CSE 473 – Introduction to Computer Networks

Lab 2 Report

David McDonnel 9/28/2015

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

```
// Name: David McDonnel
// WUID: 420325
// Date: 9/30/2015
// Description: TcpMapServer
// - Input:      port - optional argument to specify port #
//               myIp - optional argument to specify host ip
//
// - Behavior:  This program runs a server,
//               accepts input from a client,
//               performs the specified operations on the
//               argument(s),
//               stores arguments to or updates a hashmap


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

public class TcpMapServer {

        private HashMap<String, String> hMap;

        public TcpMapServer() {
                hMap = new HashMap<String, String>();
        }

        /**
         * input String key perform hashmap get on input return success message
         **/
        public String get(String key) {
                String ans = hMap.get(key);
                return ans == null ? "no match" : "success:" + ans;
        }

        /**
         * perform hashmap get on all entries return success message
         **/
        public String getAll() {
```

```
        String ans = "";
        for (String key : hMap.keySet()) {
                ans += "key:" + key + "::" + hMap.get(key) + " ";
        }
        return ans.substring(0, ans.length() - 1);
}

/**
 * input String key, String value perform hashmap put operation on inputs
 * return success messsage
 **/
public String put(String key, String value) {
        boolean b = hMap.containsKey(key);
        String ans = hMap.put(key, value);
        return b ? "updated:" + key : "success";
}

/**
 * input String key perform hashmap remove on input return success message
 **/
public String remove(String key) {
        String ans = hMap.remove(key);
        return ans != null ? "success" : "no match";
}

/**
 * input String key, String value swap two hashmap key value pairs return
 * success message
 **/
public String swap(String key1, String key2) {
        boolean b = hMap.containsKey(key1) && hMap.containsKey(key2);
        String tmp = hMap.get(key1);
        hMap.put(key1, hMap.get(key2));
        hMap.put(key2, tmp);
        return b ? "success" : "no match";
}

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

        int port = 31357;
        InetAddress myIp = null;

        if (args.length == 2) {
                myIp = InetAddress.getByName(args[0]);
                port = Integer.parseInt(args[1]);
        }
        ServerSocket socket = new ServerSocket(port, 0, myIp);
```

```
		byte[] buf = new byte[1000];
		while (true) {
			Socket client = socket.accept();
			// Once connected, create read stream to get client input
			BufferedInputStream sin = new BufferedInputStream(
					client.getInputStream());
			BufferedOutputStream sout = new BufferedOutputStream(
					client.getOutputStream());

			BufferedReader in = new BufferedReader(new
InputStreamReader(sin,"US-ASCII"));
			BufferedWriter out = new BufferedWriter(new
OutputStreamWriter(sout,"US-ASCII"));


			String[] input;
			byte[] outBytes = null;
			String response;
			int length;
			String decoded;
			//int nbytes=0;
			// Instance of class to hold hashmap
			TcpMapServer ms = new TcpMapServer();

			while (true) {

				// wait for incoming packet
				// parse incoming packet
				//nbytes = in.read(buf, 0, buf.length);
				//decoded = new String(buf, 0, nbytes);
				decoded=in.readLine();
				if(decoded == "" || decoded == null){
					break;
				}
				input = decoded.split(":");
				length = input.length;
				// handle all cases of operation input
				// For each case test if correct # of arguments
				// Then record response or mark as bad data
				switch (input[0]) {
				case "get":
					response = length == 2 ? ms.get(input[1]) : "bad";
					break;
				case "put":
					response = length == 3 ? ms.put(input[1], input[2]) : "bad";
					break;
				case "remove":
```

```
                        response = length == 2 ? ms.remove(input[1]) : "bad";
                        break;
                case "swap":
                        response = length == 3 ? ms.swap(input[1], input[2])
                                        : "bad";
                        break;
                case "get all":
                        if(ms.hMap.isEmpty()){
                                response = "empty";
                        }else{
                                response = length == 1 ? ms.getAll() : "bad";
                        }
                        break;
                default:
                        response = "bad";
                        break;
                }
                // If bad data respond with error
                if (response == "bad") {
                        response = "error:unrecognizable input:" + decoded;
                }

                // and send it back
                //outBytes = response.getBytes("US-ASCII");

                //out.write(outBytes, 0, outBytes.length);
                //out.flush();
                out.write(response);
                out.newLine();
                out.flush();
            }
            in.close();
            out.close();
            sin.close();
            sout.close();
            client.close();
            continue;
        }
    }
}
```

