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:

- 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, hard copies of all your Java code, and the anonymous feedback form.

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

Table of Content

Table of Content ____ 2

Preparing for Lab 2a ____ 2

Comments ____ 2

Part 1. Programming with Datagram Sockets and with Files ____ 3

Part 2. Traffic generators ____ 5

Part 3. Token Bucket Traffic Shaper ____ 7

Feedback Form for Lab 2a ____ *13*

Preparing for Lab 2a

This lab requires network programming with Java datagram sockets. There is an informative and short tutorial on Java datagrams is available at:
http://java.sun.com/docs/books/tutorial/networking/datagrams/index.html

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/

Comments

- **Quality of plots in lab report:** This lab asks you to produce plots for a lab 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.

- **Feedback:** To be able to improve the labs for future years, we collect data on the current lab experience. You must submit an anonymous feedback form for each lab. Please use the feedback form at the end of the lab, and return the form with your lab report.

- **Extra credit:** Complete the optional Part 4 for 10% extra credit.

- **Java:** In the Unix lab, the default version of the Java installation is relatively old. To access a more recent version use the command:

    - Compiling: /local/java/jdk1.5.0_09/bin/javac
    - Running: /local/java/jdk1.5.0_09/bin/java

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 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
      …            …             …
```

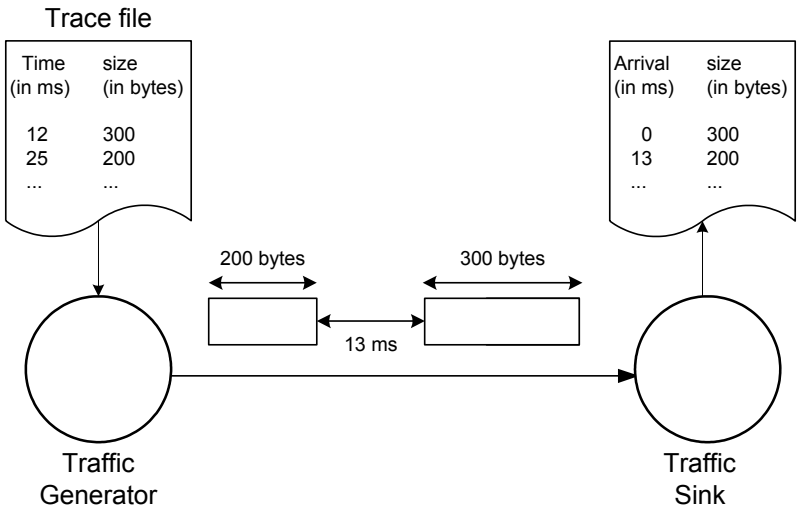
The file is read line-by-line, the values in a line are parsed and assigned to variables. Then the values are displayed, and written to a file with name *output.txt*.

- Run the program with the VBR video trace from Lab 1 as input. The video trace is available at: http://www.comm.utoronto.ca/%7Ejorg/teaching/ece466/labs/lab1/movietrace.data
- Modify the program so that it computes and displays the average size of the following frame types:
    - I frames;
    - P frames;
    - B frames.

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.



In this part of the lab, 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 Poisson traffic

Write a program that is a traffic generator for the compound Poisson traffic source (see Lab 1, Exercise 1.4) and that transmits the traffic using Java datagrams to a traffic sink. The traffic sink has to be programmed as well.

- A trace for the compound Poisson source is available at URL
  http://www.comm.utoronto.ca/%7Ejorg/teaching/ece466/labs/lab1/poisson3.data

  The file has the format:

```
SeqNo        Time (in μsec)    Size (in Bytes)

