

Lab 2 Report – 85 Points

Haiyu Wang, Haodong Huang

Part A (20 points). Place a copy of your source code for *TcpMapServer* here.

```
import java.io.*;
import java.net.*;
import java.util.*;

/** TCP Map Server
 *  author: Haiyu Wang and Haodong Huang
 *
 *  usage: java TcpMapServer [IP address] [port number]
 *
 *  have a storage service for 4 operations: get, put, remove, get all
 *  IP address and port number are optional. Default IP address is
 *  wildcard and default port number is 30123.
 *
 *  If port number is specified, IP address must also be specified.
 *
 *  Use Jon Turner's program - TCP echo server (7/2013) as a starting point
 */

public class TcpMapServer {
    public static void main(String args[]) throws Exception {

        // create HashMap
        HashMap<String, String> myMap = new HashMap<String, String>();

        // process arguments
        int port = 30123;
        if (args.length > 1) port = Integer.parseInt(args[1]);
        InetAddress ipAdr = null;
        if (args.length > 0) ipAdr = InetAddress.getByName(args[0]);

        // create and bind listening socket
        ServerSocket listenSock = new ServerSocket(port, 0, ipAdr);

        while (true) {
            // wait for incoming connection request and
            // create new socket to handle it (connection Socket)
            Socket connSock = listenSock.accept();

            // create reader & writer socket's in/out streams
            BufferedReader in = new BufferedReader(new InputStreamReader(
                connSock.getInputStream(), "US-ASCII"));
            BufferedOutputStream out = new BufferedOutputStream(
                connSock.getOutputStream());
        }
    }
}
```

```

while (true) {

    String str;
    str = in.readLine();
    if (str == null || str.length() == 0) break;

    // split the string with ":"
    String[] strSplit = str.split(":");

    // initiate value
    String value = "";

    switch (strSplit[0]){

        // Operation Get
        case "get":{
            if (strSplit.length>2){
                value = "error: unrecognizable input: "
                    + str + "\n";
            }
            else{
                value = myMap.get(strSplit[1]);
                if (value == null)
                    value = "no match\n";
                else
                    value = "Ok:"+value+"\n";
            }
            break;
        }

        // Operation Put
        case "put":{
            if (strSplit.length>3){
                value = "error: unrecognizable input: "
                    + str + "\n";
            }
            else{
                if (myMap.containsKey(strSplit[1]))
                    value = "Updated:"+strSplit[1]+"\n";
                else
                    value = "Ok\n";
                myMap.put(strSplit[1],strSplit[2]);
            }
            break;
        }

        // Operation Remove
        case "remove":{
            if (strSplit.length>2){
                value = "error: unrecognizable input: "
                    + str + "\n";
            }
            else {
                if (myMap.containsKey(strSplit[1])){

```

```

        myMap.remove(strSplit[1]);
        value = "Ok\n";
    }
    else
        value = "no match\n";
    }
    break;
}

// Operation GetAll
case "get all": {
    if (strSplit.length>1) {
        value = "error: unrecognizable input: "
            + str + "\n";
    }
    else{
        for (Map.Entry<String, String> entry:
            myMap.entrySet()) {
            value += entry.getKey() + ":"
                + entry.getValue() + "::";
        }
        value = value.substring(0,value.length()-2);
        value += '\n';
    }
    break;
}

default: {
    value = "error: unrecognizable input: "
        + str + "\n";
    break;
}

// since client use readLine(), value should be end with '\n'
}
// write value to socket
out.write(value.getBytes()); out.flush();
}
connSock.close();
}
}
}

```

Part B (10 points). Place a copy of your source code for *TcpMapClient* here.

```
import java.io.*;
import java.net.*;

/** TCP Map Client
 *  author: Haiyu Wang and Haodong Huang
 *
 *  usage: java TcpMapClient serverName [port number]
 *
 *  Open a connection to the server and print the response.
 *  When typing a blank line, connection closes and program exits.
 *  Port number are optional. Default port number is 30123.
 *
 *  Use Jon Turner's program - TCP echo client (7/2012) as a starting point
 */

public class TcpMapClient {
    public static void main(String args[]) throws Exception {
        // connect to remote server
        int port = 30123;
        if (args.length > 1) port = Integer.parseInt(args[1]);
        Socket sock = new Socket(args[0], port);

        // create reader & writer for socket's I/O
        BufferedReader in = new BufferedReader(new InputStreamReader(
            sock.getInputStream(), "US-ASCII"));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(
            sock.getOutputStream(), "US-ASCII"));

        // create reader for System.in
        BufferedReader sysin = new BufferedReader(new InputStreamReader(
            System.in));

        String line;

        while (true) {
            // reminder
            System.out.print("Please type a string: ");

            // if it is a blank line, break and close connection
            line = sysin.readLine();
            if (line == null || line.length() == 0) break;

            // write line on socket and print reply to System.out
            out.write(line); out.newLine(); out.flush();
            System.out.println(in.readLine());
        }
        sock.close();
    }
}
```