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

```
// Name: David McDonnel
// WUID: 420325
```

```
// Date: 9/30/2015
// Description: TcpMapServer
// - Input:        port - optional argument to specify port #
//                 myIp - optional argument to specify host ip
//
// - Behavior:  This program runs a server,
//                 accepts input from a client,
//                 performs the specified operations on the
//                 argument(s),
//                 stores arguments to or updates a hashmap


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

/**
 * usage: TcpMapClient name port operation arg1 arg2 Input: name - String port -
 * integer operation - String arg1 - String arg2 - String Send a packet to the
 * named server:port containing the given arguments. Wait for reply packet and
 * print its contents.
 **/

public class TcpMapClient {
        public static void main(String args[]) throws Exception {
                // get server address using first command-line argument
                InetAddress serverAdr = InetAddress.getByName(args[0]);
                int port = 31357;
                if (args.length == 2) {
                        port = Integer.parseInt(args[1]);
                }

                Socket socket = new Socket(serverAdr, port);
                BufferedReader in = new BufferedReader(new InputStreamReader(
                                socket.getInputStream(), "US-ASCII"));
                BufferedWriter out = new BufferedWriter(new OutputStreamWriter(
                                socket.getOutputStream(), "US-ASCII"));

                BufferedReader sysin = new BufferedReader(new InputStreamReader(
                                System.in));
                String line;

                while (true) {
                        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());
                }

                socket.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.
paste your output here

```
success
success
success
success:slim jim
success:ho ho
success:world
key:goodbye::world key:hah::ho ho key:foo bar::slim jim
error:unrecognizable input:get
error:unrecognizable input:foo:who
no match
success
success:toast is tasty
updated:hah
success
key:goodbye::world key:foo::yolo key:foo bar::slim jim
key:hah::toast is tasty
empty
no match
empty
no match
empty
```

Part D (15 points). In the remaining parts of the lab, you will be testing your application in ONL. 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 31357
and paste a copy of the output below.

```
dmcdonnel@pc1core07:~$ netstat -an | grep 31357
tcp6       0      0 :::31357                :::*
LISTEN
```

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

```
dmcdonnel@pc1core07:~$ netstat -an | grep 31357
tcp6       0      0 :::31357                :::*
LISTEN
tcp6       0      0 192.168.7.1:31357       192.168.4.2:50093
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). (Ignore the fact that it may say 'tcp6' when you expect 'tcp' or 'tcp4'. That is an artifact of netstat and sockets.)

When only running the server, the output of netstat is a tcp connection running on localhost at port 31357, listening for all foreign addresses. There are 0 bytes in the recv-q and the send-q.

When running the server and client, the server client is the same. The client socket is an established tcp socket connected to h7x1:31357, from h4x2:50093 with 0 bytes in both the send and receive q's.

