Advanced Operating Systems: Lab 3 - TCP

Dr Robert N. M. Watson

2020-2021

The goals of this lab are to:

- Investigate the the TCP implementation state machine – as distinct from the protocol state machine in the TCP specificatioon – and its interactions with network latency. (**Part II only**)
- Investigate the effects of network latency on TCP performance, and in particular interactions with congestion control. (**L41 only**)
- Evaluate the effects of socket-buffer size on effective TCP bandwidth. (**L41 only**)

To do this, you will:

- Employ the same benchmark used in Lab 2, but in the TCP socket mode.
- Use FreeBSD's DUMMYNET facility to simulate network latencies on the loopback interface.
- Use DTrace to inspect both network packet data and internal protocol control-block state.

Part II and L41 students should submit work based only on the lab questions they are assigned. However, students are welcome to investigate the problems assigned to the other course if they are interested.

1 Background

1.1 The Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) protocol provides reliable, bi-directional, ordered octet (byte) streams over the Internet Protocol (IP) between two communication endpoints.

1.1.1 The TCP 4-tuple

TCP connections are built between a pair of IP addresses, identifying host network interfaces, and port numbers selected by applications (or automatically by the kernel) on either endpoint. Collectively, the two addresses and port numbers that uniquely identify a TCP connection are referred to as the *TCP 4-tuple*, which is used to look up internal connection state.

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*. The other side is the *client*, which builds a connection to the service from a local *ephemeral port*. Ephemeral ports are allocated randomly (historically, sequentially).

1.1.2 Sockets

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.

1.1.3 Acknowledgment, loss, and retransmit

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. The TCP sender is not permitted to discard data until it has been explicitly acknowledged by the sender, so that it can retransmit packets that may have been lost. When and how agressively to retransmit are complex topics heavily impacted by congestion control.

2 The benchmark

The IPC benchmark introduced in Lab 2, `ipc-benchmark`, also supports a `tcp` IPC type that requests use of TCP over the *loopback interface*. 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`, for latency simulation using DUMMYNET, and also via DTrace predicates. We recommend TCP port `10141` for this purpose. Data segments carrying benchmark data from the sender to the receiver will have a *source port* of `10141`, and acknowledgements from the receiver to the sender will have a *destination port* of `10141`.

2.1 Compiling the benchmark

While the Lab 3 benchmark is very similar to the Lab 2 benchmark, the Lab 3 version contains some updates, including to flush the TCP host cache automatically. Please make sure that you are using the Lab 3 version, found in `/advopsys-packages/labs/` on your RPi4. Follow the instructions present in that assignment to build and use the IPC benchmark.

2.2 Running the benchmark

As before, you can run the benchmark using the `ipc-benchmark` command, specifying various benchmark parameters. This lab requires you to:

- Use `2thread` mode (as described in Lab 2)
- Hold the total I/O size (16M) constant
- Use verbose mode to report additional benchmark configuration data (`-v`)
- Use JSON machine-readable output mode (`-j`)

Be sure to pay specific attention to the parameters specified in the experimental questions – e.g., with respect to socket-buffer sizing and block size.

2.3 Example benchmark command

This command instructs the IPC benchmark to perform a transfer over TCP in 2-thread mode, generating output in JSON, and printing additional benchmark configuration information:

```
ipc/ipc-benchmark -j -v -i tcp 2thread
```

3 Configuring the kernel

3.1 Increasing the maximum socket-buffer size

As in Lab 2, increase the kernel's maximum socket-buffer size:

```
# sysctl kern.ipc.maxsockbuf=33554432
```

3.2 netisr worker CPU pinning

In the default FreeBSD kernel configuration, a single kernel `netisr` thread is responsible for deferred dispatch, including loopback input processing. In our experimental configuration, we pin that thread to CPU 0, where we also run the IPC benchmark. This simplifies tracing and analysis in your assignment. We have put this setting in the boot-loader configuration file for you, and no further action.

3.3 Flushing the TCP host cache

FreeBSD implements a *host cache* that stores sampled round-trip times, bandwidth estimates, and other information to be used across different TCP connections to the same remote host. Normally, this feature allows improved performance as, for example, by allowing past estimates of bandwidth to trigger a transition from slow start to steady state without 'overshooting', potentially triggering significant loss. However, in the context of this lab, carrying of state between connections reduces the independence of our experimental runs. The IPC benchmark flushes the TCP host cache before each iteration is run, preventing information that may affect congestion-control decisions from being carried between runs.

3.4 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 2$\times$ one-way DUMMYNET pipes, each imposing 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.

3.5 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
```

4 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 (`args[1]`) 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.

4.1 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.

4.2 DTrace ARMv8-A probe argument limitation

Due to a limitation of the DTrace implementation on FreeBSD/arm64, at most five probe arguments are available. This impacts some `tcp` and `fbt` probes that have larger numbers of probes – e.g., `tcp:::state-change`, hence our recommending `fbt::tcp_state_change:entry`.