Linux Lab Activity #2: Network emulation and transport layer performance measurements

# author: Aitor Martin Rodriguez ([aitor.martinrodriguez@uis.no](mailto:aitor.martinrodriguez@uis.no))

Department of Electrical Engineering and Computer Science, University of Stavanger

# Objectives

In this lab, you will learn the following items:

* How to emulate a network based on delay, bandwidth, and loss, using *tc/netem*.
* How to measure transport layer performance with *iperf3*.

Later, you will directly use these tools to explore some fundamental concepts of the transport layer:

* Study the impact of latency and packet loss on TCP performance.
* Observe and study TCP congestion control behavior: NewReno, CUBIC and BBR.
* Learn the concept of TCP fairness.

# Background / Scenario

In this activity you will mainly use two tools: **tc/netem** for network emulation and **iperf3** for performance measurement.

**tc/netem** is the default **network emulator** in Linux, and it allows to emulate any real-world network by setting impairments in network interfaces, such as packet delay, packet loss and link bandwidth, among others. This proves very useful for prototyping and testing Layer 4 protocols and network applications, as it provides abstraction of the network elements underneath.

A diagram of a network emulation

Description automatically generated

Figure 1. Concept of network emulation

You can read the full tc/netem documentation in the Linux documentation: <https://man7.org/linux/man-pages/man8/tc-netem.8.html>

# Requirements

* A linux environment, preferably Ubuntu 20.04 or 22.04 LTS.
* The following linux packages:
  + Essential packages: build-essential, net-tools, git
  + Tools for network emulation: mininet, xterm
  + Traffic generation and performance measurement tools: iperf3
  + Other tools for processing and plotting: jq, gnuplot-x11

# INSTRUCTIONS FOR APPROVAL

* Parts 1 and 2 are **MANDATORY FOR EVERYONE**.
* Among parts 3,4 and 5**, YOU MUST COMPLETE AT LEAST TWO PARTS.**

# Instructions

## *Part 1:* Set up the workspace.

In this lab, you will be performing experiments, saving the results and generating plots. To accelerate the process, we have prepared a few bash scripts which automate some tasks (e.g., setting up link emulation, and generating plots from iperf3 results).

There are three scripts:

* **link\_emulation.sh** is used to automate the process of configuring link emulation, which is explained in Part 2. You can use this script in Parts 3-5 to accelerate the process of changing the link emulation parameters.
* **plot\_datarate.sh** is used to plot the received download data rate over time, given an iperf3 trace file in JSON format. You can use it in Part 3 to observe how the TCP download throughput evolves over time.
* **plot\_cwnd.sh** is used to plot evolution of the congestion window (cwnd) over time, given an iperf3 trace file in JSON format. This script only works for iperf3 traces generated from the sender (i.e., the end host that sends the data). You will use this script in Part 4.

In this part, you will download the necessary files and set up the workspace.

### Download assignment materials.

* + - 1. Clone the GitHub repository with the materials: <https://github.com/amartin320/DAT300-Assignment2>

Make sure you write the link correctly!

$ cd /home/$USER

$ git clone <https://github.com/amartin320/DAT300-Assignment2>

* + - 1. To verify if you have execution privileges for the scripts, run each script without any input arguments. If everything is good, you should get a message saying “missing arguments” and explaining how to use the scripts.

$ ./link\_emulation.sh

$ ./plot\_cwnd.sh

$ ./plot\_datarate.sh

* + - 1. If you are missing execution privileges for any script, run the following command:

$ chmod +x <script\_path>

Then try step 1.b. again.

### Organize directory.

* + - 1. Create two folders named **‘results’** and **‘plots’** and inside the DAT300-Assignment2 folder.
* **results** will be the folder where you store the results for each Part 3-5. You can create subfolders for each Part if you wish.
* **plots** will be the directory where the plotting scripts (plot\_datarate.sh and plot\_cwnd.sh) will store the plots in PDF format.

### Install necessary packages.

* + - 1. Install the necessary packages specified in the “Requirements” section above. After LLA#1, you only need to install the following additional packages: **jq** and **gnuplot-x11**. These will be necessary to run the **plot\_datarate.sh** and **plot\_cwnd.sh** scripts.

$ **sudo apt install jq gnuplot-x11**

### Set up the mininet topology.

* + - 1. Start mininet with the same topology you created at the end of LLA#1.