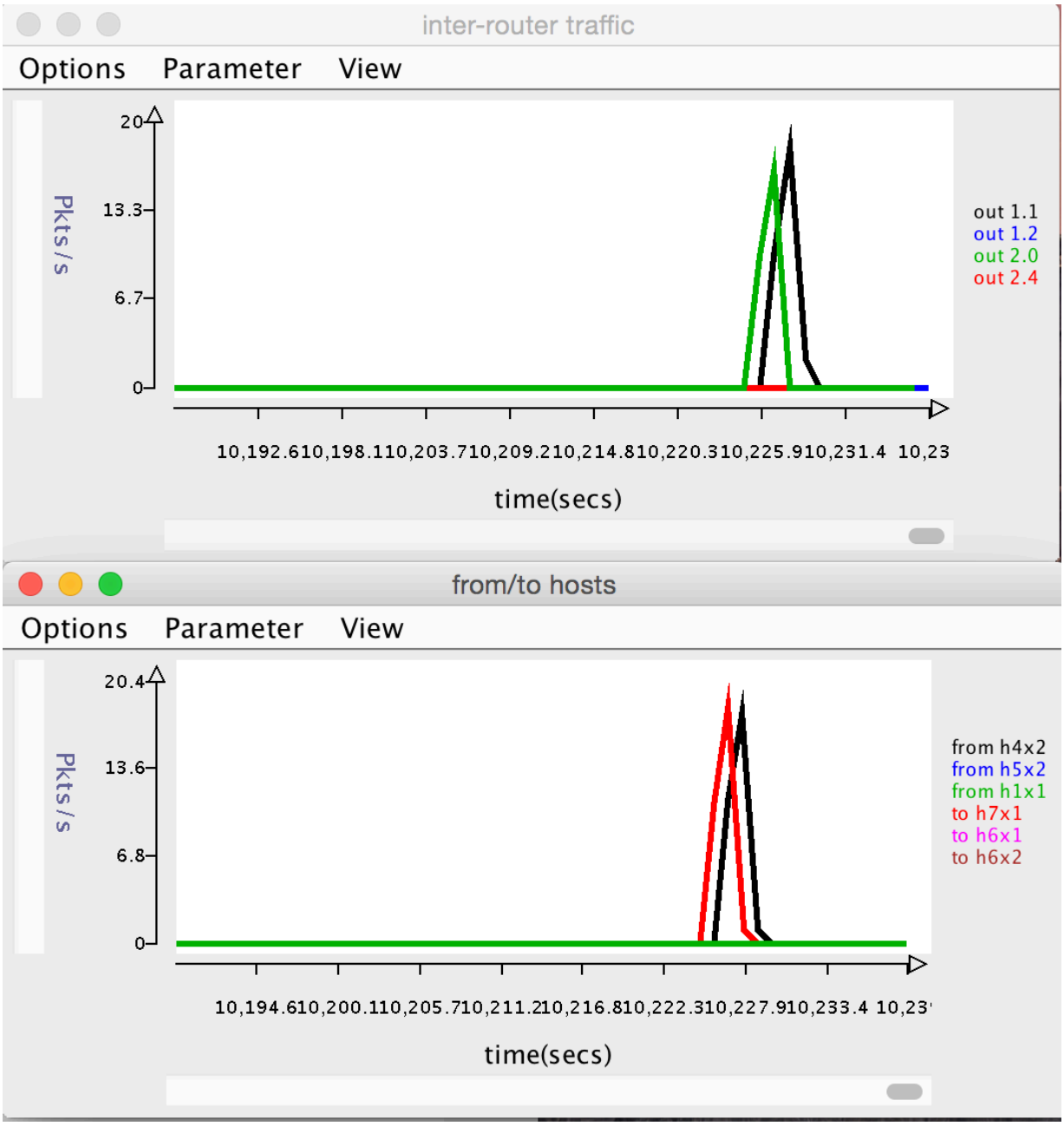
In the first case only one tcp socket was open on 31357, the listener.
After the client was started, the listener socket remained, and the client tcp socket was opened.

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

```
dmcdonnel@pc1core12:~$ ./remoteScript h7x1
success
success
success
```

```
success:slim jim
success:ho ho
success:world
key:goodbye::world key:hah::ho ho key:foo bar::slim jim
error:unrecognizable input:get
error:unrecognizable input:foo:who
no match
success
success:toast is tasty
updated:hah
success
key:goodbye::world key:foo::yolo key:hah::toast is tasty key:foo
bar::slim jim
empty
no match
empty
no match
empty
```

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.
paste your screenshot here



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 inter-router traffic, 1.1 and 2.0 have traffic, which is consistent with my expectations. The route from router 1:1 to router 2:0 is the quickest route between these two machines. The other possible route was through the gige0.

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. You will need X-windows support on your local computer to make this work. The PCs in the Urbauer labs are configured to support X, so we'll start with the procedure used there. Start by opening a new command window on your local computer and type

```
startxwin
```

This will open a new window. In this new window, type

```
ssh –X myLogin@onl.wustl.edu
```

This creates an *ssh* connection that forwards "X-windows" commands from *onlusr* back to your local computer. 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 –X $h4x2
```

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

