Lab Title: *General introduction to Wireshark and Networking*
Name: *M. Hasnain Naeem (212728)*
Class: *BSCS-7B*

Objective of this lab:

The basic purpose of this lab is to introduce you to Wireshark, a popular protocol analyzer. By the end of this lab you will be familiar to its environment and will know how to capture and interactively browse the traffic running on a computer network using it.

Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *You are supposed to provide the answers to the questions listed end of this document and upload this completed document to your course's LMS site.*
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.*

Background:

A protocol analyzer is a tool that can be used to inspect what exactly is happening on a network with respect to traffic flow. For example, if your TCP/IP sessions are "hanging", a protocol analyzer can show which system sent the last packet, and which system failed to respond. If you are experiencing slow screen updates, a protocol analyzer can display delta time stamps and show which system is waiting for packets, and which system is slow to respond.

A protocol analyzer can show runaway traffic (broadcast or multicast storms) and its origin, system errors and retries, and whether a station is sending, trying to send, or only seeming to communicate. You will get information that is otherwise unavailable, which results in more efficient troubleshooting and better LAN health.

1. Introduction to Networking:

A **computer network**, often simply referred to as a network, is a collection of hardware components and computers interconnected by communication channels that allow sharing of resources and information. In the world of computers, networking is the practice of linking two or more computing devices together for the purpose of sharing data. In networking, the communication language used by computer devices is called the protocol. Yet another way to classify computer networks is by the set of protocols they support. Networks often implement multiple protocols to support specific applications.

2. What is a protocol analyzer?

Protocol analyzers capture conversations between two or more systems or devices. A protocol analyzer not only captures the traffic, it also decodes (interprets) the traffic. Decoding allows you to view the conversation in English, as opposed to binary language. A sophisticated protocol analyzer will also provide statistics and trend information on the captured traffic. Protocol analyzers provide information about the traffic flow on your local area network (LAN), from which you can view device-specific information.

3. Introduction to Wireshark

Wireshark is a free and open-source packet analyzer, used for network troubleshooting, analysis, software and communications protocol development, and education.

The basic tool for observing the messages exchanged between executing protocol entities is called a **packet sniffer**. As the name suggests, a packet sniffer captures ("sniffs") messages being sent/received from/by your computer; it will also typically store and/or display the contents of the various protocol fields in these captured messages. A packet sniffer itself is passive. It observes messages being sent and received by applications and protocols running on your computer, but never sends packets itself. Similarly, received packets are never explicitly addressed to the packet sniffer. Instead, a packet sniffer receives a copy of packets that are sent/ received from/by application and protocols executing on your machine.

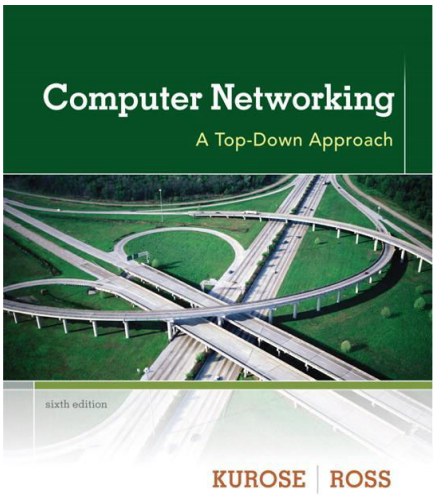
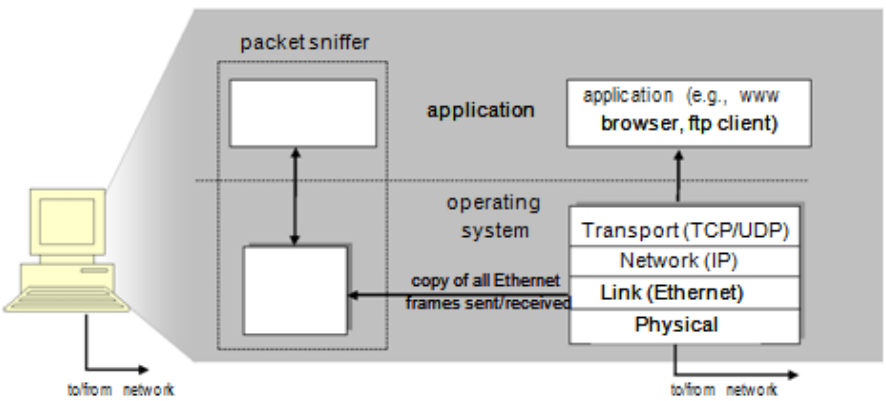


Figure 1 shows the structure of a packet sniffer. At the right of Figure 1 are the protocols (in this case, Internet protocols) and applications (such as a web browser or ftp client) that normally run on your computer. The packet sniffer, shown within the dashed rectangle in Figure 1 is an addition to the usual software in your computer, and consists of two parts. The **packet capture library** receives a copy of every link-layer frame that is sent from or received by your computer. Recall from the discussion from Section 1.5 in the textbook (Figure 1.24) that messages exchanged by higher layer protocols such as HTTP, FTP, TCP, UDP, DNS, or IP all are eventually encapsulated in link-layer frames that are transmitted over physical media such as an Ethernet cable. In Figure 1, the assumed physical media is an Ethernet, and so all upper-layer protocols are eventually encapsulated within an Ethernet frame. Capturing all link-layer frames thus gives you all messages sent/received from/by all protocols and applications executing in your computer.

