UNIVERSITY OF TECHNOLOGY SYDNEY

# 49202: Communication Protocols
# Laboratory Notes: Laboratory 3

Dr. Daniel R. Franklin

March 3, 2021

## Contents

## 1 Introduction

This laboratory covers material from Module 3, covering the Transport Layer. The main aims of this laboratory are:

1. To see transport protocols in action in the laboratory using Wireshark.
2. To measure the performance of different transport protocols in the laboratory and link this to theory - that is, verify that the results are what is expected, and if not, to explain why.

## 2 Pre-Lab Work

Before attending the laboratory, you must complete the following pre-lab questions. This will greatly improve your understanding of what is happening in the laboratory.

1. TCP and encapsulated protocols introduce an overhead into the transmission of information that limits the payload capacity of a link. Assume that you are transmitting a payload of P bytes over TCP/IP over Ethernet II. The overheads are: TCP - 20 bytes; IP - 20 bytes; Ethernet II - 14 bytes. Assume that the MTU size for Ethernet II is 1514 bytes.
   (a) Derive a formula for the payload capacity, $C$, as a function of $P$ and the link rate, $L$. Assume that $C$ and $L$ are in units of bits per second and $P$ is in bytes. For instance, if $L$ is 100 Mb/s and the total protocol overhead is 20% for a given size of $P$, then $C$ will be equal to 80 Mb/s.
   (b) Use MATLAB, GNU Octave (included in your VM image), Microsoft Excel or similar (or some other software) to plot $C$ as a function of $P$ for $1 \leq P \leq 10000$. Ensure that your plot has sufficient points in it to capture all the detail of the relationship (do you expect any discontinuities in your plot - why or why not?). Comment on the results.
2. TCP uses a slow-start mechanism to carefully ramp up transmission speed, while quickly backing off when network congestion is detected through packet loss.
   (a) Assume that you are to transmit a file size of 100000 bytes over TCP/IP over Ethernet II. Assume that transmission rate is 100 Mb/s and that the window size is 4 with an end-to-end delay of 1 ms. Draw a packet flow diagram of the entire transmission and hence or otherwise, calculate the total time required for transmission assuming that there are no errors or lost segments. You can assume that each second segment is acknowledged by the receiver.
   (b) Redo the preceding exercise but this time, assume that the receive buffer size is 5000 bytes. Assume received PDUs require 10 $\mu$s to process after removal from the receive buffer.