```
sudo wireshark
```

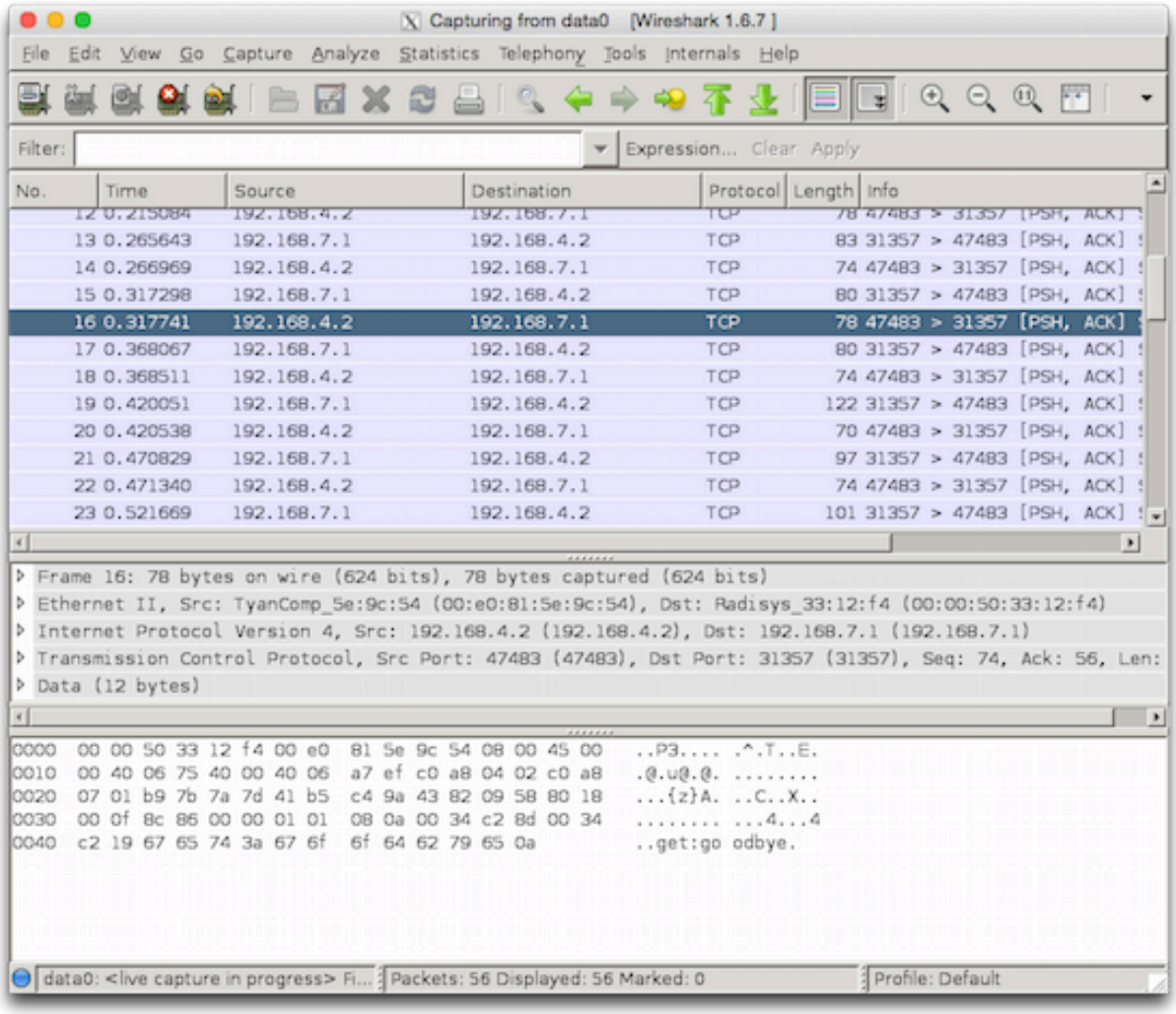
After you enter your *onl* password, *Wireshark* will start running on *h4x2*, and the *Wireshark* window will open on your local computer. If you want to do this part of the lab using your own computer, you may have to do some initial configuration. If you have a Mac or a Linux computer, with *Wireshark* 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 (note, you don't need the *startxwin* step). If you are using a Windows computer, there are several options for installing X, but the most straightforward is probably to install *cygwin*, and