Figure 1: *Packet sniffer structure*



The second component of a packet sniffer is the **packet analyzer**, which displays the contents of all fields within a protocol message. In order to do so, the packet analyzer must "understand" the structure of all messages exchanged by protocols. For example, suppose we are interested in displaying the various fields in messages exchanged by the HTTP protocol in Figure 1. The packet analyzer understands the format of Ethernet frames, and so can identify the IP datagram within an Ethernet frame. It also understands the IP datagram format, so that it can extract the TCP segment within the IP datagram. Finally, it understands the TCP segment structure, so it can extract the HTTP message contained in the TCP segment. Finally, it understands the HTTP protocol and so, for example, knows that the first bytes of an HTTP message will contain the string "GET," "POST," or "HEAD," as shown in Figure 2.8 in the textbook.

We will be using the Wireshark packet sniffer [http://www.wireshark.org/] for these labs, allowing us to display the contents of messages being sent/ received from/by protocols at different levels of the protocol stack. (Technically speaking, Wireshark is a packet analyzer that uses a packet capture library in your computer). Wireshark is a free network protocol analyzer that runs on Windows, Linux/Unix, and Mac computers. It's an ideal packet analyzer for our labs – it is stable, has a large user base and well-documented support that includes a comprehensive user-guide (http://www.wireshark.org/docs/wsug_html_chunked/), man pages (http://www.wireshark.org/docs/man-pages/), and a FAQ (http://www.wireshark.org/faq.html), rich functionality that includes the capability to analyze more than 500 protocols, and a well-designed user interface. It operates in computers using Ethernet, Token-Ring, FDDI, serial (PPP and SLIP), 802.11 wireless LANs and ATM connections (if the OS on which it's running allows Wireshark to do so).

3.1 Getting Wireshark

In order to run Wireshark, you will need to have access to a computer that supports both Wireshark and the *libpcap* or *WinPCap* packet capture library. The *libpcap* software will be installed for you alongside Wireshark automatically. See http://www.wireshark.org/download.html for a list of supported operating systems and download sites

Download and install the Wireshark software:

- Go to http://www.wireshark.org/download.html and download and install the Wireshark binary for your computer. Wireshark can be installed on both Windows and Linux. See the documentation page of Wireshark for more details.

- Download the Wireshark user guide.

The Wireshark FAQ has a number of helpful hints and interesting tidbits of information, particularly if you have trouble installing or running Wireshark.

3.2 Running Wireshark

On *Windows*, you should be able be able to find the link by clicking on the Start option of the Windows taskbar and thereby finding the wireshark program in All Programs.
On Linux machines, wireshark can be run by typing "wireshark" at the command prompt (in case there is a problem with your path, type "*which wireshark"* that would show path /usr/bin/wireshark where wireshark is typically installed). When you run the Wireshark program, the Wireshark graphical user interface shown in Figure 2 will be displayed. Initially, no data will be displayed in the

various windows.

The Wireshark interface has five major components:

- The **command menus** are standard pull down menus located at the top of the window. Of interest to us is the File and Capture menus. The File menu allows you to save captured packet data or open a file containing previously captured packet data, and exits the Wireshark application. The Capture menu allows you to begin packet capture.

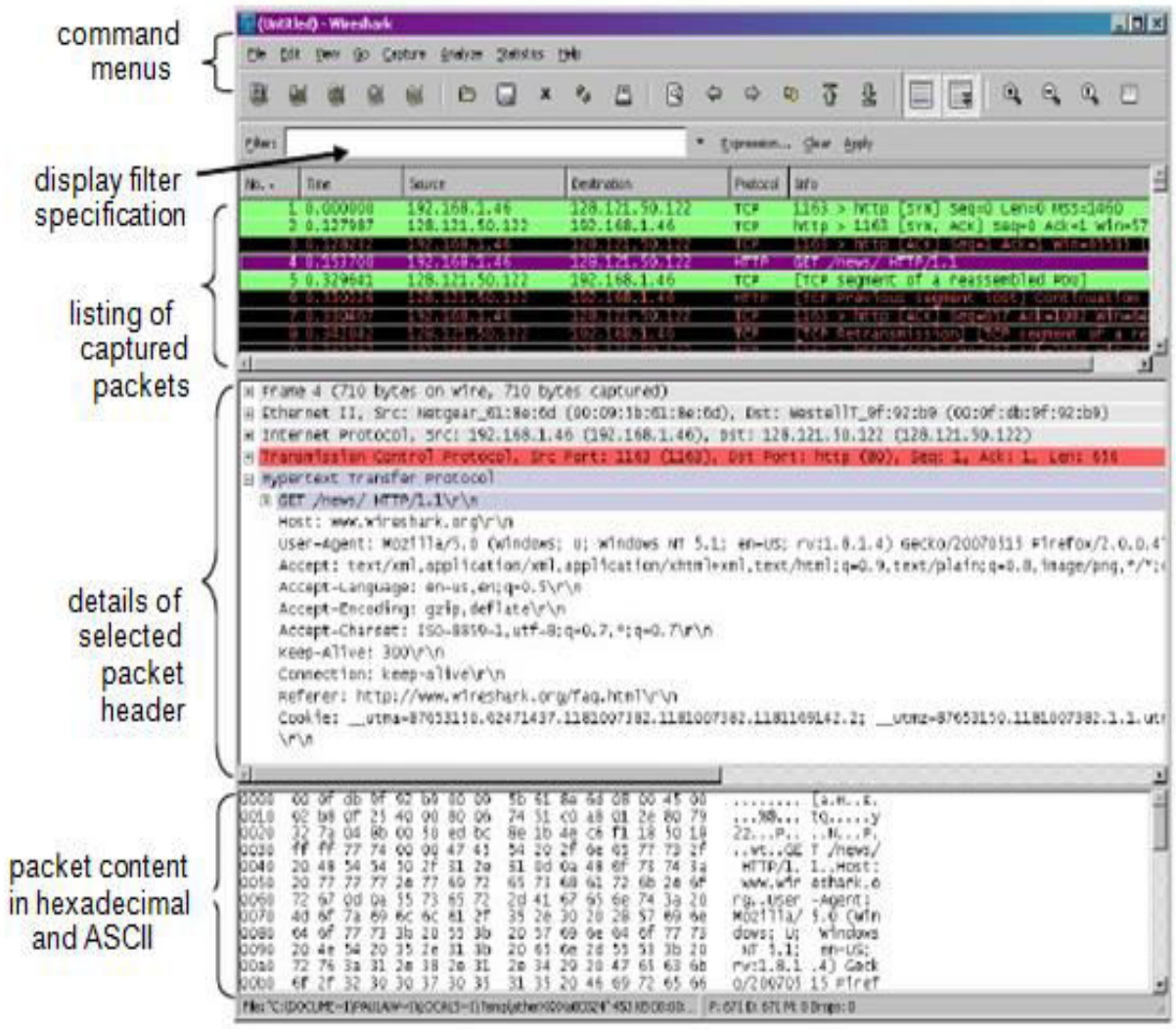
- **The packet-listing window** displays a one-line summary for each packet captured, including the packet number (assigned by Wireshark; this is not a packet number contained in any protocol's header), the time at which the packet was captured, the packet's source and destination addresses, the protocol type, and protocol-specific information contained in the packet. The packet listing can be sorted according to any of these categories by clicking on a column name. The protocol type field lists the highest-level protocol that sent or received this packet, i.e., the protocol that is the source or ultimate sink for this packet.

