#### 

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Name: Richard Wu Student ID: 464493

Name: Jiaming Qiu Student ID: 467620

Date: 2019/9/13

This program system includes a client program(TcpMapClient) and a server

program (TcpMapServer). The server will store a set of key and value pairs

and the client can send instructions to server to get, get all, put, or

remove any pair stored. The server will accept TCP connections from

clients, one at a time, and will process multiple operations based on

client's input until the client closes the connection. The format of using

the client program is listed as follows: the first argument is the name/IP

address of the host that the server is running on; the second argument

is the server's port number, which is optional and will use 30123 if not

specified. In the input line, the "get" instruction should be in the format

of "get:key"; the "get all" instruction should just be "get all"; the "put"

instruction should be in the format of "put:key:value"; the "remove"

instruction should be in the format of "remove:key". Any other

instruction will be considered ill-formatted. When the server receives

an instruction, it will send back a feedback to the client. When the user

inputs a blank line, the connection will be closed. To start the server, the

first optional argument should be the IP address (wildcard address if not

specified), and the second optional argument should be the port number.

(30123 by default and if the port number is specified, IP address should

also be specified)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class TcpMapServer {

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

// Process arguments

int port = 30123; // Default port number

if (args.length > 1) port = Integer.parseInt(args[1]);

InetAddress bindAdr = null;

if (args.length > 0) bindAdr = InetAddress.getByName(args[0]);

// Create and bind listening socket

ServerSocket listenSock = new ServerSocket(port,0,bindAdr);

// Stream buffer

byte[] buf = new byte[1000];

// Create HashMap to store the data

HashMap<String, String> map = new HashMap<>();

while (true) {

// Wait for incoming connection request and

// Create new socket to handle it

Socket connSock = listenSock.accept();

// Create buffered versions of socket's in/out streams

BufferedInputStream in = new BufferedInputStream(

connSock.getInputStream());

BufferedOutputStream out = new BufferedOutputStream(

connSock.getOutputStream());

// Connect with one client

while (true) {

int nbytes = in.read(buf, 0, buf.length);

if (nbytes < 0) break; // -1: the end of the stream

// Turn the data into string

String dataString = new String(buf, 0, nbytes, "US-ASCII");

// Manipulate the data

dataString = dataString.trim();

String[] dataStrings = dataString.split(":");

// Return string

String returnString = "";

// get instruction

if("get".equals(dataStrings[0]) &&

dataStrings.length == 2) {

String value = map.get(dataStrings[1]);

if(value == null) {

returnString = "no match";

} else {

returnString = "ok:"+value;

}

}

// put instruction

else if("put".equals(dataStrings[0]) &&

dataStrings.length == 3) {

String previous = map.put(dataStrings[1], dataStrings[2]);

if(previous == null) {

returnString = "Ok";

} else {

returnString = "updated:"+dataStrings[1];

}

}

// remove instruction

else if("remove".equals(dataStrings[0]) &&

dataStrings.length == 2) {

String removed = map.remove(dataStrings[1]);

if(removed == null) {

returnString = "no match";

} else {

returnString = "Ok";

}

}

// get all instruction

else if ("get all".equals(dataStrings[0]) &&

dataStrings.length == 1) {

Iterator<Map.Entry<String,String>> it = map.entrySet().iterator();

// Iterate all entries

if(it.hasNext()) {

Map.Entry<String,String> pair = it.next();

returnString+=(pair.getKey()+":"+pair.getValue());

}

while(it.hasNext()) {

Map.Entry<String,String> pair = it.next();

returnString+=("::"+pair.getKey()+":"+pair.getValue());

}

} else { // invalid instruction

returnString = "error:unrecognizable input:"+dataString;

}

returnString+="\n"; // String terminates with '\n'

out.write(returnString.getBytes(),0,returnString.length());

out.flush(); // Flush the buffer

}

connSock.close();

}

}

}

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Name: Richard Wu Student ID: 464493