configure it to enable X (this is the approach used in the Urbauer lab computers). *Cygwin* is a Unix emulation environment that runs on Windows. Use your favorite search engine to find the *cygwin* site, read the instructions and then download/install. Once you have things setup, just follow the procedure given above for the Urbauer computers.

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.

Paste a copy of the Wireshark window here.



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.)

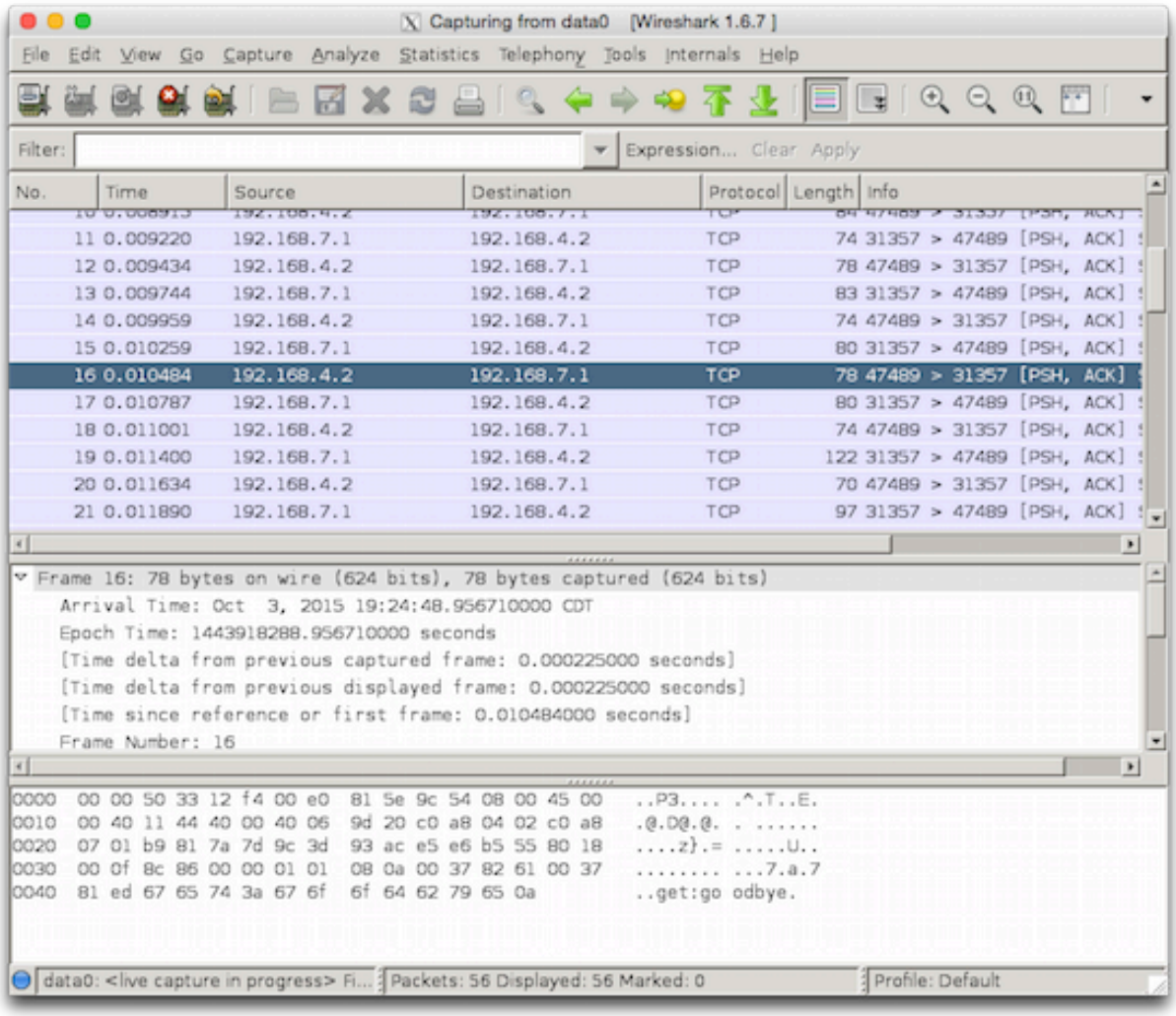
.050326000 seconds

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?

