Monitoring FreeBSD Systems
What to (Not) Monitor

Andrew Fengler
ScaleEngine Inc.
andrew.fengler@scaleengine.com

Abstract

Operators of computer systems need to be aware of the state of their systems. As systems and networks become more intricate, the need for this information increases. This increase in complexity also leads to an increase in failure modes, often creating modes unique to the environment.

This means that any monitoring setup must be as unique as the environment it operates on if it is going to be of use. As a result, it is frequently not possible to simply run an off-the-shelf solution, and an understanding of the principles behind monitoring systems must be applied to the implementation of your solution.

This paper will cover many of the basic areas for monitoring, and how they can be applied to FreeBSD systems.

1 Introduction

Monitoring of large numbers of systems is a complex balance between being sensitive to abnormal conditions that indicate problems, and filtering out false positives. Modern operating systems offer a large number of metrics to monitor, but not all are useful, and many of the things that are useful to monitor are not immediately apparent, or are difficult to find, especially for someone new to monitoring servers. The same error can have a different meaning or cause depending on other factors on the system, and without proper monitoring, it will be difficult to track down the source.

1.1 Reasoning for Monitoring

It is important to understand why we are interested in monitoring a computer system before we implement it. A computer or the FreeBSD operating system is a very intricate thing. To monitor all aspects of the system would be an incredible undertaking. So we need to reduce what we monitor down to an achieveable quantity.

The reason for running a computer at all is because they perform useful work. Parts of the computer or operating system that is needed to perform that work are important and need to be monitored. Conversely, any part that is not needed for this work is unimportant and can be ignored. Attempting to collect and monitor unimportant information simply increases the difficulty of determining whether the system is working as intended.

1.2 An Abridged History of Monitoring Technologies

In the beginning, all observation of a computer's state was done manually by a human operator. Whether this was lights on a status panel, or messages printed to `STDERR`, this would go directly to a human's attention. Daemons that are not run interactively log any problems to `syslog`, and any log entries with a high enough error level would be shown to the operator.

This methodology has persisted in many ways. Although on most servers `TTY1` is is not watched by anyone, all log messages of level `err`[1] or higher are printed there by default in FreeBSD. Many daemons will log errors, or print them to `STDERR`, and promptly forget about them, making the error impossible to spot unless you were filtering through the log file. Although logging of errors is useful for diagnosing an issue and determining the cause, is is significantly less useful for detecting the problem in the first place.

Technologies such as `SNMP`[2] solve some of this problem by allowing information on the system's state to be procedurally retrieved by a remote system. Unfortunately this is limited to information about the operating system, as few programs have support for providing data through `SNMP`.

Many programs now offer some form of status output that can be externally retrieved. BIND dumps statistics to a file, Nginx has a status page, and Varnish offers statistics through `varnishstat`. Although this allows each program to offer information tailored to how the program operates, this has the limitation that each one is unique, and any monitoring

[1] LOG_ERR is the 4th highest level in syslog. LOG_EMERG, the highest, is normally broadcast to all users terminals

[2] SNMP - Simple Network Management Protocol

must be custom-built for the combination of your monitoring system and the program being monitored.

2 Availability

The simplest form of monitoring is availability checks; i.e. "does it work?". These tests ensure that at some higher level, the system works as intended. This could be as simple as sending a `ping` and seeing if the system responds. If the system answers the ping, then we can infer many things: the system is running, it has power, it has a working network connection, and so on. There are advantages and disadvantages to high-level tests. They simplify the monitoring by not requiring a separate check at each level, but if the check fails, the reason why might take some digging to solve.

One check of this sort is to use `netcat` to check if a service is listening on a given port. Figure 1 shows a simple check to see if a webserver is listening on port 80 on IPv4. This can be useful for testing services that don't have an easy way to do a functionality check, but this way you can at least tell that something is listening.

Often, a program can listen on multiple ports or addresses. A typical web server will be listening on ports 80 and 443, on both IPv4 and IPv6. Doing 4 seperate checks of the webserver's status page would be redundant and incur pointless overheat. If you only are checking one port, you might not notice a problem on the other ports, such as the server failing to listen on IPv6, but otherwise working normally on IPv4. A simple TCP connection check is enough to catch that failure case.

Other examples of availability checks include loading a web page, logging in with SSH, and sending an email to a monitored mailbox.