$ **sudo mn --topo=linear,n=2,k=2**

| Device | IP address |
| --- | --- |
| h1s1 | 10.0.0.1 |
| h2s1 | 10.0.0.3 |
| h1s2 | 10.0.0.2 |
| h2s2 | 10.0.0.4 |

## Add link/queue emulation rules with tc/netem

Working with the topology from Part 1, which is depicted in the figure below, we are now going to set up an **emulated link between S1 and S2**. This will allow us to set more realistic end-to-end scenarios where packets have to go through router buffers which introduce delays and cause some packets to be dropped.

Link emulation is implemented using the **tc/netem,** a tool which allows to implement queues in network interfaces. We can therefore emulate a link by setting up custom queues on each side of the link between s1 and s2. (i.e., on interfaces **s1-eth3** and **s2-eth3**).

**These rules queues are implemented only on egress traffic** – i.e., the traffic leaving through that interface, and not the incoming traffic. This means that the network traffic will go through *queue1* on the uplink (from s1 to s2) and through *queue2* on the downlink (from s2 to s1).

**A diagram of a link

Description automatically generated**

To configure the queues in s1-eth3 and s2-eth3, you can use the following commands:

* **Add a queue:** tc qdisc **add** dev <interface> root netem [parameters and values]
* **Modify queue parameters:** tc qdisc **change** dev <interface> root netem [new parameters and values]
* **Remove a queue:** tc qdisc **del** dev <interface> root

All these commands must be run as **sudo.**

### Add delay to the link and verify its effect.

* + - 1. Open a new terminal and use the **tc** tool to verify the queue configuration for the interfaces.

$ tc qdisc show

**Q: Are there any queues configured in the links s1-eth3 and s2-eth3?**

* + - 1. Create a new queue in s1-eth3 and set a delay of 25 milliseconds.

$ sudo tc qdisc add dev s1-eth3 root netem delay 25ms

* + - 1. Repeat the command for the s2-eth3 interface in s2. Doing this, we set up a base round-trip time (RTT) of 50 milliseconds: 25 milliseconds in each direction. Then use the command from Step 1.a. to check that the queues have been appropriately configured.
      2. Measure the latency between h1s1 and h1s2 using *ping*.

**Q: What is the average latency or round-trip time you have measured? Does it match the queue configuration?**

### Set a maximum link bandwidth.

* + - 1. Configure a link with a symmetric bandwidth of 20 Mbps and a round-trip time (RTT) of 50 ms. Note that in this case, we must use the keyword “change” instead of “add” to modify the configuration of the currently existing queues.

$ sudo tc qdisc change dev s1-eth3 root netem delay 25ms rate 20Mbit

$ sudo tc qdisc change dev s2-eth3 root netem delay 25ms rate 20Mbit

* + - 1. Measure the TCP download throughput in Mbit/s with iperf3.

h1s2: $ iperf3 -s -p <your\_port>

h1s1: $ iperf3 -c 10.0.0.2 -p <your\_port> -R

**Q: What is the average throughput measured? Does it match the queue bandwidth configured?**

### Introduce random packet loss.

* + - 1. Modify the previous queues to add a random packet loss of 50% on each queue. NOTE: This is an exaggerated value that is not common in real life. We only use it for demonstration purposes.

$ sudo tc qdisc change dev s1-eth3 root netem delay 25ms rate 20Mbit loss 50%

$ sudo tc qdisc change dev s2-eth3 root netem delay 25ms rate 20Mbit loss 50%

* + - 1. Measure the packet loss between h1s1 and h1s2, sending 500 pings (option -c 500) in 0.1 second intervals (option -i 0.1) and verifying how many responses are received.

**Q: How many ICMP packets have gotten lost? Does it match the packet loss configured in the queues?**

* + - 1. Measure the TCP data rate with iperf3, using the same commands as in Step 2.

**Q: What can you observe when you compare the results to the ones you obtained in Part 2?**

* + - 1. Remove the queues.

$ sudo tc qdisc del dev s1-eth3 root

$ sudo tc qdisc del dev s2-eth3 root

## Study the impact of latency and packet loss on TCP performance.

In this part, the goal is to study how round-trip time (RTT) and packet loss affect TCP performance. To do this, you will perform several tests changing link emulation parameters, as you learnt in Part 2.