- The **packet-header details window** provides details about the packet selected (highlighted) in the packet-listing window. (To select a packet in the packet-listing window, place the cursor over the packet's one-line summary in the packet-listing window and click with the left mouse button.). These details include information about the Ethernet frame and IP datagram that contains this packet. The amount of Ethernet and IP-layer detail displayed can be expanded or minimized by clicking on the right-pointing or down-pointing arrowhead to the left of the Ethernet frame or IP datagram line in the packet details window. If the packet has been carried over TCP or UDP, TCP or UDP details will also be displayed, which can similarly be expanded or minimized. Finally, details about the highest-level protocol that sent or received this packet are also provided.

- The **packet-contents window** displays the entire contents of the captured frame, in both ASCII and hexadecimal format.

- Towards the top of the Wireshark graphical user interface, is the **packet display filter field**, into which a protocol name or other information can be entered in order to filter the information displayed in the packet-listing window (and hence the packet-header and packet-contents windows). In the example below, we'll use the packet-display filter field to have Wireshark hide (not display) packets except those that correspond to HTTP messages.

Figure 2: Wireshark Graphical User Interface

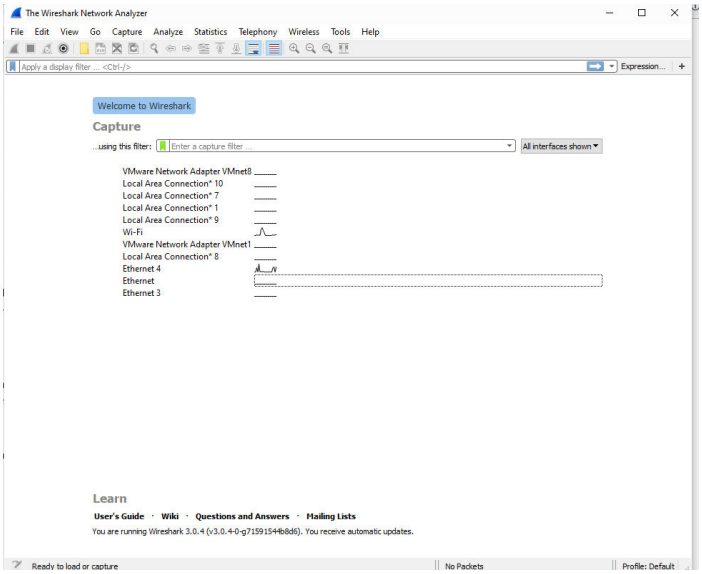


Steps for performing this lab:

The best way to learn about any new piece of software is to try it out! We'll assume that your computer is connected to the Internet via a wired Ethernet interface. Do the following:

1. ***Start up your favorite web browser***, which will display your selected homepage.

2. ***Start up the Wireshark software.*** You will initially see a window similar to that shown in Figure 2, except that no packet data will be displayed in the packet-listing, packet-header, or packet-contents window, since Wireshark has not yet begun capturing packets.



3. ***To begin packet capture,*** select the Capture pull down menu and select *Options.* This will cause the "Wireshark: Capture Options" window to be displayed, as shown in Figure 3.

