TCP Tracepoints

22 Mar 2018

TCP tracepoints have arrived in Linux! Linux 4.15 added five of them, and 4.16 (not quite released yet) added two more (`tcp:tcp_probe`, and `sock:inet_sock_set_state` – a socket tracepoint that can be used for TCP analysis). We now have:

```
# perf list 'tcp:*' 'sock:inet*'

List of pre-defined events (to be used in -e):

  tcp:tcp_destroy_sock                               [Tracepoint event]
  tcp:tcp_probe                                      [Tracepoint event]
  tcp:tcp_receive_reset                              [Tracepoint event]
  tcp:tcp_retransmit_skb                             [Tracepoint event]
  tcp:tcp_retransmit_synack                          [Tracepoint event]
  tcp:tcp_send_reset                                 [Tracepoint event]

  sock:inet_sock_set_state                           [Tracepoint event]
```

This includes one that's versatile: `sock:inet_sock_set_state`. It can be used to track when the kernel changes the state of a TCP session, such as from TCP_SYN_SENT to TCP_ESTABLISHED. One example use is my tcplife tool in the open source bcc collection:

```
# tcplife
PID   COMM       LADDR           LPORT RADDR           RPORT TX_KB RX_KB MS
22597 recordProg 127.0.0.1       46644 127.0.0.1       28527     0     0 0.23
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46644     0     0 0.28
22598 curl       100.66.3.172    61620 52.205.89.26    80        0     1 91.79
22604 curl       100.66.3.172    44400 52.204.43.121   80        0     1 121.38
22624 recordProg 127.0.0.1       46648 127.0.0.1       28527     0     0 0.22
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46648     0     0 0.27
[...]
```

I wrote tcplife before this tracepoint existed, so I had to use kprobes (kernel dynamic tracing) of the tcp_set_state() kernel function. That works, but it's relying on various kernel implementation details that may change from one kernel version to the next. To keep tcplife working, it would need to include different code every time the kernel changed, which would become difficult to maintain and enhance. Imagine needing to test changes on several different kernel versions, because tcplife has special code for each!

Tracepoints are considered a "stable API," so their details shouldn't change from one kernel version to the next, making programs that use them easier to maintain. I say "shouldn't" on purpose, because I consider these "best effort" and not "set in stone". If they are considered set in stone, then it will be harder to convince kernel maintainers to accept new tracepoints (for good reason). As a case in point: `tcp:tcp_set_state` was added in 4.15, and then `sock:inet_sock_set_state` was added in 4.16. Since the sock one is a superset, the tcp one was disabled in 4.16 and will be removed. We try to avoid changing tracepoints like this, but in this case it was short-lived and removed before anyone had used it.

tcplife isn't a great example of using tracepoints anyway, as it goes beyond the tracepoint API in three places (tx and rx bytes, and best-effort process context on tracepoints), so it may still need some maintenance. But it's a large improvement over the kprobes version, and other tools can stick to the tracepoints API only.

Another way to show `sock:inet_sock_set_state` is to compare it with kprobes on tcp_set_state() using Sasha Goldshtein's bcc trace tool. The first command uses kprobes, and the second the tracepoint:

```
# trace -t -I net/sock.h 'p::tcp_set_state(struct sock *sk) "%llx: %d -> %d", sk, sk->sk_state, a
TIME     PID     TID     COMM            FUNC             -
2.583525 17320   17320   curl            tcp_set_state    ffff9fd7db588000: 7 -> 2
2.584449 0       0       swapper/5       tcp_set_state    ffff9fd7db588000: 2 -> 1
2.623158 17320   17320   curl            tcp_set_state    ffff9fd7db588000: 1 -> 4
2.623540 0       0       swapper/5       tcp_set_state    ffff9fd7db588000: 4 -> 5
2.623552 0       0       swapper/5       tcp_set_state    ffff9fd7db588000: 5 -> 7
^C
# trace -t 't:sock:inet_sock_set_state "%llx: %d -> %d", args->skaddr, args->oldstate, args->news
TIME     PID     TID     COMM            FUNC             -
1.690191 17308   17308   curl            inet_sock_set_state ffff9fd7db589800: 7 -> 2
1.690798 0       0       swapper/24      inet_sock_set_state ffff9fd7db589800: 2 -> 1
1.727750 17308   17308   curl            inet_sock_set_state ffff9fd7db589800: 1 -> 4
1.728041 0       0       swapper/24      inet_sock_set_state ffff9fd7db589800: 4 -> 5
1.728063 0       0       swapper/24      inet_sock_set_state ffff9fd7db589800: 5 -> 7
^C
```