You are going to use four different links:

* **Link #1: Short-distance terrestrial link** (e.g., within a region in Norway)
  + RTT = 10 ms
* **Link #2: Long-distance terrestrial link** (e.g., between Norway and central Europe)
  + RTT = 50 ms
* **Link #3: Inter-continental terrestrial link** (e.g., between Europe and USA)
  + RTT = 150 ms
* **Link #4: Geostationary satellite link**
  + RTT = 600 ms

In all cases, we will set the **link** **bandwidth (rate parameter)** to **20 Mbps**.

You can emulate each of these links by setting up queues in s1-eth3 and s2-eth3 with tc/netem, using the commands learnt in Part 2.

* delay: must be configured in each queue as RTT/2.
* rate: fixed to 20Mbit in both queues.
* loss: we will change between 0, 0.1% and 1% in the different steps.

Before starting with Step 1, run the following command in both **h1s1** and **h1s2**:

**$ sysctl net.ipv4.tcp\_congestion\_control=cubic**

This is to make sure that the congestion control algorithm is set to CUBIC, which is the default TCP congestion control. You will learn more about congestion control algorithms in Part 4.

🌟 To make it easier for you, we have prepared a shell script named **“link\_emulation.sh”**, which automatically cleans previously configured queues and configures the new queues according to the parameters given.

How to use the script.

$ **./link\_emulation.sh <interface1> <interface2> [<parameter1> <value1> <parameter2> <value2> …]**

*<interface1> and <interface2> are the names of the interfaces where we want to set the queues.*

*For this assignment, we will use s1-eth3 and s2-eth3.*

*Examples:*

* *Set an RTT of 600 ms: $* ***./link\_emulation.sh s1-eth3 s2-eth3 delay 300ms***
* *In addition, set a bandwidth of 100Mbit/s: $* ***./link\_emulation.sh s1-eth3 s2-eth3 delay 300ms rate 100Mbit***
* *In addition, introduce a packet loss of 0.1%: $* ***./link\_emulation.sh s1-eth3 s2-eth3 delay 300ms rate 100Mbit loss 0.1%***

*NOTE: Make sure to check out the code of the link\_emulation.sh script to understand what it does!*

During Part 3, you must fill the following table with the results you obtain:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Link name** | **Link #1** | **Link #2** | **Link #3** | **Link #4** |
| **Round-trip time (RTT)** | **10 ms** | **50 ms** | **150 ms** | **600 ms** |
| **Loss = 0**  (Step 1) | **Average TCP download throughput [Mbit/s]** |  |  |  |  |
| **Time needed to fully utilize the available bandwidth (s)** |  |  |  |  |
| **Loss = 0.1%**  (Step 2) | **Average TCP data rate [Mbit/s]** |  |  |  |  |
| **Loss = 1%**  (Step 2) | **Average TCP data rate [Mbit/s]** |  |  |  |  |

### Measure TCP performance under ideal link conditions: no packet loss

Using iperf3 between a pair of hosts (e.g., h1s1 and h1s2), measure the TCP performance for each link and complete the table. Since the link emulation is fully symmetric, it does not matter if you measure it on the downlink or on the uplink – i.e., it does not matter in which direction the data flows.

You are going to measure two performance metrics:

* **Average TCP download throughput [Mbit/s]:** This metric measures how fast the download is. At the end of any iperf3 test, an average throughput value is provided. It is important to note down the value on the **receiver side**.
* **The time needed to fully utilize the available bandwidth (s):** This metric measures how long it takes for the TCP connection to reach the maximum available bandwidth (~20 Mbps).

To save the results and plot them, follow these instructions:

* + - 1. Run the iperf3 server in h1s2:

h1s2: $ iperf3 -s -p <your\_port>

* + - 1. Run the iperf3 client in h1s1 with the following options:

h1s1: $ cd /home/$USER/DAT300-Assignment2

h1s1: $ iperf3 -c 10.0.0.2 -i 0.5 -t 30 -p <your\_port> -R -J > results/part3\_loss0\_link1.json

NOTE: The -J option will write the output in JSON format, which can be saved into a JSON file given after the redirection operator (>). This will hide the output from the terminal, but the results will be saved in the JSON file. For each test for different links, you can change the name of the output file accordingly (e.g., link2, link3…).

