ECE 1545

Lab 1 - Assignment 3

Packet Scheduling in Linux

Purpose:

Become familiar with the configuration of packet scheduling algorithms in Linux. Conduct experiments to evaluate the impact of scheduler configuration in Linux.

Software Tools:

- The lab/assignment assumes that there is a Ubuntu (or similar) Linux installation. On Windows or Mac systems, you can install Ubuntu as a virtual machine.

What to turn in:

- For the lab (to be completed in the lecture) there is nothing to turn in. The assignment consists of parts 6 and 7, which request to prepare a lab report.

Version 1 (February 2019)

Table of Content

Table of Content 2
Overview 3
Part 1. Software installation 5
Part 2. Iperf3 – a throughput measurement tool 6
Part 3. Piping the iperf3 output 7
Part 4. Visualization 12
Part 5. A script to speed things up 14
Part 6. Measuring scheduling algorithms 15
Part 7. Hierarchical Token Bucket (HTB) 17
startiperf.sh *19*
bokehserver.py *20*
HTB-example.sh *22*

Overview

The objective of this lab is to measure the data rate obtained by a set of flows at a Linux packet scheduler. The setup of the measurements involves a number of components:

- Scheduler (Qdisc) configuration using the tc (traffic control) command;
- Traffic generators and traffic sinks using the iperf3 command;
- Visualizing the transmitted and received traffic using the Python bokeh library;

In this lab/assignment, all components will be run on a single system. Since the senders and receivers are on the same computer, we will send all transmissions to the loopback interface. (The loopback interface – also called localhost – is a virtual network interface with IPv4 address 127.0.0.1. All transmissions to destination address 127.0.0.1 arrive at the local system.)
The next figure gives an overview of the components of the lab.

- A set of iperf3 clients transmits data at a given rate to iperf3 servers via the loopback interface.
- The transmissions by the iperf3 client pass through a Qdisc packet scheduler.
- Both iperf3 clients and servers generate output that plots the amount of data transmitted over a time interval. The output is written to files.
- The data in the files is formatted for the visualization.
- The visualization, done by the Python bokeh library, shows the data rate of the iperf3 clients and servers over a moving time window.

The following exercises introduce the components of the measurement experiment one-by-one. Initially, the scheduler (Qdisc) uses the default FIFO scheduling discipline. Once the measurement infrastructure is in place, we move on to hierarchical scheduling disciplines.

