

## Exercise 3: Using Wireshark to understand basic HTTP request/response messages (marked, include in your report)

We will not be running Wireshark on a live network connection (You are strongly encouraged to try this on your own machine. Pointers provided at the end of this exercise). The CSE network administrators do not permit live traffic monitoring for security reasons. Instead, for all our lab exercises we will make use of several trace files, which were collected by running Wireshark by one of the textbook's authors. For this particular experiment download the following trace file: [http-ethereal-trace-1](http://ethereal-trace-1)

The following indicate the steps involved:

**Step 1:** Start Wireshark by typing *wireshark* at the command prompt.

**Step 2:** Load the trace file *http-ethereal-trace-1* by using the *File* pull down menu, choosing *Open* and selecting the appropriate trace file. This trace file captures a simple request/response interaction between a browser and a web server.

**Step 3:** You will see a large number of packets in the packet-listing window. Since we are currently only interested in HTTP we will filter out all the other packets by typing "http" in lower-case in the *Filter* field and press Enter. You should now see only HTTP packets in the packet-listing window.

**Step 4:** Select the first HTTP message in the packet-listing window and observe the data in the packet-header detail window. Recall that since each HTTP message was carried inside a TCP segment, which was carried inside an IP datagram, which was carried within an Ethernet frame, Wireshark displays the Frame, Ethernet, IP, and TCP packet information as well. We want to minimize the amount of non-HTTP data displayed (we're interested in HTTP here, and will be investigating these other protocols in later labs), so make sure the boxes at the far left of the Frame, Ethernet, IP and TCP information have a right-pointing arrowhead (which means there is hidden, undisplayed information), and the HTTP line has a down-pointing arrowhead (which means that all information about the HTTP message is displayed).

**NOTE:** Please neglect the HTTP GET and response for *favicon.ico*, (the third and fourth HTTP messages in the trace file. Most browsers automatically ask the server if the server has a small icon file that should be displayed next to the displayed URL in the browser. We will ignore references to this pesky file in this lab.)

By looking at the information in the HTTP GET and response messages (the first two messages), answer the following questions:

**Question 1: What is the status code and phrase returned from the server to the client browser?**

**Answer 1: The status code is '200' and phrase is 'OK'.**

```
HTTP/1.1 200 OK\r\n
```

**Question 2:** When was the HTML file that the browser is retrieving last modified at the server? Does the response also contain a DATE header? How are these two fields different?

**Answer 2:** Last-Modified time is Tuesday, 23/09/2003, 05:29:00. There is a DATE header in the response.

Last-Modified means that the HTML file last modified at the server and the DATE header can be compared with current time, which can be determined whether the response is from cache or the original server.

```
Last-Modified: Tue, 23 Sep 2003 05:29:00 GMT\r\n
```

```
Date: Tue, 23 Sep 2003 05:29:50 GMT\r\n
```

**Question 3:** Is the connection established between the browser and the server persistent or non-persistent? How can you infer this?

**Answer 3:** The connection established is persistent, because I find the 'Keep-alive' and 'connection: Keep-Alive', which asks server to keep the connection. Therefore, it can use the same TCP connection to keep communicating. In addition, the HTTP 1.1 version has default persistent status, so I can infer this.

```
Keep-Alive: timeout=10, max=100\r\nConnection: Keep-Alive\r\n
```

**Question 4:** How many bytes of content are being returned to the browser?

**Answer 4:** 73 bytes.

```
Content-Length: 73\r\n
```

**Question 5:** What is the data contained inside the HTTP response packet?

**Answer 5:** It's a text/html file. The content is:

```
<html>\nCongratulations. You've downloaded the file lab2-1.html!\n</html>\n
```

## Exercise 4: Using Wireshark to understand the HTTP CONDITIONAL GET/response interaction (marked, include in your report)

For this particular experiment download the second trace file: <http-ethereal-trace-2>

The following indicate the steps for this experiment:

**Step 1:** Start Wireshark by typing *wireshark* at the command prompt.

**Step 2:** Load the trace file *http-ethereal-trace-2* by using the *File* pull down menu, choosing *Open* and selecting the appropriate trace file. This trace file captures a request response between a client browser and web server where the client requests the same file from the server within a span of a few seconds.

**Step 3:** Filter out all the non-HTTP packets and focus on the HTTP header information in the packet-header detail window.

By looking at the information in the HTTP GET and response messages (the first two messages), answer the following questions:

**Question 1:** Inspect the contents of the first HTTP GET request from the browser to the server. Do you see an “IF-MODIFIED-SINCE” line in the HTTP GET?