Part C (10 points). Use the provided *localScript* to test your client and server. You may do this testing on any Unix (including MacOS) or Linux computer (shell.cec.wustl.edu or onl.wustl.edu). Paste a copy of the output below.

```
WHYdeMacBook-Air:javatest a1100$ java TcpMapClient WHYdeMacBook-Air.local
Please type a string: put:foo bar:slim jim
Ok
Please type a string: put:hah:ho ho
Ok
Please type a string: put:goodbye:world
Ok
Please type a string: get:foo bar
Ok:slim jim
Please type a string: get:hah
Ok:ho ho
Please type a string: get:goodbye
Ok:world
Please type a string: get all
goodbye:world::foo bar:slim jim::hah:ho ho
Please type a string: get
error: unrecognizable input: get
Please type a string: foo:who
error: unrecognizable input: foo:who
Please type a string: get:bar
no match
Please type a string: put:foo:toast is tasty
Ok
Please type a string: get:foo
Ok:toast is tasty
Please type a string: put:hah:yolo
Updated:hah
Please type a string: get all
goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo
Please type a string:
WHYdeMacBook-Air:javatest a1100$ java TcpMapClient WHYdeMacBook-Air.local
30123
Please type a string: get all
goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo
Please type a string: remove:rab oof
no match
Please type a string: get all
goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo
Please type a string: remove:foo bar
Ok
Please type a string: get all
goodbye:world::foo:toast is tasty::hah:yolo
Please type a string:
WHYdeMacBook-Air:javatest a1100$
```

Part D (15 points). In the remaining parts of the lab, you will be testing your application in ONL. Begin by logging on to a Linux desktop (see Using a Remote Linux Desktop). Use the RLI to reserve an experimental network using the provided configuration file, *cse473-lab2.onl* (remember to first open an *ssh* connection to ONL with the tunnel required by the RLI), and commit your network. Open two separate *ssh* windows, one connecting to the host *h4x2* and the other to the host *h7x1* (remember to load the topology file first). First, start the server using the window for host *h7x1*. When starting the server, you should specify the host name (*h7x1*) or IP address (192.168.7.1) for the host in the experimental network. Use the default port number. Run the command in the “background” by putting an ampersand (&) at the end of the line. This will allow you to use the window for command input, even while the server is running (read the job control section of the *bash* manual to learn more about running jobs in the foreground and background). Note that once you start the server, it will “run forever” until you stop it. One simple way to do this is to type `kill %1`. Note that if you have multiple jobs running in the background, you will need to substitute the appropriate job number for %1. See the *bash* manual for details.

Now that your server is running in the background, type the following command in the window for *h7x1*.

```
netstat -an | grep 30123
```

and paste a copy of the output below.

```
haiyu@pclcore17:~/whycode$ netstat -an | grep 30123
tcp6          0      0 192.168.7.1:30123      :::*                  LISTEN
```

Now, start the client on *h4x2* (supplying the appropriate arguments) and then re-run *netstat* on *h7x1* and paste the output below.

```
haiyu@pclcore17:~/whycode$ netstat -an | grep 30123
tcp6          0      0 192.168.7.1:30123      :::*                  LISTEN
tcp6          0      0 192.168.7.1:30123      192.168.4.2:58820    ESTABLISHED
```

Explain the *netstat* output in the two cases. You should read the man page on *netstat* before answering this part (type “man netstat” to get the man page).

In the first case, the TcpMapServer opens a socket to listen to clients for connection. Since h7x1 runs the server and its port number is 30123, the IP address is 192.168.7.1 and it also shows the port number. Moreover, the status of the server is LISTEN, which means it listen for connection requests from remote TCP ports

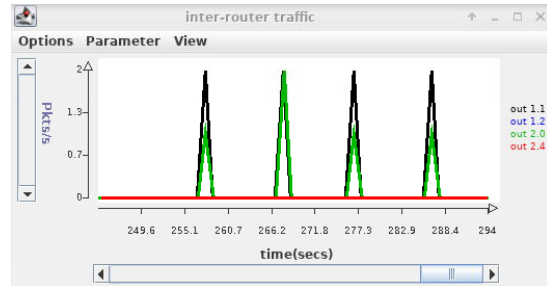
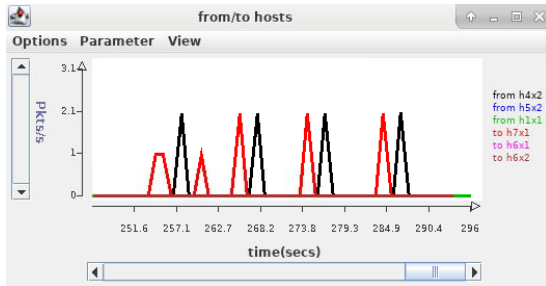
In the second case, TcpMapServer connects to TcpMapClient. The first line is the same as the first case which means the server is listening for connection requests. The second line shows the status of the server has changed into ESTABLISHED, which means the connection has been established. The server’s address and port number is 192.168.7.1:30123, and the client’ address and port number is 192.168.4.2:58820.

Now, run the provided *remoteScript* on *h4x2*. Paste the output from your run below.

