ECE 1545

Assignment 1

Traffic Shaping

Purpose:

Build (program) a network element for traffic shaping, called a **token bucket**, that runs over a real network.

Software Tools:

- The programming for this lab is done in Java and requires the use of *Java datagrams*.

What to turn in:

- Turn in a report with your answers to the questions in this lab, including the plots, and your Java code.

Version 1 (January 27, 2007)
Version 2 (January 28, 2011)
Version 3 (January 25, 2013)
Version 4 (January 2019)

Table of Content

Preparation

This lab requires network programming with Java datagram sockets. There are numerous informative and short tutorial on Java datagrams is available at:
https://www.baeldung.com/udp-in-java

Java datagram sockets use the UDP transport protocol to transmit traffic. The relationship between Java datagrams and the UDP protocol is described in:
http://www.roseindia.net/java/example/java/net/udp/

Java: Java can be downloaded from https://www.java.com/en/download/

Java for Beginners: There are numerous tutorials available on the Internet for beginners or advanced users. For absolute beginners, the following online course comes highly recommended: https://www.udemy.com/java-tutorial/

Comments

- **Quality of plots:** This assignment asks you to produce plots for a report. It is important that the graphs are of high quality. All plots must be properly labeled. This includes that the units on the axes of all graphs are included, and that each plot has a header line that describes the content of the graph.

Part 1. Programming with Datagram Sockets and with Files

The purpose of this part is to become familiar with programming Datagram sockets and with writing Java programs that read and write data to/from a file. The programs provided in this part intend to offer guidance for the programming tasks needed later on.

Exercise 1.1 Programming with datagram sockets

Compile and run the following two programs. The program *Sender.java* transmits a string to the receiver over a datagram socket. The program *Receiver.java* displays the string when it is received.

Sender.java

```
import java.io.*;
import java.net.*;
public class Sender {
    public static void main(String[] args) throws IOException {
    InetAddress  addr = InetAddress.getByName(args[0]);
    byte[] buf  = args[1].getBytes();
    DatagramPacket packet =
                new DatagramPacket(buf, buf.length, addr, 4444);
    DatagramSocket socket = new DatagramSocket();
    socket.send(packet);
  }
}
```

Receiver.java

```
import java.io.*;
import java.net.*;
public class Receiver {
  public static void main(String[] args) throws IOException {
    DatagramSocket socket = new DatagramSocket(4444);
    byte[] buf = new byte[256];
    DatagramPacket packet = new DatagramPacket(buf, buf.length);
    System.out.println("Waiting ...");
    socket.receive(packet);
    String s = new String(p.getData(), 0, p.getLength());
    System.out.println(p.getAddress().getHostName() + ": " + s);
  }
}
```

- Compile the programs.
- Start the receiver by running “java Receiver”.
- Assuming that the receiver is running on a host with IP address 128.100.13.131, start the sender by running:

  java Sender 128.100.13.131 “My String”
- The receiver program should now display the string “My String”.
- Repeat this exercise, with the difference, that you run the sender and receiver on two different hosts.

Exercise 1.2 Reading and Writing data from a file

Download the Java program *ReadFileWriteFile.java*. The program reads an input file "data.txt" which has entries of the form

```
0     0.000000     I      536    98.190 92.170 92.170
4     133.333330   P      152    98.190 92.170 92.170
1     33.333330    B      136    98.190 92.170 92.170
      …            …             …
```

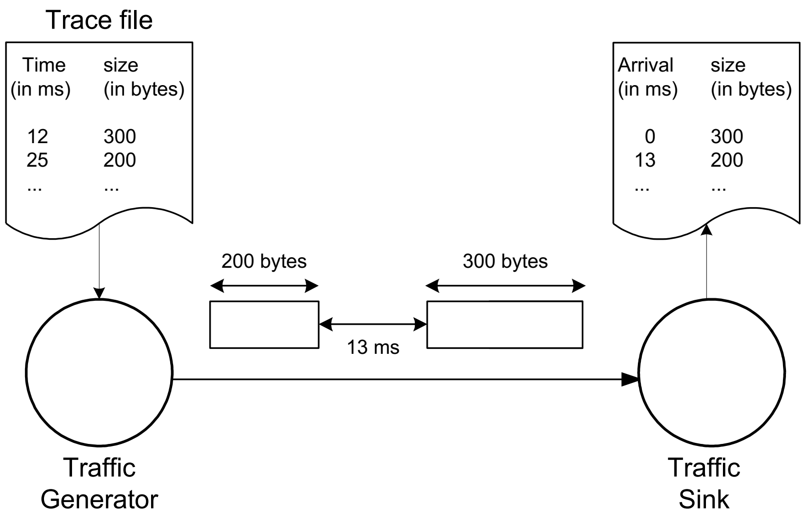
The file is read line-by-line, the values in a line are parsed and assigned to variables. Then the values are displayed, and written to a file with name *output.txt*.

