# L41 - Advanced Operating Systems: Lab 3 - TCP

Dr Robert N. M. Watson

2020-2021

**This is L41 Lab 3. If you are a Part II student, please see the other lab variant.**

Your lab report will analyse how network latency impacts TCP throughput, with a particular interest in its effects on congestion control. You will do this using the `tcp` socket mode of the IPC benchmark, using DUMMYNET to simulate various network latencies.

Students may also wish to investigate latency and state-machine interactions explored in the corresponding Part II assignment, and are welcome to do so. However, they should not submit that work as part of this lab report.

## Hypotheses

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

1. *Longer round-trip times extend the period over which TCP slow start takes place, but peak bandwidths achieved at different latencies rapidly converge once slow start has completed.*

2. *Socket buffer auto-resizing uniformly improves performance by allowing the TCP window to open more quickly during slow start.*

We will test these hypotheses by measuring net throughput between two IPC endpoints in two different threads. 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. We will also be interested in how the probe effect, as well as potential simulation effects (e.g., due to using DUMMYNET), affect the fidelity of our analysis.

# 1 Background: TCP transmission control

## 1.1 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.

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

**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 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.

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.

## 1.2 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

[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.

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.
The IPC benchmark allows socket buffers to be configured in one of two ways:

**Automatic socket-buffer sizing** The default configuration for this benchmark, the kernel will detect when socket-buffer sizes become full, and automatically expand them.

**Fixed socket-buffer sizes** When run using the `-s` argument, the benchmark will automatically set the sizes of the send and receive socket buffers to the buffer size passed to the benchmark.

# 2 Background: Using DTrace to trace TCP state

The `tcp_do_segment` FBT probe allows us to track TCP input in the steady state. In some portions ofo 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 the two types of graphs described above 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.

# 3 Plotting TCP connections

TCP time-bandwidth graphs plot time on a linear X axis, and bandwidth achieved by TCP on a linear or log Y axis. Bandwidth may be usefully calculated as the change in sequence number (i.e., bytes) over a window of time – e.g., a second. Care should be taken to handle wrapping in the 32-bit sequence space; for shorter measurements this might be accomplished by dropping traces from experimental runs in which sequence numbers wrap.

This graph type may benefit from overlaying of additional time-based data, such as specific annotation of trace events from the congestion-control implementation, such as packet-loss detection or a transition out of slow start. Rather than directly overlaying, which can be visually confusing, a better option may be to "stack" the graphs: place them on the same X axis (time), horizontally aligned but vertically stacked. Possible additional data points (and Y axes) might include advertised and congestion-window sizes in bytes.

# Approach

You will run a series of experience using the IPC benchmark using the `tcp` IPC type, using DUMMYNET to simulate varying network latency. Configure the benchmark as follows:

- To use TCP: `-i tcp`
- To use a 2-thread configuration: `2thread`
- To set (or not set) the socket-buffer size: `-s`
- Flush the TCP host cache between all benchmark runs

# 4 Experimental questions: Latency and TCP bandwidth

Explore the following experimental questions, which consider only the TCP steady state (`ESTABLISHED`), and not the three-way handshake or connection close. For both questions, use a fixed 1MiB buffer (`-b 1048576`):

1. Plot latency (0ms .. 40ms in 5ms intervals) on the X axis, and effective bandwidth on the Y axis, considering two cases: where the socket-buffer size is set, and where it auto-resizes.

   Characterize the two performance behaviours. Explain why socket-buffer auto-resizing helps, hurts, or fails to affect performance as latency varies, in comparison to the fixed buffer size.

2. Plot time on the X axis, and effective bandwidth on the Y axis, considering the effects of auto-resizing and fixed socket-buffer sizes (`-s`) at synthetic latencies of 0ms and 20ms. Stack additional graphs showing the sender last received advertised window and congestion window on the same X axis, and use `snd_ssthresh` to label portions of the plot as being in slow start. For readability reasons, you may wish to generate two independent sets of plots, one for each latency, rather than overload a single set.

   Characterise the two performance behaviours. Document their differences, and explaining how the two buffering stratgies affect performance.

Be sure, in your lab report, to describe any apparent simulation or probe effects.

# Notes

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.