# Advanced Operating Systems:
# Lab 3 – TCP
# **General Information**

Prof. Robert N. M. Watson

2023-2024

The goals of this lab are to:

- Investigate the effects of network latency on TCP performance, and in particular interactions with congestion control.
- Assess the impact of the probe effect on the tracing and analysis you have performed to understand the effects of latency on TCP performance. (**ACS/Part III 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.

## 1 Assignment documents

This document provides Lab 3 background and technique information.

**Part II students** should perform the assignment found in *Advanced Operating Systems: Lab 3 – TCP – Part II Assignment*.

**ACS/Part III students** should perform the assignment found in *Advanced Operating Systems: Lab 3 – TCP – ACS/Part III Assignment*. Follow the lab-report guidance found in *ACS/Part III: Lab Reports*, and use the lab-report LaTeX template, `l341-labreport-template.tex`.

## 2 Background: The Transmission Control Protocol (TCP)

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

### 2.1 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).

## 2.2 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. In our case, the server sends data and the client receives data.
- 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.

## 2.3 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.

# 3 Background: TCP transmission control

## 3.1 TCP flow control and congestion control

TCP specifies two rate-control mechanisms:

**Flow control** allows a receiver to limit the amount of unacknowledged data transmitted by the remote sender, preventing receiver buffers from being overflowed. This is implemented via *window advertisements* sent via acknowledgments back to the sender. When using the sockets API, the advertised window size is based on available space in the *receive socket buffer*, meaning that it will be sensitive to both the size configured by the application (using socket options) and the rate at which the application reads data from the buffer.

Contemporary TCP implementations *auto-resize* socket buffers if a specific size has not been requested by the application, avoiding use of a constant default size that may substantially limit overall performance (as the sender may not be able to fully fill the *bandwidth-delay product* of the network)[1]. Note that this requirement for large buffer sizes is in tension with local performance behaviour explored in prior IPC labs.

**Congestion control** allows the sender to avoid overfilling the network path to the receiving host, avoiding unnecessary packet loss and negative impacting on other traffic on the network (*fairness*). This is implemented via a variety of congestion-detection techniques, depending on the specific algorithm and implementation – but most frequently, interpretation of packet-loss events as a congestion indicator. When a receiver notices a gap in the received sequence-number series, it will return a *duplicate ACK*, which hints to the sender that a packet has been lost and should be retransmitted[2].

TCP congestion control maintains a *congestion window* on the sender – similar in effect to the flow-control window, in that it limits the amount of unacknowledged data a sender can place into the network. When a connection first opens, and also following a timeout after significant loss, the sender will enter *slow start*, in

[1] Bandwidth (bits/s) * Round Trip Time (s)

[2] This is one reason why it is important that underlying network substrates retain packet ordering for TCP flows: misordering may be interpreted as packet loss, triggering unnecessary retransmission.

which the window is 'opened' gradually as available bandwidth is probed. The name 'slow start' is initially confusing as it is actually an exponential ramp-up. However, it is in fact slow compared to the original TCP algorithm, which had no notion of congestion and overfilled the network immediately!

In slow start, TCP performance is directly limited by latency, as the congestion window can be opened only by receiving `ACKs` – 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.

When congestion is detected (i.e., because the congestion window has gotten above available bandwidth triggering a loss), a cycle of congestion recovery and avoidance is entered. The congestion window will be reduced, and then the window will be more slowly reopened, causing the congestion window to continually (gently) probe for additional available bandwidth, (gently) falling back when it re-exceeds the limit. In the event a true timeout is experienced – i.e., significant packet loss – then the congestion window will be cut substantially and slow start will be re-entered.

The steady state of TCP is therefore responsive to the continual arrival and departure of other flows, as well as changes in routes or path bandwidth, as it detects newly available bandwidth, and reduces use as congestion is experienced due to over utilisation.

