How BCC helped change how I approach debugging

Brendan Good

2018-04-09 Mon 18:22

BCC, a series of scripts using eBPF (a packet filter turned tracer for Linux), helped change my debugging perspective from "what is my code doing" to "why is the system behaving in such a way" and then answering those questions efficiently. By not digging into the source code prematurely, I spent more time understanding the behavior of the system and, when it was necessary to dive into the source code, I had a much better idea of where to look. This prevented me from wasting my time stumbling through unfamiliar code.

In this post, I'll cover:

- The problem I worked on
- The questions I asked while I debugged it
- How I used the tools I found helpful along the way (which were mostly from BCC)

Problem description

As part of a job interview, I was given a data pipeline in a Docker environment and told to find as many things wrong with it as possible. There were a few things that needed to be addressed, but I'll focus on how I found a bottleneck in the pipeline. Since this was a part of a real job interview, I've intentionally obscured the names of some technologies used (namely the database and the transaction log).

Architecture

The architecture was fairly straightforward: A dummy service (written in Python) writes to a database and there's a loader service (written in Java) that reads from the operation log of the database and loads it into a transaction log.

$$\text{Dummy} \rightarrow \text{ Database } \rightarrow \text{Loader } \rightarrow \text{ Transaction Log}$$

Aside: Difficulties posed by Docker

It seems like Docker's default security model makes it a bit non-trivial to run BCC in a container. Indeed, since a Docker container doesn't contain its own kernel (and BPF often involves instrumenting the kernel), this seems like an understandable constraint (although such a thing is possible in principle [1]). A lot of very meaningful information can still be gleaned from running BCC tools on the host. On the one hand, that does, admittedly, make some useful scripts like profile more difficult to use. On the other hand, having the whole pipeline running on one host made it easy to access system wide metrics such as TCP traffic and latency, so it's not all doom and gloom.

Starting point

- Question: What's the TCP throughput? I started off with seeing how much traffic was going through the pipeline; even though the exact data going from one point to the next may be different (because data may be added or removed), it ended up being useful information down the line. After looking around the available tcp tools, I found tcptop to be the best tool for this particular job.

  ```
  ./tcptop -i 10
  ```

  -i 10 means show the data over a 10 second period, this made it easier to comprehend the data being displayed (the default is 1 second)

  `https://b-t-g.github.io/assets/tcp_before.png`

  Note the processes going through the loader service (the Java processes with RADDR 27017) are quite low in comparison to the dummy service (python3). This gives some early evidence that the loader service is a bottleneck.

- Question: What services are causing the most latency? I also wanted to look at the processes responsible for the most latency across the

[1] I suspect the reason you can get DTrace working with zones is because a zone is a first-class construct in Illumos whereas that is not the case with Docker and Linux.

entire system. The syscount command with the -L flag (aggregate by latency of a syscall and not the amount of times a syscall is made) and the -P flag (aggregate by processes and not syscalls), was what I needed:

```
./syscount -LP
```

The computer was running things other than just the pipeline, so the output was fairly noisy, but I did see that Python and the database were pretty high up there.

`https://b-t-g.github.io/assets/global_latency.png` Python being that high was surprising to me. To investigate further, I ran:

```
./syscount -L -i 1 -p $(pgrep python3)
```

Where the -i 1 option just means "print output every second" and -p means "for this PID only".

`https://b-t-g.github.io/assets/python_latency.png`

Why is so much time being spent in the select syscall? Looking at the arguments of select using:

```
strace -fp $(pgrep python3) -e select
```

it wasn't looking at a particular file descriptor. The linked man page has something to say about this:

> Some code calls select() with all three sets empty, nfds zero, and a non-NULL timeout as a fairly portable way to sleep with subsecond precision.

So Python is sleeping for half a second for – some reason? There were some timeouts in the source code, but tinkering with those didn't do anything. This prompted another question; is this responsible for latency downstream? Let's look at the database.

Database

Running

```
./ext4slower
```

I was able to see that the latency for syncing the log had some of the highest latency and contained most of the filesystem operations taking over 1 ms across the entire system. The exact time varied wildly; the distribution seemed to be roughly bimodal with 15 milliseconds and about 30 milliseconds being the most common. I saw syncs take as long as over 150 milliseconds, but they were atypical. The syncs were relatively consistent in the sense that they were never idle (although their latencies did vary), this did not seem like a bottleneck.

