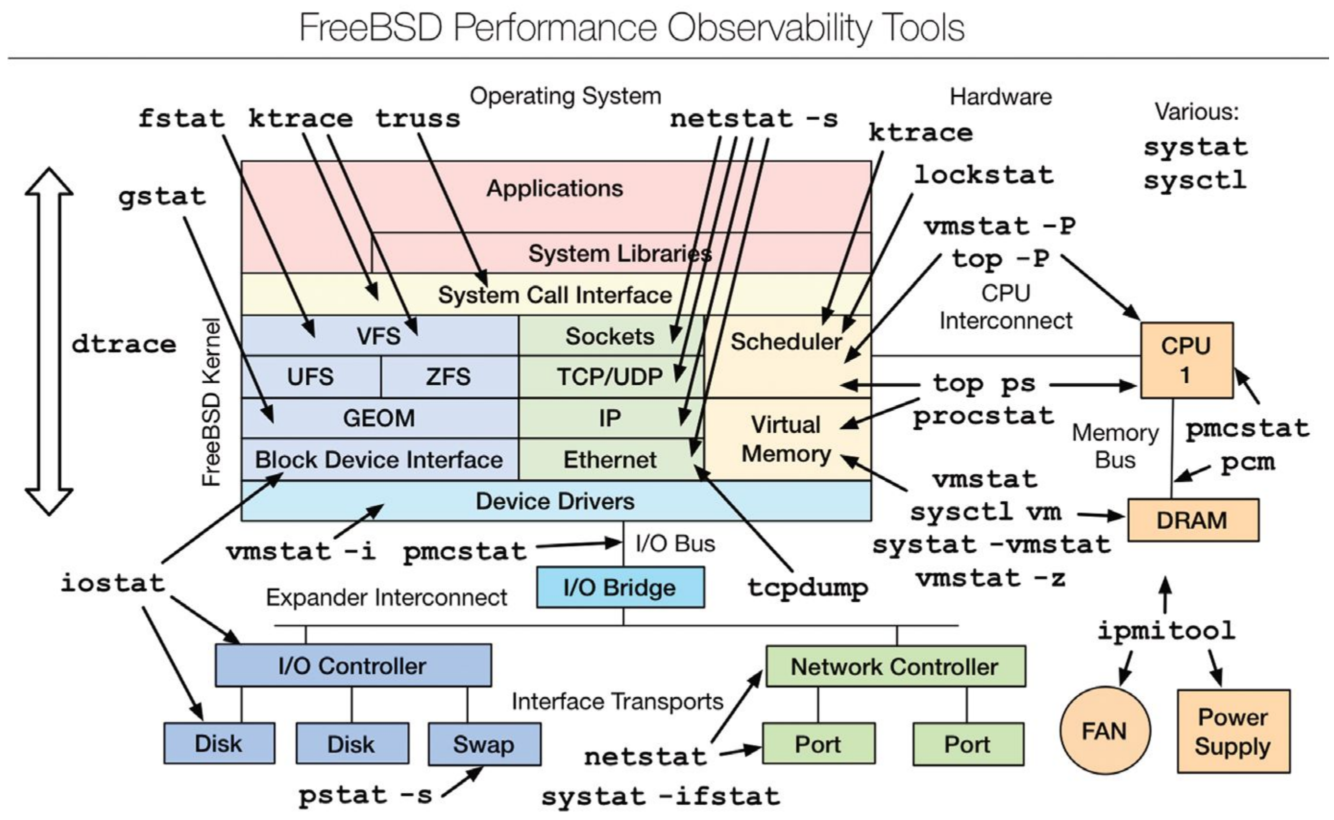
ScaleEngine hosts the package repository for PC-BSD using ZFS replication. A dataset was created on one of our storage servers and the PC-BSD team uploads the latest packages into a new directory. As they are uploading, our server takes snapshots every 15 minutes and replicates those to an array of edge servers. Once the upload is complete, the directory is moved in place of the old repository and the edges atomically update. Combined with our Global Server Load Balancer (GSLB), which stops routing traffic to an edge if its replication is delayed, this strategy has worked exceedingly well. Now that ZFS has support for resuming interrupted replication and the PC-BSD project has upgraded the connection of its build servers, we will start experimenting with having PC-BSD push a ZFS dataset directly to us rather than transferring the files individually.

