L41: Lab 4 - The TCP State Machine

Dr Robert N. M. Watson   Dr Graeme Jenkinson

2019-2020

The goals of this lab are to:

- Use DTrace to investigate the actual TCP state machine and its interactions with the network stack.
- Use DTrace and DUMMYNET to investigate the effects of latency on TCP state transitions.

In this lab, we begin that investigation, which will be extended to include additional exploration of TCP bandwidth, as affected by latency, in the following lab.

Background: Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) is a near-universally used protocol that provides reliable, bi-directional, ordered octet streams over the Internet Protocol (IP) between two communication endpoints. TCP connections are built between a pair of IP addresses, identifying host network interfaces, and port numbers selected applications (or automatically by the kernel) on either endpoint – collectively, a *4-tuple*. While other models are possible, typical TCP use has one side play the role of a 'server', which provides some network-reachable service on a *well-known port*, and the other the 'client', building a connection to reach that service from an *ephemeral port* randomly selected by the client TCP implementation.

The BSD (and now POSIX) sockets API offers a portable and simple interface for TCP/IP client and server programming. The server opens a socket using the `socket(2)` system call, binds a well-known or previously negotiated port number using `bind(2)`, and performs `listen(2)` to begin accepting new connections, returned as additional connected sockets from calls to `accept(2)`. The client application similarly calls `socket(2)` to open a socket, and `connect(2)` to connect to a target address and port number. Once open, both sides can use system calls such as `read(2)`, `write(2)`, `send(2)`, and `recv(2)` to send and receive data over the connection. The `close(2)` system call both initiates a connection close (if not already closed) and releases the socket – whose state may persist for some further period to allow data to drain and prevent premature re-use of the 4-tuple.

As discussed in lecture, TCP connections are implemented by a pair of state machine instances, one on each communications endpoint. Once in the `ESTABLISHED` steady state, data passes in each direction via *segments* that are *acknowledged* by packets passing in the other direction. The rate of data flow is controlled by TCP's *flow-control* and *congestion-control* mechanisms that respectively prevent the sender from sending more data than the receiver or network can handle. Congestion control operates in three phases: *slow start*, in which use of bandwidth is rapidly ramped up to exploit available network bandwidth either at the start of the connection or following a timeout, and two tightly coupled phases of *congestion avoidance* and *fast recovery* as TCP discovers and maintains a *congestion window* close to available fair bandwidth limit.

TCP identifies every byte in one direction of a connection via a sequence number. Data segments contain a starting sequence number and length, describing the range of transmitted bytes. Acknowledgment packets contain the sequence number of the byte that follows the last contiguous byte they are acknowledging. Acknowledgments are piggybacked onto data segments traveling in the opposite direction to the greatest extent possible to avoid additional packet transmissions. In slow start, TCP performance is directly limited by latency, as the congestion window can be opened only by receiving `ACK`s – which require successive round trips. These periods are referred to as *latency bound* for this reason, and network latency a critical factor in effective utilisation of path bandwidth.

The benchmark

Our IPC benchmark also supports a `tcp_socket` IPC type which requests use of TCP over the *loopback interface* on port `10141`. Use of a fixed port number makes it easy to identify and classify experimental packets on the loopback interface using packet-sniffing tools such as `tcpdump`, and also via DTrace predicates. You are advised to minimise network activity during the running of TCP-related benchmarks, and when using DTrace, to reduce the degree of interference both from the perspective of analysing behaviour, and for reasons of the probe effect.

Compiling the benchmark

Labs 4 and 5 use the same IPC benchmark utilized in Labs 2 and 3. Follow the instructions present in those lab assignments to build and use the IPC benchmark.

Running the benchmark

As before, you can run the benchmark using the `ipc-static` command, specifying various benchmark parameters. For the purposes of this benchmark, we recommend the following configuration:

- Use 2-thread mode (as described in Lab 2)
- Do not set the socket-buffer size flag
- Do not modify the total I/O size

Do ensure that, as in Labs 2 and 3, you have increased the kernel's maximum socket-buffer size.

IPFW and DUMMYNET

To control latency for our experimental traffic, we will employ the IPFW firewall for packet classification, and the DUMMYNET traffic-control facility to pass packets over simulated 'pipes'. To configure two 1-way DUMMYNET pipes, each carrying a 10ms one-way latency, run the following commands as root:

```
ipfw pipe config 1 delay 10
ipfw pipe config 2 delay 10
```

During your experiments, you will wish to change the simulated latency to other values, which can be done by reconfiguring the pipes. Do this by repeating the above two commands but with modified last parameters, which specify one-way latencies in milliseconds (e.g., replace '10' with '5' in both commands). The total Round-Trip Time (RTT) is the sum of the two latencies – i.e., 10ms in each direction comes to a total of 20ms RTT. Note that DUMMYNET is a simulation tool, and subject to limits on granularity and precision. Next, you must assign traffic associated with the experiment, classified by its TCP port number and presence on the loopback interface (`lo0`), to the pipes to inject latency:

```
ipfw add 1 pipe 1 tcp from any 10141 to any via lo0
ipfw add 2 pipe 2 tcp from any to any 10141 via lo0
```

You should configure these firewall rules only once per boot.