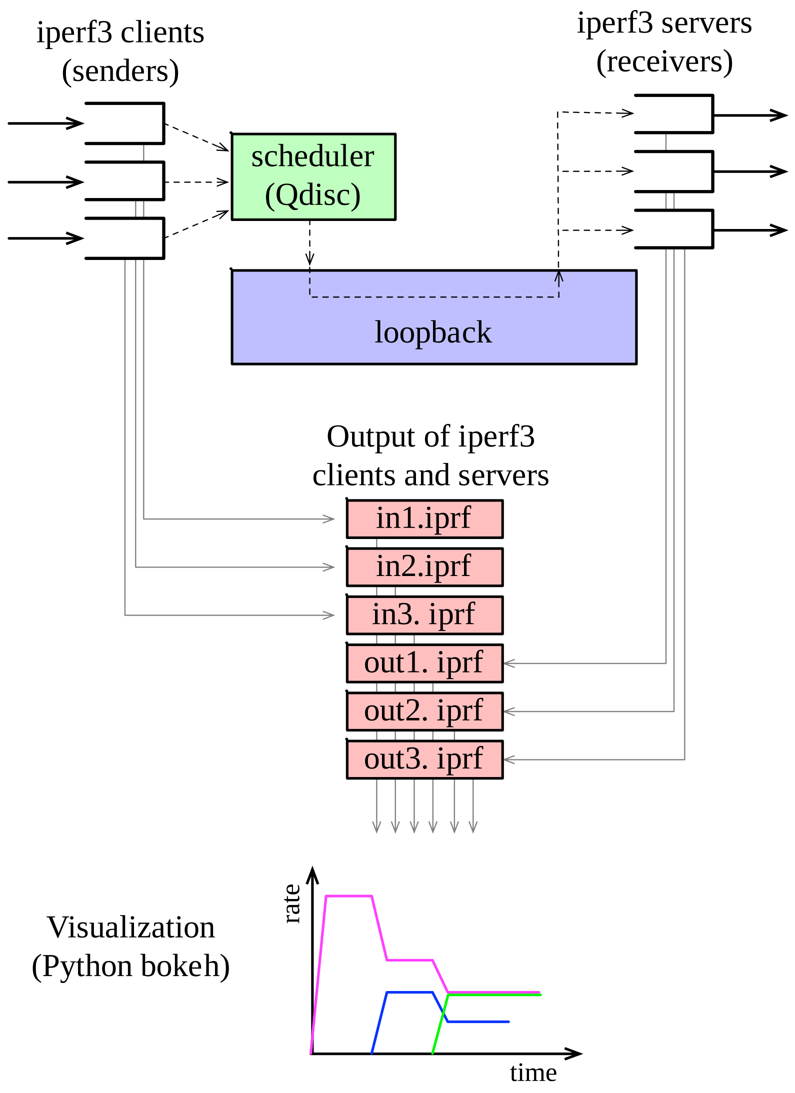


Figure 1. Overview of measurements and their visualization.

Part 1. Software installation

We assume that you have an operational Ubuntu Linux system available. If you are working on a Windows or Mac, you can install the VirtualBox virtualization software and install Ubuntu as a virtual machine.

- Install VirtualBox: https://www.virtualbox.org/wiki/Downloads
- Instructions for installing Ubuntu on VirtualBox: https://askubuntu.com/questions/142549/how-to-install-ubuntu-on-virtualbox

Exercise 1.1 Install iperf3, python and bokeh library

Open a terminal window and type the following commands:

```
$ sudo apt update                  (updates the Ubuntu package manager)
$ sudo apt install iperf3          (installs iperf3)
$ sudo apt install python3         (installs Python)
$ sudo apt install python3-pip     (installs Python package manager)
$ pip3 install bokeh               (installs bokeh library)
```

The bokeh library is installed in the directory ~/.local/bin. To test whether the user account is setup to search in this directory, type

```
$ bokeh -v
```

If the command is not found, then the directory ~/.local/bin must be added to the PATH environment variable. This is done with

```
$ export PATH="$PATH:$HOME/.local/bin"
```

Re-trying

```
$ bokeh -v
```

should now be successful, and display the version of bokeh.

Part 2. Iperf3 – a throughput measurement tool

Iperf3 was designed for throughput measurement over a network. A measurement is done by first setting up an iperf3 server, e.g.,

```
$ iperf3 -s
```

and then start an iperf3 client, e.g.,

```
$ iperf3 -c 127.0.0.1
```

Here, the client transmits messages to the loopback address, that is, client and server are running on the same system. The client transmits data as fast as possible for a duration of 10 seconds. In regular time intervals, client and server display the amount of transmitted data and the data rate of the previous interval.

Iperf3 has many options. Default values are used when options are not specified. Here is a list of options that are relevant for us:

```
   -i value     specifies the elapsed time (in seconds) between reports of
                the data rate (default: 1)

   -f value     specifies units of  displayed data rates
                (k: kbit/s, K: kB/s, m: Mbit/s, M: MB/s, g: Gbit/s, G:
                GB/s)

   --logfile filename
                sends output to file filename

   -u           use UDP (default: TCP). This option is only specified at
                the client.

   -t value     specifies the duration of the measurement (default: 10
                sec)

   -b value M   specifies the target bit rate of transmission in Mbit/s,
                e.g., -b 10M is 10 Mbit/s.
                (default: unlimited for TCP, 1 Mbit/s for UDP; -b 0
                disables the rate control)

   -p value     species the port number of the iperf3 server (Default:
                5201)
```

Exercise 2.1 Throughput Measurements
Familiarize yourself with iperf3 tool by running a set of measurement tests.

Part 3. Piping the iperf3 output

The output of iperf3 has the following format:

```
$ iperf3 -c 127.0.0.1  -f M
Connecting to host 127.0.0.1, port 5201
[  5] local 127.0.0.1 port 53744 connected to 127.0.0.1 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  6.97 GBytes  7140 MBytes/sec
[  5]   1.00-2.00   sec  6.98 GBytes  7146 MBytes/sec
[  5]   2.00-3.00   sec  7.15 GBytes  7324 MBytes/sec
[  5]   3.00-4.00   sec  7.19 GBytes  7360 MBytes/sec
[  5]   4.00-5.00   sec  7.41 GBytes  7585 MBytes/sec
[  5]   5.00-6.00   sec  7.17 GBytes  7341 MBytes/sec
[  5]   6.00-7.00   sec  7.11 GBytes  7278 MBytes/sec
[  5]   7.00-8.00   sec  7.12 GBytes  7295 MBytes/sec
[  5]   8.00-9.00   sec  7.15 GBytes  7318 MBytes/sec
[  5]   9.00-10.00  sec  7.13 GBytes  7302 MBytes/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.00  sec  71.4 GBytes  7309 MBytes/sec         sender
[  5]   0.00-10.00  sec  71.4 GBytes  7308 MBytes/sec         receiver
```

As input for the visualization, we want to write each line of output into a file, such that each new line of output is immediately available as input to the visualization software;

The following exercise show how to accomplish these tasks.