Observability

Any good system administrator should be like a petulant 5-year-old asking "why" in a recursive loop. FreeBSD provides the tooling required to answer such questions, which goes a long way to being able to solve problems. The `top(1)` command in FreeBSD is very powerful while at the same time not consuming a large amount of resources itself, which would limit its usefulness for measuring the load on the system. It also has an I/O mode which can answer the question: which process(es) are causing all of this I/O? Then `fstat(1)` can tell which files that process has open. For a broader view, `gstat(8)` shows the I/O broken down by GEOM, which corresponds to disks and partitions. If throughput is not as high as expected, look at the number of IOPS being consumed. Does the device have any IOPS remaining with which to service your requests? The average latency measure can tell you if a disk is struggling to keep up with the load. A higher than expected latency under modest load can suggest a failing disk.

The excellent diagram below by Brendan Gregg (http://www.brendangregg.com/Perf/freebsd_observability_tools.png) illustrates the various parts of a FreeBSD system, and which tools can be used to observe them.

Dtrace has been invaluable in helping us track down issues in our application stack and follow them through the kernel. Whether you want to



watch the congestion window of a specific TCP connection or generate a graph of the average write latency to your disk array, it is just a matter of a few lines of D code. We used a series of DTrace probes to monitor a server during a local ZFS replication–duplication 10-TB of customer data to a second dataset. By looking at how long it took to flush all dirty data to the pool, the "sync latency," we were able to optimize the settings and achieve a 25% performance gain. By adjusting the maximum amount of dirty data, the dirty data threshold (the buffer is nearing full, so a sync is started), and the transaction timeout, we were able to delay the sync cycle (which suspends reads to reduce write latency) to only happen once 24 GB of data was waiting to be written, which would take around 5 seconds to write. The default tuning meant that a write happened every time 4 GB of data was dirty, causing the pool to frequently switch between reading and writing.

Networking

Advancements in the FreeBSD network stack have been extremely helpful to us. The pluggable TCP congestion-control algorithm system allowed us to experiment with different algorithms and determine that HTCP provided the best performance for trans-Atlantic ZFS replication. In cases like ours, often called long, fat networks, fluctuations in latency are not an indicator of congestion, and minor packet loss is not unexpected. HTCP is more aggressive than the default New Reno algorithm and gets up to speed faster and recovers from packet loss more quickly.

Support for modifying the default, initial-congestion window (RFC6928) was required to allow us to remain competitive with the small-file performance of the large competing CDNs. This setting limits the number of segments of data that can be sent before receiving acknowledgment from the other side. The initial value before the experimental RFC was between 2 and 4 segments and was changed to 10. This greatly reduced the load time for small HTTP objects, as the total number of round-trips would drop by 4. Increasing the size of the initial congestion window also allows for faster recovery in the case of packet loss during the early stages of the connection. This can greatly improve performance when the cause of the packet loss is not congestion, especially in the case of wireless connections.

With FreeBSD 11, the entire network stack has become pluggable, in such a way that different applications can use a different network stack. Theoretically this would allow us to use one set of optimizations for traffic from our edges to end users, and another for traffic over our backhaul network. This could become especially important because of the different types of traffic we serve. Flash video streams are a single long-lived connection at a constant modest bitrate. To provide the best experience, the focus should be on consistency and avoiding latency and packet loss. HLS video streams are a constant series of medium-sized (~1 MB) HTTP requests. Maximum burst performance is the most desirable outcome; however, if the device is mobile, via 4G or WiFi, packet loss can be expected, but does not necessarily indicate congestion. HTTP Progressive streaming, is similar to a regular download, but done as a series of HTTP Range Requests. This method is typically used by mobile devices, desktop browsers, HTML5 players, and set-top boxes. In this case, the optimal strategy is to avoid congestion while sending the data as quickly as possible. In the final case, HTTP Downloads, like the weekly episodes of the BSDNow.tv podcast, or PC-BSD packages, the goal is to avoid contention with the more latency- and congestion-sensitive streams coming from the same server, while offering the best possible speed.

Final Thoughts

To summarize why we choose FreeBSD to power our business: the world's most respected networking stack combined with the most reliable filesystem ever developed were brought together as an actively developed but extremely stable operating system all under a liberal, copyfree license. It is no wonder more bits are pushed by FreeBSD than anything else. ●

ALLAN JUDE is VP of Operations at ScaleEngine Inc., a global HTTP and Video Streaming CDN, where he makes extensive use of ZFS and FreeBSD. He is also the host of the video podcasts BSDNow.tv (with Kris Moore) and TechSNAP.tv. He is a FreeBSD src and doc committer, and was elected to the FreeBSD Core team in the summer of 2016. Allan is the coauthor of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* with Michael W. Lucas.