

## Traffic Regulation

This lab must be completed individually

### Purpose of this lab:

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

### 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 hard copy of all 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)

© Jörg Liebeherr, 2007. All rights reserved. Permission to use all or portions of this material for educational purposes is granted, as long as use of this material is acknowledged in all derivative works.

---

## Table of Content

Table of Content	2
Preparing for Lab 2	2
<i>Lab 2</i>	<i>Error! Bookmark not defined.</i>
Part 1. Programming with Datagram Sockets and with Files	3
Part 2. Traffic generators	5
Part 3. Leaky Bucket Traffic Regulator	7
Part 4. Regulating a VBR video source with Multiple Leaky buckets	10
<i>Feedback Form for Lab 2</i>	<i>12</i>

## Preparing for Lab 2

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(packet.getData(), 0, packet.getLength());
        System.out.println(packet.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
...    ...    ...
```

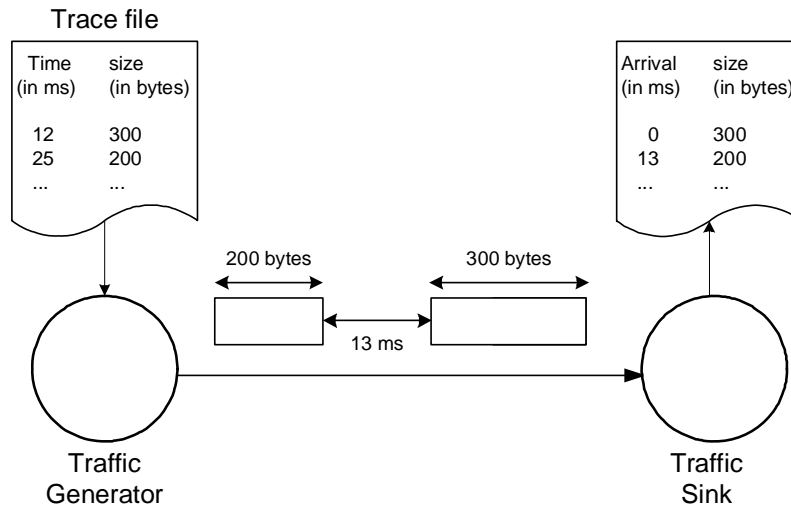
The file is read line-by-line, the values in a line are parsed and assigned to variables. Then the values are displayed, and written to a file with name *output.txt*.

- 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 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 $\mu$ sec)	Size (in Bytes)
1	273	30
2	934	99
3	2293	27
...	...	...

Re-scale the data in the file as follows:

- Multiply the time values in the file by a factor of 10
- Multiply the packet size by a factor of 10.

This results in a compound Poisson process with packet arrival rate  $\lambda = 125$  packets/sec, and the packet size has an exponential distribution with average size  $1/\mu = 1000$  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**

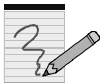
Run experiments where you transmit traffic from the traffic generator to the traffic sink. Evaluate the accuracy of the traffic generator by comparing the entries in the trace file (at the traffic generator) to the results written to the output file (at the sink).

- 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 (Optional) 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.



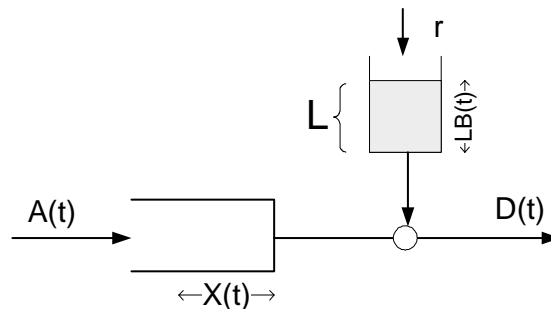
#### **Lab Report:**

Provide discussions and graphs as requested in Exercise 2.3 and (the optional) Exercise 2.4.

## Part 3. Leaky Bucket Traffic Regulator

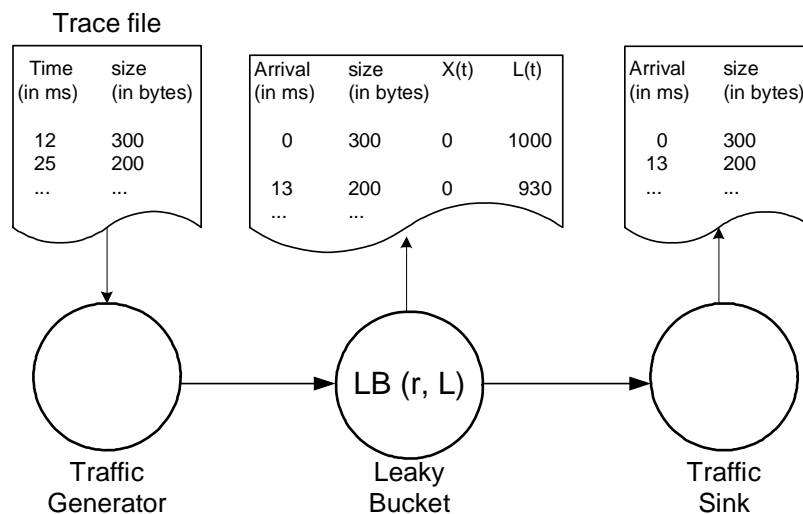
You will design and implement a leaky bucket regulator with size  $L$  and rate  $r$ , as shown in the figure. Tokens are fed into the bucket at rate  $r$ . No more tokens are added if the bucket contains  $L$  tokens. Data can be transmitted only if there are sufficient tokens in the bucket. To transmit a packet of  $N$  bytes, the bucket must contain at least  $N$  tokens. If there are not sufficient tokens, the packet must wait until there are enough tokens in the buffer.

Initially, the leaky bucket is full and the buffer is empty, i.e.,  $LB(t) = L$  bytes and  $X(t) = 0$  bytes. Note that  $L$  should not be smaller than the maximum packet size.



### Exercise 3.1 Build a Leaky Bucket traffic regulator

Your task is to build a leaky bucket regulator that receives data from a traffic generator and transmits the output to a traffic sink. This is illustrated below.