* + - 1. Run the **plot\_datarate.sh** script, providing the path to the JSON file generated. This will process the file and print the average download throughput in Mbps on the terminal. It will also create a PDF file including a plot of the download throughput over time, which will be stored in the “plots” directory with the same name as the JSON file provided as input.

h1s1: $ ./plot\_datarate.sh results/part3\_loss0\_link1.json

You can then note down the value you obtained in the table above. To measure the “Time needed to fully utilize the available bandwidth (s)”, you can look at the PDF plot generated and give an approximate answer based on that.

**Repeat this process for the four links and fill the first two rows of the table above. Based on your results, answer the questions**.

**Q: How does RTT affect the average TCP throughput? Why?**

*Hint: Think of how TCP connections are established and TCP congestion control mechanisms.*

**Q: Why does the average data rate never reach 20 Mbps?**

*Hint: Think about encapsulation in the TCP-IP stack, and remember that iperf3 is running on the application layer.*

### Measure TCP performance under lossy conditions: loss 0.1% and 1%

**Repeat the previous experiments setting packet loss values of 0.1% and 1%. Fill in the rest of the table with your new results and answer the questions:**

**Q. What impact does packet loss have on TCP performance?**

**Q. Why does only 1% of packet loss cause such a big performance reduction? – i.e., if only 1-2% of the packets are lost, why is the data rate reduced so drastically?**

## Study the impact of TCP congestion control (NewReno, CUBIC, BBR)

TCP congestion control is an essential mechanism for the well-being of the Internet as we know it.

The goal of TCP congestion control is to adjust the data sending rate to the level of congestion in the network:

* **When the network is not congested**, congestion control must aim to utilize the network bandwidth as fully as possible.
* **When the network is congested**, congestion control must be able to dynamically detect congestion and reduce the sending rate, to avoid introducing more congestion.

TCP congestion control is usually based on a **congestion window (*cwnd*).** Different congestion control algorithmsuse different techniques to adapt this *cwnd* to the varying network conditions. In this part, we are going to study three different congestion control algorithms: **TCP NewReno, TCP CUBIC, and TCP BBR.**

**NewReno** (RFC 6582) and **CUBIC** (RFC 8312) are similar in the way they detect congestion: they use packet loss as an indicator of network congestion. The difference between them is the function they use to adjust the cwnd over time and how they behave when they detect loss.

**BBR** was initially proposed by Google researchers in 2016 as an alternative to loss-based congestion control. This algorithm constantly attempts to estimate the bottleneck bandwidth and RTT to create a model of the network and adjust the sending rate according to it. Its goal is to find an optimal sending rate that maximizes bandwidth utilization without saturating the network buffers.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **TCP NewReno** | **TCP CUBIC** (default in Linux) | **TCP BBR** |
| **Standardization** | RFC 6582 | RFC 8312 | IETF draft  draft-cardwell-iccrg-bbr-congestion-control-02 |
| **Supported in Linux since** | Linux 2.6.13 | Linux 2.6.13 | Linux 4.9 |
| **Congestion indicator** | Packet loss | Packet loss. | Bottleneck bandwidth and RTT estimation. |
| **Slow start** | Yes | Yes | Yes |

We are going to observe these three congestion control mechanisms in action.

### Step 0: Calculate the buffer size and set up the queues.

The packets that travel through the network must go through several network devices, which have limited hardware resources and therefore cannot store an infinite number of packets in the queue. This means that all queues will have a limit of bytes that they can store before starting to drop packets. When measuring network performance, it is interesting to observe how different transport layer mechanisms perform under different buffer conditions: e.g., shallow buffers or deep buffers.

An important concept when talking about buffer sizes is the **bandwidth-delay product (BDP)**. The BDP is a value that represents how many bytes can be “in-flight” between sender and receiver before being acknowledged. The higher the round-trip-time (RTT) and the bottleneck bandwidth (BtlBw), the higher will be the number of bytes that must be stored in buffer queues.

In order to ensure that network buffers do not limit the bottleneck bandwidth, we must ensure that the buffer size is at least **1 BDP:**

In this Part, we will work with a link of the following characteristics:

* + - * RTT = 60 ms
      * Bandwidth = 20Mbps
      * Loss = 0%.