**Answer 1:** No.

**Question 2:** Does the response indicate the last time that the requested file was modified?

**Answer 2:** Yes.

```
Last-Modified: Tue, 23 Sep 2003 05:35:00 GMT\r\n
```

**Question 3:** Now inspect the contents of the second HTTP GET request from the browser to the server. Do you see an “IF-MODIFIED-SINCE:” and “IF-NONE-MATCH” lines in the HTTP GET? If so, what information is contained in these header lines?

**Answer 3:** Yes, “IF-MODIFIED-SINCE:” is from browser. According to previous Last-Modified time from server, the browser will include “IF-MODIFIED-SINCE:” in GET request and server can find out whether the browser gets the latest HTTP file. If it is the latest HTTP file in browser, the server will return 304 to tell browser the HTTP file is the latest. It will decrease data transmission on the Internet. “IF-NONE-MATCH” is similar to “IF-MODIFIED-SINCE:”, but it can find more detail.

```
If-Modified-Since: Tue, 23 Sep 2003 05:35:00 GMT\r\n
If-None-Match: "1bfef-173-8f4ae900"\r\n
```

**Question 4: What is the HTTP status code and phrase returned from the server in response to this second HTTP GET? Did the server explicitly return the contents of the file? Explain.**

**Answer 4: The status code is 304 and phrase is Not Modified. The server didn't return the contents of file. As I mentioned above, if the HTTP file doesn't change, the server will just return 304 and the browser will load the webpage that was stored in browser cache. This method not only can decrease the congestion of Internet, but also can reduce the work of server.**

```
HTTP/1.1 304 Not Modified\r\n
```

**Question 5: What is the value of the Etag field in the 2nd response message and how it is used? Has this value changed since the 1<sup>st</sup> response message was received?**

**Answer 5: The Etag is 1bfef-173-8f4ae900. The function of Etag and "IF-NONE-MATCH" is similar to "LAST-MODIFIED" and "IF-MODIFIED-SINCE:". Their difference is that the Etag and "IF-NONE-MATCH" can find out more detail. And we can find that in the 2<sup>nd</sup> response, the content of "IF-NONE-MATCH" is same as Etag. Therefore, we can infer that the HTTP file isn't changed. This value in the 1<sup>st</sup> response is same as in the 2<sup>nd</sup> response because the HTTP file isn't changed.**

**2<sup>nd</sup> response:**

```
Etag: "1bfef-173-8f4ae900"\r\n
```

```
If-None-Match: "1bfef-173-8f4ae900"\r\n
```

**1<sup>st</sup> response:**

```
Etag: "1bfef-173-8f4ae900"\r\n
```

## (\*) Exercise 5: Ping Client

**Note: This exercise is to be included in the report and demonstrated to the tutor during your lab.**

In this exercise, you will study a simple Internet ping server (written in Java), and implement a corresponding client. The functionality provided by these programs are similar to the standard ping programs available in modern operating systems, except that they use UDP rather than Internet Control Message Protocol (ICMP) to communicate with each other. Note that, we will study ICMP later in the course. (Most programming languages do not provide a straightforward means to interact with ICMP)

The ping protocol allows a client machine to send a packet of data to a remote machine and have the remote machine return the data back to the client unchanged (an action referred to as echoing). Among other uses, the ping protocol allows hosts to determine round-trip times to other machines.

### **Ping Server (provided)**

You are given the complete code for the Ping server: [PingServer.java](#). The server sits in an infinite loop listening for incoming UDP packets. When a packet comes in, the server simply sends the encapsulated data back to the client.

Your task is to write the corresponding Ping client. You can use either C, Java or Python to write your client. You should read through the server code thoroughly, as it will help you with the development of your client program.

### **Packet Loss & Delay**

UDP provides applications with an unreliable transport service, because messages may get lost in the network due to router queue overflows or other reasons. In contrast, TCP provides applications with a reliable transport service and takes care of any lost packets by retransmitting them until they are successfully received. Applications using UDP for communication must therefore implement any reliability they need separately in the application level (each application can implement a different policy, according to its specific needs). Because packet loss is rare or even non-existent in typical campus networks, the server in this lab injects artificial loss to simulate the effects of network packet loss. The server has a parameter `LOSS_RATE` that specifies the percentage of packets that are lost (i.e. dropped). The server also has another parameter `AVERAGE_DELAY` that is used to simulate the delay incurred by packets in the Internet. You should set `AVERAGE_DELAY` to a positive value when testing your client and server on the same machine, or when machines are close by on the network. You can set `AVERAGE_DELAY` to 0 to find out the true round-trip time between the client and server.