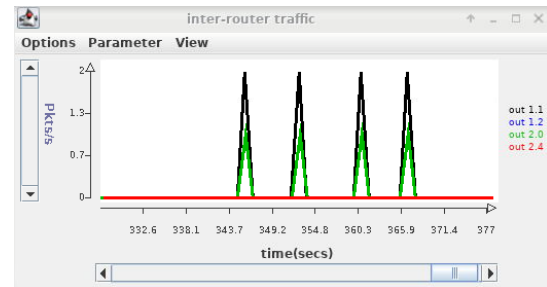
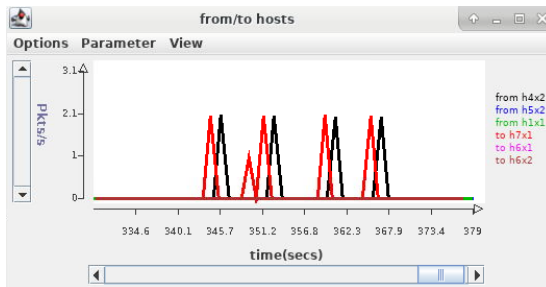
```
haiyu@pclcore21:~/whycode$ java TcpMapClient h7x1
Please type a string: put:foo bar:slim jim
Ok
Please type a string: put:hah:ho ho
Ok
Please type a string: put:goodbye:world
Ok
Please type a string: get:foo bar
Ok:slim jim
Please type a string: get:hah
Ok:ho ho
Please type a string: get:goodbye
Ok:world
Please type a string: get all
goodbye:world::foo bar:slim jim::hah:ho ho
Please type a string: get
error: unrecognizable input: get
Please type a string: foo:who
error: unrecognizable input: foo:who
Please type a string: get:bar
no match
Please type a string: put:foo:toast is tasty
Ok
Please type a string: get:foo
Ok:toast is tasty
Please type a string: put:hah:yolo
Updated:hah
Please type a string: get all
goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo
Please type a string:
haiyu@pclcore21:~/whycode$ java TcpMapClient h7x1
Please type a string: get all
goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo
Please type a string: remove:rab oof
no match
Please type a string: get all
goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo
Please type a string: remove:foo bar
Ok
Please type a string: get all
goodbye:world::foo:toast is tasty::hah:yolo
Please type a string:
haiyu@pclcore21:~/whycode$
```

Part E (10 points). In this part, you are to re-run the *remoteScript* and take a screen capture of the two monitoring windows showing the traffic that results from running the *remoteScript* (ignore the queue length window). You can pause a monitoring window by selecting *Stop* from its *Options* menu. This makes it easier to do the screen capture. Restart the paused window by select *Stop* a second time.

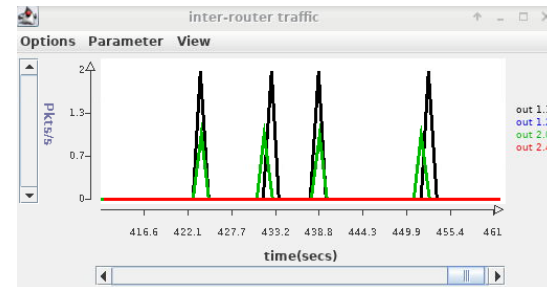
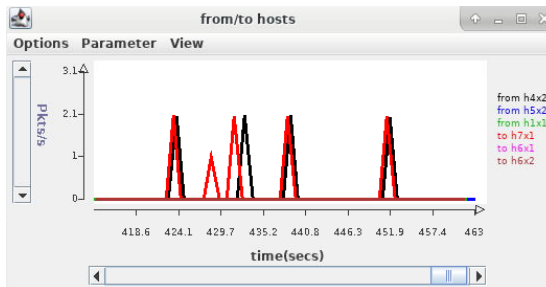
Commands: 1-4



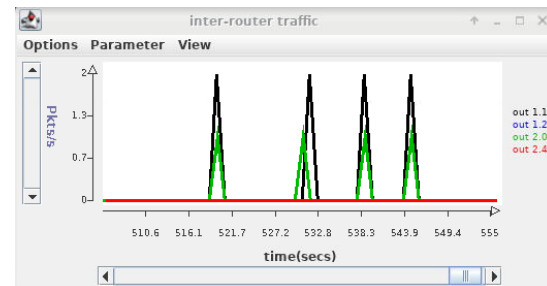
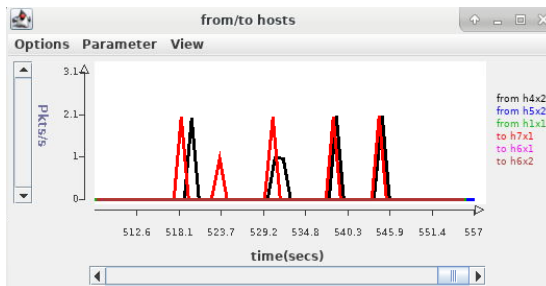
Commands: 5-8