- Run the program the VBR video trace from available at: http://www.comm.utoronto.ca/%7Ejorg/teaching/ece466/labs/lab1/movietrace.data

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. 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.



In this part, you will build a traffic generator for the trace files with Poisson traffic. In Part 3, you work with the Bellcore trace file. In Part 4, you use the VBR video traffic.

Exercise 2.1 Traffic Generator for VBR video traffic

Write a program that is a traffic generator for the VBR video trace used in Part 1, and that transmits traffic via Java datagrams to a traffic sink. The traffic sink has to be programmed as well.

Note: The size of a video frame may exceed the maximum size of a datagram, and you need to send the frame in multiple datagrams.

Exercise 2.2 Build the Traffic Sink

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).

- Test the traffic sink with the traffic generator from Exercise 2.1.

Exercise 2.3 Evaluation

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).

- 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.

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

Exercise 2.4 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.

- Indicate in your plots from the evaluation any packet losses.

Assignmnt Report:

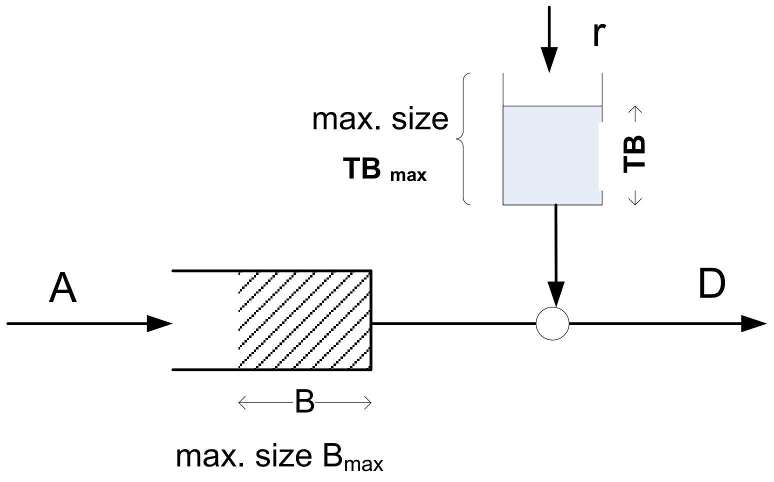
Provide discussions and graphs as requested in Exercise 2.3 and Exercise 2.4.

Part 3. Token Bucket Traffic Shaper

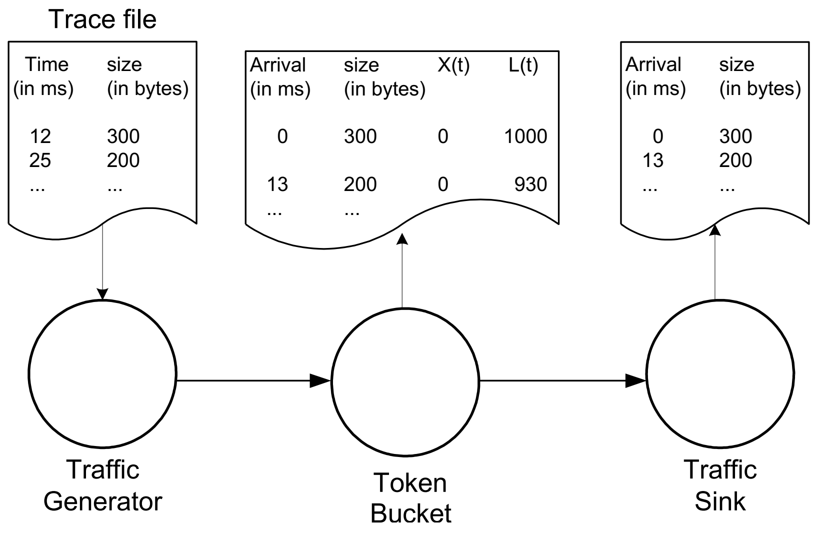
In this part, you familiarize yourself with a reference implementation of a Token Bucket. The implementation of the token bucket (in Java) and documentation for using the implementation is available from the course web page.