Exercise 3.1 Processing the iper3 output for visualization

Setup iperf3 server with

```
$ iperf3 -s
```

In a separate terminal window, start an iperf3 client that writes the output to file 'mylog'

```
$ iperf3 -c 127.0.0.1 --logfile mylog
```

This will create iperf3 client that write the output data as seen above to file 'mylog'

Assuming that the output of iperf3 is as given above we want to

1. Deliver the output from iperf3 as it occurs in real-time, that is, line by line. For this we resort to Unix mechanisms, called pipes and named pipes, which allow us to concatenate commands in such a way that the output of one command becomes the input to the next command.

Creating a file without content / Deleting content of a file

If the file ‘mylog’ does not exist you will see an error message. In this case, create an empty file with that name with the command

```
$ touch mylog
```

Alternatively, you can open the file with an editor and immediately save and close the file, without writing any characters in the file.

If you run iperf3 multiple times with the same logfile, new entries will be appended to the existing file. To ensure that the logfile is empty and the logfile exists when you start a new iperf measurement, use the command

```
$ >mylog
```

Alternatively, you can open the file with an editor, delete all content, and save the file. If you issue this command when file ‘mylog’ does not exist, the file will be created with no content.

The next figure gives an overview of the task at hand.

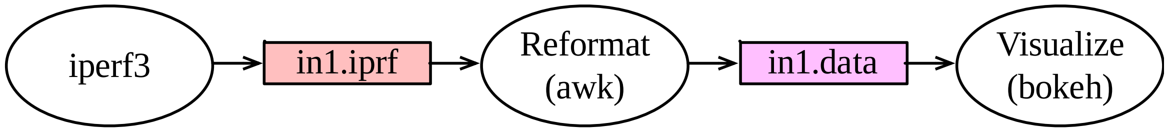


Figure 2. Processing of iperf3 output.

Suppose that iperf3 uses the file “in1.iprf” as logfile. When iperf3 writes a new line to the logfile, we want this line to be immediately made available to bokeh for visualization. This is done by creating a named paper with name “in1.data”.

We complete the output processing illustrated in Figure 2. We first process the output of iperf3 line-by-line as new lines are written to the logfile, and then write the results into a named pipe.

1. Open two terminal windows and type the commands

   ```
   $ iperf3 -s
   $ iperf3 -c 127.0.0.1 --logfile mylog
   ```

   (Type one command in one window and one in the other.) Extend the running time of the client as necessary, e.g., by adding the option `-t 100’.

2. In a separate window, run the command

   ```
   $ tail -f mylog
   ```

   The command `tail’ displays the last (default: 10) lines of a file. With the option `-f ’, lines are displayed as they are appended to the logfile.

   Terminate the command before proceeding to the next step. If necessary, restart the iperf3 client.

3. The final step is to write the output of awk into a named pipe. First we create a named pipe with

   ```
   $ mkfifo mypipe
   ```

   And then use the redirection operator `>' to write the output of tail to the named pipe. This is done by typing

   ```
   $ tail -f mylog > mypipe
   ```

   Convince yourself that the desired content is written to the named pipe by listing its content with

   ```
   $ more mypipe
   ```

   You should see how the size of file 'mypipe' grows when iperf3 is running.

4. To start the iperf3 client and the awk processor at the same time, the command is

   ```
   $ iperf3 -c 127.0.0.1 --logfile mylog & tail -f mylog > mypipe
   ```

Running Multiple Linux Commands in Parallel

(Note: If you are familiar with Unix/Linux, the following description will rightfully appear overly simplified.)

When typing commands in a terminal window, you can only type one command at a time. If you want to run multiple commands in parallel, one option is to open multiple terminal windows and type commands in each of the windows.

An alternative to this is to run commands "in the background". In each terminal window you can run one command in the foreground process and arbitrarily many commans in the background. The commands that you usually type run in the foreground. To start a command in the background, add an ampersand ('&') at the end of the command. For example,

```
$ cmd1 &
```

starts the command `cmd1' in the backround. After typing the command, the terminal window immediately displays a command prompt, which allows you to enter a new command. To execute two commands in parallel, you can therefore type

```
$ cmd1 &
$ cmd2
```

If you want to start both commands at the same time, you enter

```
$ cmd1 & cmd2
```

You will get a new command appears after both commands are completed.   If you add an ampersand after `cmd2', you immediately get a command prompt.