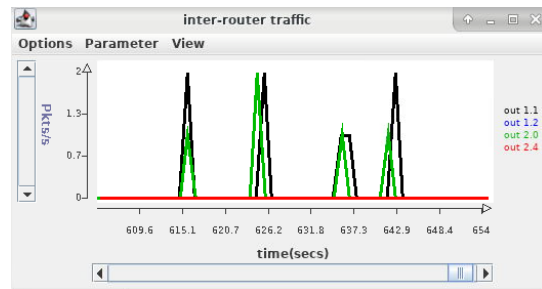
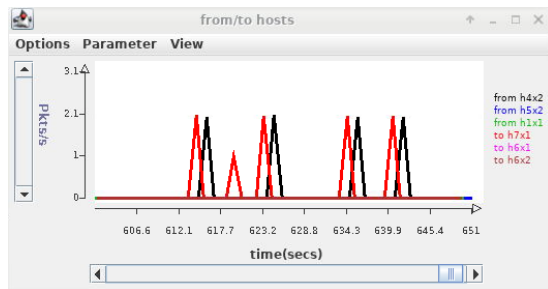
Commands: 9-12



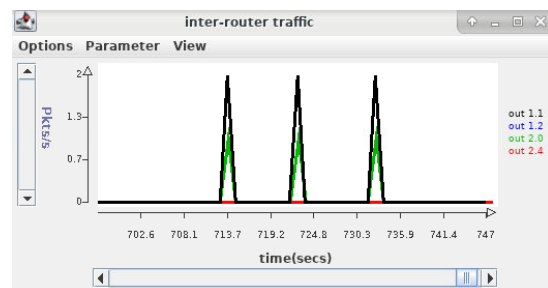
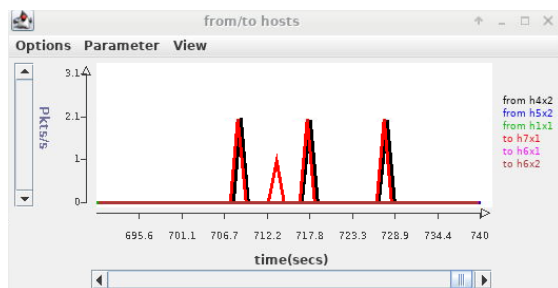
Commands: 13-16



Commands: 17-20



Commands: 21-23



The charts should show a burst of traffic for some of the curves and no traffic for others. Which curves show a burst of traffic? Is this consistent with what you expect? Note that there are two possible routes between the two end hosts. Which of the two routes are used in this case?

In the from/to hosts chart, the "from h4x2" and "to h7x1" curves show burst traffic. It's consistent with what I expect, because the data flows between host h4x2 and h7x1. Thus, the burst traffic should occurs between these 2 hosts. There is no data between other hosts, so there is no burst traffic in their curves.

In the inter-router traffic, the "out 1.1" and "out 2.0" curves show burst traffic. '

In this case, there are 2 possible routes between the 2 end hosts. They are:

- (1) h4x2 -> out 1.1 -> h7x1*
- (2) h4x2 -> out 2.0 -> h7x1*

Part F (10 points). In this next part, you are to run *remoteScript* once again, but this time, you will be using *Wireshark* to capture packets as seen at both hosts. Using *Wireshark* in *onl* requires a little extra effort, since *Wireshark* itself must run on the target computer within *onl*, while the graphical interface needs to appear on your local computer. Start by opening a new shell window on your Linux desktop (again, use the Virtual Linux Lab to run a remote Linux desktop if you don't have access to a Linux/Unix machine), and type

```
ssh -YC blowfish-cbc,arcfour <username>@onl.wustl.edu
```

This creates an *ssh* connection that forwards "X-windows" commands from *onlusr* back to your Linux desktop, turns compression on, and specifies a more efficient cipher for encryption. X-windows is a generic windowing system developed at MIT in the 1980s. It is still used for a number of *unix/linux* applications, including *Wireshark*. Now, type

```
source /users/onl/.topology  
ssh -YC $h4x2
```

This will log you into host *h4x2* and forward X-windows commands from *h4x2* back through *onlusr* to your Linux desktop. Next, type

```
sudo wireshark
```

After you enter your *onl* password, *Wireshark* will start running on *h4x2*, and the *Wireshark* window will open on your Linux desktop. If you want to do this part of the lab using your own computer, you may have to do some initial configuration. If you have a Mac or a Linux computer, with *Wireshark* installed, you're probably good to go. Just open a terminal window and type

```
ssh -X myLogin@onl.wustl.edu
```

and proceed as described above. Again, if you are not using a Linux/Unix machine, you must use a remote Linux desktop.

Now, configure *Wireshark* to capture packets on the *data0* interface and then re-run *remoteScript* in the original terminal window connected to *h4x2*. Find the packet going from *h4x2* to *h7x1* that includes the “get:goodbye” command. Highlight that packet in the upper sub-window and make sure that the packet contents are visible in the lower sub-window.