Right now, the objective is to run the code of the reference implementation, but without a need to modify the provided source code.

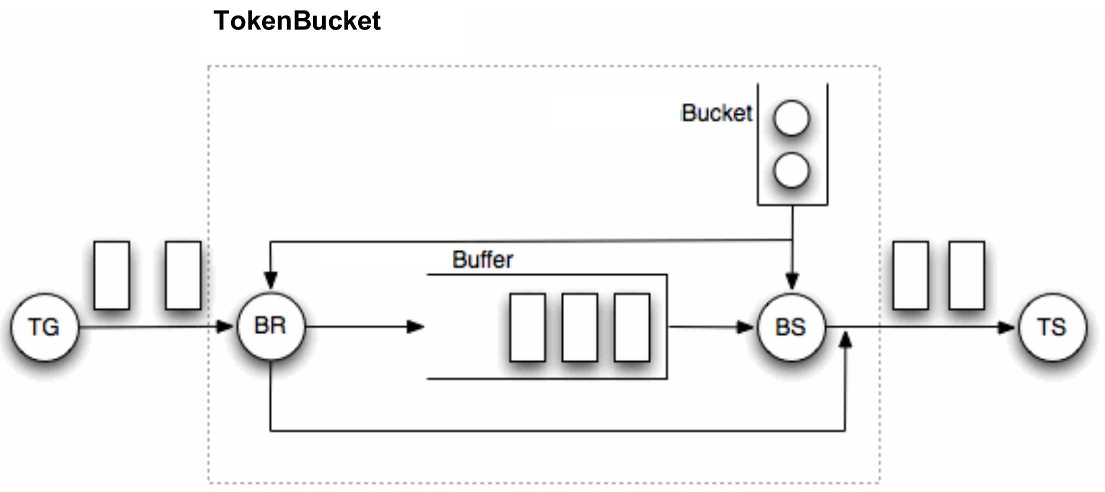
A token bucket traffic shaper with burst size TB_{max} and rate r is shown in the figure. Tokens are fed into the bucket at rate r. No more tokens are added if the bucket contains TB_{max} tokens. Data can be transmitted only if there are sufficient tokens in the bucket. To transmit a packet of L bytes, 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 a 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 send. 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 package TokenBucket, which consists of four Java classes:

- **Class 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:

    - `inPort` – UDP Port number, where BR expects arriving packets.
    - `outAddress` - IP address to which BS sends packets.
    - `outPort` – UDP port number to which BS sends packets
    - `maxPacketSize` - Maximum size of packet that can be processed.
    - `bufferCapacity` - Capacity of buffer in bytes (B_{max}).
    - `bucketSize` – Maximum number of tokens in token bucket (TB_{max}).
    - `bucketRate` - Token generating rate in tokens/sec (r).
    - `fileName` - Name of file, where arrival times are recorded.

- **Buffer**
  Thread safe implementation of a FIFO buffer that is meant to be used in producer/consumer scenario. This buffer stores packets that must not exceed *MAX_PACKET_SIZE* and has inherent capacity of *MAX_VALUE* packets. The capacity of the buffer 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 reading the SendingInProgress variable of BucketSender. Note that BucketReceiver and BucketSender never send at the same time, since BucketReceiver 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;
          sendingInProgress = true;
          remove packet form buffer;
          send packet;
          sendingInProgress = false;
    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
```

- **Bucket**

This class updates the content of the tokens in the token bucket. 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 request the Bucket about its content 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 nanoseconds) 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 3.1 Running the reference implementation of the Token Bucket

Your task is to execute 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 following code segment shows how you start the TokenBucket of the reference implementation in a process. The implementation assumes that the class TokenBucket is in a subdirectory with name “TokenBucket”:

```
import TokenBucket.TokenBucket;

public class Main {

    public static void main(String[] args)  {
    // listen on port 4444, send to localhost:4445,
    // max. size of received packet is 1024 bytes,
    // buffer capacity is 100*1024 bytes,
    // token bucket has 10000 tokens, rate 5000 tokens/sec, and
    // records packet arrivals to  bucket.txt).
    TokenBucket lb = new TokenBucket(4444, "localhost", 4445,
                   1024, 100*1024, 10000, 5000, "bucket.txt");
     new Thread(lb).start();
   }
}
```

Exercise 3.2 Evaluate the reference implementation for the VBR video trace

Consider the transmission the VBR video trace used in Parts 1 and 2.

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.

Provide a second plot that shows the content of the token bucket and the backlog in the Buffer as a function of time.

Assignmnt Report:

- For Exercises 3.2 provide the set of plots and include a description of the plots.

Part 4. A Reference Traffic Generator

In this part you create a simple traffic generator that can be used for testing and tuning your token bucket implementation. This reference traffic generator creates a predictable periodic traffic pattern of UDP datagrams. The traffic generator takes as input parameters three values *T, N, L,* with the interpretation:

> "Every *T* msec transmit a group of *N* packets back-to-back, each with a size of *L* bytes"

If the group of N packets are sent quickly, then the traffic generator sends traffic according to a cumulative arrival function A with:

$$\begin{aligned} A(t) &= \lceil t/T \rceil \cdot N \cdot L \ \text{ Bytes} \\ &= \lceil t/T \rceil \cdot N \cdot L \cdot 8 \ \text{ Bits} \end{aligned}$$

The long term average traffic rate of the traffic generator is given by $Rate_{source} = NL8/T$ bps (bits per second).

For each transmitted packet, write a line to an output file that records the size of the packet and the time since the transmission of the previous packet (For the first packet, the time is zero), e.g., the file has the format:

```
SeqNo        Time since last arrival         Size
                  (in µsec)               (in Bytes)