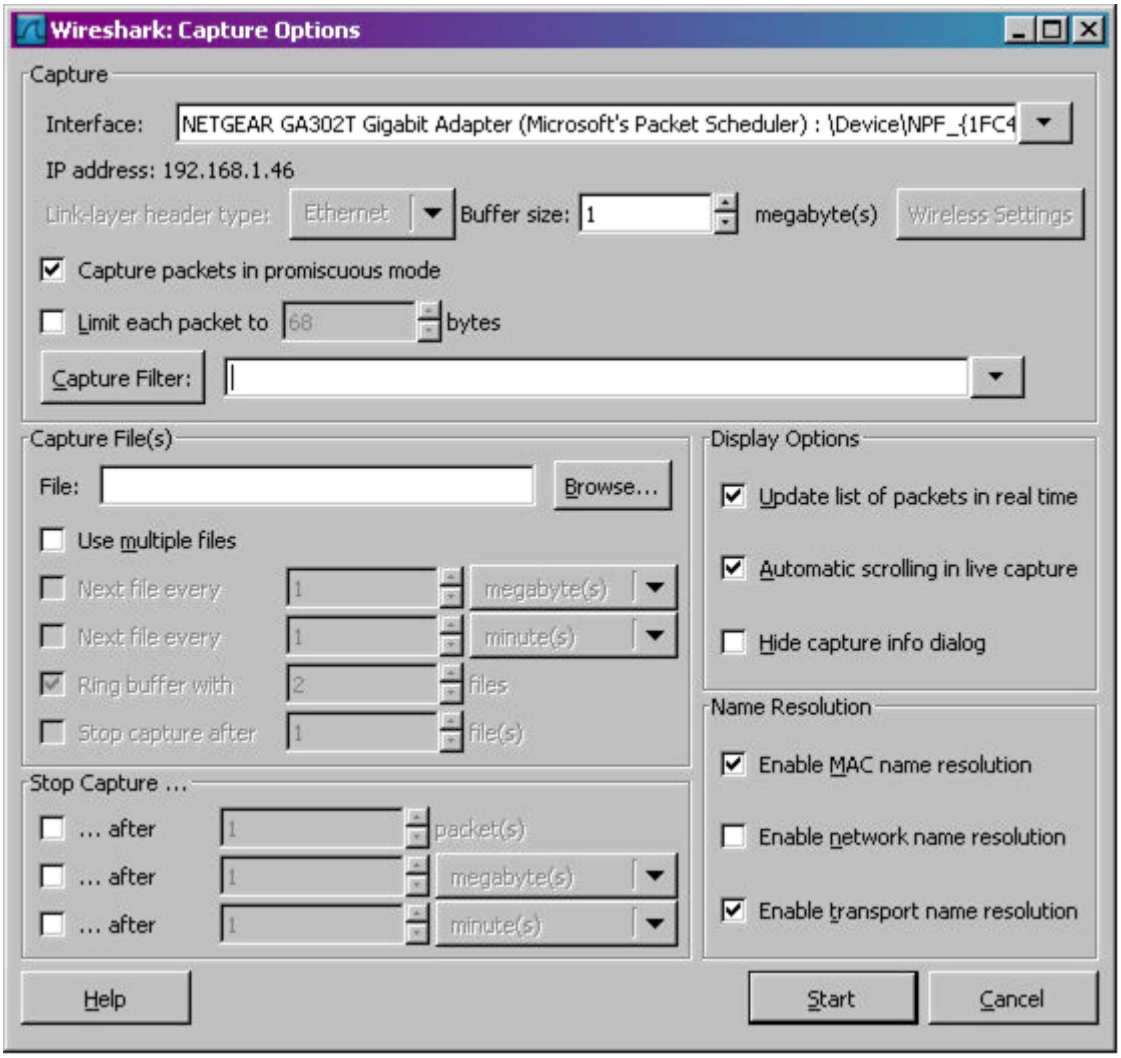
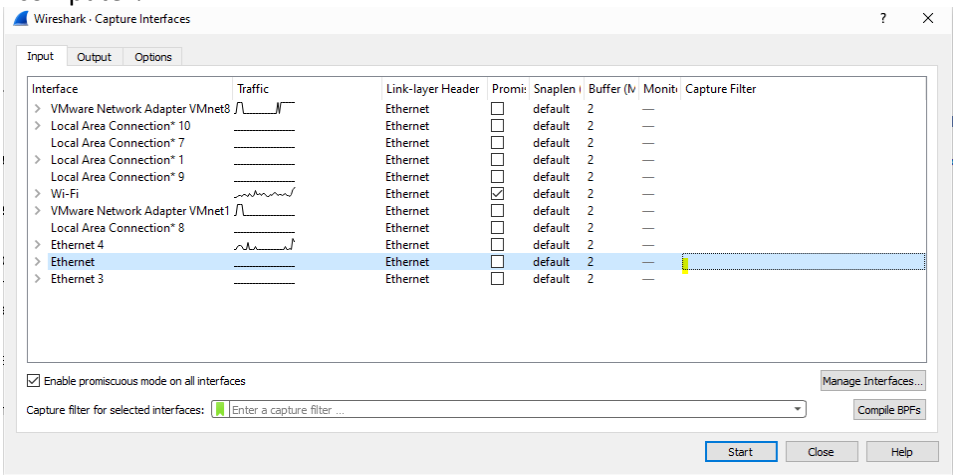


Figure 3: *Wireshark Capture Options Window*

4. ***Selecting the network interface on which packets would be captured:*** You can use most of the default values in this window, but uncheck "Hide capture info dialog" under Display Options. The network interfaces (i.e., the physical connections) that your computer has to the network will be shown in the Interface pull down menu at the top of the Capture Options window. In case your computer has more than one active network interface (e.g., if you have both a wireless and a wired Ethernet connection), you will need to select an interface that is being used to send and receive packets (most likely the wired interface). After selecting the network interface (or using the default interface chosen by Wireshark), click Start. Packet capture will now begin - Wireshark is now capturing all packets being sent/received from/ by your computer!



