CSE 473 – Introduction to Computer Networks

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 TcpMapCient 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 bank 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@pc1core17:~/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@pc1core17:~/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

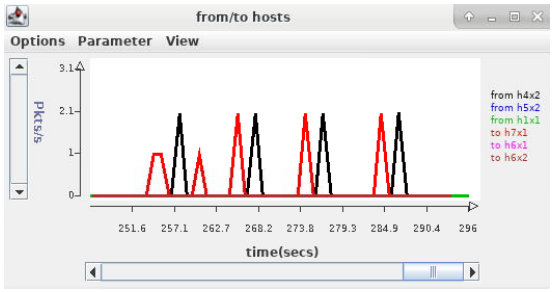
In the second case, TcpMapServer connects to TcpMapClient. The first line is the same as the first case which means the server is listening for connection requests. The second line shows the status of the server has changed into ESTABLISHED, which means the connection has been established. The server's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

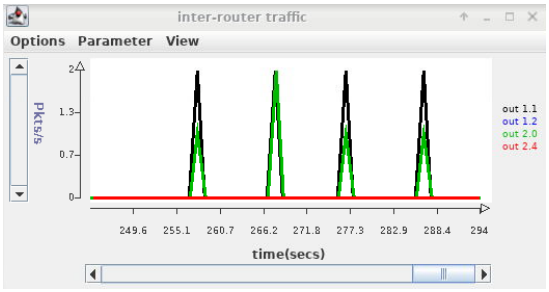
Now, run the provided *remoteScript* on *h4x2*. Paste the output from your run below.

```
haiyu@pc1core21:~/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@pc1core21:~/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@pc1core21:~/whycode$
```

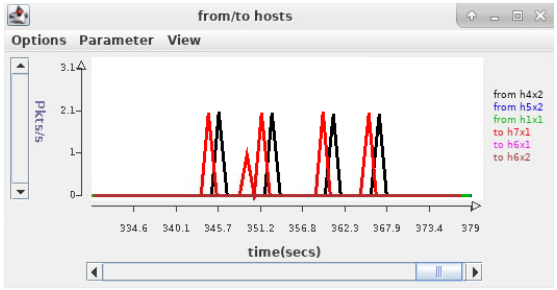
Part E (10 points). In this part, you are to re-run the *remoteScript* and take a screen capture of the two monitoring windows showing the traffic that results from running the *remoteScript* (ignore the queue length window). You can pause a monitoring window by selecting *Stop* from its *Options* menu.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