  1                     0                    320
  2                   934                     99
  3                  2293                     27
               ...
```

Exercise 4.1 Create and Test Reference Traffic Generator

Build the traffic generator by modifying the implementation of the traffic generator Part 2.

Run experiments for specified values of T, N, and L, where you transmit traffic from the traffic generator to the traffic sink (without a token bucket). 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).

- Use at least 10,000 data points for your evaluation.
- 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.

Assignmnt Report:

- Include the plot from Exercise 4.1 and include a description of the plots.

Part 5. Token Bucket Implementation

You are asked to implement the missing features from the token bucket implementation provided to you on the course web page.

You only need to modify the implementation of the TokenBucket class. The provided implementation does not correctly update the value of TB and does not correctly compute the time for waking up the Bucket Sender. An instance of the TokenBucket is created by calling the constructor of this class with two parameters *size* and *rate*, where

- `size_TB` is the maximum content of the token bucket (TB_{max}),
- `rate_TB` is the token generation rate in tokens per second (r).

Other classes can query the TokenBucket about the content of the token bucket, 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 nanoseconds) 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 5.1 Complete the implementation of the Token Bucket

Complete the implementation of the TokenBucket class by providing the algorithms for updating the contents of the token bucket, and executing the requests `getNoTokens, removeTokens, getWaitingTime.`

Hints:

- The contents of the token bucket does not need to be kept updated at all times. It is sufficient to update the token bucket only when there is a request made to TokenBucket.
- Be consistent about whether a single token corresponds to a Bit or a Byte.
- Refer to the documentation of the classes available on the course web page.

Exercise 5.2 Validate your implementation

Design and conduct a series of measurement experiments that show that your implementation of the token bucket is correct. The setup of the experiments is as seen in Part 3.

Use the reference traffic generator from Part 4 to create a deterministic traffic stream with known properties.

When you conduct a measurement experiment, record relevant data (using the code from previous parts):

- **At the traffic generator:** For each transmitted packet, record the size of the packet and the time since the previous packet transmission (For the first packet, the time is zero).
- **At the Token Bucket:** Upon each packet arrival, write 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(t)) in the token bucket and the backlog in the buffer (B(t)) after the arrival of the packet.
- **At the traffic sink:** 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).

Use a packet size L=100 Bytes for all experiments. For the rate parameter at the traffic generator ($Rate_{source}$ < rate_TB) and at the Token Bucket ($Rate_{source}$ < rate_TB), select rates in the range 0.5 – 5 Mbps.

Conduct the following measurement experiments:

1. **$Rate_{source}$ < rate_TB, N =1, size_TB = 100 Bytes:**
   Here, the long term rate of the source is smaller than the rate allowed by the token bucket. With N=1, the traffic source generates single packets that are equally spaced.

   Expected observations: The Token Bucket has always sufficient tokens. There is never a backlog in the Buffer.

2. **$Rate_{source}$ < rate_TB, N =10, size_TB = 500 Bytes:**
   Here, the long term rate of the source is smaller than the rate allowed by the token bucket. The maximum burst size at the source is 10 packets, however, the token bucket allows bursts to consist of at most five packets.

   Expected observations: Whenever a burst (group) of 10 packets arrives, the Token Bucket releases 5 packets quickly, and then spaces the remaining packets out as determined by the rate parameter. The backlog in the Buffer depends on the rate at which the burst of packets arrives, and the rate at which the Token Bucket can transmit a packet. The maximum backlog should be somewhere between 5-9 packets.

3. **$Rate_{source} \approx$ rate_TB, N =1, size_TB = 100 Bytes:**
   Here, the arrival rate of traffic is approximately equal to the long term rate of the Token Bucket. (To prevent persistent overflows, it is recommended that the arrival rate is a little smaller than the rate of the Token Bucket).

   Expected observations: This setting can be used to test if your Token Bucket implementation can support the data rates of the source. If the Token Bucket can keep up with the arrival rate, then there should never be a backlog in the Buffer. On the other hand, if the Buffer builds up and eventually overflows, then the Token Bucket implementation is too slow.

Exercise 5.3 Generate plots for the experiments

Use the data saved in the measurement experiments to generate a series of plots. For each experiment above, prepare a single plot that shows the cumulative arrival function as a function of time of:

- The transmissions of the reference traffic generator;
- The arrivals at the token bucket;
- The arrivals at the traffic sink.

For each experiment, provide a second plot that shows the content of the token bucket and the backlog in the Buffer as a function of time.

Exercise 5.4 Maximum rate of Token Bucket

Determine the maximum arrival rate of traffic that can be supported by your Token Bucket. Support your findings with an experiment and (at least) one data plot.

Assignment Report:

- Describe the design of your implementation for the Token Buffer. Include your source code in the lab report. (Only include the source code of methods that you have modified).

- For Exercises 5.2 and 5.3, provide the set of plots and include a description of the plots. Describe your observations in each of the experiments.

- For Exercise 5.4 describe how you determined the maximum supported arrival rate, and discuss any graph that you generated to support your findings.

Part 6. Engineering Token Bucket Parameters