5. Once you begin packet capture, a packet capture summary window will appear, as shown in Figure 4. This window summarizes the number of packets of various types that are being captured, and (importantly!) contains the *Stop* button that will allow you to stop packet capture. Don't stop packet capture yet.

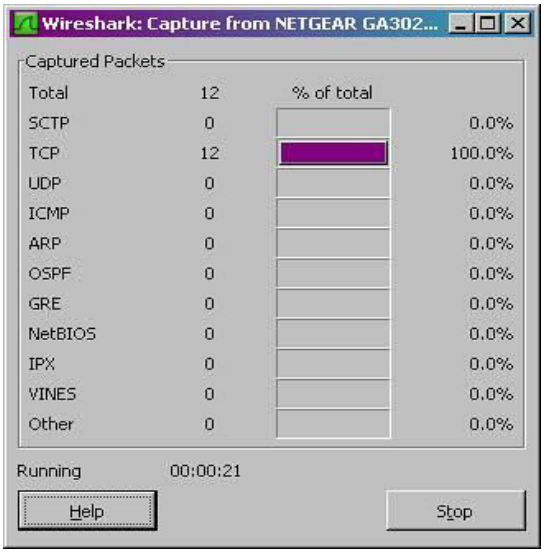


Figure 4: *Wireshark Packet Capture Window*

6. Capturing an HTTP interaction on Wireshark: While Wireshark is running, enter the URL: http://seecs.nust.edu.pk/ and have that page displayed in your browser. In order to display this page, your browser will contact the HTTP server at http://seecs.edu.pk, and exchange HTTP

messages with the server in order to download this page, as discussed in section 2.2 of the text. Wireshark will capture the Ethernet frames containing these HTTP messages.

7. ***Stopping the capture and inspecting captured packets:*** After your browser has displayed the page, stop Wireshark packet capture by selecting stop in the Wireshark capture window. This will cause the Wireshark capture window to disappear and the main Wireshark window to display all packets captured since you began packet capture. The main Wireshark window should now look similar to Figure 2. You now have live packet data that contains all protocol messages exchanged between your computer and other network entities! The HTTP message exchanges with the seecs.nust.edu.pk web server should appear somewhere in the listing of packets captured. But there will be many other types of packets displayed as well (see, e.g., the many different protocol types shown in the *Protocol* column in Figure 2). Even though the only action you took was to download a web page, there were evidently many other protocols running on your computer that are unseen by the user. We'll learn much more about these protocols as we progress through the text! For now, you should just be aware that there is often much more going on than "meets the eye".

8. ***Filtering:*** Type in "http" (without the quotes, and in lower case – all protocol names are in lower case in Wireshark) into the display filter specification window at the top of the main Wireshark window. Then select *Apply* (to the right of where you entered "http"). This will cause only HTTP message to be displayed in the packet-listing window.

9. ***Details of a packet:*** Select the first http message shown in the packet-listing window. This should be the HTTP GET message that was sent from your computer to the seecs.nust.edu.pk HTTP server. When you select the HTTP GET message, **the Ethernet frame, IP datagram, TCP segment, and HTTP message header** information will be displayed in the packet-header window. By clicking on right-pointing and down-pointing arrows heads to the left side of the packet details window, *minimize* the amount of Frame, Ethernet, Internet Protocol, and Transmission Control Protocol information displayed. *Maximize* the amount information displayed about the HTTP protocol. Your Wireshark display should now look roughly as shown in Figure 5. (Note, in particular, the minimized amount of protocol information for all protocols except HTTP, and the maximized amount of protocol information for HTTP in the packet-header window).

10. ***Statistics of packet captured:*** Click on the 'Statistics' option on the upper toolbar of Wireshark to explore the various ways in which statistics may be obtained about network traffic. Explore specifically the 'Conversation' options in 'Statistics' option on the upper toolbar of Wireshark.

11. ***Obtaining credit for this lab:*** Answer the questions listed at the end of this lab. Please note that this is an individual activity and every student must upload the answer file (after duly filling in the answers) through the appropriate link at your LMS course site for the specific date of your lab (an upload link would be made available) to obtain credit. Please clarify with your instructor/ lab engineer if you have any queries.