If you are running multiple commands in the background you sometimes want to forcefully terminate one of these commands. One option to do this is to list the currently running processes with the command `ps' and then terminate a process with the command `kill -9'.

For example,

```
$ ps
        PID      TTY          TIME CMD
        4886     ttys000      0:00.34 -bash
        12279    ttys002      0:00.08 -bash
        37293    ttys002      8:40.97 iperf3 -s
$ kill -9 37293
```

This terminates the `iperf3 -s' process.

A generally more convenient way to control commands in execution is to work with `jobs'. In Unix, every command that is entered at a command prompt constitutes a `job'. You can control the execution of jobs as follows.

1. Listing current jobs

```
$ jobs
```

This provides a list of currently active jobs. Each entry has a job number (1,2,…) and a status (Running, Stopped, Done).

2. Terminating a job in the background

```
$ kill %1
```

This terminates the job with job number 1.

3. Stopping and terminating the foreground job

```
$ Ctrl-Z
```

Stops the foreground job

```
$ Ctrl-C
```

Terminates the foreground job.

4. Moving a job from the foreground to the background

```
$ Ctrl-Z
$ bg
```

5. Moving a job from the background to the foreground

```
$  %1
```

This moves the job with job number 1 to the foreground.

Unix Pipes and Named Pipes

We next address the issue of running a sequence of commands such that the output of one command becomes the input of another command. This can be done by using the Unix concept of a pipeline, or, in short a **pipe**. Let us suppose we want to list all files in a directory whose names contain the substring "bin". To list all files in a directory, we use the command

```
$ ls
```

To list all lines of a file with name file1 that contain the string "bin", we can use the command

```
$ grep bin file1
```

Using a pipe, we can concatenate the two commands to obtain a command

```
    $ ls | grep bin
```

The symbol “|” defines a pipe. Now, the input to the “grep” command is the output of the “ls” command.

A **named pipe**, also called a FIFO special file, is a version of the Unix pipe, that appears as a file in the operating system. That is, the named pipe has a name, it is located in a directory, and can be read from or written to just as a regular file. Suppose we have a named pipe with filename “mypipe”. We can `pipe’ two processes by having the first process write to “mypipe” and the second process read from “mypipe”. The advantage of named pipes over regular files is that

- named pipes always reside in main memory; and
- data written into the named pipe is immediately available for reading.

This is not ensured when using regular files. A named pipe with name `mypipe’ is created with the command

```
    $ mkfifo mypipe
```

Example: To see how a named pipe works open two terminal windows. In one window, type

```
    $ ls -l > mypipe
```

Then, in the other window, type

```
    $ cat < mypipe
```

The result is that the command `cat’, which displays the content of a file), reads and displays the content of the directory from the first window. Now you display the file before and after issuing the ‘cat’ command with

```
    $ more mypipe
```

you will see that the named pipe contains the output of `ls -l’ before it is read, and it is empty after it has been read. If ‘mypipe’ had been a regular file, the content would not have been cleared by reading its content.

Part 4. Visualization

We will show how to visualize the transmitted and received data from an iperf3 measurement.

Exercise 4.1 Visualization with bokeh

1. Set up logfiles and named pipes for the iperf3 client and the iperf3 server. The filenames are:

in1.iprf	-- logfile for the client
out1.ipfrf	-- logfile for the server
data/in1.data	-- named pipe for client
data/out1.data	-- named pipe for server

   To ensure that the files exist and the logfiles are empty issue the commands

```
$ >in1.iprf
$ >out1.iprf

$ mkdir data

$ mkfifo data/in1.data data/out1.data
```

   Note that you need to create the directory data and the named pipes only once.

2. Download the Python file

   https://www.comm.utoronto.ca/~jorg/teaching/ece1545/labs/serverbokeh.py

   and place it in the directory `data'.

3. Go to directory data and start a bokeh server with the commands:

```
$ cd data
$ bokeh serve bokehserver.py
```

4. Start a web client, e.g., Firefox, and type the following URL:

   http://localhost:5006/bokehserver

5. Start an iperf3 server and an iperf3 client that each write to a logfile, and write them to the named pipes.

```
$ iperf3 -s -i 0.1 -f bytes -p 10000 --logfile out1.iprf  &  \
     tail -f out1.iprf > data/out1.data

$ iperf3 -c 127.0.0.1  -i 0.1 -u -f bytes -p 10000 -t 100 -b 10M  \
            --logfile in1.iprf & tail -f in1.iprf > data/in1.data
```

   We set the options so that iperf3

- runs the client for 100 sec,
- transmits data using UDP,
- sets the targeted transmission rate of the client to 10 Mb/sec,
- creates an output line every 0.1 sec, and
- writes the data rate in bits/sec

Note that:

- If a command gets really long and you want to break it up into several lines, you can create a linebreak with a backslash (\).