To calculate the BDP, we can use the formula above:

We can set a buffer size using the **limit** parameter in tc/netem, specifying the buffer size in “number of packets”. Since by default interfaces use a Maximum Transmission Unit (MTU) of 1500 bytes:

### Step 1: Configure the queues and start monitoring latency.

* + - 1. Configure the queues for a link with the following characteristics:
         1. RTT = 60 ms
         2. Bandwidth = 20Mbit/s
         3. Loss = 0%
         4. Buffer size = 1 BDP = 150 kB = 100 packets

$ ./link\_emulation.sh s1-eth3 s2-eth3 delay 30ms rate 20Mbit loss 0% limit 100

* + - 1. Start a continuous ping from the mininet console and keep it running while you continue with the next steps. It will be useful to observe what happens to the RTT when the network starts to get congested.

mininet> h1s1 ping h1s2

You should measure around between 60 and 70 milliseconds of RTT approximately.

### Step 2: Observe congestion control behaviour with NewReno.

1. Select NewReno congestion control on the sender host, i.e., h1s1.

h1s1: $ sysctl net.ipv4.tcp\_congestion\_control=reno

1. Run an iperf3 test of 60 seconds in upload mode (i.e., without the -R option) and export the results to a JSON file, as you did in Part 3. The option “-i 0.1”is used to get better time granularity in the exported results.

h1s2: $ iperf3 -s -p <your\_port>

h1s1: $ iperf3 -c 10.0.0.2 -i 0.1 -t 60 -p <your\_port> -J > results/part4\_reno\_noloss.json

* + - 1. Note down the average throughput on the receiver side (h1s2) provided by the iperf3 server and write it down in the corresponding field in the table at the end of Part 4.
      2. Use the **plot\_cwnd.sh** script providing the generated JSON file as input. This will create a PDF file including a plot of the evolution of the congestion window (cwnd) over time.

$ ./plot\_cwnd.sh results/part4\_reno\_noloss.json

Observe the plot generated and answer the questions:

**Q: How does the “cwnd” evolve over time in NewReno?**

*Hint: you should observe a “sawtooth” behaviour.*

**Q: Did the latency measured by ping increase during the test? How much?**

* + - 1. Repeat the steps 2.b to 2.d with a packet loss of 1%. Observe the new results and compare them to the no loss scenario.

**How does packet loss affect NewReno congestion control?**

### Step 4: Observe congestion control behaviour with CUBIC.

* + - 1. Select CUBIC congestion control on the sender host, i.e., h1s1.

h1s1: $ sysctl net.ipv4.tcp\_congestion\_control=cubic

* + - 1. Repeat the commands from Step 3 and answer the questions.

**Q: How does the “cwnd” evolve over time in CUBIC?**

*Hint: You should observe several CUBIC functions.*

**Q: Did the latency measured by ping increase during the test? How much?**

* + - 1. Repeat the test with a packet loss value of 1%. Observe the new results and compare them to the no loss scenario.

**Q: How does packet loss affect CUBIC congestion control?**

### Step 5: Observe congestion control behaviour with BBR

* + - 1. Select BBR congestion control on the sender host, i.e., h1s1.

h1s1: $ sysctl net.ipv4.tcp\_congestion\_control=bbr

* + - 1. Repeat the same process as before and answer the questions.

**Q: How does the “cwnd” evolve over time in BBR?**

**Q: Did the latency measured by ping increase during the test? How much?**

* + - 1. Repeat the test with a packet loss value of 1%.

**Q: How does packet loss affect BBR?**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **NewReno** | **CUBIC** | **BBR** |
| **Average TCP download throughput (Mbit/s)** | **Loss = 0%** |  |  |  |
| **Loss = 1%** |  |  |  |

**Reflection question:**

**Q: Based on your results, what are the benefits of BBR when compared to NewReno and CUBIC?**

## Study transport layer fairness: TCP and UDP

In this part, we are going to observe what happens when multiple data flows traverse a shared link and study the concept of **fairness**.

Imagine the following scenario with two flows:

* **Flow A** will go from h1s2 to h1s1.
* **Flow B** will go from h2s2 to h2s1.

**Flows A** and **B** simultaneously go through the bottleneck link (the emulated link).

A diagram of a flowchart

Description automatically generated

The interactions between these flows will depend on the transport protocol specifics, such as protocol choice, congestion control and flow control.