The FreeBSD version used in this lab defaults to the NewReno congestion-control algorithm (later versions have changed). Authoritative information on TCP congestion control can be found in RFC 5681. [3]

TCP composes these two windows by taking the minimum: it will neither send too much data for the remote host (flow control), nor for the network itself (congestion control). One limit is directly visible in the packets themselves (the advertised window from the receiver), but the other must either be intuited from wire traffic, or given suitable access, monitored using end-host instrumentation – e.g., using DTrace. Two further informal definitions will be useful:

**Latency** is the time it takes a packet to get from one endpoint to another. TCP implementations measure *Round-Trip Time (RTT)* in order to tune timeouts detecting packet loss. More subtlely, RTT also limits the rate at which TCP will grow the congestion window, especially during slow start: the window can grow only as data is acknowledged, which requires round-trip times as ACKs are received. As latency increases, congestion-window-size growth is limited.

**Bandwidth** is the throughput capacity of a link (or network path) to carry data, typically measured in bits or bytes per second. TCP attempts to discover the available bandwidth by iteratively expanding the congestion-control window until congestion is experienced, and then backing off. The rate at which the congestion-control window expands is dependent on round trip times; as a result, it may take longer for TCP to achieve peak bandwidth on higher latency networks.

### 3.2 TCP and the receive socket buffer

The TCP stack will not advertise a receive window that will not fit in the available space in the socket buffer. This is calculated by subtracting current buffer occupancy from the socket-buffer limit. In early TCP, the advertised window was solely present to support flow control, allowing the sender to avoid transmitting data that the recipient could not reliably buffer.

However, the size of the buffer also has a secondary effect: It limits bandwidth utilization by constraining the bandwidth-delay product, which must fit within that window. As latency increases, TCP must have more unacknowledged data in flight in order to fill the pipe, and hence achieve maximum bandwidth. More recent TCP and sockets implementations allow the socket buffer to be automatically resized based on utilization: as it becomes more full, the socket-buffer limit is increased to allow the TCP window to open further.

## 4 Using DTrace to trace TCP state

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:

[3] `https://datatracker.ietf.org/doc/html/rfc5681`

**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 Tracing connections: Packets and internal TCP state

The `tcp_do_segment` FBT probe allows us to track TCP input in the steady state. In some portions of this lab, you will take advantage of access to the TCP control block (`tcpcb` structure – `args[3]` to the `tcp_do_segment` FBT probe) to gain additional insight into TCP behaviour. The following fields may be of interest:

**`snd_wnd`** On the sender, the last received advertised flow-control window.

**`snd_cwnd`** On the sender, the current calculated congestion-control window.

**`snd_ssthresh`** On the sender, the current *slow-start threshold* – if `snd_cwnd` is less than or equal to `snd_ssthresh`, then the connection is in slow start; otherwise, it is in congestion avoidance.

When writing DTrace scripts to analyse a flow in a particular direction, you can use the port fields in the TCP header to narrow analysis to only the packets of interest. For example, when instrumenting `tcp_do_segment` to analyse received acknowledgments, it will be desirable to use a predicate of `/args[1]->th_dport == htons(10141)/` to select only packets being sent to the server port (e.g., ACKs), and the similar (but subtly different) `/args[1]->th_sport == htons(10141)/` to select only packets being sent from the server port (e.g., data). Note that you will wish to take care to ensure that you are reading fields from within the `tcpcb` at the correct end of the connection – the 'send' values, such as last received advertised window and congestion window, are properties of the server, and not client, side of this benchmark, and hence can only be accessed from instances of `tcp_do_segment` that are processing server-side packets.

To calculate the length of a segment in the probe, you can use the `tcp:::send` probe to trace the `ip_length` field in the `ipinfo_t` structure (`args[2]`):

```
typedef struct ipinfo {
        uint8_t ip_ver;                 /* IP version (4, 6) */
        uint16_t ip_plength;            /* payload length */
```

```
        string ip_saddr;                /* source address */
        string ip_daddr;                /* destination address */
} ipinfo_t;
```

As is noted in the DTrace documentation for this probe this `ip_plength` is the expected IP payload length so no further corrections need be applied.