Part 5. A script to speed things up

The following file is a bash script file that starts an iperf3 server and an iperf3 client, and writes the output into a named pipe as done in Part 3 and Part 4. The bash script file takes four parameters, and is started with the command

```
$ ./startiperf.sh class length rate offset
```

where

- **class** is the class identifier which is used for filenames and to construct the port number;
- **length** is the length of experiment in seconds;
- **rate** is the target rate of the client in Mb/sec;
- **offset** is a time offset for the the start of experiment in sec.

This file is available at
https://www.comm.utoronto.ca/~jorg/teaching/ece1545/labs/startiperf.sh

The script cleans up files and creates directories and named pipes as necessary. When the iperf3 client has completed, all processes started by the script – including the iperf3 server – are terminated.

After downloading the file, change the permission so that the file can be run as a program. Suppose the file is downloaded to your home directory. You change the permission with the commands:

```
$ cd ~
$ chmod 700 startiperf.sh
```

The 'chmod' command makes the file readable, writeable, and executable by the owner of the file.

Part 6. Measuring scheduling algorithms

Exercise 6.1 FIFO

Setup the following scenario and visualize the results:

- Use FIFO scheduling;
- Use netem to limit the maximum rate of the scheduler to 10 Mbps;
- Generate three UDP traffic flows with iperf3 which sent traffic as follows:
    - Flow 1: Sends at rate 2 Mbps, starts at t=0 sec, stops at t=60 sec;
    - Flow 2: Sends at rate 4 Mbps, starts at t=10 sec, stops at t=50 sec;
    - Flow 3: Sends at rate 8 Mbps, starts at t=20 sec, stops at t=40 sec.

Save a screen capture of the complete experiment visualization to a file.

When all flows are active (20 sec ≤ t ≤ 40 sec), determine the bandwidth obtained by each flow. Relate the bandwidth obtained by a flow to the rate at which a flow is sending.

Exercise 6.2 Priority Scheduling

Repeat the previous experiment with a static priority scheduler:

- Use PRIO scheduling;
  Recall: By default a prio qdisc has three classes with minor numbers "1", "2", and "3", where "1" is the highest priority, and "3" is the lowest priority.
- Use netem to limit the maximum rate of the scheduler to 10 Mbps;
- Generate three UDP traffic flows with iperf3 which sent traffic as follows:
    - Flow 1 (class 1): Sends at rate 2 Mbps, starts at t=0 sec, stops at t=60 sec;
    - Flow 2 (class 2): Sends at rate 4 Mbps, starts at t=10 sec, stops at t=50 sec;
    - Flow 3 (class 3): Sends at rate 8 Mbps, starts at t=20 sec, stops at t=40 sec.
- Note: If a class has minor number "1", the script "startiperf.sh" sets the port number for that class to "10001". If a class has minor number "2", the script "startiperf.sh" sets the port number for that class to "10002".

Save a screen capture of the complete experiment visualization to a file.

When all flows are active (20 sec ≤ t ≤ 40 sec), determine the bandwidth obtained by each flow. Explain the results.

Exercise 6.3 DRR (Deficit Round Robin)

Repeat the previous experiment with the DRR scheduler:

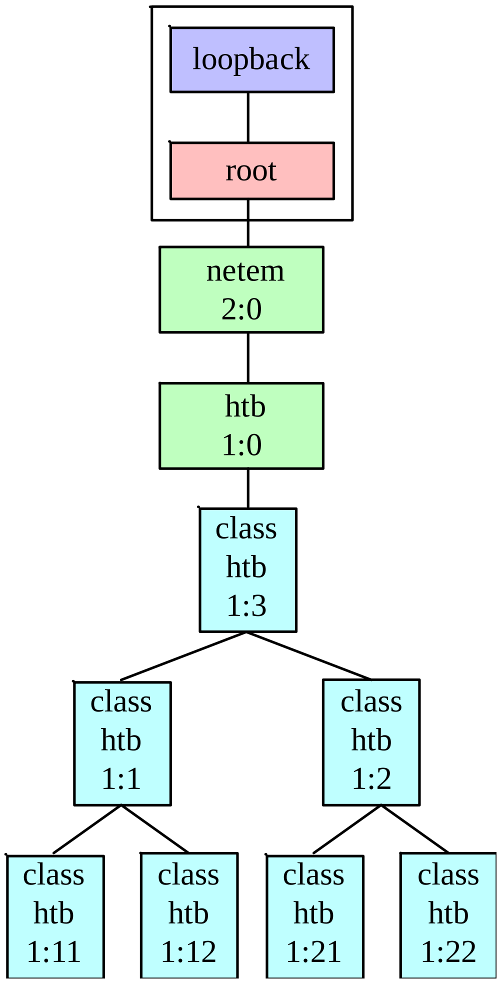
- Use DRR scheduling;
- Use netem to limit the maximum rate of the scheduler to 10 Mbps;
- You need to define a class as well as filter for each flow.
  Note: If a class has minor number “1”, the script “startiperf.sh” sets the port number for that class to “10001”. If a class has minor number “33”, the script “startiperf.sh” sets the port number for that class to “10033”.
- Generate three UDP traffic flows with iperf3 which sent traffic as follows:
  - Flow 1: Sends at rate 2 Mbps, starts at t=0 sec, stops at t=60 sec;
  - Flow 2: Sends at rate 4 Mbps, starts at t=10 sec, stops at t=50 sec;
  - Flow 3: Sends at rate 8 Mbps, starts at t=20 sec, stops at t=40 sec.

Save a screen capture of the complete experiment visualization to a file.

When all flows are active (20 sec ≤ t ≤ 40 sec), determine the bandwidth obtained by each flow. Relate the bandwidth obtained by a flow to the rate at which a flow is sending.

Part 7. Hierarchical Token Bucket (HTB)

The final example creates a hierarchical token bucket with classes as shown here:



Exercise 6.1 HTB

Setup the HTB scheduler with parameters for the classes as follows:

Class Id	Assured Rate (Mbps)	Ceiling Rate (Mbps)	Burst Size (Bytes)
1:3	10	10	4000
1:1	5	10	2000
1:2	5	10	2000
1:11	2.5	10	1000
1:12	2.5	10	1000
1:21	1	10	1000
1:22	4	10	1000

Note: A configuration script for this setup is given at:

https://www.comm.utoronto.ca/~jorg/teaching/ece1545/labs/HTB-example.sh

Your task is to setup (5) different experiments to evaluate whether HTB provides a hierarchical fair bandwidth allocation. Flows may start and stop at different times.

- Include a scenario, where the bandwidth demand of each flow exceeds the assured rate.

- Include a scenario, where 1:12 has a bandwidth demand of 10 Mbps, and no other flow is transmitting.

- Include a scenario, where the bandwidth demands are:

1:21	10 Mbps	start time: 0 sec
1:22	1 Mbps	start time: 0 sec
1:12	5 Mbps	start time: 20 sec
1:11	does not send	

- Include two other scenarios of your own design.

For each scenario, save a screen capture of the complete experiment visualization to a file, and provide a discussion of the expected versus the observed outcomes.

Appendix

startiperf.sh

```
#!/bin/bash

# sets up an iperf3 sender and client
# writes output to a named pipe in directory ./data
# Arguments:
#  $1  - flow id used for filenames and port
#  $2  - length of experiment in seconds
#  $3  - target rate  of client in Mb/sec
#  $4  - offset of start of experiment in sec

# Check arguments
if [ $# -ne 4 ]; then
  echo "ERROR: Incorrect number of arguments."
  echo " 1.  Class id used for filenames and port"
  echo " 2.  length of experiment in seconds"
  echo " 3.  target rate  of client in Mb/sec"
  echo " 4.  offset of start of experiment"
  exit 1
fi

# create "./data" directory, if necessary
if [ ! -d "data" ]; then
  echo "Directory './data' does not exist  ... will be created"
  mkdir data
fi

# Portnumber is flow 10000 + classID
portno=$(($1 + 10000))


# Clear output files
>in${1}.iprf
>out${1}.iprf

# Create named pipes in ./data and delete if they exist
rm -f data/${1}.in
rm -f data/${1}.out
mkfifo data/${1}.in
mkfifo data/${1}.out


# Start iperf3 server
iperf3 -s -i 0.2  -f bytes -p $portno --logfile out${1}.iprf  &  tail -F out${1}.iprf  --retry >
data/${1}.out &

sleep $4

# Start iperf3 client
# We reverse the order of iperf3 and tail so that we can run iperf3 in foreground
tail -F in${1}.iprf --retry > data/${1}.in & iperf3 -c 127.0.0.1  -i 0.2 -u  -f bytes -p $portno -t
${2} -b ${3}M --logfile in${1}.iprf


# once client has completed, kill all processes started by this script
echo "Done with client of class ${1}. Kill all processes"
pkill -P $$
```