Both are showing the same output. For reference:

- 1: TCP_ESTABLISHED
- 2: TCP_SYN_SENT
- 3: TCP_SYN_RECV
- 4: TCP_FIN_WAIT1
- 5: TCP_FIN_WAIT2
- 6: TCP_TIME_WAIT
- 7: TCP_CLOSE
- 8: TCP_CLOSE_WAIT

I know, I know, I should just add that as a lookup hash and ... a little while later, here's a new tool I just contributed to bcc – tcpstate – that does the translations, and shows per-state durations:

```
# tcpstates
SKADDR           C-PID C-COMM     LADDR           LPORT RADDR          RPORT OLDSTATE    -> NEWSTA
ffff9fd7e8192000 22384 curl       100.66.100.185  0     52.33.159.26   80    CLOSE       -> SYN_SE
ffff9fd7e8192000 0     swapper/5  100.66.100.185  63446 52.33.159.26   80    SYN_SENT    -> ESTABL
ffff9fd7e8192000 22384 curl       100.66.100.185  63446 52.33.159.26   80    ESTABLISHED -> FIN_WA
ffff9fd7e8192000 0     swapper/5  100.66.100.185  63446 52.33.159.26   80    FIN_WAIT1   -> FIN_WA
ffff9fd7e8192000 0     swapper/5  100.66.100.185  63446 52.33.159.26   80    FIN_WAIT2   -> CLOSE
^C
```

I'm demonstrating this on Linux 4.16, after Yafang Shao wrote an enhancement to show all state transitions, instead of just the ones the kernel implements. Here's what it used to look like on 4.15:

```
# trace -I net/sock.h -t 'p::tcp_set_state(struct sock *sk) "%llx: %d -> %d", sk, sk->sk_state, a
TIME     PID    TID    COMM         FUNC             -
3.275865 29039  29039  curl         tcp_set_state    ffff8803a8213800: 7 -> 2
3.277447 0      0      swapper/1    tcp_set_state    ffff8803a8213800: 2 -> 1
3.786203 29039  29039  curl         tcp_set_state    ffff8803a8213800: 1 -> 8
3.794016 29039  29039  curl         tcp_set_state    ffff8803a8213800: 8 -> 7
^C
# trace -t 't:tcp:tcp_set_state "%llx: %d -> %d", args->skaddr, args->oldstate, args->newstate'
TIME     PID    TID    COMM         FUNC             -
2.031911 29042  29042  curl         tcp_set_state    ffff8803a8213000: 7 -> 2
2.035019 0      0      swapper/1    tcp_set_state    ffff8803a8213000: 2 -> 1
2.746864 29042  29042  curl         tcp_set_state    ffff8803a8213000: 1 -> 8
2.754193 29042  29042  curl         tcp_set_state    ffff8803a8213000: 8 -> 7
```