The task in this part is to determine token bucket parameters for the video traffic source from Part 1.
The tracefile is available from

- http://www.comm.utoronto.ca/~jorg/teaching/ece466/labs/lab1/movietrace.data

The Token Bucket parameters must be selected subject to (all) of the following constraints:

- The objective is to smooth the traffic as much as possible, by spacing out packets, but without delaying traffic by more than 100ms (milliseconds).

- The required size of the buffer in the Token Bucket should be as small as possible, but the buffer should be large enough so that no overflows occur.

- The rate of the token bucket should be at least equal to the average rate of the traffic source, and less than the peak rate of the traffic source (In Lab 1, average and peak rate where calculated for each of the traffic sources).

There is generally a whole family of solutions for size_TB and rate_TB that can satisfy the above constraints. Your choice of parameters must trade-off the desire to smooth the traffic rate (small value for rate_TB) or to reduce the burst size of the output of the token bucket (small value for size_TB)

Note: Using the topics covered in the lecture, you can compute the required parameters without resorting to trial and error. To do this, you need to compute the empirical envelope of each traffic source, and use the service curve S(t) = size_TB + rate_TB * t.

Exercise 6.1 (Long-term) Bandwidth is cheap

Determine the token bucket parameters that satisfy the above constraints for the video traffic source. When you have to choose between increasing rate_TB or size_TB, your preference should be to increase rate_TB.

- For the parameters that you select, generate a plot that shows the cumulative arrival function as a function of time for

    - arrivals at the token bucket;

    - arrivals at the traffic sink.

  Also plot the content of the token bucket and the backlog in the Buffer as a function of time.

Exercise 6.2 (Long-term) Bandwidth is expensive

Repeat the steps of Exercise 3.2, but with a different assumption. Now, when you have to choose between increasing rate_TB or size_TB, your preference should be to increase size_TB.

Assignment Report:

- Discuss your method for selecting the token bucket parameters.
- For Exercises 6.1 and 6.2, provide the set of plots and include a description of the plots. Describe your observations in each of the experiments. Compare the results for the cheap bandwidth and expensive bandwidth scenarios.

Part 7. (Optional) Shaping VBR video with Multiple Token Buckets

The regulation of VBR video traffic with a token bucket has to address the following two problems:

- Since the size of a frame can be much larger than the maximum packet size (in the video trace file, the maximum frame size exceeds 600 KB), yet the time lag between frames is relatively long, one would like to select token bucket parameters that stretch the transmission of a frame over the entire 33 ms.

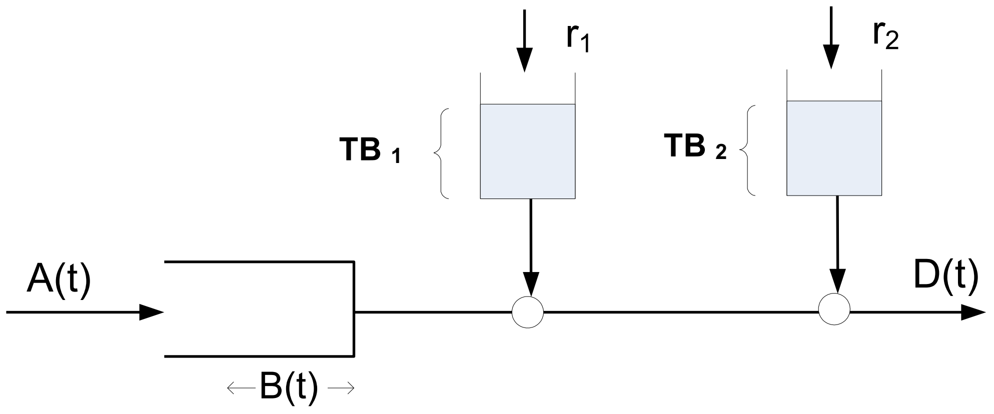
- In order to reduce the long-term resource consumption of the VBR video source, one would like to set the token bucket rate parameter to a value that is close to the average rate of the VBR source.

To satisfy both constraints, one token bucket is clearly not sufficient. A possible (and realized) solution to this problem is to regulate VBR video traffic by two token buckets put in series:

- The first token bucket controls the peak rate, by spacing out the transmission of a frame over a longer duration;

- The second token bucket controls the average rate of the VBR source.

An illustration of a dual token bucket is shown in the figure below. An arriving packet must withdraw tokens from both token buckets. If one of the two buckets does not have enough tokens, the packet must wait until sufficient tokens are available in both buckets.



Exercise 7.1 Dual Token Bucket Traffic Shaper

Revise the Token Bucket implementation to build a traffic shaper that realizes a dual token bucket. The token bucket has four parameters (TB_1, r_1, TB_2, r_2), for the size and rate of the token buckets. As before, the Token Bucket receives data from a traffic generator and transmits the output to a traffic sink.

- Build a traffic generator (of your choice) that can be used to demonstrate the correctness and accuracy of your implementation.

- Provide plots that illustrate how you tested the implementation.

Exercise 7.2 Selecting dual token bucket parameters for a VBR video source

Your task is to control the video trace with your dual-token bucket implementation. Transmitted packets are not permitted to exceed 1480 Bytes. So, some frames must be divided up into multiple packets.

Your task is to make a `good' selection for the parameters of the dual leaky. The selection of parameters requires you to tradeoff multiple considerations:

1. The maximum buffer size (and the maximum waiting time) should not be too large;
2. The average rate allocation should not be much larger than the average rate of the VBR source (to avoid over-allocation of bandwidth);
3. The number of packets that can be transmitted at once (this is determined by min(TB_1, TB_2)), should be small.

- Transmit the video source from the traffic generator using the data in file: http://www.comm.utoronto.ca/~jorg/teaching/ece466/labs/lab1/movietrace.data.
- Prepare a plot that shows the differences between the trace file and the output file, as a function of time. Prepare plots that depict the size of the token bucket TB(t) and the backlog in the buffer B(t) as a function of time.
  - What percentage of time is there a backlog at the token buckets?
  - What is the longest backlog at the token buckets?
- Prepare a plot that shows the differences between the trace file and the output file, as a function of time. Also provide a plot that shows the backlog in the buffer B(t) as a function of time. Mark the longest backlog in the buffer and the maximum waiting time.

Assignment Report:

- Describe the design of your implementation for the Dual Token Bucket. Include your source code in the lab report. Include and discuss how you have tested the implementation.
- Discuss your method for selecting the token bucket parameters. Use plots to justify your choices.
- Provide the plots and the answers performed in Exercise 7.2.