### **Compiling and Running Server**

To compile the server, type the following at the command prompt:

```
$javac PingServer.java
```

To run the server, type the following:

```
$java PingServer port
```

where *port* is the port number the server listens on. Remember that you should pick a port number greater than 1024, because only processes running with root (administrator) privilege can bind to ports less than 1024. If you get a message that the port is in use, try a different port number as the one you chose may be in use.

**Note:** if you get a class not found error when running the above command, then you may need to tell Java to look in the current directory in order to resolve class references. In this case, the commands will be as follows:

```
$java -classpath . PingServer port
```

## Your Task: Implementing Ping Client

You should write the client (called [PingServer.c](#) PingClient.java or PingClient.c or PingClient.py) such that it sends 10 ping requests to the server, separated by approximately one second. Each message contains a payload of data that includes the keyword PING, a sequence number, and a timestamp. After sending each packet, the client waits up to one second to receive a reply. If one second goes by without a reply from the server, then the client assumes that its packet or the server's reply packet has been lost in the network.

You should write the client so that it executes with the following command:

```
$java PingClient host port (for Java)
```

or

```
$PingClient host port (for C)
```

or

```
$python PingClient.py host port (for Python)
```

where *host* is the IP address of the computer the server is running on and *port* is the port number it is listening to. In your lab you will be running the client and server on the same machine. So just use 127.0.0.1 (i.e., localhost) for *host* when running your client. In practice, you can run the client and server on different machines.

The client should send 10 pings to the server. Because UDP is an unreliable protocol, some of the packets sent to the server may be lost, or some of the packets sent from server

to client may be lost. For this reason, the client cannot wait indefinitely for a reply to a ping message. You should have the client wait up to one second for a reply; if no reply is received, then the client should assume that the packet was lost during transmission across the network. It is important that you choose a reasonably large value, which is greater than the expected RTT (Note that the server artificially delays the response using the `AVERAGE_DELAY` parameter). In order to achieve this your socket will need to be non-blocking (i.e. it must not just wait indefinitely for a response from the server). If you are using Java, you will need to research the API for `DatagramSocket` to find out how to set the timeout value on a datagram socket (Check:<http://java.sun.com/javase/6/docs/api/java/net/Socket.html> ). If you are using C, you can find information here:<http://www.beej.us/guide/bgnet/output/html/singlepage/bgnet.html#datagram> . Note that, the `fcntl()` function is the simplest way to achieve this.

Note that, your client should not send all 10 ping messages back-to-back, but rather sequentially. The client should send one ping and then wait either for the reply from the server or a timeout before transmitting the next ping. Upon receiving a reply from the server, your client should compute the RTT, i.e. the difference between when the packet was sent and the reply was received. There are functions in Java Python and C that will allow you to read the system time in milliseconds. The RTT value should be printed to the standard output (similar to the output printed by ping; have a look at the output of ping for yourself). An example output could be:

```
ping to 127.0.0.1, seq = 1, rtt = 120 ms
```

We prefer that you show the timeout requests in the out put. Only replace the 'rtt=120ms' in the above example with 'time out'. You will also need to report the minimum, maximum and the average RTTs of all packets received successfully at the end of your program's output.

### Message Format

The ping messages in this lab are formatted in a simple way. Each message contains a sequence of characters terminated by a carriage return (CR) character (`\r`) and a line feed (LF) character (`\n`). The message contains the following string:

*PING sequence\_number time CRLF*

where *sequence\_number* starts at 0 and progresses to 9 for each successive ping message sent by the client, *time* is the time when the client sent the message, and CRLF represent the carriage return and line feed characters that terminate the line.

*Hint: Cut and paste PingServer, rename the code PingClient, and then modify the code, if you are using Java.*