- The transmissions between traffic generator, leaky bucket, and traffic sink use UDP datagrams.
- The size and timing of packet transmissions is done as described in Part 2.
- Upon each packet arrival, the leaky 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 ( $L(t)$ ) in the leaky bucket and the backlog in the buffer ( $X(t)$ ) after the arrival of the packet.

### **Exercise 3.2 Regulating LAN traffic from the Bellcore trace**

Evaluate the accuracy and correctness of your leaky bucket implementation with the aggregate LAN traffic from Bellcore as traffic source.

- Take at least 10,000 data points of the Bellcore traffic trace, available at <http://www.comm.utoronto.ca/~jorg/teaching/ece466/labs/lab1/Bel.data>
- Modify the entries in the trace file as follows (This can be done when the data is read):
  - All packets with a size of 1518 bytes are counted as having a size of 1480 size (What is the reason for this adjustment?).
  - The time units in the first column are interpreted as seconds (as opposed to milliseconds).
- Determine the average traffic rate  $r^*$  of your trace.
- Using the setup from Exercise 3.1, set the parameters of the leaky bucket ( $L$ ,  $r$ ), such that  $L = 1480$  bytes and  $r = r^*$ .
- Transmit packets from the traffic generator using the (truncated and modified) data in the Bellcore trace.
- 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 leaky bucket  $L(t)$  and the backlog in the buffer  $X(t)$  as a function of time.
  - What percentage of time is there a backlog at the leaky bucket?
  - What is the longest backlog at the leaky bucket?

### **Exercise 3.3 Perform a stress test**

Perform a stress test of your leaky bucket implementation that determines the maximum transmission rate that can be supported by your leaky bucket

For the stress test you will create a constant-rate traffic source that generates fixed-sized packets at constant time intervals. When the time between packets is decreased, the rate of data transmission is increased. Eventually, the leaky bucket will not be able to keep up and a growing backlog will be observed at the buffer of the leaky bucket. The highest rate at which no (permanent and growing) backlog is observed marks the maximum transmission rate.

- Create a traffic generator that transmits packets of a fixed size ( $L$  bytes) with constant inter-packet spacing of  $T$  ms. Set up a leaky bucket traffic regulator with parameters ( $L$ ,  $r$ ).
- For a packet size of  $L = 100$  bytes, determine the smallest inter-packet spacing  $T_{\min}$  that can be supported by your implementation. Start with an initial spacing (chosen

by you) and reduce the time between packets, until you see a long backlog developing in the buffer of the leaky bucket.

- Determine the maximum transmission rate for scenarios, where both the sender and the regulator are running on the same system, and where sender and regulator are running on two different computers.
- Repeat the experiment with different packet sizes, and evaluate the impact of the packet size on the maximum transmission rate.

### **Exercise 3.4 Improving the implementation**

Try to improve the performance of the leaky bucket traffic generator, so that it can support higher data rates. The following are factors that may improve the performance:

- Avoid floating point operations. Perform operations in Integer arithmetic, avoiding divisions.
- Avoid unnecessary memory allocations (creation of objects) and system calls (e.g., timers).
- Note: By implementing the solution in Java, some aspects with an impact on the performance are not under your control. An important limitation is that garbage collection, i.e., the de-allocation unused memory, is not controlled by the user.

Your grade in this section depends on the maximum rate that you achieve (in comparison to other implementations). You may be asked to demonstrate the performance in the next lab session.



#### **Lab Report:**

- Describe the design of your leaky bucket implementation.
- For Exercise 3.2, provide a set of plots and include a description of the plots.
- For Exercise 3.3, provide a plot to support your the data rate the plots, a description of the plots, and your observations and discussions.
- For Exercise 3.4 describe the improvements to your code and the results that you achieved. Compare the results with those of Exercise 3.3.

## Part 4. (Optional) Regulating VBR video with Multiple Leaky buckets

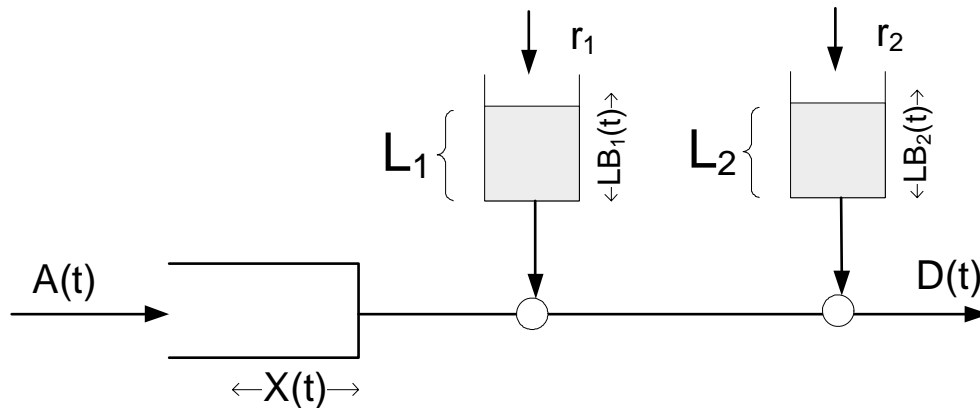
The regulation of VBR video traffic with a leaky bucket has to address the following two problems:

- Since the size of a frame can be much larger than the maximum packet size (in the file *movietrace.data*, the maximum frame size exceeds 600 KB), yet the time lag between frames is relatively long, one would like to select leaky 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 leaky bucket rate parameter to a value that is close to the average rate of the VBR source.

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

- The first leaky bucket controls the peak rate, by spacing out the transmission of a frame over a longer duration;
- The second leaky bucket controls the average rate of the VBR source.

An illustration of a dual leaky bucket is shown in the figure below. An arriving packet must withdraw tokens from both leaky 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 4.1 Dual Leaky Bucket Traffic regulator**

Implement a traffic regulator that realizes a dual leaky bucket. The leaky bucket has four parameters ( $L_1, r_1, L_2, r_2$ ), for the size and rate of the leaky buckets. As in Exercise 3.1, the regulator 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 (similar to Exercise 3.2) that illustrate how you tested the implementation.

### **Exercise 4.2 Selecting dual leaky bucket parameters for a VBR**

#### **video source**

Your task is to control the VBR video trace from Exercise 1.2 with your dual-leaky 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 a dual leaky bucket for the parameters. 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  $\min(L_1, L_2)$ ), should be small.
- Transmit the video source from the traffic generator using the data in file *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 leaky bucket  $L(t)$  and the backlog in the buffer  $X(t)$  as a function of time.
    - What percentage of time is there a backlog at the leaky buckets?
    - What is the longest backlog at the leaky 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  $X(t)$  as a function of time.
    - Determine the longest backlog  $X(t)$  and the maximum waiting time.

