

Lab 2

Objectives:

- Gain insights into operation of HTTP
- Get familiar with the basic socket programming
- Preparation for programming assignments

Prerequisites and Links:

- Week 2 & 3 Lectures.
- Relevant Parts of Chapter 2 of the textbook (Socket Programming) and Socket Programming discussion in week 3.
- Basic understanding of Linux. There are several resources online.
- Several socket programming resources for all 3 programming languages are available at the bottom of the [Sample Client Server Programs & Networking Resources](#) page
- [PingServer.java](#)
- [http-ethereal-trace-1](#)
- [http-ethereal-trace-2](#)

Marks: 4 marks

Questions to be marked: 1) Exercise 1.question3, 2) Exercise 5.

- Each lab comprises of a number of exercises. Not all the exercises for each lab are marked. Only those marked with (*) and written in bold will be marked.
- We expect the students to go through as much of the lab exercises as they can at home and come to the lab ready.
- Please attend your allocated lab and show/explain the answers of the marked exercises to your tutor.
- If lab exercise involves diagrams or plots, you require to show them to the tutor as well.
- Please make sure you **sign the marking form** once the tutor marked your lab. Signing this form implies that you agreed on the mark you received.
- There are 7 labs during this course. For each student, the 5 best performing labs will contribute to your final lab mark.

Exercise 1: Using Telnet to interact with a Web Server

Follow the steps described below. You will notice certain questions as you attempt the exercise. Write down the answers and discuss them with your lab partner to enable him/her to assess your performance. The solutions will be put up on the webpage at the end of the week. If you have any questions or are experiencing difficulty with executing the lab please consult your tutor.

Step 1: Open an *xterm* window. Enlarge the size of your *xterm* window so that it is reasonably large and covers almost the entire screen.

Step 2: Telnet to the vision.ucla.edu web server by typing:

```
$ telnet vision.ucla.edu 80
```

Note that the port number for all web servers is "80".

Step 3: Retrieve the main webpage by typing:

```
GET / HTTP/1.1  
host: www.vision.ucla.edu
```

Important Note: You will have to press the carriage return twice after typing the last line.

Question 1: What is the content type of the response? What is the size of the response? When was the webpage last modified? Do you see an "Accept-Range" header field. What may this be used for?

Step 4: Now execute the HEAD method. When a server receives a request with the HEAD method, it responds with only the message lines of the previous response (i.e. the response to the GET method minus the actual requested object).

```
HEAD / HTTP/1.1  
host: www.vision.ucla.edu
```

Question 2: What is the content type of the response? What is the size of the response?

(*) Question 3: Using telnet, find a way to get the people.html webpage from vision.ucla.edu (0.5 marks)

Question 4: Why is there the need to include the host in the GET (and HEAD) HTTP 1.1 request messages?

Exercise 2: Understanding Internet Cookies

Question 1. Repeat steps 1-3 in the previous experiment for www.google.com.au. Does the site set a cookie in your browser? How can you tell by purely examining the HTTP response message received using telnet? How about www.vision.ucla.edu? Do you think this site will set a cookie in your browser?

Question 2. Open a web browser (Firefox/IceWeasel/Mozilla preferred). Go to the browser preferences and remove all existing cookies. Open the google webpage and then view the cookies. How many cookies are stored on your machine? Which sites installed the cookies?

Question 3. Repeat the above steps for the vision.ucla.edu website. How many cookies are stored on your machine? Which sites installed the cookies? Is the answer inconsistent with the answer for Question 1? Explain why.

Exercise 3: Using Wireshark to understand basic HTTP request/response messages

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)

NOTE: IT IS NOT POSSIBLE TO RUN WIRESHARK VIA SSH. IT IS A RESOURCE INTENSIVE PROGRAM AND IT WOULD SLOW DOWN THE CSE LOGIN SERVERS. IF YOU WANT TO WORK REMOTELY, THEN YOU CAN DOWNLOAD AND INSTALL **WIRESHARK** ON YOUR PERSONAL MACHINE. WIRESHARK IS AVAILABLE ON ALL LAB MACHINES.

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?

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?

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

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

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

Note: Students are strongly encouraged to use Wireshark to capture real network traffic on their own machines. Check <https://www.wireshark.org/download.html> for details. Once you have Wireshark installed, do the following. Clear the cache of your web browser (Firefox->Tools->Clear Recent History). Launch the Wireshark tool by typing Wireshark in the command line. Start Wireshark capture by clicking on: capture -> interfaces -> click Start on the interface eth0. Run the Web browser and enter an url for a website (e.g. www.bbc.co.uk). Stop capturing packets when the web page is fully loaded. Examine the captured trace and answer the questions as above.

Exercise 4: Using Wireshark to understand the HTTP CONDITIONAL GET/response interaction

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?

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

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:" line in the HTTP GET? If so, what information is contained in the "IF-MODIFIED-SINCE:" header?

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.

(*) Exercise 5: Implementing a ping server (3.5 marks)

Note: This exercise is marked thus you need to have the ping server implemented to get your mark.

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` 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
```

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 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 time out requests in the output. Only replace the "rtt =120ms" in the above example with "time out". If you also do not show it, that should be fine.

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 represents 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.

Marking

Correct Display of ping outputs with a RTT value that makes sense (greater than 100): **2 marks**

Packet drop that conforms to the Loss Rate: **1.5 marks**