Lab

2a

Traffic Shaping (Part 1)

Purpose of this lab:

In this lab you will build (program) a network element for traffic shaping, called a token bucket, that runs over a real network. The traffic for testing the token bucket will be the Poisson, VBR video, and LAN traffic traces from Lab 1.

Note: Lab 2b will be a continuation of this lab.

Software Tools:

- This lab uses Python with the built-in `socket` module.
- Sample code is provided.

What to turn in:

- On Crowdmark: A report with your answers to the questions in this lab, including the plots.
- On Quercus: Your modified code files.

January 2026

Table of Content

Lab 2

Provided python code and data files: In addition to files with traffic trace data, we provide reference code for the Python programs. The files are available on Quercus. Use the Python code as starting points for your lab work.

- **Part 1:** **Code:** *Sender.py* *Receiver.py* *ReadWriteFile.py*
  Trace data: *data.txt* *movietrace.data*

- **Part 2:** **Trace data:** *poisson-lab2a.data*

- **Part 3:** **Code:** *byte_queue.py* *bucket_receiver.py*
  bucket_sender.py *token_bucket.py*
  Trace data: *poisson-lab2a.data* *movietrace.data* *BC-pAug89.TL.Z*

Part 1. Programming with Datagram Sockets and with Files

The purpose of this part of the lab is to become familiar with programming Datagram sockets and with reading from and writing to a file. The programs provided here intend to offer guidance for the programming tasks needed later on.

Exercise 1-a. Programming with datagram sockets

Run the following two programs. The program *Sender.py* transmits a string to the receiver over a datagram socket. The program *Receiver.py* displays the string when it is received.

- **Sender.py**

```
import socket
import sys

if len(sys.argv) != 3:
    print(f"Usage: {sys.argv[0]} <hostname> <message>")
    sys.exit(1)
host = sys.argv[1]
message = sys.argv[2].encode('utf-8')  # convert string to bytes

addr = (host, 4444)

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock.sendto(message, addr)  # Send the datagram
sock.close()
```

- **Receiver.py**

```
import socket

# Create a UDP socket bound to port 4444
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(("", 4444))   # "" means all local interfaces

print("Waiting for a message on UDP port 4444 ...")

data, addr = sock.recvfrom(256)  # Receive up to 256 bytes
message = data.decode('utf-8')  # Decode bytes to string

print(f"Received '{message}' from {addr}")
```

Step 1: Compile the programs.
Step 2: Start the receiver by running "python3 Receiver.py".
Step 3: Assuming that the receiver is running on a host with IP address 128.100.13.131, start the sender by running:

```
python3 Sender.py 128.100.13.131 “My String”
```

The receiver program should now display the string "My String".

Determining the IP address
Replace the IP address 128.100.13.131 with the IP address of the computer where Receiver.py is running. If sender and receiver are running on the same system, you can use the loopback address 127.0.0.1 or use the name "localhost".

If the receiver is running on a different system, you can use the command *ifconfig* (on Linux or Mac) or *ipconfig* (on Windows) to list the configured IP addresses.

Step 4: Repeat this exercise, with the difference, that you run the sender and receiver on two different hosts.

Exercise 1-b. Reading and Writing data from a file

Step 1: Download the Java program *ReadFileWriteFile.py*. The program involves the pandas software library. Pandas that you encountered in Lab 1. The program reads an input file *data.txt* which has entries of the form

```
0   0.000000     I   536    98.19092.17092.170
4   133.333330   P   152    98.19092.17092.170
1   33.333330    B   136    98.19092.17092.170
       …         …          …
```

Step 2: The file is read line-by-line, the values in a line are parsed and assigned to variables. Then the values are displayed and written to a file with name *output.txt*.

Step 3: Run the program with the VBR video trace (*movietrace.data*) from Lab 1 as input.

Step 4: Modify the program so that it computes and displays the average size of the following frame types:

- I frames;
- P frames;
- B frames.