3 System Resources

A computer offers resources, and services running on the computer will use some of those resources. Ensuring that those resources are available is one of the main tasks for monitoring at the operating system level. Some of these resources include CPU time, memory, disk space, and network bandwidth.

3.1 CPU

SNMP offers 2 ways to get your CPU usage. One is to use the `UCD-SNMP-MIB::ssCpuRaw*` values to get a counter of the CPU time spent and average it on your monitoring interval. The other is to use the `UCD-SNMP-MIB::ssCPU*` values which are an integer, pre averaged over 1 minute, divided down by the number of processors, and rebased as a percentage. See figure 2.

These 2 collection methods represent 2 different types of measurement: the rolling average, and the snapshot. A rolling average shows an average usage over a given window, in this case the window is the measurement interval. Rolling averages will account for all usage in that window, but as a side effect will smooth out spikes shorter than the window size. A snapshot shows the exact level at the time of the measurement. This shows a more exact level, but only at the time of the measurement. Any spikes outside of the time of the measurement will be missed.[3]

Neither is necessarily better, both have their strengths and weaknesses. If you have a cron job running every 5 minutes that is using all available CPU for 30 seconds, and you check CPU usage every 5 minutes, a rolling average will show only 10% usage, and a snapshot might show either 100%, or 0%.

3.2 Load Average

Although it seems like the easiest thing to check, load average has a number of issues. It fails to account for multiple cores, and the distinction between cores and hyperthreads is lost on it. It varies wildly between workloads: our monitoring servers will show loadavg of 16 on a 4 core machine, with only 30% CPU usage, and video server with its CPU fully used might only show a load average of ˜8 on an 8 core machine.

3.3 Memory

We need to ensure that there is free memory available for programs that need it. However not all memory usage is equal.

[3] While the `UCD-SNMP-MIB::ssCPU*` values are strictly a rolling average over a window of 1 minute, for checks with an interval > 1 minute it is functionally a snapshot.

```
$ nc -z -4 host 80
Connection to host 80 port [tcp/http] succeeded!
```

Figure 1: A simple check with netcat to see if something is listening on port 80

```
$ snmpget -c public -v 2c server.example.com UCD-SNMP-MIB::ssCpuIdle.0
UCD-SNMP-MIB::ssCpuIdle.0 = INTEGER: 83
$ snmpget -c public -v 2c server.example.com UCD-SNMP-MIB::ssCpuRawIdle.0
UCD-SNMP-MIB::ssCpuRawIdle.0 = Counter32: 1653347551
```

Figure 2: Getting CPU data out of SNMP

FreeBSD will put unused memory to use, by using it for cache and for the ZFS ARC.[4] ARC and cache will frequently consume most free memory, but the system will free up memory being used for cache when needed. This creates a measurement problem, as memory being used for ARC is counted as wired. You can see an example of this in figure 3. To get useful numbers, you have to count the used memory minus the ARC, count the ARC and cache as memory that can become available, and the memory that is truly free.

All of the statistics from the arc and regular memory usage can be found in `sysctl: kstat.zfs.misc.arcstats`, `vm.stats`, and `vm.stats.vm.v_page_size`.[5] You can even get a count of how many times the ARC has been throttled due to memory pressure from `kstat.zfs.misc.arcstats.memory_throttle_count`

3.4 Network

Because packets are routed from network provider to network provider on their route, a problem in the middle of the route can cause connection issues. Frequently, these problems will cause packets to be lost, or to be routed on sub-optimal connections and cause the packets to take longer from source to destination. We can use ICMP between remote systems to detect loss and high latency along the route.

The status of the network connection itself deserves checking. Ethernet link speeds are typically autonegotiated, and negotiations can sometimes return a less than optimal result. Some service providers will throttle your connection speed if you're over your usage quota. In any case, it's good to know if the connection is operating at the speed we want it to be. FreeBSD's `ifconfig` will show you the media settings for an interface, which includes the speed:

```
$ ifconfig | grep media
media: Ethernet autoselect (1000baseT )
```

[4] ARC: Adaptive Replacement Cache

[5] Note that ARC stats are in bytes, and memory stats are in pages, so you need to multiply memory stats by page size to get an even comparison.