Name: Jiaming Qiu Student ID: 467620

Date: 2019/9/13

This program system includes a client program(TcpMapClient) and a server

program (TcpMapServer). The server will store a set of key and value pairs

and the client can send instructions to server to get, get all, put, or

remove any pair stored. The server will accept TCP connections from

clients, one at a time, and will process multiple operations based on

client's input until the client closes the connection. The format of using

the client program is listed as follows: the first argument is the name/IP

address of the host that the server is running on; the second argument

is the server's port number, which is optional and will use 30123 if not

specified. In the input line, the "get" instruction should be in the format

of "get:key"; the "get all" instruction should just be "get all"; the "put"

instruction should be in the format of "put:key:value"; the "remove"

instruction should be in the format of "remove:key". Any other

instruction will be considered ill-formatted. When the server receives

an instruction, it will send back a feedback to the client. When the user

inputs a blank line, the connection will be closed. To start the server, the

first optional argument should be the IP address (wildcard address if not

specified), and the second optional argument should be the port number.

(30123 by default and if the port number is specified, IP address should

also be specified)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.io.\*;

import java.net.\*;

public class TcpMapClient {

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

// Connect to remote server

int port = 30123; // Default port number

if (args.length > 1) port = Integer.parseInt(args[1]); // port number

Socket sock = new Socket(args[0], port); // Socket created

// Create buffered reader & writer for socket's in/out streams

BufferedReader in = new BufferedReader(new InputStreamReader(

sock.getInputStream(),"US-ASCII"));

BufferedWriter out = new BufferedWriter(new OutputStreamWriter(

sock.getOutputStream(),"US-ASCII"));

// Create buffered reader for System.in

BufferedReader sysin = new BufferedReader(new InputStreamReader(

System.in));

String line; // Input line

while (true) {

line = sysin.readLine(); // Read the line

if (line == null || line.length() == 0) break; // Blank line

// Write line on socket and print feedback to System.out

out.write(line); out.newLine(); out.flush();

System.out.println(in.readLine());

}

sock.close(); // Close connection

}

}***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.

wuyuanpeidembp:lab2 wuyuanpei$ ./localScript

Ok

Ok

Ok

ok:slim jim

ok:ho ho

ok:world

goodbye:world::foo bar:slim jim::hah:ho ho

error:unrecognizable input:get

error:unrecognizable input:foo:who

no match

Ok

ok:toast is tasty

updated:hah

goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo

goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo

no match

goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo

Ok

goodbye:world::foo:toast is tasty::hah:yolo3

***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.

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.

tcp6 0 0 192.168.7.1:30123 :::\* LISTEN

tcp6 0 0 192.168.7.1:30123 192.168.4.2:54618 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).

*The first line from both cases indicate that our socket uses tcp6 protocol and is listening for incoming connections, which means that the server is running and has built the listening socket successfully.*

*The second line from the second case indicates that out socket now has an established connection, which means that the client is started and connected to the server. Connection Socket has been created to exchange data.*Now, run the provided *remoteScript* on *h4x2*. Paste the output from your run below.

