Building a Resilient Private Cloud with FreeBSD

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

BY DANIEL J. BELL

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

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

Everyone sees a breach or a failure sooner or later.

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

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

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

Preparing Our Infrastructure

Evaluating when [not] to use bare metal

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

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

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

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

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

Designing our network

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

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

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

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

Host configurations

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

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

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

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

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

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

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

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

Server for durability and redundancy

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

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

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

Maintenance of Our Cloud Environment

Snapshots

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

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

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

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

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

Replication

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

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

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

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

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

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

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

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

Figure out our latest snapshot:

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

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

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

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

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

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

Migration and restoration with ZFS

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

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

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

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

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

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

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

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

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

Tips and Tricks

Migrating instances faster

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

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

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

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

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

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

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

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

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

Escaping someone else's cloud (or another hypervisor)

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

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

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

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

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

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

On our target FreeBSD host:

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

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

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

Conclusion

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

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