**This is my code (java):**

```
import java.io.*;

import java.net.*;

import java.util.*;

public class PingClient {

    private static final int TIMEOUT=1000;    // Timeout is 1 second.

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

        if (args.length != 2) {

            System.out.println("Required arguments: Server's address and port");

            return;

        }

        InetAddress address = InetAddress.getByName(args[0]);

        int port = Integer.parseInt(args[1]);


        long RTTarr[] = new long[10];        //store RTT array to compare max,min,ave.

        int num_of_RTTarr=0;

        for (int i = 0; i < 10; i++){          //10 packets

            long startTime = System.currentTimeMillis();

            String msg = "PING " + i + " "+ startTime+" \r\n";        //message

            byte[] data = msg.getBytes();

            DatagramSocket socket = new DatagramSocket();

            // set timeout(1 second)to make client infer whether it is timeout.

            socket.setSoTimeout(TIMEOUT);

            // create packets including message, ip address and port.

            DatagramPacket sendPacket = new DatagramPacket(data, data.length, address,

port);


            byte[] data2 = new byte[1024];
```



```
DatagramPacket receivePacket = // null packets to receive.

    new DatagramPacket(data2, data2.length);

boolean receivedResponse = false;

socket.send(sendPacket);

try{

    socket.receive(receivePacket);

    receivedResponse = true; //gets response

} catch (InterruptedException e){ //if not gets, print timeout.

    System.out.println("ping to 127.0.0.1, seq = " + i + ", Timed out" );

}

if (receivedResponse) { //if gets, execute this function.

    long finishTime = System.currentTimeMillis(); //calculate the RTT.

    long RTT = finishTime - startTime;

    RTTarr[num_of_RTTarr] = RTT ; //store the RTT to compare.

    num_of_RTTarr++;

    System.out.println("ping to 127.0.0.1,seq = " + i + " rtt = " + RTT + "ms");

//print result.

}

}

long min,max,sum,ave; //RTT of max,min,ave.

min=max=RTTarr[0];

sum=0;

for(int k = 0; k < num_of_RTTarr; k++)

{

    if(RTTarr[k]>max)

        max=RTTarr[k];

    if(RTTarr[k]<min)

        min=RTTarr[k];

    sum = sum + RTTarr[k];

}
```

```
ave = sum/(num_of_RTTarr);

    System.out.println("min    RTT:"+min+"ms,    max    RTT:"+max+"ms,    Ave
RTT:"+ave+"ms");

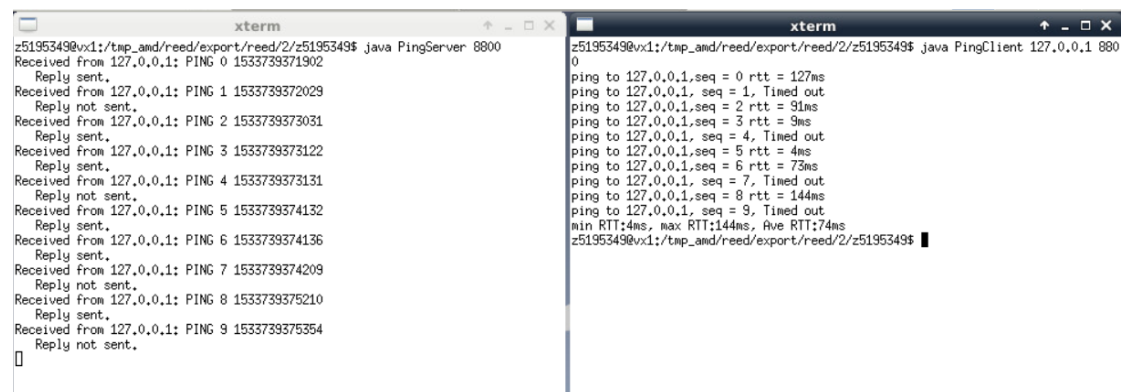
    // close

    // socket.close();

}

}
```

**This is the result:**



```
z5195349@vx1:/tmp_and/reed/export/reed/2/z5195349$ java PingServer 8800
Received from 127.0.0.1: PING 0 1533739371902
  Reply sent.
Received from 127.0.0.1: PING 1 1533739372029
  Reply not sent.
Received from 127.0.0.1: PING 2 1533739373031
  Reply sent.
Received from 127.0.0.1: PING 3 1533739373122
  Reply sent.
Received from 127.0.0.1: PING 4 1533739373131
  Reply not sent.
Received from 127.0.0.1: PING 5 1533739374132
  Reply sent.
Received from 127.0.0.1: PING 6 1533739374136
  Reply sent.
Received from 127.0.0.1: PING 7 1533739374209
  Reply not sent.
Received from 127.0.0.1: PING 8 1533739375210
  Reply sent.
Received from 127.0.0.1: PING 9 1533739375354
  Reply not sent.
[]

z5195349@vx1:/tmp_and/reed/export/reed/2/z5195349$ java PingClient 127.0.0.1 880
0
ping to 127.0.0.1,seq = 0 rtt = 127ms
ping to 127.0.0.1, seq = 1, Timed out
ping to 127.0.0.1,seq = 2 rtt = 91ms
ping to 127.0.0.1,seq = 3 rtt = 9ms
ping to 127.0.0.1, seq = 4, Timed out
ping to 127.0.0.1,seq = 5 rtt = 4ms
ping to 127.0.0.1,seq = 6 rtt = 73ms
ping to 127.0.0.1, seq = 7, Timed out
ping to 127.0.0.1,seq = 8 rtt = 144ms
ping to 127.0.0.1, seq = 9, Timed out
min RTT:4ms, max RTT:144ms, Ave RTT:74ms
z5195349@vx1:/tmp_and/reed/export/reed/2/z5195349$
```