The image shows the Wireshark network protocol analyzer interface. The top pane displays a list of captured packets. Packet 20 is highlighted in blue. The middle pane shows the details of packet 20, including Ethernet II, Internet Protocol Version 4, and Transmission Control Protocol (TCP) fields. The bottom pane shows the raw packet data in hexadecimal and ASCII format.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.4.2	192.168.7.1	TCP	74	58868 → 30123 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=5020475 TSecr=0 WS=1024
2	0.050183	192.168.7.1	192.168.4.2	TCP	74	30123 → 58868 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=5018142 TSecr=502047...
3	0.050222	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=5020487 TSecr=5018142
4	17.781586	192.168.4.2	192.168.7.1	TCP	87	58868 → 30123 [PSH, ACK] Seq=1 Ack=1 Win=29696 Len=21 TSval=5024920 TSecr=5018142
5	17.832066	192.168.7.1	192.168.4.2	TCP	66	30123 → 58868 [ACK] Seq=1 Ack=22 Win=29696 Len=0 TSval=5022587 TSecr=5024920
6	17.834090	192.168.7.1	192.168.4.2	TCP	69	30123 → 58868 [PSH, ACK] Seq=1 Ack=22 Win=29696 Len=3 TSval=5022588 TSecr=5024920
7	17.834126	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=22 Ack=4 Win=29696 Len=0 TSval=5024933 TSecr=5022588
8	25.300414	192.168.4.2	192.168.7.1	TCP	80	58868 → 30123 [PSH, ACK] Seq=22 Ack=4 Win=29696 Len=14 TSval=5026800 TSecr=5022588
9	25.351103	192.168.7.1	192.168.4.2	TCP	69	30123 → 58868 [PSH, ACK] Seq=4 Ack=36 Win=29696 Len=3 TSval=5024467 TSecr=5026800
10	25.351144	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=36 Ack=7 Win=29696 Len=0 TSval=5026813 TSecr=5024467
11	34.762984	192.168.4.2	192.168.7.1	TCP	84	58868 → 30123 [PSH, ACK] Seq=36 Ack=7 Win=29696 Len=18 TSval=5029166 TSecr=5024467
12	34.813704	192.168.7.1	192.168.4.2	TCP	69	30123 → 58868 [PSH, ACK] Seq=7 Ack=54 Win=29696 Len=3 TSval=5026833 TSecr=5029166
13	34.813746	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=54 Ack=10 Win=29696 Len=0 TSval=5029178 TSecr=5026833
14	43.994581	192.168.4.2	192.168.7.1	TCP	78	58868 → 30123 [PSH, ACK] Seq=54 Ack=10 Win=29696 Len=12 TSval=5031473 TSecr=5026833
15	44.045591	192.168.7.1	192.168.4.2	TCP	78	30123 → 58868 [PSH, ACK] Seq=10 Ack=66 Win=29696 Len=12 TSval=5029141 TSecr=5031473
16	44.045635	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=66 Ack=22 Win=29696 Len=0 TSval=5031486 TSecr=5029141
17	47.318892	192.168.4.2	192.168.7.1	TCP	74	58868 → 30123 [PSH, ACK] Seq=66 Ack=22 Win=29696 Len=8 TSval=5032305 TSecr=5029141
18	47.369531	192.168.7.1	192.168.4.2	TCP	75	30123 → 58868 [PSH, ACK] Seq=22 Ack=74 Win=29696 Len=9 TSval=5029972 TSecr=5032305
19	47.369572	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=74 Ack=31 Win=29696 Len=0 TSval=5032317 TSecr=5029972
20	55.636268	192.168.4.2	192.168.7.1	TCP	78	58868 → 30123 [PSH, ACK] Seq=74 Ack=31 Win=29696 Len=12 TSval=5034384 TSecr=5029972
21	55.686967	192.168.7.1	192.168.4.2	TCP	75	30123 → 58868 [PSH, ACK] Seq=31 Ack=86 Win=29696 Len=9 TSval=5032051 TSecr=5034384
22	55.687013	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=86 Ack=40 Win=29696 Len=0 TSval=5034397 TSecr=5032051
23	58.884554	192.168.4.2	192.168.7.1	TCP	74	58868 → 30123 [PSH, ACK] Seq=86 Ack=40 Win=29696 Len=8 TSval=5035196 TSecr=5032051
24	58.936883	192.168.7.1	192.168.4.2	TCP	109	30123 → 58868 [PSH, ACK] Seq=40 Ack=94 Win=29696 Len=43 TSval=5032864 TSecr=5035196
25	58.936924	192.168.4.2	192.168.7.1	TCP	66	58868 → 30123 [ACK] Seq=94 Ack=83 Win=29696 Len=0 TSval=5035209 TSecr=5032864

Frame 20: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
 Ethernet II, Src: TyanComp_5e:9c:5c (08:e0:81:5e:9c:5c), Dst: Radsys_2c:81:f2 (00:00:50:2c:81:f2)
 Internet Protocol Version 4, Src: 192.168.4.2, Dst: 192.168.7.1
 Transmission Control Protocol, Src Port: 58868, Dst Port: 30123, Seq: 74, Ack: 31, Len: 12
 Data (12 bytes)

```

0000  00 00 50 2c 81 f2 00 e0 81 5e 9c 5c 08 00 45 00  ..P.....E-
0010  00 40 17 e9 40 00 00 06 96 7b c0 a8 04 02 c0 a8  @.@.@.{}
0020  07 01 e5 f4 75 ab 47 39 8d 30 09 f2 dd f0 80 18  ....u.G.0....
0030  00 1d 8c 86 00 00 01 01 08 0a 00 4c d1 90 00 4c  .......L...L
0040  c0 54 67 65 74 3a 67 6f 6f 64 62 79 65 0a      .Tget:go odbye-
  
```

How much time passes between the time this packet is sent and the time the reply arrives?
 (Note, the reply appears on the next line and the second column of the displays shows the times relative to the start of the capture.)