NPR.1 contains the delay plugin.

Find the filter that causes packets to pass through the plugin and turn it off, using the RLI. Don't forget to commit after disabling the filter. Now, start a new *Wireshark capture* and re-run *remoteScript*.

Paste a copy of the new Wireshark window here.



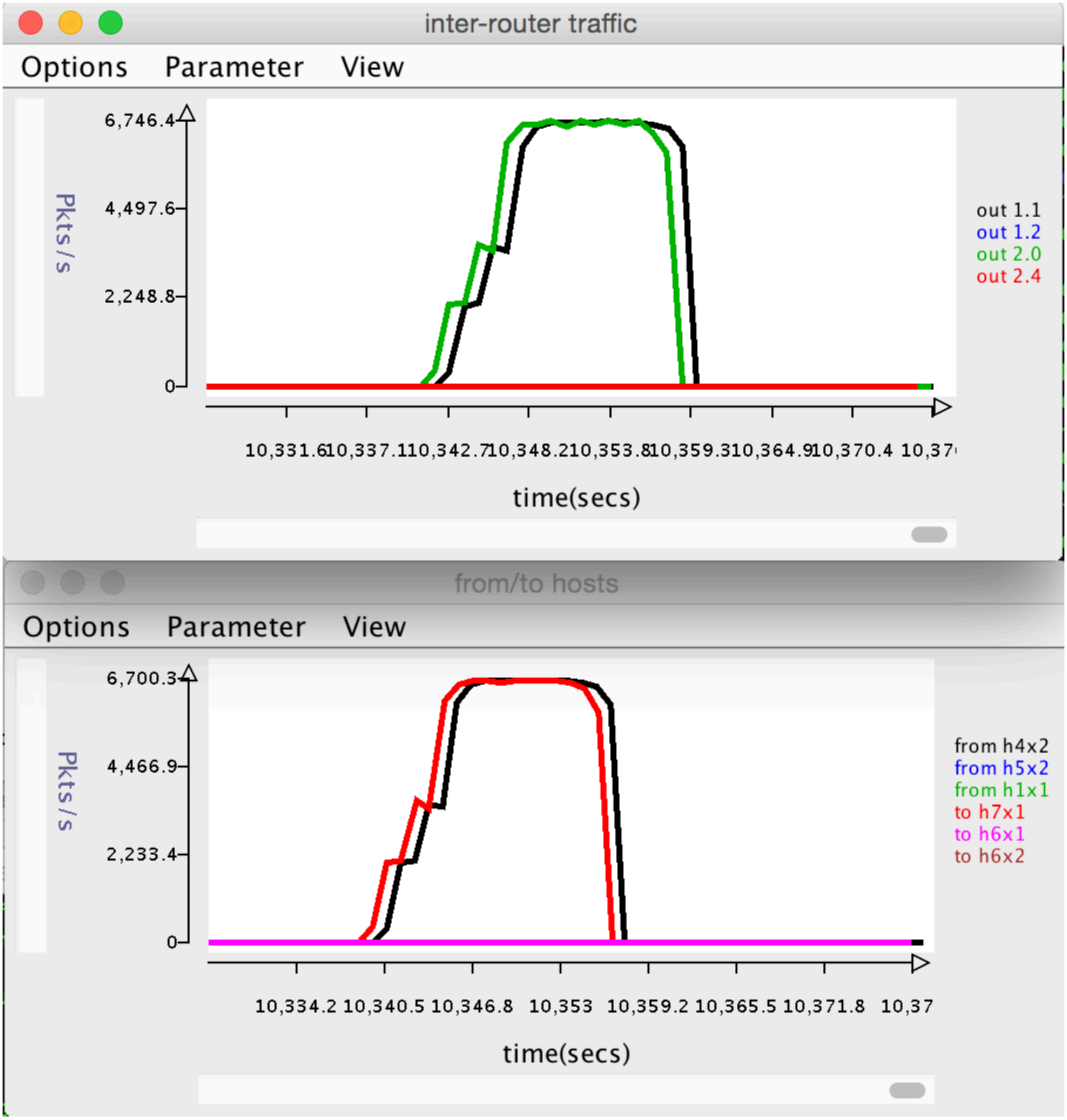
Now, how much time passes between the sending of the packet and the response?

.000303000 seconds

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.

(The filter directing packets to the delay plugin should still be turned OFF!)

Paste the screenshot here