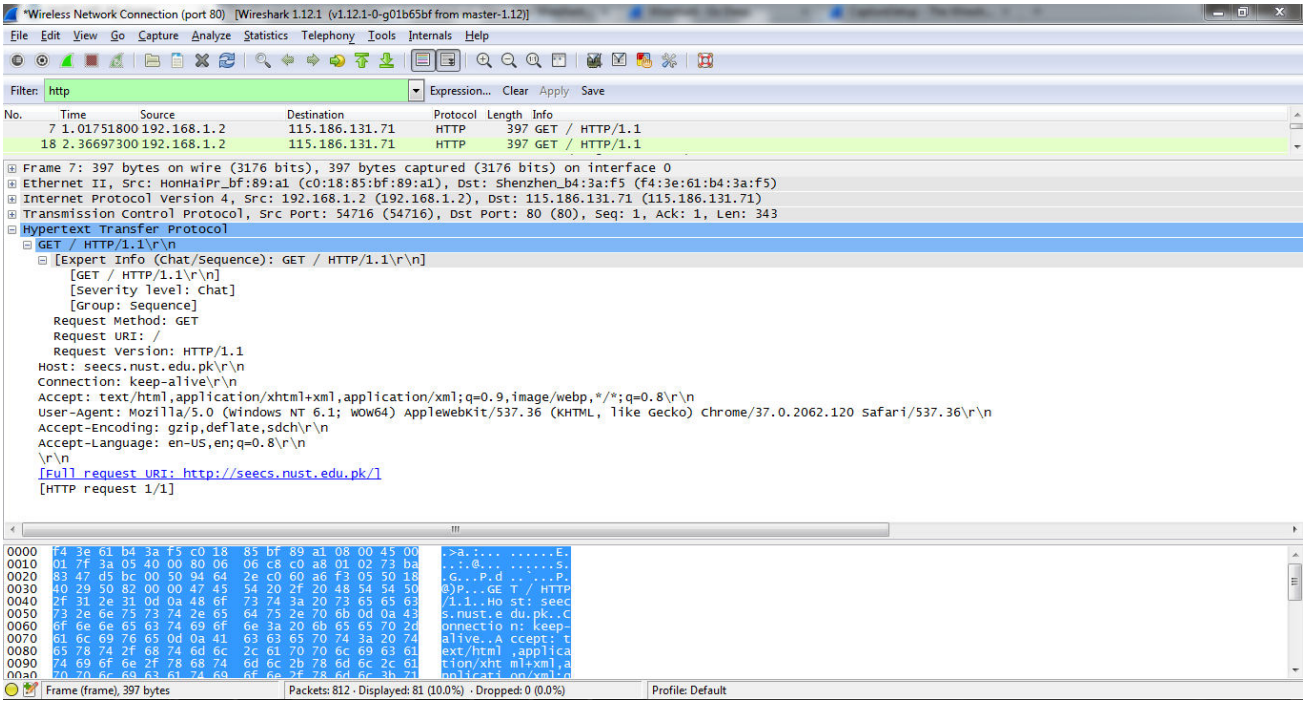


Figure 5: *Wireshark display after step 9*

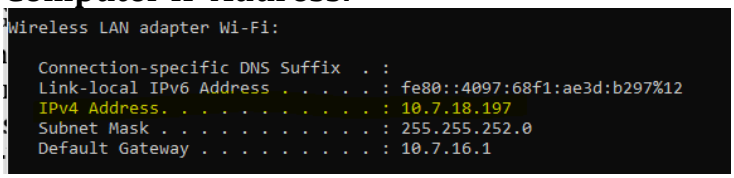
Objectives:

- Understanding different layers in a networking processs
- Getting familiar with wireshark to inspect packets sent during the networking process
- Inspecting packets using wireshark

Conclusion:

Networking process uses different layers and header data increases in each of the layer whilst data moves from the application layer to physical layer. We can inspect the details of that data using wireshark.

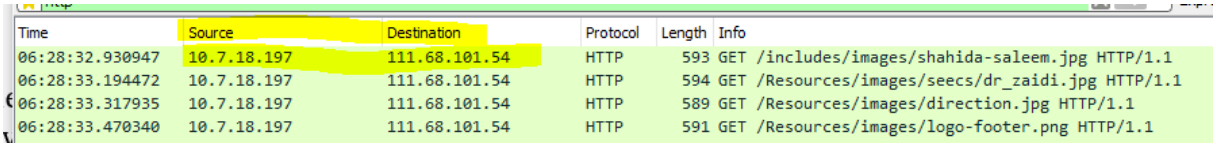
Computer IP Address:



Questions:

1. ***Finding IP address of your machine in Wireshark:*** What is the IP address of 'seecs.nust.edu.pk'? What is the IP address of your computer? How did you find it in Wireshark? Compare the IP address of your machine by using ipconfig command.

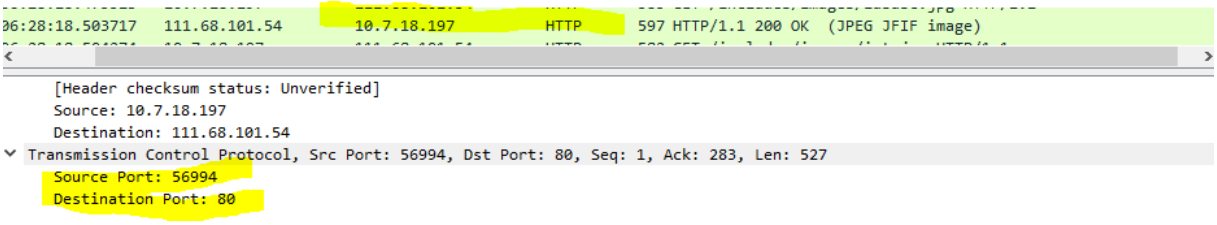
IP address of client (as shown above and in wireshark source field): 10.7.18.197
IP address of SEECS server (destination in wireshark destination field): 111.68.101.54



Time	Source	Destination	Protocol	Length	Info
06:28:32.930947	10.7.18.197	111.68.101.54	HTTP	593	GET /includes/images/shahida-saleem.jpg HTTP/1.1
06:28:33.194472	10.7.18.197	111.68.101.54	HTTP	594	GET /Resources/images/seecs/dr_zaidi.jpg HTTP/1.1
06:28:33.317935	10.7.18.197	111.68.101.54	HTTP	589	GET /Resources/images/direction.jpg HTTP/1.1
06:28:33.470340	10.7.18.197	111.68.101.54	HTTP	591	GET /Resources/images/logo-footer.png HTTP/1.1

2. What is the ***port number*** used by the HTTP server 'seecs.nust.edu.pk'. How did you note it in Wireshark?

Destination port: 80
It can be inspected in TCP segment details.



