Linux bcc tcptop

15 Oct 2016

I recently wrote a tcptop tool using the new Linux BPF capabilities, which summarizes top active TCP sessions:

```
# tcptop
Tracing... Output every 1 secs. Hit Ctrl-C to end

19:46:24 loadavg: 1.86 2.67 2.91 3/362 16681

PID    COMM         LADDR                 RADDR                  RX_KB  TX_KB
16648  16648        100.66.3.172:22       100.127.69.165:6684        1      0
16647  sshd         100.66.3.172:22       100.127.69.165:6684        0   2149
14374  sshd         100.66.3.172:22       100.127.69.165:25219       0      0
14458  sshd         100.66.3.172:22       100.127.69.165:7165        0      0

PID    COMM         LADDR6                           RADDR6                           RX_KB  TX_
16681  sshd         fe80::8a3:9dff:fed5:6b19:22      fe80::8a3:9dff:fed5:6b19:16606       1
16679  ssh          fe80::8a3:9dff:fed5:6b19:16606   fe80::8a3:9dff:fed5:6b19:22          1
16680  sshd         fe80::8a3:9dff:fed5:6b19:22      fe80::8a3:9dff:fed5:6b19:16606       0
```

The output has a single line system summary, then groups for IPv4 and IPv6 traffic, if present. tcptop is in the open source bcc project, along with many other BPF tools that work on the Linux 4.x series.

This information can be useful for performance analysis and general troubleshooting: who is this server talking to, and how much. You might discover unexpected traffic that can be eliminated with application changes, improving overall performance.

The current version has these options:

```
# tcptop -h
usage: tcptop [-h] [-C] [-S] [-p PID] [interval] [count]

Summarize TCP send/recv throughput by host

positional arguments:
  interval           output interval, in seconds (default 1)
  count              number of outputs

optional arguments:
  -h, --help         show this help message and exit
  -C, --noclear      don't clear the screen
  -S, --nosummary    skip system summary line
  -p PID, --pid PID  trace this PID only

examples:
    ./tcptop           # trace TCP send/recv by host
    ./tcptop -C        # don't clear the screen
    ./tcptop -p 181    # only trace PID 181
```

I'm fond of using -C, so that it prints rolling output without clearing the screen. That way I can examine it for time patterns, or copy interesting output to share with others. I'm tempted to make -C the default behavior, but instead I follow the expected screen-clearing behavior of the original top by William LeFebvre.

Other options and behavior may be added later. This current version doesn't truncate to the screen size, but should (unless -C).

Overhead

tcptop currently works by tracing send/receive at the TCP level, and summarizing session data in kernel context. The user-level bcc program wakes up each interval, fetches this summary, and prints it out.

You should be extremely cautious about anything that instruments the networking send/receive path due to the event rates possible: over 1 million per second. Reasons this bcc/BPF approach lowers overhead:

- The kernel->user transfer is the summary only, and not a dump of every packet.
- The kernel->user transfer is infrequent: once per interval.
- Tracing at the TCP level may have a lower event rate than at the packet level, due to TCP buffering (I've seen a 3x difference, although by using jumbo frames the difference can be much smaller).
- The BPF instrumentation is JIT compiled.

The overhead is relative to TCP event rate (the rate of the TCP functions traced by the program: tcp_sendmsg() and tcp_recvmsg() or tcp_cleanup_rbuf()). Due to TCP buffering, this should be lower than the packet rate. You can measure the rate of these kernel functions using funccount, also in bcc.

I tested some sample production servers and found total rates of 4k to 15k TCP events per second. The CPU overhead for tcptop at these rates would range from 0.5% to 2.0% of one CPU. Maybe your workloads have higher rates and therefore higher overhead, or, lower rates. The most extreme production case I've seen so far was a repository server under a load test of 5 Gbits/sec, where the TCP event rate was around 300k per second. I'd estimate the overhead of tcptop for that workload to be 40% of one CPU, or 1.3% overall (across 32 CPUs). Not very big, but not negligible either.

Can overhead be lowered further? I was wondering if we could add counters to struct sock or stuct tcp, when Alexei Starovoitov (BPF lead developer) suggested I look at the new tcp_info enhancements.

tcp_info and RFC-4898

Here's the latest tcp_info from include/uapi/linux/tcp.h:

```
struct tcp_info {
    __u8    tcpi_state;
[...]
    __u64   tcpi_bytes_acked;    /* RFC4898 tcpEStatsAppHCThruOctetsAcked */
    __u64   tcpi_bytes_received; /* RFC4898 tcpEStatsAppHCThruOctetsReceived */
    __u32   tcpi_segs_out;       /* RFC4898 tcpEStatsPerfSegsOut */
    __u32   tcpi_segs_in;        /* RFC4898 tcpEStatsPerfSegsIn */

    __u32   tcpi_notsent_bytes;
    __u32   tcpi_min_rtt;
    __u32   tcpi_data_segs_in;  /* RFC4898 tcpEStatsDataSegsIn */
    __u32   tcpi_data_segs_out; /* RFC4898 tcpEStatsDataSegsOut */

    __u64   tcpi_delivery_rate;
};
```

These counters are **new**, with Eric Dumazet (Google) beginning to add the RFC4898 counters in 2015, and Martin KaFai Lau (Facebook) adding more this year.