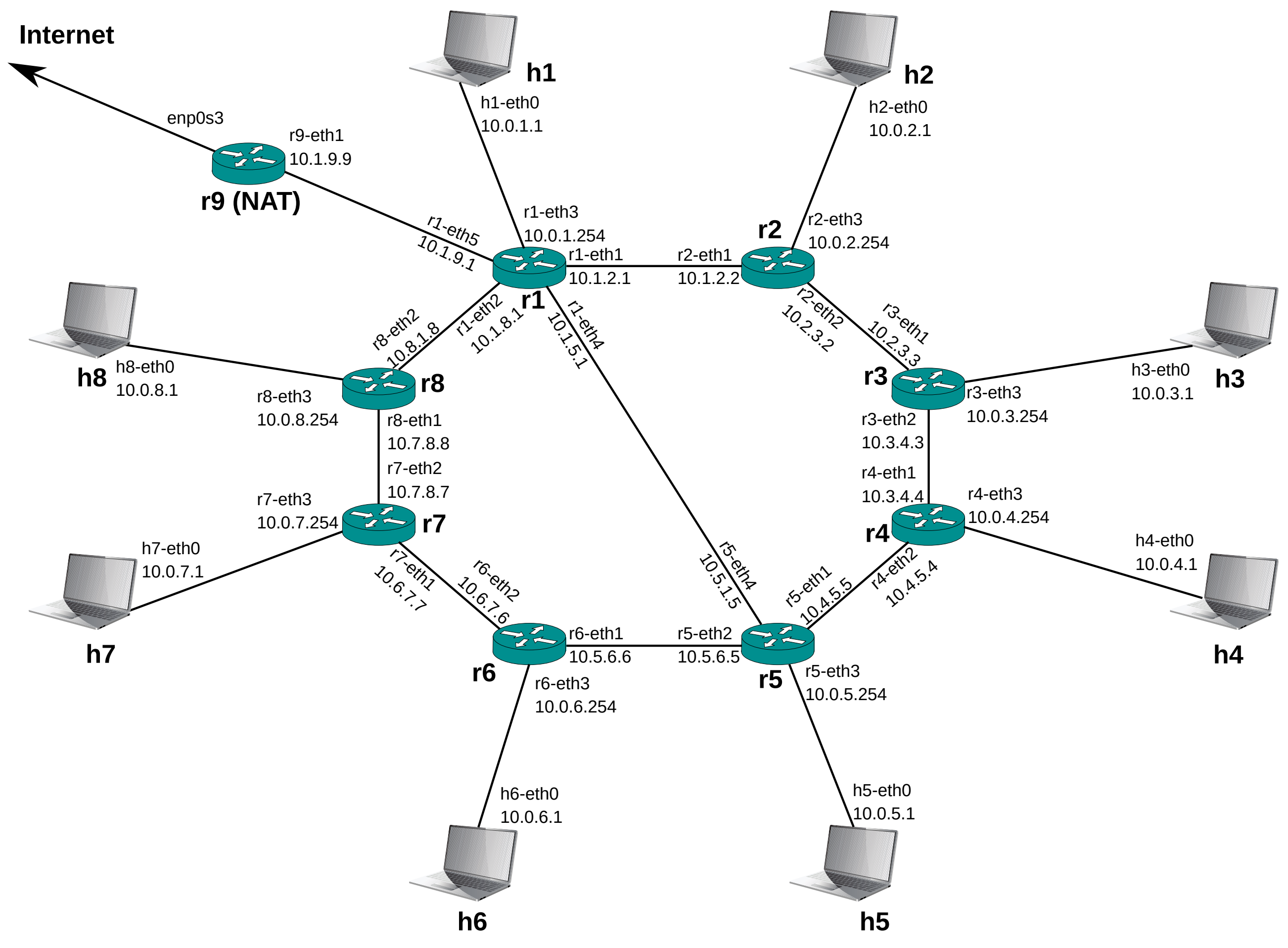

Figure 1: Network topology for Lab 3. The routing protocol is OSPF.

## 3 Setup

Prior to starting the lab, start your virtual machine (refer to Lab 0 notes) and make sure you have the latest version of the lab scripts:

```
cd ~/comm-protocols-labs

git pull
```

Start your VM (as described in Lab 0), open a terminal window (note: you may always open more than one terminal window if desired). For this lab, we will be using a generic OSPF-based network topology with full Internet access, as shown in Figure 1.

To start the Mininet emulated network with OSPF, enter the folder `~/comm-protocols-labs/ospf_ring` if you haven't already done so, and run the command

```
sudo ../start.py
```

Remember, it will take about another 30 seconds for the network to fully converge after you start mininet (this is the time needed for OSPF to fully converge across all routers in the network).

For this laboratory, you will be using a tool called `iperf3`, which is a cross-platform network performance testing utility. `iperf3`, which has a number of graphical front-ends such as `flent` and `jperf`, is a command-line tool which can generate TCP or UDP traffic with many different options, and can operate as a client or a server. You need to first start `iperf3` in server mode on one host or router, then connect to it from another instance of `iperf3` running in client mode.

The different `iperf3` command-line options used in this laboratory are summarised in Table 1.

The main options we will be using in this lab are the socket buffer size (`-w`) and the total amount of data to be sent (either specified via the time-limit option `-t` or the data-limit options `-n` or `-k`). `iperf3` generates pseudorandom data and transmits it sequentially using the chosen protocol (either TCP, UDP, or SCTP with the various options that you specify) until the time or data limit is reached. By running it in server mode on one machine and then connecting to it with another instance running in client mode, buffer sizes, TCP flags etc. can be specified individually at each end. While running, it periodically prints out some throughput statistics (by default, once per second).

## 4 Laboratory Tasks

### 4.1 Basic functionality of `iperf3`

1. Open an xterm on hosts `h1` and `h4`. We will use `h1` as the client and `h4` (initially) as the server. Packets will be sent from the client to the server by default. On `h4`, start `iperf3` in server mode:

   ```
   iperf3 -s
   ```

   This will continue to run unless interrupted (via ctrl-C).

2. Open additional xterms on `h1` and `h5`. Start a continuous `ping` to `h5`. The ICMP packets will traverse some of the same links as used by the UDP (or TCP) traffic flow:

   ```
   ping h5
   ```

   We will use this to monitor the effects of TCP and UDP packet flows on network latency.

3. Open a wireshark session on `h1`. Don't start the capture until just before you are ready to start the next task (i.e. running the `iperf` command).

4. Now, on `h1`, run iperf with the following options (immediately after starting Wireshark:

   ```
   iperf3 -c h4 -t 30 -O 5 -C reno -Z
   ```

   This initiates a TCP connection from the client to the server. The client then starts a 30 second data transmission to the server (by default `iperf3` sends data from the client to the server; this can be reversed if you add the `-R` option to the command line).

   The options for this command are as follows: `-c h4` requests `iperf3` to start in client mode and connect to the server running on `h4`. The test will run for a time (`-t`) of 30 seconds (you can increase this if you want), but will omit (`-O`) the first 5 seconds to allow TCP slow start to stabilise (you can check to see what the congestion window size is while iperf3 runs in the