The command “get:goodbye” was sent at 55.636268 and the reply “Ok:world” was sent back at 55.686967, so the time between the request and reply is 0.050699sec.

The observed time is caused primarily by an artificial delay that has been configured in one of the routers, using a special *delay plugin*. Which router contains the delay plugin?

After checking the plugin table of the routers, NPR.1 contains the delay plugin. Moreover, I checked the filter table of each port of NPR.1, port 0, port 3, port 4 have this plugin.

Find the filter that causes packets to pass through the plugin and turn it off, using the RLI. You can find filters by clicking on an interface and looking at the Filter Table under Configuration. Don't forget to commit after disabling the filter. Note that if you accidentally disable the wrong thing, the whole system could stop working. So don't be afraid to reopen the original onl file and start over. Now, start a new *Wireshark capture* and re-run *remoteScript*.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.4.2	192.168.7.1	TCP	74	44888 → 30123 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=10516523 TSecr=0 WS=1024
2	0.000358	192.168.7.1	192.168.4.2	TCP	74	30123 → 44888 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=10518049 TSecr=10516523
3	0.000424	192.168.4.2	192.168.7.1	TCP	66	44888 → 30123 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=10516524 TSecr=10518049
4	5.008392	TyanComp_5c:d0:72	Radisys_33:13:14	ARP	42	Who has 192.168.4.1? Tell 192.168.4.2
5	5.009605	Radisys_33:13:14	TyanComp_5c:d0:72	ARP	60	192.168.4.1 is at 00:00:50:33:13:14
6	21.804473	192.168.4.2	192.168.7.1	TCP	87	44888 → 30123 [PSH, ACK] Seq=1 Ack=1 Win=29696 Len=21 TSval=10521975 TSecr=10518049
7	21.804951	192.168.7.1	192.168.4.2	TCP	66	30123 → 44888 [ACK] Seq=1 Ack=22 Win=29696 Len=0 TSval=10523500 TSecr=10521975
8	21.807026	192.168.7.1	192.168.4.2	TCP	69	30123 → 44888 [PSH, ACK] Seq=1 Ack=22 Win=29696 Len=3 TSval=10523501 TSecr=10521975
9	21.807063	192.168.4.2	192.168.7.1	TCP	66	44888 → 30123 [ACK] Seq=22 Ack=4 Win=29696 Len=0 TSval=10521975 TSecr=10523501
10	27.716052	192.168.4.2	192.168.7.1	TCP	80	44888 → 30123 [PSH, ACK] Seq=22 Ack=4 Win=29696 Len=14 TSval=10523452 TSecr=10523501
11	27.716750	192.168.7.1	192.168.4.2	TCP	69	30123 → 44888 [PSH, ACK] Seq=4 Ack=36 Win=29696 Len=3 TSval=10524978 TSecr=10523452
12	27.716787	192.168.4.2	192.168.7.1	TCP	66	44888 → 30123 [ACK] Seq=36 Ack=7 Win=29696 Len=0 TSval=10523453 TSecr=10524978
13	34.645527	192.168.4.2	192.168.7.1	TCP	84	44888 → 30123 [PSH, ACK] Seq=36 Ack=7 Win=29696 Len=18 TSval=10525185 TSecr=10524978
14	34.646198	192.168.7.1	192.168.4.2	TCP	69	30123 → 44888 [PSH, ACK] Seq=7 Ack=54 Win=29696 Len=3 TSval=10526710 TSecr=10525185
15	34.646239	192.168.4.2	192.168.7.1	TCP	66	44888 → 30123 [ACK] Seq=54 Ack=10 Win=29696 Len=0 TSval=10525185 TSecr=10526710
16	40.348951	192.168.4.2	192.168.7.1	TCP	78	44888 → 30123 [PSH, ACK] Seq=54 Ack=10 Win=29696 Len=12 TSval=10526611 TSecr=10526710
17	40.349924	192.168.7.1	192.168.4.2	TCP	78	30123 → 44888 [PSH, ACK] Seq=10 Ack=66 Win=29696 Len=12 TSval=10528136 TSecr=10526611
18	40.349964	192.168.4.2	192.168.7.1	TCP	66	44888 → 30123 [ACK] Seq=66 Ack=22 Win=29696 Len=0 TSval=10526611 TSecr=10528136
19	44.853091	192.168.4.2	192.168.7.1	TCP	74	44888 → 30123 [PSH, ACK] Seq=66 Ack=22 Win=29696 Len=8 TSval=10527737 TSecr=10528136
20	44.853726	192.168.7.1	192.168.4.2	TCP	75	30123 → 44888 [PSH, ACK] Seq=22 Ack=74 Win=29696 Len=9 TSval=10529262 TSecr=10527737
21	44.853767	192.168.4.2	192.168.7.1	TCP	66	44888 → 30123 [ACK] Seq=74 Ack=31 Win=29696 Len=0 TSval=10527737 TSecr=10529262
22	51.957736	192.168.4.2	192.168.7.1	TCP	78	44888 → 30123 [PSH, ACK] Seq=74 Ack=31 Win=29696 Len=12 TSval=10529513 TSecr=10529262
23	51.958459	192.168.7.1	192.168.4.2	TCP	75	30123 → 44888 [PSH, ACK] Seq=31 Ack=86 Win=29696 Len=9 TSval=10531039 TSecr=10529513
24	51.958672	192.168.4.2	192.168.7.1	TCP	66	44888 → 30123 [ACK] Seq=86 Ack=40 Win=29696 Len=0 TSval=10529513 TSecr=10531039
25	78.493877	192.168.4.2	192.168.7.1	TCP	74	44888 → 30123 [PSH, ACK] Seq=86 Ack=40 Win=29696 Len=8 TSval=10536147 TSecr=10531039
26	78.496227	192.168.7.1	192.168.4.2	TCP	109	30123 → 44888 [PSH, ACK] Seq=40 Ack=94 Win=29696 Len=43 TSval=10537673 TSecr=10536147