3.5 Bandwidth

Checking the network utilization is pretty straightforward with `SNMP`. `Net-SNMP` implements the IF-MIB::ifXTable, which shows statistics for each interface. We can get the number of octets[6] sent and received on the second interface in the table, as in figure 4. Usage is stored as a counter of total octets sent since system boot. If we store the value and subtract the old value from the new one, we get a delta of the total bytes sent over that time interval.

Note that we are using `ifXTable` which uses 64 bit counters. `SNMP` also has `ifTable`, which only uses 32 bit counters. A 32 bit counter of bytes will roll over at ~4 GB, which can be sent in less than 5 minutes on a 10 Gb/s interface.

For those who rent their servers, the total data transfered can also be a limited resource. Many providers limit the amount of data you can send and receive, usually some number of terrabytes per month. Since the method we've discussed for monitoring network usage measures in deltas of bytes, if we store those deltas, we can tally them up and measure usage over the interval of interest. ScaleEngine uses a program called `RTG` [1] for this, which logs the deltas to a MySQL database, where we get the data for usage information.

3.6 Disk Space

Free space on disk is important if anything needs to write to the disk. Most filesystems, ZFS included, will perform poorly if they do not have some free space.

A common way to watch your disk space is to use `SNMP` to check free space on `/`. That will not work as expected with ZFS. `Net-SNMP`'s disk checks are not ZFS aware, and see each dataset as a partition, and any data stored on a different "partition" will cause the root partition to shrink, rather than show more space used. A stock FreeBSD installation will only have ~5 GB on the root dataset, meaning the free space on the `/` "partition" won't drop below 50% free space until there is only 5 GB free for the root dataset. Since most programs will not be writing to the root dataset, this will only ensure that there is free space on that one dataset. If you set a reservation on the

[6] octets = bytes

```
Mem: 9588K Active, 103M Inact, 2786M Wired, 4992K Cache, 55M Free
ARC: 1935M Total, 603M MFU, 1202M MRU, 560K Anon, 19M Header, 111M Other
```

Figure 3: Memory usage from `top`

```
$ snmpget -c public -v 2c server IF-MIB::ifHCOutOctets.2
IF-MIB::ifHCOutOctets.2 = Counter64: 26790537050371
$ snmpget -c public -v 2c server IF-MIB::ifHCInOctets.2
IF-MIB::ifHCInOctets.2 = Counter64: 17901892810225
```

Figure 4: Getting counters for the second NIC

root dataset, that will be seen as free space, even though other datasets that do not have reservations will be out of space.

It's better to use ZFS to check ZFS. ZFS has some useful tools, and also has good output handling for automatic parsing:

```
$ zfs list -pH -o name,used,avail
mjolnir 46128820224 67251605504
mjolnir/ROOT 12187820032 67251605504
mjolnir/ROOT/default 12187729920 67251605504
mjolnir/tmp 157851648 67251605504
mjolnir/usr 29228777472 67251605504
mjolnir/usr/home 27944427520 67251605504
```

ZFS offers a number of advantages over `SNMP` for disk space checks, making it worth the exta labour to implement the checks. ZFS has per-dataset granularity, quota information is readily available, and jail usage is easy to break down. If you're only able to work with `SNMP`, then make sure you're checking on all datasets you care about, not just `/`.

3.7 Disk Health

Disks are consumables, whether they are hard disks or solid state drives. Staying on top of the health of the drives will allow you to have some chance of anticipating a failure. We can get the disk information using `smartctl`.

A study by Google [2] found that the only SMART errors that correllate to drive failure is reallocated sectors, weakly correlating for online reallocations, and more strongly for offline reallocations. SMART value 5 is online reallocation, and value 198 is offline reallocations. Value 197 is the same as 198 for most manufacturers, Toshiba being the only exception we've encountered. We can monitor these 3 values along with the overall health check using `smartctl`. Figure 5 shows an example of these values.

Also of interest is the SSD wear values, but these vary between manufacturers and we have yet to find a way to measure them that gives usable data.

SMART will not predict all failures, and there are failures external to the drive that can make it unusable. This could be a cable error, data corruption, or some other error that SMART doesn't catch. Checking zpool health is pretty simple: To see if the drive is in a working state, we can check if ZFS sees the drive as usable or not.

```
$ zpool list -o name,health
NAME     HEALTH
mjolnir  ONLINE
```

The health column shows the state of the pool, `ONLINE` is ok, other states indicate a problem.[7] If you want more details, you can also parse through `zpool status` to find the exact drive that has the problems.