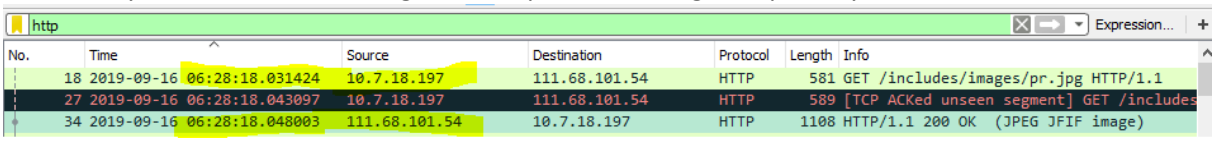
3. ***Delay between request and reply:*** How long did it take from when the HTTP GET message was sent until the HTTP OK reply was received? (By default, the value of the Time column in

the packet-listing window is the amount of time, in seconds, since Wireshark tracing began. To display the Time field in time-of-day format, select the Wireshark *View* pull down menu, then select Time *Display Format*, then select *Time-of-day*.)

Get request time: 06:28:18.031424

OK response time: 06:28:18.048003

It took only 0.016579 seconds to get the response of first get request by client to server.



Lab Title: Wireshark – HTTP (Hypertext Transfer Protocol)
***Submitted By*:** M. Hasnain Naeem (212728)
Section: BSCS-7B
Objective of this lab:

In this lab, we'll explore several aspects of the HTTP protocol: the basic GET/response interaction, and HTTP message formats.

Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *You are supposed to provide the answers to the in-line questions in this document and upload the completed document to your course's LMS site.*
- ***For all questions, you must not only answer the question, but also supply all necessary information regarding how you arrived at the answer (e.g., use screenshots/ accompanying text, etc.) Use red font color to distinguish your replies from the rest of the text.***
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.*

Background:

The world's web browsers, servers and related web applications all talk to each other through HTTP, the Hypertext Transfer Protocol. Before proceeding to the experiments, it is recommended that you read introductions to some general terms used in this lab, to avoid any confusion.

1. What is a web page?

A Web page (also called a document) consists of objects. An object is a simple file -- such as a HTML file, a JPEG image, a GIF image, a Java applet, an audio clip, etc. -- that is addressable by a single URL. Most Web pages consist of a base HTML file and several referenced objects. For example, if a Web page contains HTML text and five JPEG images, then the Web page has six objects: the base HTML file plus the five images. The base HTML file references the other objects in the page with the objects' URLs. Each URL has two components: the host name of the server that houses the object and the object's path name. For example, the URL www.someSchool.edu/someDepartment/picture.gif has www.someSchool.edu for a host name and /someDepartment/picture.gif for a path name.

2. What is a web browser?

A browser is a user agent for the Web; it displays to the user the requested Web page and provides numerous navigational and configuration features. Web browsers also implement the client side of HTTP. Thus, in the context of the Web, we will interchangeably use the words "browser" and "client". Popular Web browsers include Google Chrome, Netscape Communicator, Apple Safari and Microsoft Explorer.

3. What is a web server?

A Web server hosts Web objects, each addressable by a URL. Web servers also implement the server side of HTTP. Popular Web servers include Apache, Microsoft Internet Information Server, and the Netscape Enterprise Server.

4. Introduction to HTTP:

The Hypertext Transfer Protocol (HTTP), the Web's application-layer protocol, is at the heart of the Web. HTTP is implemented in two programs: a client program and server program. The client program and server programs, executing on different end systems, talk to each other by exchanging HTTP messages. HTTP defines the structure of these messages and how the client and server exchange the messages. HTTP defines how Web clients (i.e., browsers) request Web pages from servers (i.e., Web servers) and how servers transfer Web pages to clients. When a user requests a Web page (e.g., clicks on a hyperlink), the browser sends HTTP request messages for the objects in the page to the server. The server receives the requests and responds with HTTP response messages that contain the objects.

Steps for performing this lab:

For all the experiments we will use *Wireshark* packet analyzer.

Exercise 01: The Basic HTTP GET/response interaction

Aim of this exercise: We will now learn about what packets are exchanged during a HTTP conversation---we will learn about the HTTP GET message that is sent from the HTTP client to the HTTP server and the HTTP message that is sent as response to this message.

Follow the steps below to complete this exercise and to provide answers to the questions below

- Start up your web browser.
- Start up the Wireshark packet sniffer (but don't yet begin packet capture). Enter "http" (just the letters, not the quotation marks) in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window. (We're only interested in the HTTP protocol here, and don't want to see the clutter of all captured packets).
- Begin Wireshark packet capture.
- Enter the following to your browser http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html. Your browser should display the very simple, one-line HTML file.
- Stop Wireshark packet capture.

The example in Figure 1 shows in the packet-listing window that two HTTP messages were captured: the GET message (from your browser to the gaia.cs.umass.edu web server) and the response message from the server to your browser. The packet-contents window shows details of the selected message (in this case the HTTP GET message, which is highlighted in the packet- listing window). Recall that since the 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.