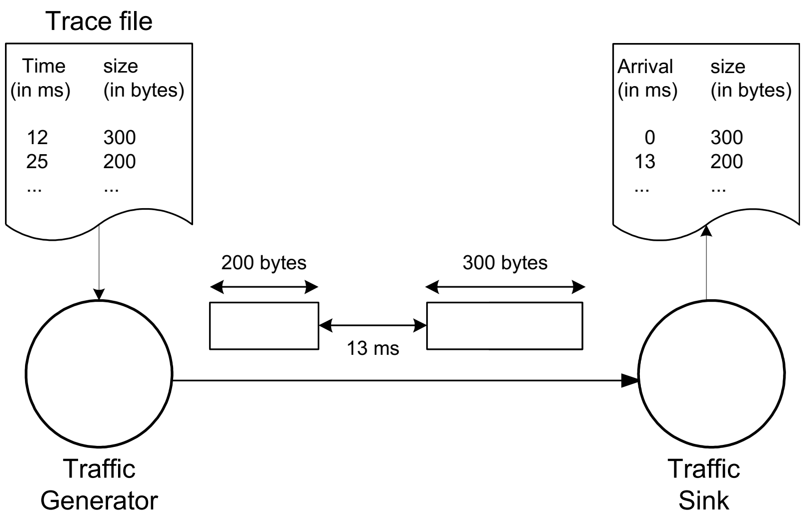
Lab Report

- For this part, only submit the code for Exercise 1-b from Step 4.

Part 2. Traffic generators

The goal in this part of the lab is to build a traffic generator that emulates realistic traffic sources.

The traffic generator is driven by a traffic trace file, i.e., one of the trace files from Lab 1. The entries in the file permit to determine the size of transmitted packets and the elapsed time between packet transmissions. For each entry in the trace file, the traffic generator creates a UDP datagram of the indicated size and transmits the datagram to the traffic sink. The traffic sink records time and size of each incoming datagram, and writes the information into a file. The scenario is depicted in the figure below.



Exercise 2-a. Traffic Generator for Poisson traffic

Write a program that is a traffic generator for the compound Poisson traffic source (similar to Lab 1, Exercise 1-b) and that transmits the traffic to a traffic sink via a datagram socket. The traffic sink has to be programmed as well.

- The traffic trace file for the compound Poisson source is in *poisson-lab2a.data*. The file has the format:

SeqNo	Time (in msec)	Size (in bytes)
1	0	30
2	1	99
3	2	27
…		

  The average data rate of the traffic source is 1 Mbps. The file contains only 200 packet arrivals. Note that some entries have the same timestamp. Packets with the same timestamp are batch arrivals that are transmitted back-to-back.

- The traffic generator reads each line from the trace file, re-scales the values, and transmits a UDP datagram packet using the following considerations:
  - The size of the transmitted datagram is equal to the (re-scaled) packet size value;

    - The time of transmission is determined from the (re-scaled) time values. (The first packet should be transmitted without delay).

Exercise 2-b. Build the Traffic Sink

Step 1: Write a program that serves as traffic sink for the traffic generator from the previous exercise. The requirements for the traffic sink are as follows:

- Read packets from a specified UDP port.
- For each incoming packet, write a line to an output file that records the size of the packet and the time since the arrival of the previous packet (For the first packet, the time is zero).

Step 2: Test the traffic sink with the traffic generator from Exercise 2-a.

Exercise 2-c. Evaluation

Step 1: Run experiments where you transmit traffic from the traffic generator to the traffic sink. Evaluate the accuracy of the traffic generator by comparing the entries in the trace file (at the traffic generator) to the results written to the output file (at the sink).

Step 2: Prepare a plot that shows the difference of trace file and the output file. For example, you may create two functions that show the cumulative arrivals of the trace file and the output file, respectively, and plot them as a function of time.

Step 3: Try to improve the accuracy of the traffic generator. Evaluate and graph your improvements by comparing them to the initial plot.

Exercise 2-d. Account for packet losses.