bokehserver.py

The following file is a python script file that runs the visualization with the bokeh library. The command is started by with the command

```
$ bokeh serve bokehserver.py
```

```
from bokeh.plotting import figure, curdoc
from bokeh.driving import linear
from bokeh.models.sources import ColumnDataSource
from bokeh.models import DataRange1d
from bokeh.palettes import Spectral

from queue import Queue, Empty
from threading import Thread
from collections import defaultdict
import os
import re

# parameter
rate_factor = 1000 # Conversion factor from bps
rate_unit = 'kbps' # Unit to print to y label

time_window = 30 # time window for default boundary, in second
y_max = None # upper bound on y axis, may be None, in `rate_unit`
base_path = '.' # path to search for files, '.' for directory at start up
max_data_point = None # Max number of point to plot in each file, or None for unlimited

# List all files in this directory
paths = sorted(os.listdir(base_path))
names = [ os.path.basename(path) for path in paths ]

# Split file names into first and last name using first `.` as separator
name_pattern = re.compile('(\S+?)(\.\S*)?')
match = [ name_pattern.fullmatch(name) for name in names ]
first_names = sorted(set([ m.group(1) for m in match ]))
last_names = sorted(set([ m.group(2) or '' for m in match ]))

# Set color/style for each first/last name
available_colors = Spectral[min(max(3, len(first_names)), 11)]
available_styles = ['solid', 'dashed', 'dotted', 'dotdash']
colors = { name: available_colors[i % len(available_colors)] for (i, name) in enumerate(first_names) }
styles = { name: available_styles[i % len(available_styles)] for (i, name) in enumerate(last_names) }

# Shared queue
queue = Queue()

# This thread reads each file, line-by-line, try to parse it to match `data_pattern` and not
`reject_pattern` and submit parsed data to `queue`
data_pattern = re.compile('\A\[\s*\d*\]\S*\s+\d*.?\d*-
(\d*.?\d*)\s+sec\s+\d*.?\d*\s+(?:[KMGT]?(?:Byte|bit))s\s+(\d*.?\d*)\s+([KMGT]?)(Byte|bit)s/sec\s')
reject_pattern = re.compile('(?:sender|receiver|local|remove)\\n?\Z')
def filler(name, path):
    try:
        with open(os.path.join(base_path, path), 'r') as f:
            for line in f:
                match = data_pattern.match(line)
                if not match or reject_pattern.search(line):
                    continue # Skip if doesn't match, or match reject_pattern

                time = float(match.group(1))
                rate = float(match.group(2))
                # Kilo, Mega, Giga, Tera
                rate *= {'': 1, 'K': 1e+3, 'M': 1e+6, 'G': 1e+9, 'T': 1e+12}[match.group(3)]
                # Byte/bit
                rate *= {'Byte': 8, 'bit': 1}[match.group(4)]

                # Submit to queue
                queue.put((path, time, rate / rate_factor))

    except IOError as e:
        print(f'\nError reading {os.path.join(base_path, path)} with error: {e}\n')

# Set default zoom to [end-time_window, end] on x-axis, and [0, y_max] on y-axis
p = figure(sizing_mode='stretch_both', x_range=DataRange1d(follow='end', follow_interval=time_window),
y_range=DataRange1d(start=0, end=y_max))
# Set grid color, label, etc.
```

```
p.grid.minor_grid_line_color = '#eeeeee'
p.grid.grid_line_color = '#888888'
p.yaxis.axis_label = f"Rate ({rate_unit})"
p.xaxis.axis_label = "time (s)"

found = False

sources = {}
# Create lines to draw to graph, and their data sources
for name, path in zip(names, paths):
    if os.path.isdir(os.path.join(base_path, path)): continue # ignore directories

    # Create thread to read the file
    thread = Thread(target=filler, args=(name, path))
    thread.daemon = True
    thread.start()

    # Get first/last name for line color/style
    match = name_pattern.fullmatch(name)
    assert(match)
    first_name = match.group(1)
    last_name = match.group(2) or ''

    # Create data source, line
    sources[path] = ColumnDataSource({'x': [], 'y': []})
    p.line(x='x', y='y', source=sources[path], line_width=2, line_color=colors[first_name],
line_dash=styles[last_name], legend=dict(value=name))
    found = True

if found:
    # You need to have at least one legend, otherwise these 2 lines will crash. Make sure you created
at least one line
    p.legend.location = "top_left"
    p.legend.click_policy = "hide"

@linear()
def update(step):
    new_data = defaultdict(lambda: {'x': [], 'y': []})

    # Keep reaading from `queue`
    while True:
        try:
            name, time, size = queue.get_nowait()
        except Empty:
            # Queue empty
            break

        # Group data by source (each file has its own source)
        # data will be dictionary {'x': xdata, 'y': ydata} and xdata, ydata will have same size
        new_data[name]['x'].append(time)
        new_data[name]['y'].append(size)

    if not new_data:
        return

    for (name, data) in new_data.items():
        # Push new data onto appropriate source
        sources[name].stream(data, rollover=max_data_point)

# Add `p` to current document and setup callback
curdoc().add_root(p)
curdoc().add_periodic_callback(update, 100) # Interval in ms
```