Data for graphs in this assignment is typically gathered at (or close to) one endpoint in order to provide timeline consistency – i.e., the viewpoint of just the client or the server, not some blend of the two time lines. As we will be measuring not just data from packet headers, but also from the TCP implementation itself, we recommend gathering most data close to the sender. As described here, it may seem natural to collect information on data-carrying segments on the receiver (where they are processed by `tcp_do_segment`), and to collect information on ACKs on the server (where they are similarly processes). However, given a significant latency between client and server, and a desire to plot points coherently on a unified real-time X axis, capturing both at the same endpoint will make this easier.

It is similarly worth noting that `tcp_do_segment`'s entry FBT probe is invoked before the ACK or data segment has been processed – so access to the `tcpcb` will take into account only state prior to the packet that is now being processed, not that data itself. For example, if the received packet is an ACK, then printed `tcpcb` fields will not take that ACK into account.

### 4.2 Sample DTrace script

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 {
  printf("%u %u %s",
    (unsigned int)args[1]->th_seq,
    (unsigned int)args[1]->th_ack,
    tcp_state_string[args[3]->t_state]);
}'
```

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

### 4.3 DTrace ARMv8-A probe argument limitation

Due to a limitation of the DTrace implementation on FreeBSD/arm64, at most five probe arguments are available, which corresponds to the number of argument registers available in the aarch64 ABI. This impacts some `tcp` and `fbt` probes that have larger numbers of arguments.

## 5 Useful kernel source code

The source code for the version of the kernel we are using can be found in `/usr/src/sys`:

- The socket implementation can be found in `/usr/src/sys/kern`. Here, you mind the files `*socket*.c` useful reading.
- The TCP/IP implementation can be found in `/usr/src/sys/netinet` (and `netinet6` for IPv6). Here, you may find `tcp_input.c` useful reading.
- The TCP NewReno implementation can be found in `/usr/src/sys/netinet/cc`.

## 6 Hypotheses

In this lab, we provide you with this hypothesis that you will test and explore through benchmarking:

- *Longer round-trip times extend the period over which TCP slow start takes place, but TCP is able to achieve equivalent performance through rapid identification of, and adaptation to, available bandwidth.*

- *Despite TCP's sensitivity to execution timing, the probe effect arising from using DTrace leaves your analysis valid.* (**ACS/Part III only**)

We will test these hypotheses by measuring net throughput between two TCP endpoints in two different processes. We will use DTrace to establish the causes of divergence from these hypotheses, and to explore the underlying implementations leading to the observed performance behavior.

# 7 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 TCP 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 (the server) to the receiver (the client) will have a *source port* of `10141`, and acknowledgements from the receiver to the sender will have a *destination port* of `10141`.

# 8 Getting Started

You do not need to recompile `ipc-benchmark` for Lab 3. No additional lab tarball is provided for this lab.

## 8.1 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 `-i tcp` to select the TCP benchmark mode
- Use `2proc` 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`)
- Collect `getrusage()` information (`-g`)
- As needed, set the buffer size (`-b bufferisze`)

Be sure to pay specific attention to the parameters specified in the experimental questions, and carefully check DUMMYNET settings.

## 8.2 Example benchmark command

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

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

# 9 Configuring the kernel

## 9.1 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.

### 9.2 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.

### 9.3 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× 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 out via lo0
ipfw add 2 pipe 2 tcp from any to any 10141 out via lo0
```

You should configure these firewall rules only once per boot.

### 9.4 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
```

## 10 Notes on plots and tables

Graphs and tables should be used to illustrate your measurement results. Ensure that, for each question, you present not only results, but also a causal explanation of those results – i.e., why the behaviour in question occurs, not just that it does. For the purposes of performance graphs in this assignment, use achieved bandwidth, rather than total execution time, for the Y axis, in order to allow you to more directly visualise the effects of configuration changes on efficiency.