Packet losses may occur due to bit errors, buffer overflows, collisions of transmissions, or other reasons. Packet losses are less likely if both the sender and the receiver are on the same machine. The UDP protocol does not recover packet losses.

Step 1: Indicate in your plots from the evaluation any packet losses.

Lab Report:

- Provide the plots as requested in Exercise 2-c and Exercise 2-d with a description.

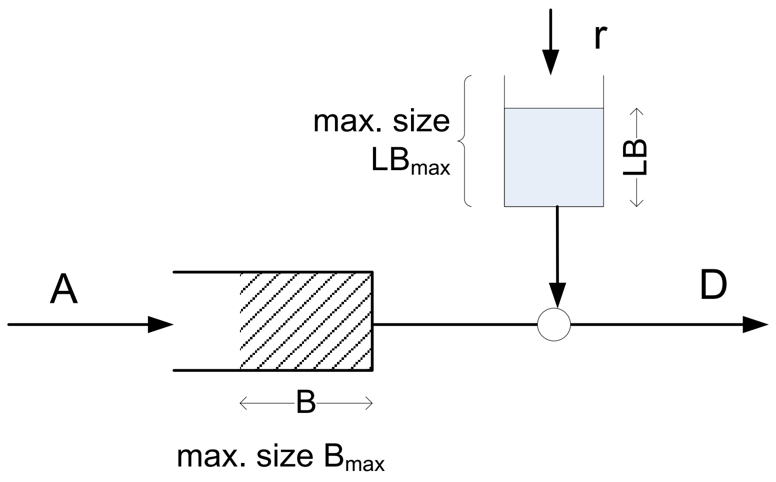
Part 3. Token Bucket Traffic Shaper

In this part of the lab, you familiarize yourself with a reference implementation of a Token Bucket. The implementation of the token bucket (in python) is available on Quercus.

In the next lab (Lab 2b) you will modify the source code of the implementation. Right now, the objective is to run the code of the reference implementation, but without modifying the provided source code.

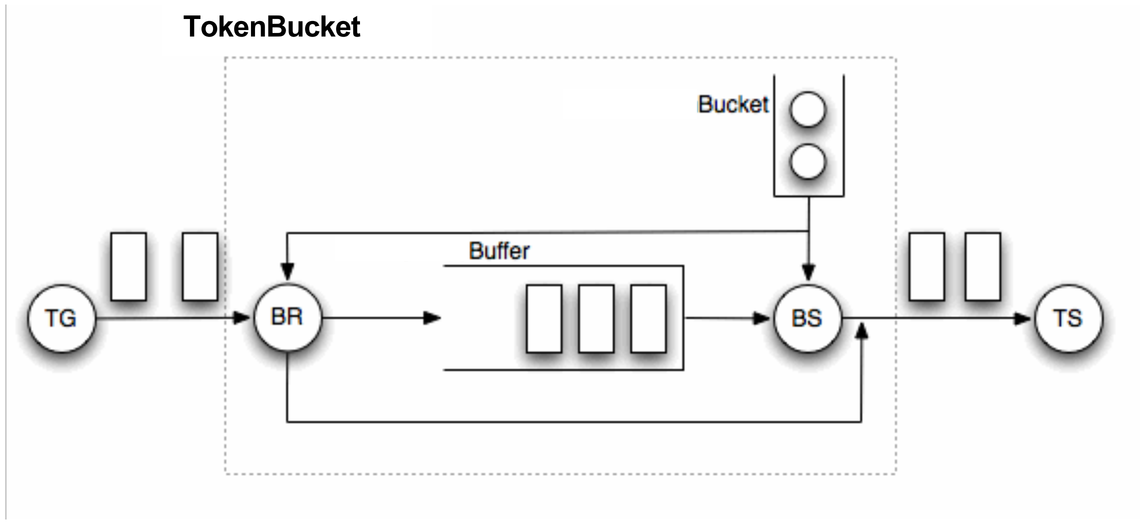
A token bucket traffic shaper with burst size TB_{max} and rate r is shown in the figure. Tokens are fed into the bucket at rate r. No more tokens are added if the bucket contains TB_{max} tokens. Data can be transmitted only if there are sufficient tokens in the bucket. To transmit a packet of L bytes, the bucket must contain at least L tokens. If there are not sufficient tokens, the packet must wait until there are enough tokens in the bucket. The maximum size of the buffer is B_{max}.