Table 1: Some of the main command-line options of `iperf3`

| **Option** | **Function** |
|---|---|
| `-c servername` | Operate `iperf3` in client mode, connecting to server `servername` |
| `-s` | Operate `iperf3` in server mode (already done on all hosts and routers in the emulated network) |
| `-V` | Provide more verbose output |
| `-t time` | Run the test for *time* seconds |
| `-n size[K\|M\|G]` | Run the test with *size* bytes (or KB, MB or GB if the K, M or G suffix is added) |
| `-k numblocks` | Run the test with *numblocks* blocks (packets) |
| `-C algorithm` | Use TCP variant *algorithm* - one of `reno`, `cubic` (the default) or `bbr` |
| `-Z` | Use a zero-copy data transfer mechanism, reducing CPU overhead (recommended!) |
| `-O time` | Discard the first `time` seconds of data (to eliminate TCP slow-start from the statistics) |
| `-R` | Run in reverse mode - with server sending data to the client |
| `-N` | Use TCP no-delay option, disabling Nagle's algorithm |
| `-u` | Use UDP rather than TCP |
| `-b bandwidth[K\|M\|G]` | For UDP, set the target bandwidth to `bandwidth` in b/s (or Kb/s, Mb/s or Gb/s if the K, M or G suffix is added) |
| `-w windowsize` | Set socket buffer size |
| `-l length[K\|M\|G]` | Buffer size for reading or writing (128 kB by default) |

terminal window). The TCP congestion control algorithm **reno** is selected (`-C`), which is the classical TCP variant described in the lectures (you can also try **cubic** or **bbr**). Finally, the `-Z` option requests that `iperf3` uses a zero-copy method of data transmission to reduce CPU load (recommended, particularly in the VM environment).

The final two lines of output summarise total throughput from the sender and receiver perspective.

Run the command several times. Record the average throughput observed on each run. Do you see a lot of variation either during a run or between runs? Do you observe any impact on the latency reported by `ping`?

5. Stop the Wireshark capture. Filter out the data transfer TCP stream - you can do this with the following filter:

   ```
   tcp.stream eq 1
   ```

   Note: TCP stream zero is a control connection used to negotiate analysis options with the `iperf3` server - if you have trouble identifying the actual data stream, check with the lab demonstrator. Set a reference mark on the first packet in the stream (the one with the SYN flag set). Now go to **Statistics** $\rightarrow$ **TCP Stream Graphs** $\rightarrow$ **Throughput** to plot throughput over time. Can you identify the part of the graph where slow start is operating?

6. Repeat the wireshark analysis with streams using TCP-BBR and TCP-cubic (using the `-C cubic` or `-C bbr` options to `iperf3`). Do you notice any significant difference in TCP behaviour? Qualitatively describe what you observe in each case.

7. Try running the test with iperf3 servers running on several other hosts or routers in the network, both 'near' (e.g. `h1` to `r1`) and 'far'. Do you observe any differences in their behaviour? In particular, observe the maximum value that the congestion window reaches during the run (reported on the client side). How does this change with 'distance' (essentially, delay) in our network (where all links have the same bandwidth - think about bandwidth-delay product; look it up if you aren't sure what this means)?

8. Now repeat the test with the `-u` option added (client side only - leave the server instance running on `h4`):

   ```
   iperf3 -u -c h4 -t 30 -Z
   ```

   (no need to omit the first packets now since we aren't using TCP - there's no slow start).

   This switches the protocol to UDP. By default, packets will be sent with a target overall bandwidth of 1 Mb/s. `iperf3` now reports two other important parameters: packet loss and jitter. Packet loss is just the proportion of packets which are transmitted but which are lost before arriving at the receiver; jitter is the variation in the difference between consecutive packet arrival times. An unloaded network should see zero packet loss and zero jitter. Is that what you observe here?

9. Try increasing the UDP bandwidth from the default. To do this, repeat the preceding comamnd with the `-b bandwidth` option. For example, to produce UDP traffic with a constant bitrate of 10 Mb/s, use the following command:

```
iperf3 -u -c h4 -t 30 -Z -b 10M
```

Try increasing the load until you see some packet loss. Compare the bitrate as observed on the client side and server side as you increase the offered load. At what point do they diverge?

10. How does the latency (as reported by `ping` change as you increase the offered load?

## 4.2 Protocol Fight!

Now we will observe the change in network behaviour as we start to load up the network with multiple traffic flows of different types.

1. Open yet another pair of xterms, this time on `h2` and `h5`. Start an `iperf3` server on `h5`, and establish a long TCP flow (say 1000 seconds) from `h2` to `h5`.
2. Restart the UDP flow from `h1` to `h4`, this time at 10 Mb/s. As soon as it starts, observe the impact on the TCP flow (it may be fairly small). If you prefer, you can also leave the UDP flow running for a long time and run multiple short TCP flows and record the average.
3. In steps of 10 Mb/s, start increasing the UDP load until you are transmitting at 100 Mb/s (you will certainly see some packet loss at this point). What has happened to the TCP flow?
4. Terminate the UDP flow, while leaving the TCP flow running. Instead of UDP, now start a TCP flow from `h1` to `h4`. Do both TCP flows share the link bandwidth roughly equally?
5. Try changing the TCP variant. We have been using `reno` up to this point; try switching both flows to `cubic` and `bbr` (so both flows use the same TCP congestion control algorithm). Is there any change in their behaviour?
6. What happens if you are using `reno` on one flow and `cubic` on the other? Also try `reno` vs. `bbr` and `cubic` vs. `bbr`. Comment on the results.

## References

[1] James F. Kurose and Keith W. Ross. *Computer Networking A Top-Down Approach*. 8th edition, 2020.

[2] Behrouz A. Forouzan. *TCP/IP Protocol Suite*. 4th edition, 2017.