4 Precursor Metrics

In addition to system resources, there are many things that while they do not directly affect the system's operation, they can cause problems in some situations.

The temperature of the system can be monitored to watch for overheating. FreeBSD does not make it obvious when thermal throttling kicks in, other than some entries in `syslog`. You can get the CPU temperatures out of `sysctl`.

```
dev.cpu.#.temperature
```

NTP is important for keeping your clock accurate. If you drift out of sync, this will cause problems with anything time-sensitive, including a lot of cryptography. You can check your offset against an NTP server with `ntpdate`. Use a server that is different than the one you're syncronizing against in order to catch problems with that server.

```
$ ntpdate -q 0.pool.ntp.org
... offset -0.007518, ...
```

While it might not seem useful, monitoring the uptime of a server can be a good way to catch unexpected reboots. Check if the uptime is less than twice the alert interval to have an alert whenever a server reboots. The advantage to using uptime rather than a cron entry for `@reboot` is that checking the uptime with `SNMP` works on switches, allowing you to catch reboots that might just apear as a brief period of packet loss otherwise.[8]

5 Jails

Jails offer a lot of challenges in obtaining meaningful statistics, especially if you care about isolating to the jail. CPU

[7] A state of OFFLINE indicates the device was taken offline. While this is not an error, it is probably not a desired state.

[8] Switches can reboot really fast, and if it happens between checks there might not be anything else to give it away

```
SMART overall-health self-assessment test result: PASSED
  5 Retired_Block_Count     0x0033   100   100   003    Pre-fail  Always       -       0
197 Current_Pending_Sector  0x0022   100   100   000    Old_age   Always       -       0
198 Offline_Uncorrectable   0x0008   100   100   000    Old_age   Offline      -       0
```

Figure 5: SMART values of interest

usage information will include usage from processes outside the jail. Isolated network metrics are only possible if you are using vnet, which only became the default in FreeBSD 12. If you want to run `SNMP` in the jail, you have to build `Net-SNMP` with special options so it works at all. Since much of the resource usage information will only be available on the host, make sure you can corellate services in jails to the host they are running on to be able to examine the state of resource availability for that jail.

6 Different Types of Measurements

Regardless of the source of the measurement, the data that can be retrieved will fit into 2 broad categories.

State measurements are of something that fits into a limited set of states. This could be a service that is up or down, whether a connection succeeds or fails, or an HTTP status code. Each state is typically tied to some stable meaning: a degraded zpool is a degraded zpool, whether it's on a supercomputer or your Raspberry Pi.

Metrics are a measurement in some range of possible values. CPU usage, ICMP RTA[9] times, and a count of logged in users are all examples of this. The administrator's judgement is required to establish thresholds for mapping values in this range to a stable interpretation of the state it represents. These thresholds will frequently vary from system to system. 600 Mb/s of network traffic might be unremarkable on a router, but could be worrisome on a server that's supposed to only be serving DNS. Tools such as graphs are useful to track historical values, as the definition of typical may change over time and as the system itself grows.

7 Conclusions

While the information shown here is by no means exhaustive, this paper covers a selection of key methods and services for FreeBSD systems in a server environment. Each environment is unique, and this paper was based off of only one. However, the basic principles will remain the same, and the techniques here can be applied to far more than what the examples hold.

Availability

Most of the monitoring described here has been implemented by the author, and is available on ScaleEngine's Github: https://github.com/scaleengine/se-nagios-plugins

Acknowledgments

Thanks to every sysadmin who solved a problem they faced, or wrote some script, and provided their work to the world. The author has been saved many wheel reinventions by simple blog posts.

Thanks to Allan Jude, for humoring my many experiments that never panned out.

[9] Round trip average, or the average time it takes a packet to travel to the other system and back

References

[1] Robert Beverly. Rtg. http://rtg.sourceforge.net/.

[2] Eduardo Pinheiro, Wolf-Deitrich Weber, and Luiz Andre Barroso. Failure trends in a large disk drive population. In *USENIX Conference on File and Storage Technologies (FAST'07)*, 2007. https://research.google.com/archive/disk_failures.pdf.