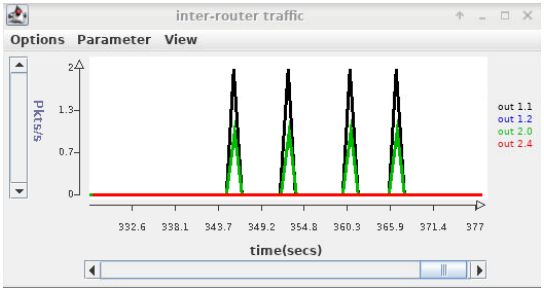
`Commands: 1-4`



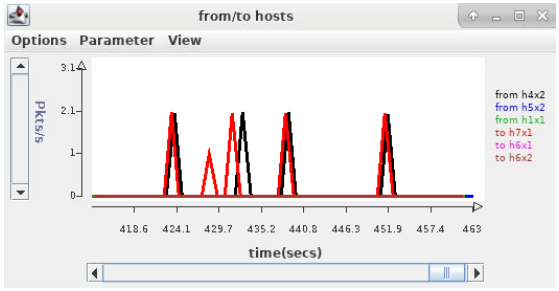


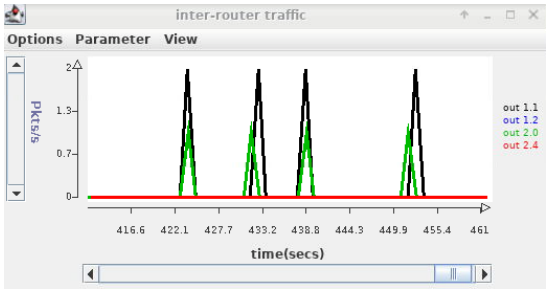
`Commands: 5-8`



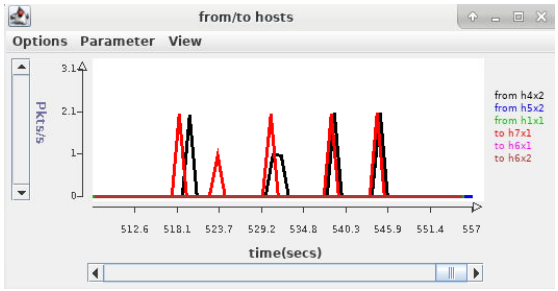


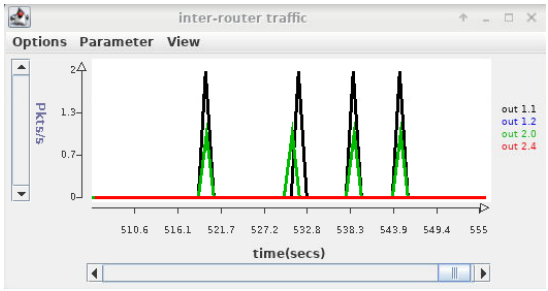
`Commands: 9-12`



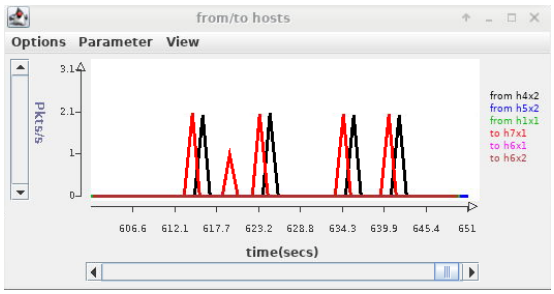