Ok

Ok

Ok

ok:slim jim

ok:ho ho

ok:world

goodbye:world::foo bar:slim jim::hah:ho ho

error:unrecognizable input:get

error:unrecognizable input:foo:who

no match

Ok

ok:toast is tasty

updated:hah

goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo

goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo

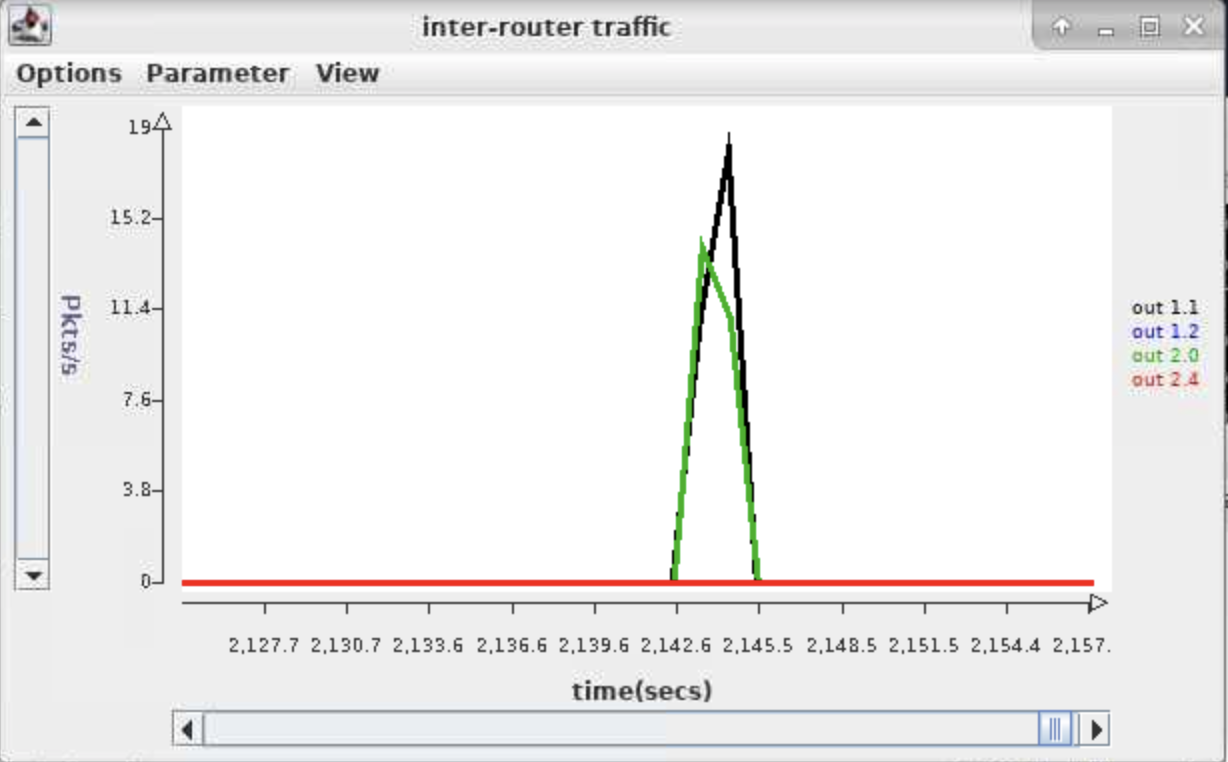
no match

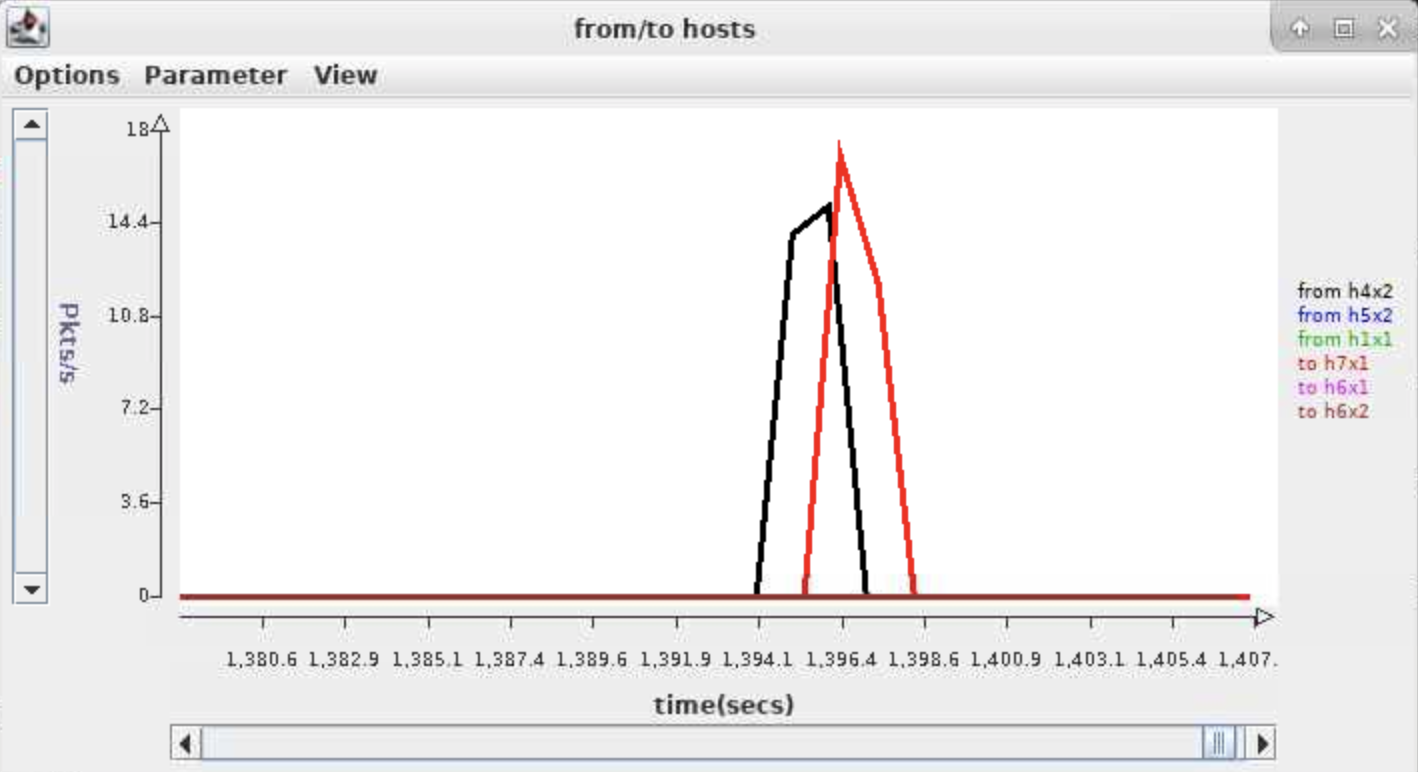
goodbye:world::foo:toast is tasty::foo bar:slim jim::hah:yolo

Ok

goodbye:world::foo:toast is tasty::hah:yolo

***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.





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 first graph, out 1.1 and out 2.0 show a burst of traffic. In the second graph, from h4x2 and to h7x1 show a burst of traffic. This is consistent with what we expect about the communication between h4x2 and h7x1. There are two possible routes between the two end hosts. One is from NPR 1.1 to NPR 2.0 and the second is from NPR 1.3 to GigE switch to NPR 2.4. In this case the route from NPR 1.1 to NPR 2.0 is used, noticed that out 1.1 and out 2.0 show a burst of traffic.*

