Linux bcc/BPF tcplife: TCP Lifespans

30 Nov 2016

"i really wish i had a command line tool that would give me stats on TCP connection lengths on a given port"

```
# ./tcplife -D 80
PID   COMM       LADDR           LPORT RADDR           RPORT TX_KB RX_KB MS
27448 curl       100.66.11.247   54146 54.154.224.174  80        0     1 263.85
27450 curl       100.66.11.247   20618 54.154.164.22   80        0     1 243.62
27452 curl       100.66.11.247   11480 54.154.43.103   80        0     1 231.16
27454 curl       100.66.11.247   31382 54.154.15.7     80        0     1 249.95
27456 curl       100.66.11.247   33416 52.210.59.223   80        0     1 545.72
27458 curl       100.66.11.247   16406 52.30.140.35    80        0     1 222.29
27460 curl       100.66.11.247   11634 52.30.133.135   80        0     1 217.52
27462 curl       100.66.11.247   25660 52.30.126.182   80        0     1 250.81
[...]
```

That's tracing destination port 80, you can also trace local port (-L), or trace all ports (default behavior).

The quote and good idea is from Julia Evans on twitter, and I added it as a tool to the Linux BPF-based bcc open source collection.

The output of tcplife, short for TCP lifespan, shows not just the duration (MS == milliseconds) but also throughput statistics: TX_KB for Kbytes transmitted, and RX_KB for Kbytes received. It should be useful for performance and security analysis, and network debugging.

I am NOT tracing packets

The overheads can cost too much to examine every packet, especially on servers that process millions of packets per second. To do this I'm not using tcpdump, libpcap, tethereal, or any network sniffer.

So how'd I do it? There were four challenges:

1. Measuring lifespans

The current version of tcplife uses kernel dynamic tracing (kprobes) of tcp_set_state(), and looks for the duration from an early state (eg, TCP_ESTABLISHED) to TCP_CLOSE. State changes have a much lower frequency than packets, so this approach greatly reduces overhead. But it ends up tricker than it sounds, since it's tracing the Linux implementation of TCP, which isn't guaranteed to use tcp_set_state() for every state transition. But it works well enough for now, and we can add a stable tracepoint for TCP state transitions to future kernels so that this will always work. (Or, if that proves untenable, tracepoints for the creation and destruction of TCP sessions or sockets.)

2. Fetching addresses and ports

tcp_set_state() has `struct sock *sk` as an argument, and we can dig the details from that.

3. Fetching throughput statistics

Those TX_KB and RX_KB columns. This was only possible in the last couple of years thanks to the RFC-4898 additions to `struct tcp_info` in the Linux kernel: tcpi_bytes_acked and tcpi_bytes_received. I discussed these in my tcptop blog post.

4. Showing task context

It's useful to show the PID and COMM (process name) with these connections, but TCP state changes aren't guaranteed to happen in the correct task context, so we can't just fetch the currently running task information. Nor is there a cached PID and comm in `struct sock` to print out. So I'm caching the task context on TCP state changes where it's usually valid, by virtue of implementation. It works, but is another area where we can do better, and can be addressed if and when we add stable TCP tracepoints.

That's how tcplife works right now. It the future it may change and improve, especially if more TCP tracepoints become available. I'm also not the first to try this kind of TCP event tracing: Facebook already have their own BPF TCP event tool, although I think their implementation is a little different.

Here's the full USAGE message for tcplife:

```
# ./tcplife -h
usage: tcplife [-h] [-T] [-t] [-w] [-s] [-p PID] [-L LOCALPORT]
               [-D REMOTEPORT]

Trace the lifespan of TCP sessions and summarize

optional arguments:
  -h, --help            show this help message and exit
  -T, --time            include time column on output (HH:MM:SS)
  -t, --timestamp       include timestamp on output (seconds)
  -w, --wide            wide column output (fits IPv6 addresses)
  -s, --csv             comma seperated values output
  -p PID, --pid PID     trace this PID only
  -L LOCALPORT, --localport LOCALPORT
                        comma-separated list of local ports to trace.
  -D REMOTEPORT, --remoteport REMOTEPORT
                        comma-separated list of remote ports to trace.

examples:
    ./tcplife           # trace all TCP connect()s
    ./tcplife -t        # include time column (HH:MM:SS)
    ./tcplife -w        # wider colums (fit IPv6)
    ./tcplife -stT      # csv output, with times & timestamps
    ./tcplife -p 181    # only trace PID 181
    ./tcplife -L 80     # only trace local port 80
    ./tcplife -L 80,81  # only trace local ports 80 and 81
    ./tcplife -D 80     # only trace remote port 80
```

And there is more example output in bcc, along with a man page. For more about bcc and BPF (eBPF), see the collection of documents on my Linux performance page.

The only catch is that tcplife does need newer kernels (say, 4.4).

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).