Back to 4.16, here's the current list of tracepoints, with arguments, via bcc's tplist tool:

```
# tplist -v 'tcp:*'
tcp:tcp_retransmit_skb
    const void * skbaddr;
    const void * skaddr;
    __u16 sport;
    __u16 dport;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
tcp:tcp_send_reset
    const void * skbaddr;
    const void * skaddr;
    __u16 sport;
    __u16 dport;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
tcp:tcp_receive_reset
    const void * skaddr;
    __u16 sport;
    __u16 dport;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
tcp:tcp_destroy_sock
    const void * skaddr;
    __u16 sport;
    __u16 dport;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
tcp:tcp_retransmit_synack
    const void * skaddr;
    const void * req;
    __u16 sport;
    __u16 dport;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
tcp:tcp_probe
    __u8 saddr[sizeof(struct sockaddr_in6)];
    __u8 daddr[sizeof(struct sockaddr_in6)];
    __u16 sport;
    __u16 dport;
    __u32 mark;
    __u16 length;
    __u32 snd_nxt;
    __u32 snd_una;
    __u32 snd_cwnd;
    __u32 ssthresh;
    __u32 snd_wnd;
    __u32 srtt;
    __u32 rcv_wnd;
# tplist -v sock:inet_sock_set_state
sock:inet_sock_set_state
    const void * skaddr;
    int oldstate;
    int newstate;
    __u16 sport;
    __u16 dport;
    __u16 family;
    __u8 protocol;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
```

The first TCP tracepoint added was tcp:tcp_retransmit_skb by Cong Wang (Twitter). He cited my kprobe-based tcpretrans tool from perf-tools as an example consumer. Song Liu (Facebook) added five more tracepoints, including tcp:tcp_set_state which is now `sock:inet_sock_set_state`. Thanks to Cong and Song, and also David S. Miller (networking maintainer) for accepting these and providing feedback on the ongoing tcp tracepoint work.

During development I talked to Song (and Alexei Starovoitov) about the recent additions, so I already have an idea about why these exist and their use. Some rough notes for the current TCP tracepoints:

- **tcp:tcp_retransmit_skb**: Traces retransmits. Useful for understanding network issues including congestion. Will be used by my tcpretrans tools instead of kprobes.

- **tcp:tcp_retransmit_synack**: Tracing SYN and SYN/ACK retransmits. These are interesting to separate out, as they can show server saturation (listen backlog drops) rather than network congestion. It corresponds to LINUX_MIB_TCPSYNRETRANS.
- **tcp:tcp_destroy_sock**: Needed by any program that summarizes details in-memory for a TCP session, which would be keyed by the sock address. This probe can be used to know that the session has ended, so that sock address is about to be reused and any summarized data so far should be consumed and then deleted.
- **tcp:tcp_send_reset**: This traces a RST send during a valid socket, to diagnose those type of issues.
- **tcp:tcp_receive_reset**: Trace a RST receive.
- **tcp:tcp_probe**: for TCP congestion window tracing, which also allowed an older TCP probe module to be deprecated and removed. This was added by Masami Hiramatsu, and merged in Linux 4.16.
- **sock:inet_sock_set_state**: Can be used for many things. The tcplife tool is one, but also my tcpconnect and tcpaccept bcc tools can be converted to use this tracepoint. We could add separate `tcp:tcp_connect` and `tcp:tcp_accept` tracepoints (or `tcp:tcp_active_open` and `tcp:tcp_passive_open`), but `sock:inet_sock_set_state` can be used for this.

Use of these tracepoints is much preferred over packet capture approaches (libpcap), as tracepoints should cost lower overhead and can expose useful kernel state that's not on the wire.

I can imagine how useful these TCP tracepoints will be, as I designed and used similar tracepoints many years ago: my DTrace TCP provider which I demonstrated at CEC2006. I originally split out TCP state changes into a probe for each state, but by the time this was merged it became a single tcp:::state-change probe, as we now have in Linux via the sock tracepoint.

What's next? Tracepoints for `tcp:tcp_send` and `tcp:tcp_receive` may be handy, but special attention must be paid to the tiny overhead they can add (both enabled and especially disabled), since send/receive can be a very frequent activity. More tracepoints for error conditions would be useful too, such as for the connection refused path, which may be helpful for analyzing DoS attacks.

If you are interested in adding TCP tracepoints, I'd recommend coding a kprobe solution to start with as the proof of concept, and getting some production experience with it. This is the role my prior kprobe tools played. A kprobe solution will show whether a tracepoint would be that useful, and if so, help make the case for its inclusion with the Linux kernel maintainers.

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).