  1               273              30
  2               934              99
  3               2293             27
                 ...
```

  This results in a compound Poisson process with packet arrival rate λ = 1250 packets/sec, and the packet size has an exponential distribution with average size $1/\mu$ =100 Bytes.

- 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 2.2 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

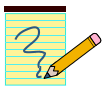
Run experiments where you transmit traffic from the traffic generator to the traffic sink. Evaluate the accuracy of the traffic generator by comparing the entries in the trace file (at the traffic generator) to the results written to the output file (at the sink).

- 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.
- 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.



Lab Report:

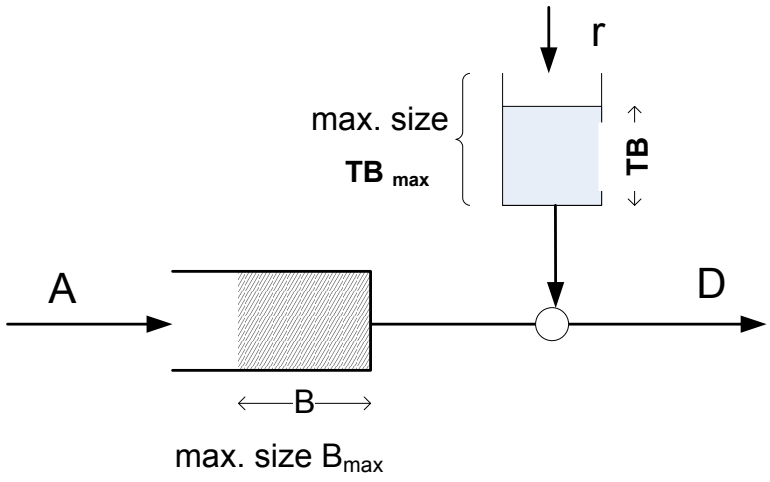
Provide discussions and graphs as requested in Exercise 2.3 and Exercise 2.4.

Part 3. Token Bucket Traffic Shaper

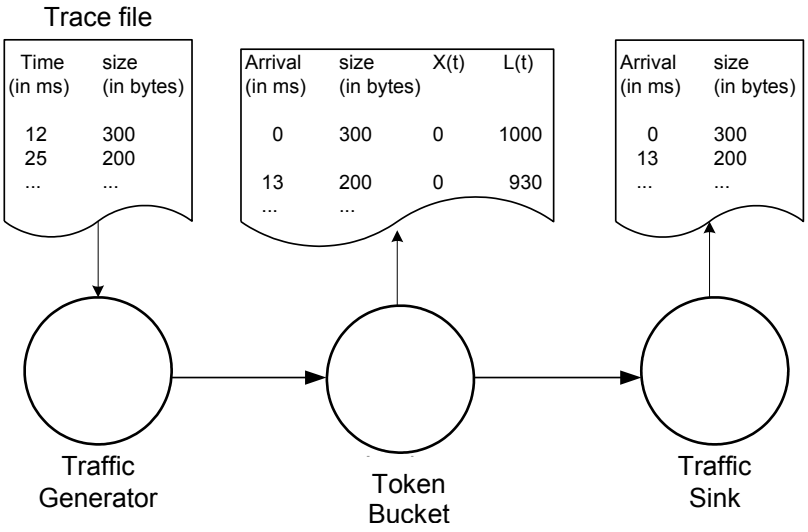
In this part of the lab, you familiarize yourself with a reference implementation of a Token Bucket. The implementation of the token bucket (in Java) and documentation for using the implementation is available at:
`http://www.comm.utoronto.ca/%7Ejorg/teaching/ece466/labs/lab2/TokenBucket`

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 a need to modify the provided source code.

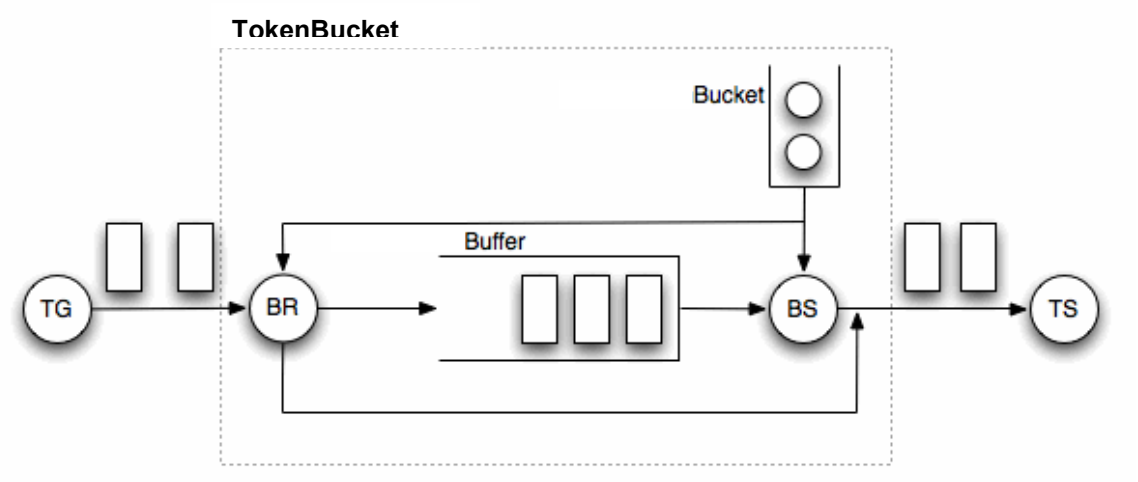
A token bucket traffic shaper with burst size TB_{max} and rate r is shown in the figure. Tokens are fed into the bucket at rate r. No more tokens are added if the bucket contains TB_{max} tokens. Data can be transmitted only if there are sufficient tokens in the bucket. To transmit a packet of L bytes,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

Consider the transmission of Poisson traffic from Exercise 2.3.

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.

Exercise 3.3 Evaluate the reference implementation for the Ethernet and Video Tracefiles

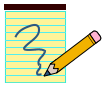
Repeat the evaluation for the video traffic trace and the Ethernet trace file that were used in Lab 1, and which are available at:

- **Video trace file:**
  http://www.comm.utoronto.ca/%7Ejorg/teaching/ece466/labs/lab1/movietrace.data
- **Ethernet traffic:**
  http://www.comm.utoronto.ca/%7Ejorg/teaching/ece466/labs/BC-pAug89-small.TL

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.



Lab Report:

- Provide the source code for your source code for Exercise 3.1. Do not include the sources for the traffic generator and traffic sink. Also, do not include source files of the reference implementation.

- For Exercises 3.2 and 3.3, provide the set of plots and include a description of the plots.

Feedback Form for Lab 2a

- Complete this feedback form at the completion of the lab exercises and submit the form when submitting your lab report.
- The feedback is anonymous. **Do not put your name on this form** and keep it separate from your lab report.
- For each exercise, please record the following:

	Difficulty (-2,-1,0,1,2) -2 = too easy 0 = just fine 2 = too hard	**Interest Level (-2,-1,0,1,2)** -2 = low interest 0 = just fine 2 = high interest	**Time to complete (minutes)**
Part 1. Programming with Datagram Sockets and with Files			
Part 2. Traffic generators			
Part 3. Token Bucket Traffic Regulator			

Please answer the following questions:

- What did you like about this lab?
- What did you dislike about this lab?
- Make a suggestion to improve the lab.