***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

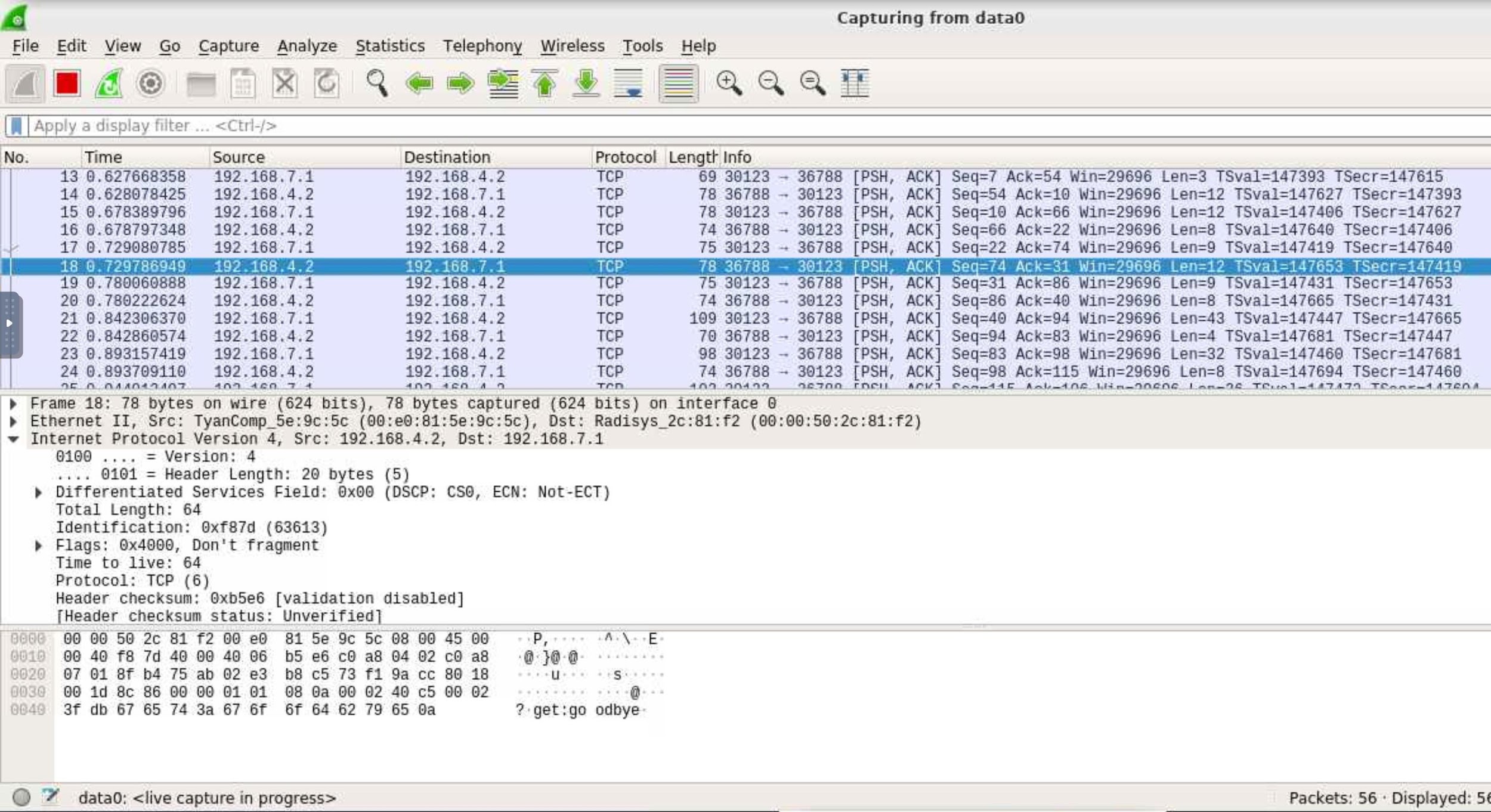
sudo wireshark

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.