HTB-example.sh

```
# Editing Loopback Interface
PATH="$PATH:/usr/bin:/sbin:/bin"
INT=lo
echo Configuring interface $INT

# Delete current Interface attached to `root`
sudo tc qdisc del dev $INT root

# Add `netem` with major id 2:
sudo tc qdisc add dev $INT root handle 2: netem rate 10Mbit

# Add htb `qdisc` with classid 1:, to class 2: (netem)
sudo tc qdisc add dev $INT parent 2: handle 1: htb

# Add class 1:3 to qdisc 1: with rate/ceil/burst parameter
sudo tc class add dev $INT parent 1: classid 1:3 htb rate 10Mbit ceil 10Mbit burst 4000

sudo tc class add dev $INT parent 1:3 classid 1:1 htb rate 5Mbit ceil 10Mbit burst 2000
sudo tc class add dev $INT parent 1:1 classid 1:11 htb rate 2.5Mbit ceil 10Mbit burst 1000
sudo tc class add dev $INT parent 1:1 classid 1:12 htb rate 2.5Mbit ceil 10Mbit burst 1000

sudo tc class add dev $INT parent 1:3 classid 1:2 htb rate 5Mbit ceil 10Mbit burst 2000
sudo tc class add dev $INT parent 1:2 classid 1:21 htb rate 1Mbit ceil 10Mbit burst 1000
sudo tc class add dev $INT parent 1:2 classid 1:22 htb rate 4Mbit ceil 10MBit burst 1000

sudo tc filter add dev $INT parent 1: protocol arp u32 match u32 0 0 flowid 1:11 # arp
sudo tc filter add dev $INT parent 1: protocol ip u32 match ip dport 10000 0xffff classid 1:11
sudo tc filter add dev $INT parent 1: protocol ip u32 match ip dport 10001 0xffff classid 1:12
sudo tc filter add dev $INT parent 1: protocol ip u32 match ip dport 10002 0xffff classid 1:21
sudo tc filter add dev $INT parent 1: protocol ip u32 match ip dport 10003 0xffff classid 1:22

# IDs are used to identify qdisc, class and filters
# ID consists of 2 parts; 16bit major number, and 16bit of minor number, written as major:minor.
# Special IDs:
# - root: 65535:65535 (all ones)
# - unspecified: 0:0 (all zeros)
#
# qdisc is identified by only major number (called `handle`)
# - 2: for netem and 1: for htb above
#
# class id uses both major number and minor number
# - major number is the same as qdisc it belongs to
# - minor number (called classid) can be arbitrarily assigned, there is no rule regarding this.
# - For example
#   - 1:4 corresponds to a class in qdisc 1: with class id 4
#
# Current setting:
#                 *---------------*
#                 |interface: root|
#                 *---------------*
#                         |
#                   *-----------*
#                   |netem 2:   |
#                   *-----------*
#                         |
#     *-----------------------------------------*
#     |htb qdisc 1:    -------                  |
#     |                   |                     |
#     |              *---------*                |
#     |              |root(htb)|                |
#     |              |   1:3   |                |
#     |              *---------*                |
#     |                   |                     |
#     |          *--------*----------*          |
#     |          |                   |          |
#     |      *-------*           *-------*      |
#     |      |A      |           |B      |      |
#     |      |  1:1  |           |  1:2  |      |
#     |      *-------*           *-------*      |
#     |          |                   |          |
#     |     *----*----*         *----*----*     |
#     |     |         |         |         |     |
#     |  *-----*   *-----*   *-----*   *--_--*  |
#     |  |A1   |   |A2   |   |B1   |   |B2   |  |
#     |  | 1:11|   | 1:12|   | 1:21|   | 1:22|  |
#     |  *-----*   *-----*   *-----*   *-----*  |
#     *-----------------------------------------*
```