`https://b-t-g.github.io/assets/ext4slower.png`

But maybe there was latency elsewhere? Once again, relying on good 'ol

```
./syscount -L -p <PID of DB>
```

I saw that it was spending some time in select again (this time about a quarter of a second). Once again using strace to see the arguments of select, it was waiting on a few file descriptors this time. Using

```
lsof -i -a -p <PID of DB>
```

I could see that it was waiting on file descriptors associated with the dummy service. This seemed troubling at first, but being blocked on that file descriptor means that it is waiting for data, which would manifest in fewer operation log syncs. However, the operation log was syncing regularly, so I concluded that there were no latency issues with the database.

Just to make sure that the database wasn't a problem, I looked at the number of current outstanding database operations, it was constantly 5 or 6, so everything checked out here! Even though the mystery of Python's sleep was still unsolved, it did not seem like the problem I was looking for.

Loader

Now here's where things get a bit more interesting. Checking for the highest latency syscalls running this:

```
./syscount -L -p <PID of loader>
```

most of the latency was coming from epoll_wait, but about a third of a millisecond. Digging around in the source code, I could see that the loader service was polling the database's operation log. In light of this, that amount of latency makes sense since that seems like a reasonable amount of latency for a request. Indeed, it seemed like most of the workload was from reading, receiving, and send/write syscalls. This ultimately led me to strongly consider that this could be a bottleneck.

Running top, this process was consuming about 100% CPU, in an attempt to get some more information about what it was doing, I got a thread dump by logging into the container and using:

```
jstack -l <PID of loader process>
```

Unfortunately, the workload was coming from short-lived threads, so not much information was gained from it (other than learning that the loader service spawns quite a few threads).

I could still test the hypothesis that the loader was the bottleneck by turning my attention to the transaction log. Checking for high latency syscalls in the same way, I noticed that most of the latency was again coming from epoll_wait, this time, about a third of a second; this seemed to be a bit too long for me and led me to believe that the loader service is, indeed the bottleneck.

Since the pipeline was running locally and not in production, I was able to do a test to see whether it was the loader service that was unable to keep up by killing the dummy service. I saw that the tcp traffic going through the loader service was exactly the same, indicating that there was still data for the loader service to read despite no new records being added to the operation log. There is another (and safer) way to check for the same thing.

First, observe the tcp throughput for the loader service:

`https://b-t-g.github.io/assets/tcp_loader.png`

We want to know how much the operation log growing was growing over a ten second period (since I ran tcptop with the -i 10 option again) and compare that to how much data is going to the loader service over tcp. I found the location of the operation log and ran wc -c several times and noticed that the file stayed the same size. After scratching my head, I found out that the size of the operation log on disk is a fixed size after it gets to a certain size. This left the question of "What is the rate of new data going into the operation log over a 10 second period"? This time I ran:

```
./biotop
```

I could see that much more data was running through the operation log than through the loader service.

`https://b-t-g.github.io/assets/biotop.png`

Now we need to figure out what is determining how much traffic is being consumed by the loader process. We go back to the source. Conveniently for me, there was a class called DatabaseReader; perfect! Looking into the details of how it's polled, I could see that there was this batchSize variable.

`https://b-t-g.github.io/assets/batchSize.png`

Hmm, curious. Well, what does the batchSize default to? I found the config file and the default batch size is 10. That's struck me as quite low given how quickly the log was growing. Bumping that batchSize up and rebuilding the container, I could see much more data running through!

`https://b-t-g.github.io/assets/tcp_after.png` So there we have it, tracking down a bottleneck from beginning to end with the help of BCC!

Impressions about BCC and eBPF

When I first heard about BCC, I had already known about DTrace; I initially wasn't as excited about BCC and eBPF as I was when I first heard about DTrace because it's far more difficult to make new scripts. However, the pre-made scripts from the BCC-toolkit are incredibly powerful and I never really felt wanting for this project. As time goes on, who knows? Maybe bpftrace or ply will catch on or I'll make another post about how I wrote my own eBPF script![2]

[2] If you are interested in writing eBPF scripts, Julia Evans has several good posts on the subject.