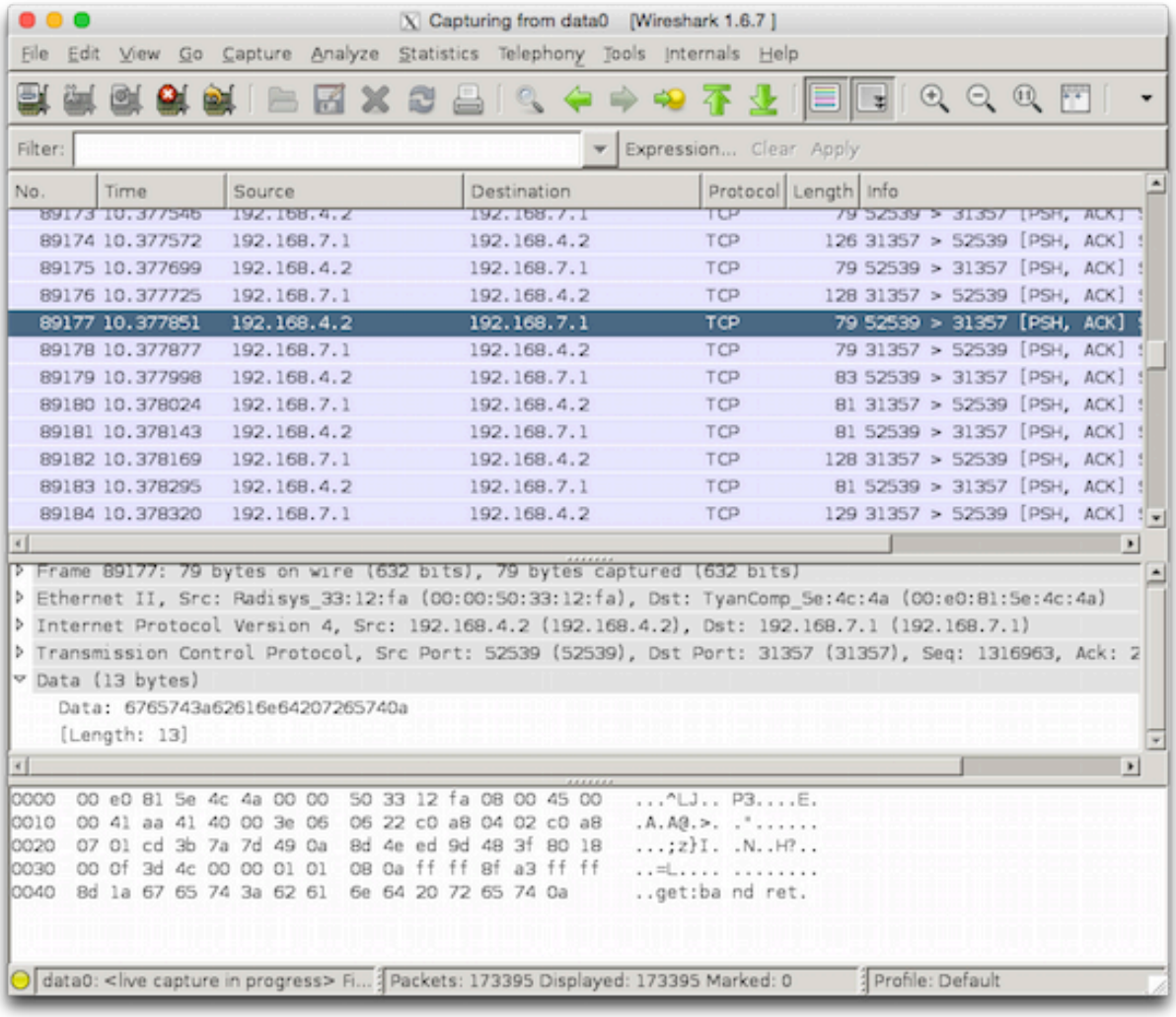
What does the traffic data tell you about the performance of the application?

This informs us that our application can transfer about 6700.3 packets/s.

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.

Paste a copy of your Wireshark window here.



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

0.000026 seconds

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

0.000147 seconds

Is this consistent with what you observed based on the packet rate chart?
Yes, 1/6700.3 = .000149 seconds which is almost identical to the lag between sent packets.