▶ Frame 22: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
 ▶ Ethernet II, Src: TyanComp_5c:d0:72 (00:e0:81:5c:d0:72), Dst: Radisys_33:13:14 (00:00:50:33:13:14)
 ▶ Internet Protocol Version 4, Src: 192.168.4.2, Dst: 192.168.7.1
 ▶ Transmission Control Protocol, Src Port: 44888, Dst Port: 30123, Seq: 74, Ack: 31, Len: 12
 ▶ Data (12 bytes)

```

0000  00 00 50 33 13 14 00 e0 81 5c d0 72 08 00 45 00  ..P3....\r..E.
0010  00 40 48 88 40 00 40 06 65 dc c0 a8 04 02 c0 a8  @H@.@e.....
0020  07 01 af 58 75 ab 07 10 30 a1 d4 6f 1f 0b 00 18  ..Xu...0-....
0030  00 1d 8c 86 00 00 01 01 08 0a 00 a0 aa e9 00 a0  .....
0040  a9 ee 67 65 74 3a 67 6f 6f 64 62 79 65 0a      ..get:go odybe
  
```

Now, how much time passes between the sending of the packet and the response?

The command "get:goodbye" was sent at 51.957736, and the response "Ok:world" was sent back at 51.958459. The time between the request and reply is 0.000723sec. It is much shorter than before.

Part G (10 points). In this part, you will measure the performance of your application in another way. Run the provided *longScript* on the client. This performs a large number of puts and gets. Make a screen capture of the two monitoring windows showing the packet traffic in the network.

These 2 screenshots show the start of running these programs. They show that after the beginning of the program the traffic rises sharply and maintain a high level during the program running.



What does the traffic data tell you about the performance of the application?

We can see from the monitoring windows that the traffic rises sharply when the programs start running and maintain a high level during the whole time of the running of the program. This shows that this program generates lots of traffic and the network allocates lots of resources to this program.

If the network can handle such large amount of traffic, the program may have a good performance. Otherwise, it will take a long time to run this program and the program will perform badly.

Repeat the above experiment while running *Wireshark* on *h7x1*. Select a packet going from *h4x2* to *h7x1* from somewhere near the middle of the capture.

The screenshot shows a Wireshark packet capture. The packet list pane displays a series of TCP packets from 1499 to 1521. Packet 1516 is selected, showing a timestamp of 38.292211. The packet details pane shows the structure of the packet: Ethernet II, Internet Protocol Version 4, Transmission Control Protocol, and Data (69 bytes). The data section shows a sequence of bytes starting with 00 00 81 5e 9c c4.