Configuring the loopback MTU

Network interfaces have a configured Maximum Transmission Unit (MTU) – the size, in bytes, of the largest packet that can be sent. For most Ethernet and Ethernet-like interfaces, the MTU is typically 1,500 bytes, although larger 'jumbograms' can also be used in LAN environments. The loopback interface provides a simulated network interface carrying traffic for loopback addresses such as 127.0.0.1 (`localhost`), and typically uses a larger (16K+) MTU. To allow our simulated results to more closely resemble LAN or WAN traffic, run the following command as root to set the loopback-interface MTU to 1,500 bytes after each boot:

```
ifconfig lo0 mtu 1500
```

Example benchmark command

This command instructs the IPC benchmark to perform a transfer over TCP in 2-thread mode:

```
ipc/ipc-static -v -i tcp 2thread
```

DTrace probes for TCP

FreeBSD's DTrace implementation contains a number of probes pertinent to TCP, which you may use in addition to system-call and other probes you have employed in prior labs:

fbt::syncache_add:entry FBT probe when a `SYN` packet is received for a listening socket, which will lead to a SYN cache entry being created. The third argument (`args[2]`) is a pointer to a `struct tcphdr`.

fbt::syncache_expand:entry FBT probe when a TCP packet converts a pending SYN cookie or SYN cache connection into a full TCP connection. The third argument (`args[2]`) is a pointer to a `struct tcphdr`.

fbt::tcp_do_segment:entry FBT probe when a TCP packet is received in the 'steady state'. The second argument (`args[1]`) is a pointer to a `struct tcphdr` that describes the TCP header (see RFC 893). You will want to classify packets by port number to ensure that you are collecting data only from the flow of interest (port `10141`), and associating collected data with the right direction of the flow. Do this by checking TCP header fields `th_sport` (source port) and `th_dport` (destination port) in your DTrace predicate. In addition, the fields `th_seq` (sequence number in transmit direction), `th_ack` (ACK sequence number in return direction), and `th_win` (TCP advertised window) will be of interest. The fourth argument (`args[3]`) is a pointer to a `struct tcpcb` that describes the active connection.

fbt::tcp_state_change:entry FBT probe that fires when a TCP state transition takes place. The first argument (`args[0]`) is a pointer to a `struct tcpcb` that describes the active connection. The `tcpcb` field `t_state` is the previous state of the connection. Access to the connection's port numbers at this probe point can be achieved by following `t_inpcb->inp_inc.inc_ie`, which has fields `ie_fport` (foreign, or remote port) and `ie_lport` (local port) for the connection. The second argument is the new state to be assigned.

When analysing TCP states, the D array `tcp_state_string` can be used to convert an integer state to a human-readable string (e.g., 0 to `TCPS_CLOSED`). For these probes, the port number will be in *network byte order*; the D function `ntohs()` can be used to convert to host byte order when printing or matching values in `th_sport`, `th_dport`, `ie_lport`, and `ie_fport`. Note that sequence and acknowledgment numbers are cast to unsigned integers. When analysing and graphing data, be aware that sequence numbers can (and will) wrap due to the 32-bit sequence space.

Sample DTrace scripts

The following script prints out, for each received TCP segment beyond the initial SYN handshake, the sequence number, ACK number, and state of the TCP connection prior to full processing of the segment:

```
dtrace -n 'fbt::tcp_do_segment:entry {
  trace((unsigned int)args[1]->th_seq);
  trace((unsigned int)args[1]->th_ack);
  trace(tcp_state_string[args[3]->t_state]);
}'
```

Trace state transitions printing the receiving and sending port numbers for the connection experiencing the transition:

```
dtrace -n 'fbt::tcp_state_change:entry {
  trace(ntohs(args[0]->t_inpcb->inp_inc.inc_ie.ie_lport));
  trace(ntohs(args[0]->t_inpcb->inp_inc.inc_ie.ie_fport));
  trace(tcp_state_string[args[0]->t_state]);
  trace(tcp_state_string[args[1]]);
}'
```

These scripts can be extended to match flows on port `10141` in either direction as needed.

Experimental questions (part 1)

These questions form the first part of your lab report spanning Labs 4 and 5. As described above, configure the IPC benchmark to use TCP in `2thread` mode. When exploring TCP state-machine behaviour, use whole-program analysis. When exploring the effects of latency on performance, use only I/O-loop analysis. Employ the Graphviz tool to plot state machines automatically from measurements captured using DTrace scripts.

1. Plot an effective (i.e., as measured) TCP state-transition diagram for the two directions of a single TCP connection: states will be nodes, and transitions will be edges. Where state transitions diverge between the two directions, be sure to label edges indicating 'client' vs. 'server'.
2. Extend the diagram to indicate, for each edge, the TCP header flags of the received packet triggering the transition, or the local system call (or other event – e.g., timer) that triggers the transition.
3. Compare the graphs you have drawn with the TCP state diagram in RFC 793.
4. Using DUMMYNET, explore the effects of simulated latency at 5ms intervals between 0ms and 40ms. What observations can we make about state-machine transitions as latency increases?
5. Be sure, in your lab report, to describe any apparent simulation or probe effects.