**We say that two or more flows are fair to each other if they share the bandwidth resources *in a fair manner***. There are several metrics to measure fairness, the simplest one being Jain’s Fairness Index, which provides a fairness value between 0 and 1, where ‘1’ represent perfect fairness. You can read about Jain’s Fairness index here: <https://en.wikipedia.org/wiki/Fairness_measure>

You can compute Jain’s fairness index (JFI) for “n” parallel flows using the following equation:

where is the download throughput of the -th flow.

*Example 1:*

*Two parallel flows get a download throughput of 6 Mbit/s and 10 Mbit/s respectively.*

*Example 2:*

*Eight parallel flows get a download throughput of [1, 2, 3, 4, 5, 6, 7,8] Mbit/s respectively.*

### Step 1: Configure the queues.

Set up link emulation with RTT=60ms, 20 Mbit/s bandwidth, no packet loss and a buffer size of 1 BDP. You can use the script you used in previous Parts.

$ **./link\_emulation.sh s1-eth3 s2-eth3 delay 30ms rate 20Mbit loss 0% limit 100**

### Step 2: Observe fairness between TCP CUBIC flows.

iperf3 allows start several parallel flows between one pair of hosts. The **option** **-P** followed by an integer “n” will start **“**n” parallel flows with the same characteristics.

* + - 1. Start an iperf3 test of 30 seconds with two concurrent flows, and measure the average throughput obtained by each flow.

h1s2: $ iperf3 -s -p <your\_port>

h1s1: $ iperf3 -c 10.0.0.2 -p <your\_port> -R -t 30 -P 2

**Q: Is one of the flows always getting more throughput than the other? Do the flows ever “swap places”?**

* + - 1. Repeat the same process 2 more times to get some statistical significance, and fill in the table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Flow 1** | **Flow 2** | **Sum** | **JFI** |
| Run #1 |  |  |  |  |
| Run #2 |  |  |  |  |
| Run #3 |  |  |  |  |
| Average | | | |  |

**Q: What is the average JFI you have measured over the three runs?**

* + - 1. Repeat the process for 4 parallel flows.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Flow 1** | **Flow 2** | **Flow 3** | **Flow 4** | **Sum** | **JFI** |
| Run #1 |  |  |  |  |  |  |
| Run #2 |  |  |  |  |  |  |
| Run #3 |  |  |  |  |  |  |
| Average | | | | | |  |

### Step 3: Observe the latecomer issue between simultaneous CUBIC flows.

The latecomer issue is the effect observed when a new flow enters the link after other flows are already utilizing the link. Depending on the transport layer mechanisms implemented, latecomer flows might be treated very unfairly.

* + - 1. Open the xterm terminals for the 4 hosts: h1s1, h1s2, h2s1 and h2s2. For better understanding, arrange the windows on your screen to match the network topology (i.e., clients on the left side and servers on the right side).
      2. Start Flow A: a very long TCP download between h1s1 and h1s2 (e.g., 600 seconds long).

h1s2: $ iperf3 -s -p <your\_port>

h1s1: $ iperf3 -c 10.0.0.2 -p <your\_port> -R -t 600

* + - 1. While the connection in 1b is running, start a Flow B: a TCP download between h2s1 and h2s2, with a 60-second duration.

h2s2: $ iperf3 -s -p <your\_port>

h2s1: $ iperf3 -c 10.0.0.4 -p <your\_port> -R -t 60

**Q: What do you observe in Flow A when Flow B joins the shared link?**

**Q: How long did it take approximately for Flow B to reach a “fair” share of the bandwidth?**

**Q: What happens to Flow A when Flow B ends?**

### Step 4: Observe simultaneous TCP and UDP traffic sharing a link.

* + - 1. Start Flow A: a very long TCP download between h1s1 and h1s2 (e.g., 600 seconds long).

h1s2: $ iperf3 -s -p <your\_port>

h1s1: $ iperf3 -c 10.0.0.2 -p <your\_port> -R -t 600

* + - 1. When the TCP connection has already stabilized, start Flow B: a UDP download at 15 Mbit/s between h2s1 and h2s2, with a 60-second duration.

h2s2: $ iperf3 -s -p <your\_port>

h2s1: $ iperf3 -c 10.0.0.4 -p <your\_port> -R -t 60 -u -b 15M

**Q: What do you observe in Flow A when Flow B joins the shared link?**

**Q: Would you say Flow A and Flow B have been “fair” to each other when sharing the link?**

**Q: What happens if you stop Flow A? Will Flow B get up to 20 Mbps?**