Initially, the token bucket is full and the buffer is empty, i.e., $TB(t) = TB_{max}$ bytes and $B(t) = 0$ bytes. Note that L should not be smaller than the maximum packet size.



The token bucket for this Lab receives data from a traffic generator on a UDP port and transmits the output to a traffic sink, specified in terms of an IP address and a UDP port. This is illustrated below.

An implementation of the token bucket for packet sizes with a variable length is sketched in the following figure. The traffic generator (TG) generates packets that are transmitted to the token bucket, which sends the packets to the traffic sink (TS) after applying a shaping policy. The token bucket is comprised of four components: Bucket Receiver (BR), Bucket Sender (BS), the Bucket (containing the tokens), and a Buffer. Upon arrival to the token bucket, packets are handled by BR which stores packets into the buffer or sends them out immediately. The Buffer works as a FIFO queue with limited capacity and holds packets until they can be sent. The Bucket generates tokens at a given rate (r) and holds up to TB_{max} tokens. BS takes packets out of buffer and sends them when enough tokens are available.



The reference implementation of the Token Bucket implements the above functions, but which is missing details on the method for updating the value of TB and for computing the time t* for waking up the Bucket Sender. This implementation is provided by the Java classes Sender, Receiver, and TokenBucket.

- **`TokenBucket`**
  Creates the components of the token bucket and starts BR, BS, and TokenBucket in different threads. An instant of TokenBucket is created with the following parameters:
    - `in_port` – UDP Port number, where BR expects arriving packets.
    - `out_ip` - IP address to which BS sends packets.
    - `out_port` – UDP port number to which BS sends packets
    - `bucket_size` – Maximum number of tokens in token bucket (TBmax).
    - `bucket_rate` - Token generating rate in bytes/sec (r).
    - --`max-packet-size` - Maximum UDP packet size in bytes
    - --`buffer-capacity` - Capacity of buffer in bytes (Bmax).
    - --`logfile` - Name of file, where arrival times are recorded.

- **`ByteQueue`**
  Thread safe implementation of a FIFO buffer that is meant to be used in a producer/consumer scenario. The capacity of the buffer, denoted as MAX_BYTES, can be specified explicitly in bytes. The buffer has methods for adding packets to the tail of the buffer and removing packets from the head of the buffer, peeking at the first packet (without removing it), and querying the currently occupied buffer size (in terms of packets or bytes).

- **`TokenBucketReceiver (BR)`**
  Waits on a specified UDP port for incoming packets. When a packet arrives, it records its arrival time, size, buffer backlog (B), and the number of tokens in the bucket (TB) to a file. A received packet is processed in the following way:

```
 if (buffer_is_empty && not(sendingInProgress) && enough_tokens_available)
        consume tokens;
        send packet;
 else
        add packet in buffer;
```

  Whether a transmission is in progress is checked by locking a mutual exclusion (mutex) variable snd_lock. Note that TokenBucketReceiver and TokenBucketSender never send at the same time, since TokenBucketReceiver only transmits when the buffer is empty. If a packet cannot be added to the buffer, e.g., because the buffer is full, the packet is dropped and an error message displayed.

- **`TokenBucketSender (BS)`**
  Removes packets from buffer and transmits them to a specified address (IP address and UDP port), when there are enough tokens. The procedure for transmission is as follows:

```
If (buffer_is_not_empty)
     if (enough_tokens)
            consume tokens;
            remove packet from buffer;
            send packet;
     else
            get expected time when there will be enough tokens;
            sleep for this time;
else
     wait for packet to arrive to buffer;
     // buffer wakes up BS when packet arrives
```

- **`TokenBucket`**
  This class updates the content of the tokens in the token bucket. Each token corresponds to one byte. A token bucket is created with two parameters size and rate, where
    - size is the maximum content of the token bucket (TB_{max}),
    - rate is the token generation rate in tokens per second (r).