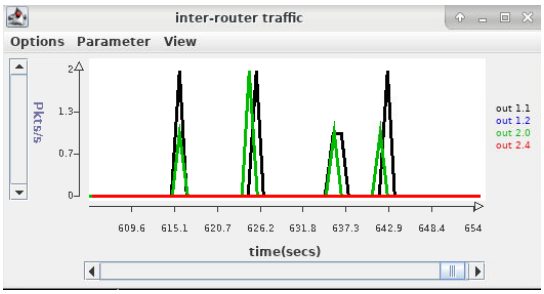
`Commands: 13-16`



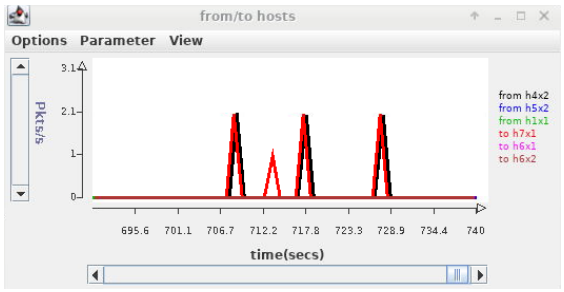


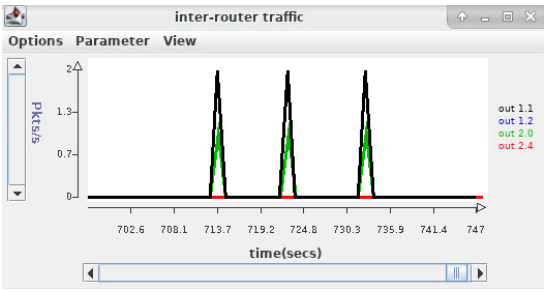
Commands: 17-20





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

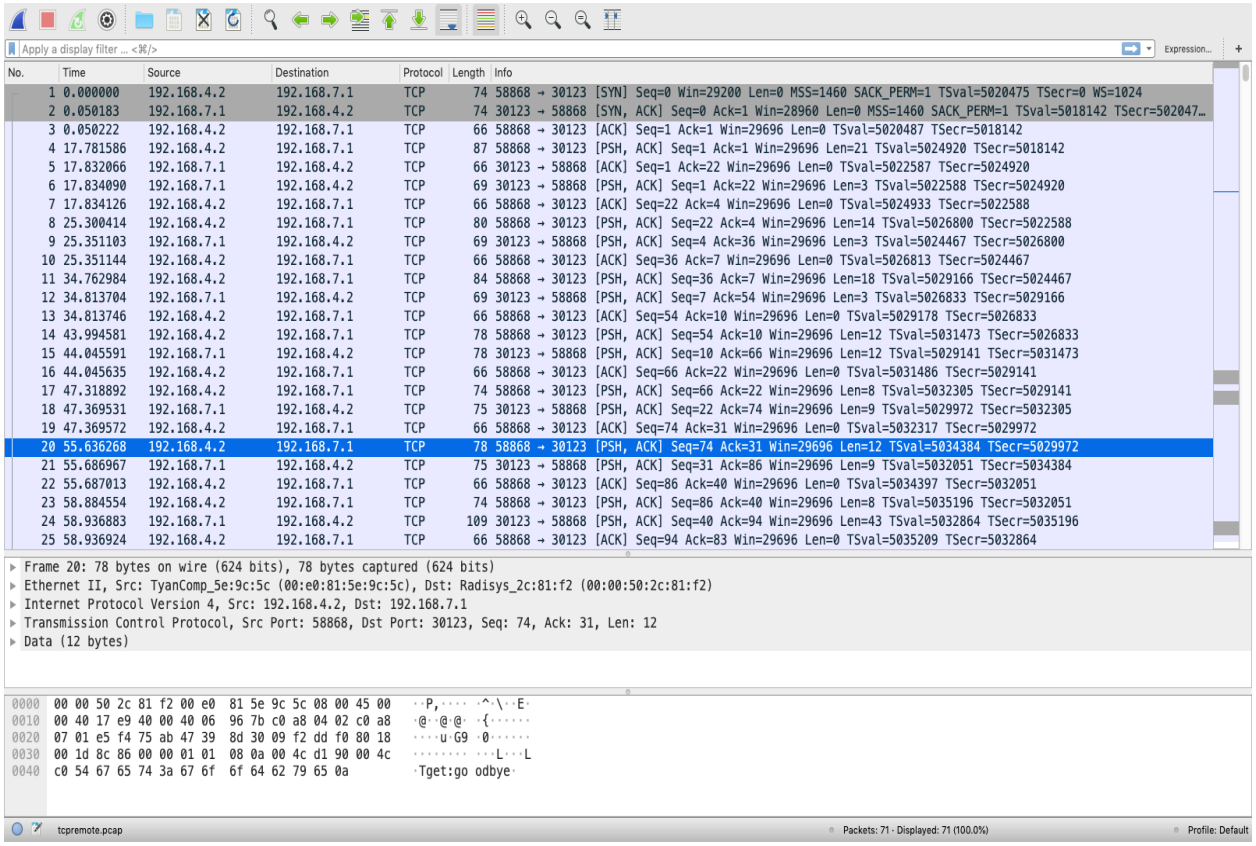
```
sudo wireshark
```

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.