No.	Time	Source	Destination	Protocol	Length	Info
1499	37.839572	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5830 Ack=47359 Win=29696 Len=3 TSval=13524127 TSecr=13522426
1500	37.889750	192.168.4.2	192.168.7.1	TCP	101	52482 → 30123 [PSH, ACK] Seq=47359 Ack=5833 Win=29696 Len=35 TSval=13522439 TSecr=13524127
1501	37.889984	192.168.7.1	192.168.4.2	TCP	92	30123 → 52482 [PSH, ACK] Seq=5833 Ack=47394 Win=29696 Len=26 TSval=13522439 TSecr=13522439
1502	37.940172	192.168.4.2	192.168.7.1	TCP	96	52482 → 30123 [PSH, ACK] Seq=47394 Ack=5859 Win=29696 Len=30 TSval=13522451 TSecr=13524140
1503	37.940426	192.168.7.1	192.168.4.2	TCP	92	30123 → 52482 [PSH, ACK] Seq=5859 Ack=47424 Win=29696 Len=26 TSval=13524153 TSecr=13522451
1504	37.990606	192.168.4.2	192.168.7.1	TCP	94	52482 → 30123 [PSH, ACK] Seq=47424 Ack=5885 Win=29696 Len=28 TSval=13522464 TSecr=13524153
1505	37.990756	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5885 Ack=47452 Win=29696 Len=3 TSval=13524165 TSecr=13522464
1506	38.040933	192.168.4.2	192.168.7.1	TCP	139	52482 → 30123 [PSH, ACK] Seq=47452 Ack=5888 Win=29696 Len=73 TSval=13522476 TSecr=13524165
1507	38.040979	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5888 Ack=47525 Win=29696 Len=3 TSval=13524178 TSecr=13522476
1508	38.091154	192.168.4.2	192.168.7.1	TCP	138	52482 → 30123 [PSH, ACK] Seq=47525 Ack=5891 Win=29696 Len=72 TSval=13522489 TSecr=13524178
1509	38.091233	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5891 Ack=47597 Win=29696 Len=3 TSval=13524190 TSecr=13522489
1510	38.141419	192.168.4.2	192.168.7.1	TCP	136	52482 → 30123 [PSH, ACK] Seq=47597 Ack=5894 Win=29696 Len=70 TSval=13522502 TSecr=13524190
1511	38.141501	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5894 Ack=47667 Win=29696 Len=3 TSval=13524203 TSecr=13522502
1512	38.191686	192.168.4.2	192.168.7.1	TCP	142	52482 → 30123 [PSH, ACK] Seq=47667 Ack=5897 Win=29696 Len=76 TSval=13522514 TSecr=13524203
1513	38.191766	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5897 Ack=47743 Win=29696 Len=3 TSval=13524215 TSecr=13522514
1514	38.241949	192.168.4.2	192.168.7.1	TCP	142	52482 → 30123 [PSH, ACK] Seq=47743 Ack=5900 Win=29696 Len=76 TSval=13522527 TSecr=13524215
1515	38.242032	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5900 Ack=47819 Win=29696 Len=3 TSval=13524228 TSecr=13522527
1516	38.292211	192.168.4.2	192.168.7.1	TCP	135	52482 → 30123 [PSH, ACK] Seq=47819 Ack=5903 Win=29696 Len=69 TSval=13522539 TSecr=13524228
1517	38.292420	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5903 Ack=47888 Win=29696 Len=3 TSval=13524241 TSecr=13522539
1518	38.342607	192.168.4.2	192.168.7.1	TCP	137	52482 → 30123 [PSH, ACK] Seq=47888 Ack=5906 Win=29696 Len=71 TSval=13522552 TSecr=13524241
1519	38.342746	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5906 Ack=47959 Win=29696 Len=3 TSval=13524253 TSecr=13522552
1520	38.392930	192.168.4.2	192.168.7.1	TCP	139	52482 → 30123 [PSH, ACK] Seq=47959 Ack=5909 Win=29696 Len=73 TSval=13522564 TSecr=13524253
1521	38.393073	192.168.7.1	192.168.4.2	TCP	69	30123 → 52482 [PSH, ACK] Seq=5909 Ack=48032 Win=29696 Len=3 TSval=13524266 TSecr=13522564

Frame 1516: 135 bytes on wire (1080 bits), 135 bytes captured (1080 bits)
 Ethernet II, Src: RadiSys_2c:81:f8 (00:00:50:2c:81:f8), Dst: TyanComp_Se9c:c4 (00:e0:81:5e:9c:c4)
 Internet Protocol Version 4, Src: 192.168.4.2, Dst: 192.168.7.1
 Transmission Control Protocol, Src Port: 52482, Dst Port: 30123, Seq: 47819, Ack: 5903, Len: 69
 Data (69 bytes)

0000 00 e0 81 5e 9c c4 00 00 50 2c 81 f8 00 00 45 00 ...P...E...
 0010 00 79 af 92 40 00 3e 06 00 99 c0 a8 04 02 c0 a8 ...y@>...
 0020 07 01 cd 02 75 ab e1 e0 58 65 b9 8c 25 b6 80 18 ...u...Xe...
 0030 00 1d ea fb 00 00 01 01 08 0a 00 ce 56 60 00 ce ...VK...
 0040 5d 04 70 75 74 3a 72 65 63 65 69 76 65 64 2e 20]put:received.
 0050 20 54 68 69 73 3a 20 6d 65 63 68 61 6e 69 73 6d This: mechanism
 0060 20 61 6c 6c 6f 77 73 20 66 6f 72 20 73 74 72 61 allows for str
 0070 69 67 68 74 2d 66 6f 72 77 61 72 64 20 64 75 70 ight-for ward dup
 0080 6c 69 63 61 74 65 0a licate

How much time passes between when *h7x1* receives the packet and the time it sends its reply?

In the command with highlight, the request was sent at 38.292211 and the reply was sent back at 38.292420. The time between the request and reply is 0.00189sec.

How much time from when *h7x1* receives this packet and the time it receives the next one.

As we can see in the screen shot, h7x1 receives this packet at 38.292211 and its number is No.1516. The next packet it receives is No.1518 and it is received at 38.342746. The time between these two packets is 0.050535sec.

Is this consistent with what you observed based on the packet rate chart?

From the screenshot we can tell that the time between 2 packets to be received at h7x1 is around 0.05sec. Thus, h7x1 receives packets at a rate of $(1/0.05) = 20$ pkts/sec. It is consistent with the packet rate in the charts.