Most interesting are tcpi_bytes_acked and tcpi_bytes_received, which made it into Linux 4.1. They are printed by "ss -nti":

```
# ss -nti
State      Recv-Q Send-Q        Local Address:Port                         Peer Address:Port
ESTAB      0      0              100.66.3.172:22                          10.16.213.254:55277
     cubic wscale:5,9 rto:264 rtt:63.487/0.067 ato:48 mss:1448 cwnd:58 bytes_acked:175897
    bytes_received:15933 segs_out:618 segs_in:917 send 10.6Mbps lastsnd:176 lastrcv:196
    lastack:112 pacing_rate 21.2Mbps rcv_rtt:64 rcv_space:28960
ESTAB      0      0              100.66.3.172:22                          10.16.213.254:52066
     cubic wscale:5,9 rto:264 rtt:63.509/0.064 ato:40 mss:1448 cwnd:54 bytes_acked:47461
    bytes_received:28861 segs_out:746 segs_in:1285 send 9.8Mbps lastsnd:10732 lastrcv:10732
    lastack:10668 pacing_rate 19.7Mbps rcv_rtt:76 rcv_space:28960
[...]
```

From RFC4898:

```
   tcpEStatsAppHCThruOctetsAcked  OBJECT-TYPE
       SYNTAX          ZeroBasedCounter64
       UNITS           "octets"
       MAX-ACCESS      read-only
       STATUS          current
       DESCRIPTION
          "The number of octets for which cumulative acknowledgments
           have been received, on systems that can receive more than
           10 million bits per second.  Note that this will be the sum
           of changes in tcpEStatsAppSndUna."
       ::= { tcpEStatsAppEntry 5 }

   tcpEStatsAppHCThruOctetsReceived  OBJECT-TYPE
       SYNTAX          ZeroBasedCounter64
       UNITS           "octets"
       MAX-ACCESS      read-only
       STATUS          current
       DESCRIPTION
          "The number of octets for which cumulative acknowledgments
           have been sent, on systems that can transmit more than 10
           million bits per second.  Note that this will be the sum of
           changes in tcpEStatsAppRcvNxt."
       ::= { tcpEStatsAppEntry 8 }
```

These counters are acknowledged octets (bytes) sent and received. Couldn't we fashion a tcptop from these, that polled tcp_info similar to "ss -nti"? That way we could avoid instrumenting send & receive, potentially lowering overhead further.

There are at least two challenges with the tcp_info polling approach:

1. **Short-lived and partial sessions**: Just like how top can miss short-lived processes and those that finish before the interval snapshot, a polling approach would miss short-lived sessions. Unfortunately, tcp_info doesn't stay around for TIME-WAIT (likely as part of DoS mitigation), and applications can make many short-lived TCP connections (eg, to dependency services) that would be missed by polling. The solution would be to cache the previously polled tcp_info session state by tcptop, and also instrument the TCP close paths with BPF, including fetching out the RFC-4898 counters. That way, short-lived sessions could be seen, and sessions that closed during an interval could also be measured as the difference between the previous poll cache and the TCP close counters.
2. **Overhead of polling tcp_info**: "ss -nti" on some servers, with over 15k sessions, can take over 100 milliseconds of CPU time. This can be optimized down somewhat, but it's possible that for some workloads the overhead of tcp_info polling (plus caching, and TCP close tracing) is higher than just TCP send/receive tracing.

It's possible the current implementation, involving TCP send/receive tracing, ends up hard to beat due to these other complications. I'm still investigating.

Prior tcptop

I wrote the original tcptop years ago using DTrace, based on my earlier work with tcpsnoop. The lesson I learned from these was to minimize the number of dynamic probes used, since the TCP/IP stack will change between kernel versions, and the more probes you use the more brittle the program will be. Better still, use static tracepoints, which won't change.

Linux needs static tracepoints for TCP send & receive, after which, tools like tcptop won't break between kernels. That's a topic for another post.

Thanks to Coburn (Netflix) for suggesting I try building this tool with BPF.

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).