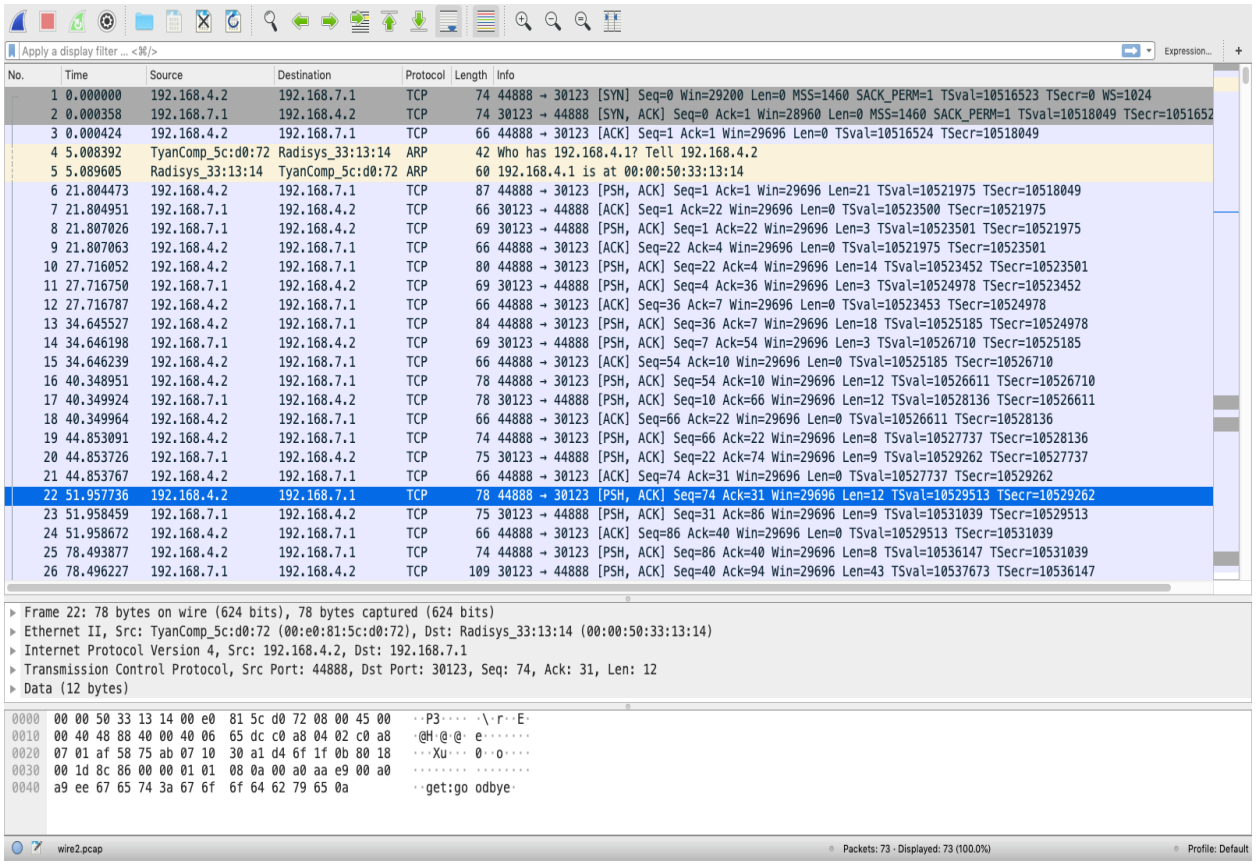
How much time passes between the time this packet is sent and the time the reply arrives? (Note, the reply appears on the next line and the second column of the displays shows the times relative to the start of the capture.)

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 accidently 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*.