**

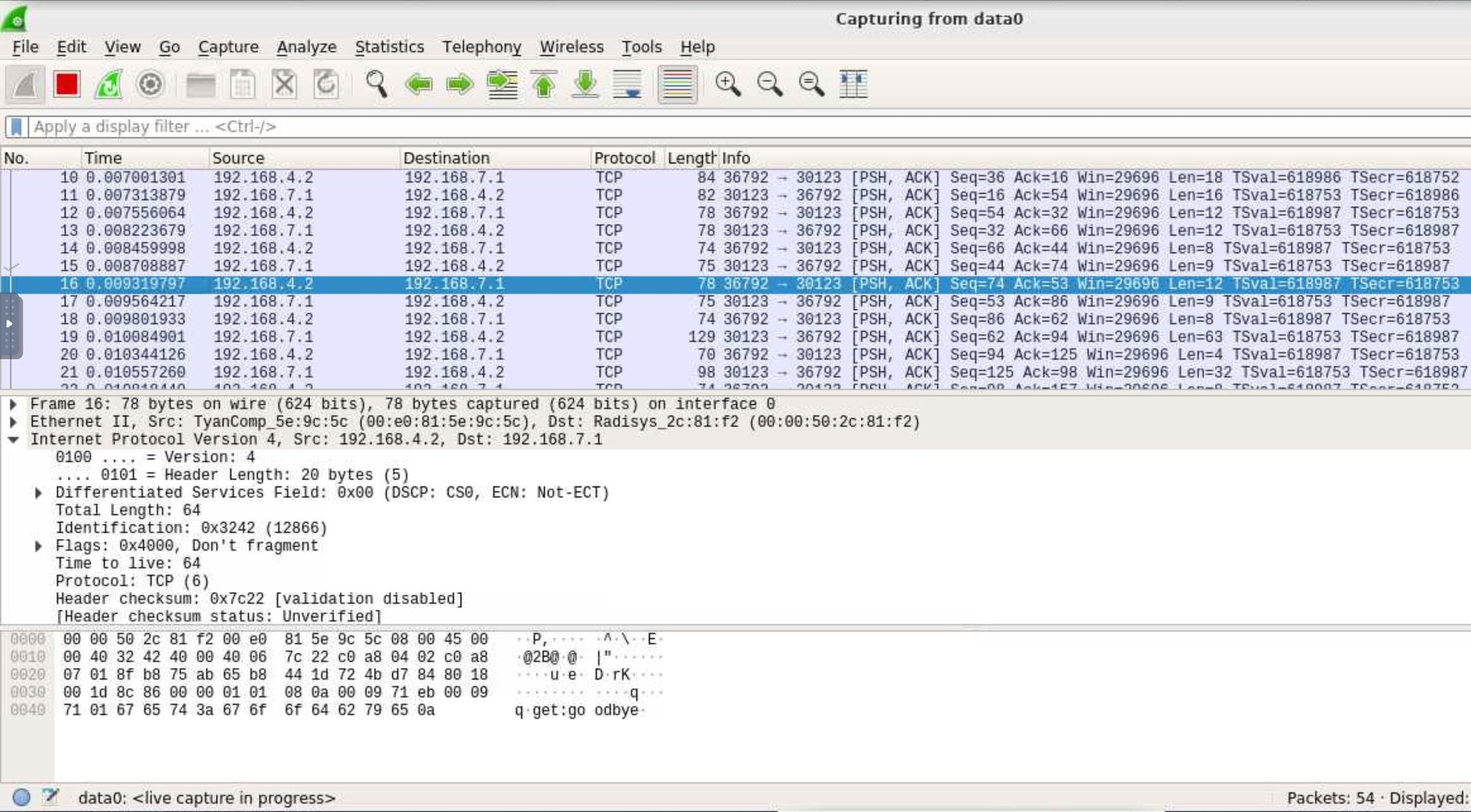
How much time passes between the time this packet is sent and the time the reply arrives? (Note, the reply appears on the next line and the second column of the displays shows the times relative to the start of the capture.)

*h4x2 sent the packet at 0.729786949. It then received the reply at 0.780060888. The time passed is 0.050273939 second.*

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*

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?

*Now, h4x2 sent the packet at 0.009319797. It then received the reply at 0.009564217. The time passed is 0.00024442 second.*

***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.

*A screenshot of a social media post

Description automatically generatedA screenshot of a cell phone

Description automatically generated*

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

*The performance of the application: sending about 5800 packets and receiving about 5800 packets per second in both the server and client sides.*

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.

*A screenshot of a social media post

Description automatically generated*

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

*Take packet No. 77838 and 77839 as an example, h7x1 received the packet at 7.807261706. It then sent the reply at 7.807297741. The time passed is 0.000036035 second.*

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

*Take packet No. 77838 and 77840 as an example, h7x1 received the first packet at 7.807261706. It received the second packet at 7.807439089. The time passed is 0.000177383 second.*

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

*Yes, because the time difference between two adjacent packets received is 0.000177383 second, which means it can receive 1/0.000177383 = 5637.52 packets per second, which is basically consistent with the packet rate chart.*