  Other classes can query the TokenBucket about the filling level of tokens, and can request to remove tokens from the bucket. The following requests are available:
    - `getNoTokens`: Returns number of tokens in bucket.
    - `removeTokens(X)` : Request to remove `X` tokens from the bucket. The method returns false if there are less than X tokens in the bucket.
    - `getWaitingTime(X)`: Returns the waiting time (in milliseconds until bucket has `X` tokens. Note that there is no guarantee that there will be enough tokens at the returned time (someone else may have removed tokens).

Exercise 3-a. Running the reference implementation of the Token Bucket

Step 1: Your task is to run the reference implementation of the token bucket so that it receives packets from your implementation of the traffic generator from Part 2 and send packets to your version of the traffic sink.

- The transmissions between traffic generator, token bucket, and traffic sink use UDP datagrams.
- The size and timing of packet transmissions by the traffic generator is done as described in Part 2.
- Upon each packet arrival, the token bucket regulator writes a line to an output file that records the size of the packet and the time since the arrival of the last packet (For the first packet, the time is zero). Also recorded are the number of tokens (TB) in the token bucket and the backlog in the buffer (B) after the arrival of the packet.

In your implementation, the traffic sink, the token bucket, and the traffic generator must be started separately as independent processes.

The TokenBucket of the reference implementation is started with the command:

```
python3 token_bucket.py <inport> <outIP> <outport> <bucket size> <bucket rate>
```

where

`inport`	– port number where TokenBucketReceiver listens to arrivals
`outIP, outport`	– IP address and port number of the traffic sink
`bucket size`	– maximum number of tokens in the token bucket
`bucket rate`	– rate at which tokens are added to the token bucket.

This command starts a token bucket, which listens on *<inport>*, and forwards packets to *<outIP>* at *<outport>*. You may add the following optional arguments:

`--max-packet-size:` maximum packet size in bytes (default: 1480 Bytes)
`--buffer-capacity:` size of the FIFO buffer in bytes (default: 200kB)
`--logfile:` the name of the log file to store the arrival time (default: arrivals.log)

Exercise 3-b. Evaluate the reference implementation for Poisson traffic

Consider the transmission of Poisson traffic from Exercise 2-c.

Step 1: Prepare a single plot that shows the cumulative arrival function as a function of time of:

- The data of the trace file (as read by the traffic generator);
- The arrivals at the token bucket;
- The arrivals at the traffic sink.

Step 2: Provide a second plot that shows the content of the token bucket and the backlog in the Buffer as a function of time.
Note: Since the given token bucket always has enough tokens, there is generally no backlog. There can be small backlog if multiple packets arrive back-to-back.

Exercise 3-c. Evaluate the reference implementation for the Ethernet and Video Tracefiles

Step 1: Repeat the evaluation for the video traffic trace and the Ethernet trace file that were used in Lab 1, and which are available at:

- Video traffic: *movietrace.data*
- Ethernet traffic: *BC-pAug89.TL.Z*

There are a few things to note:

- The format of the trace files for the Video and the Ethernet traffic is described in the description of Lab 1.
- For the video trace file, the size of a video frame may exceed the maximum size of a datagram, and you need to send the frame in multiple datagrams.
- You may truncate the files and only consider the first 1000 entries.

Lab Report

- Provide the source code of your traffic generator and traffic sink in Exercise 3-a. Do not include source files of the reference implementation.
- For Exercises 3-b and 3-c, provide the set of plots and include a description of the plots.

Feedback Form for Lab 2

The lab has been revised for the W2026 semester. We appreciate any feedback on any issues that you may have encountered, e.g.,

- Instructions that are in need of clarification;
- Issues with provided Python code;
- ...

If you have feedback, please add it to your lab report as an appendix.

Thank you!