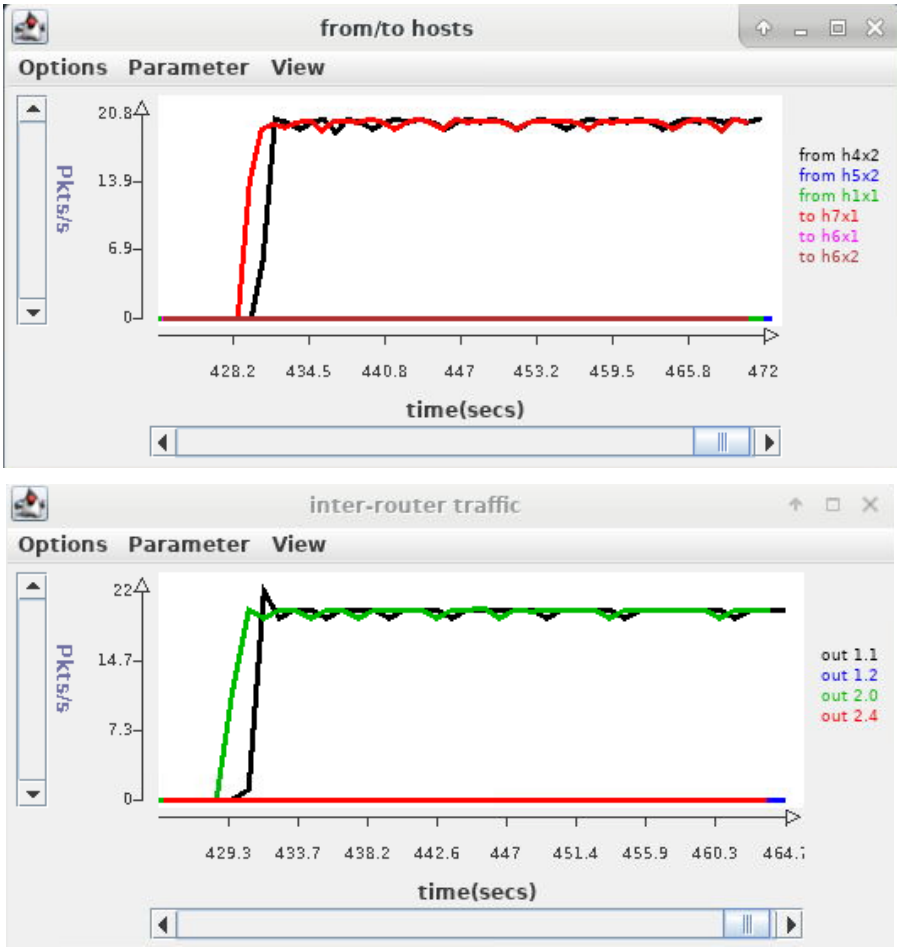


Now, how much time passes between the sending of the packet and the response?

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.

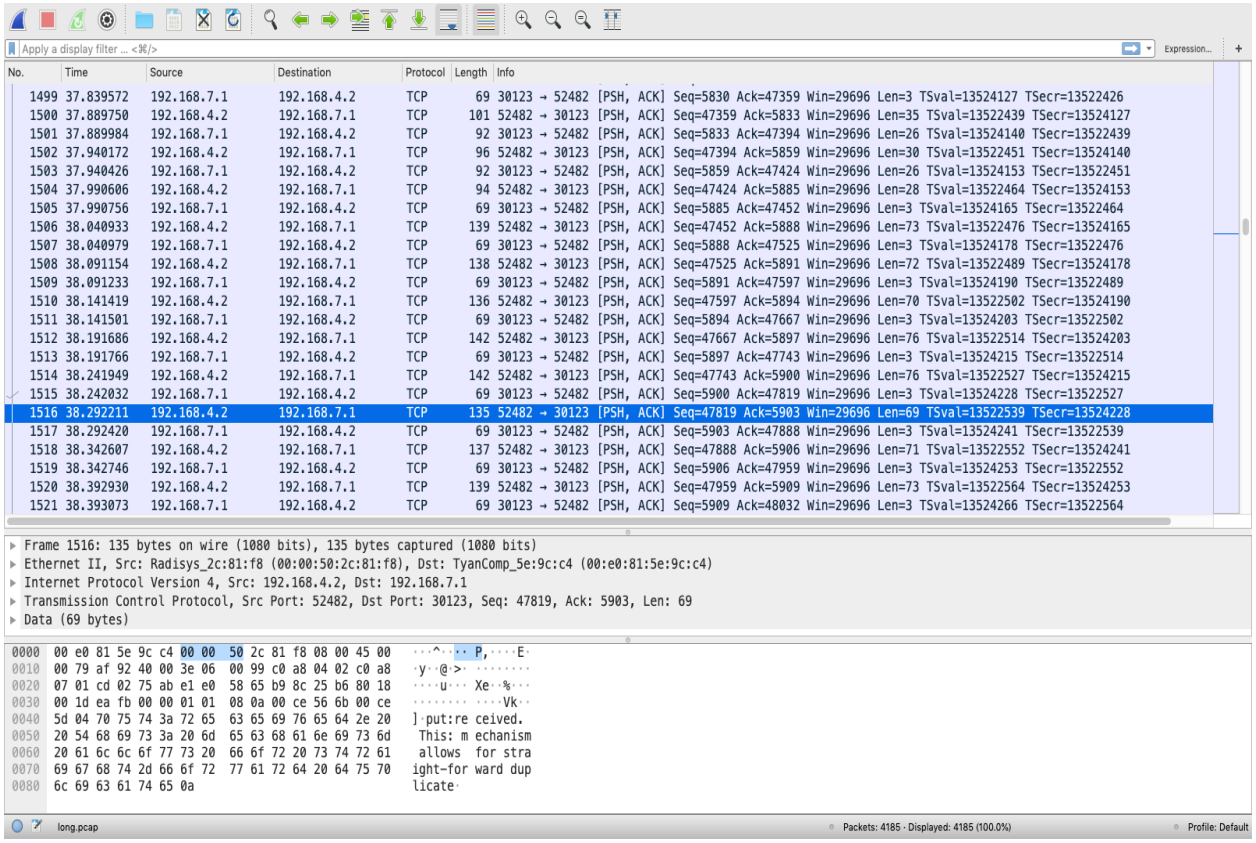


What does the traffic data tell you about the performance of the application?

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.



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.