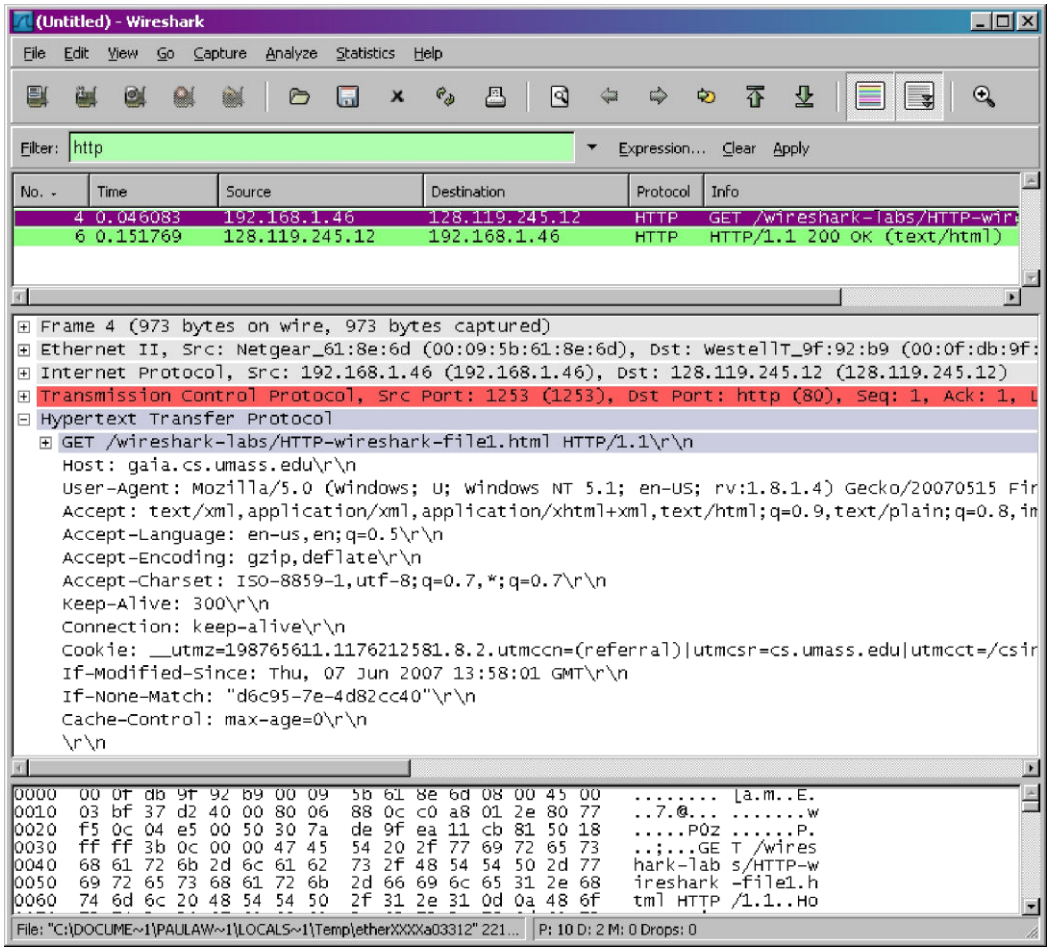


Figure 1: Wireshark display after http://gaia.cs.umass.edu/wireshark-labs/ HTTP-wireshark-file1.html has been retrieved by your browser

By looking at the information in the HTTP GET and response messages that you have captured, answer the following questions:

Objectives:

- Knowledge of protocols used in application layer
- Getting familiar with HTTP GET and response messages
- Inspecting HTTP packets using Wireshark
- Understanding the data/information/flags in HTTP packets

Conclusion:

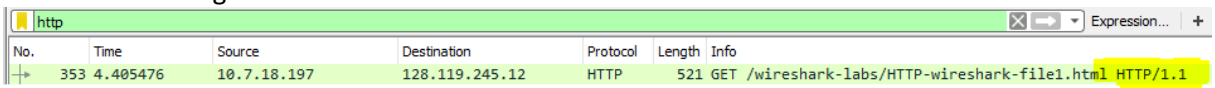
Browser uses HTTP protocol to send and receive HTTP packets to and from the server. We can inspect those packets using the wireshark software. Header files and objects sent from the server by the client and requests sent by the client to server can be analyzed through inspection of HTTP packets.

IP Address:

```
Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::4097:68f1:ae3d:b297%12
   IPv4 Address. . . . . . . . . . . : 10.7.18.197
   Subnet Mask . . . . . . . . . . . : 255.255.252.0
   Default Gateway . . . . . . . . . : 10.7.16.1
```

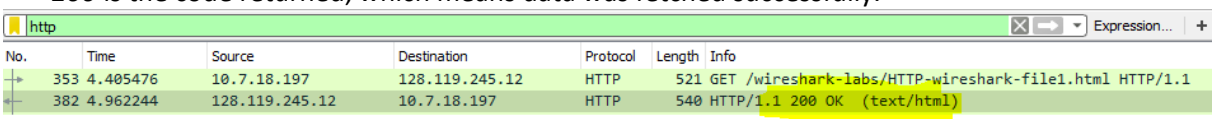
1.1 Which version of HTTP is the browser running 1.0 or 1.1? Which HTTP version is the server running?

Browser is running *HTTP 1.1*.



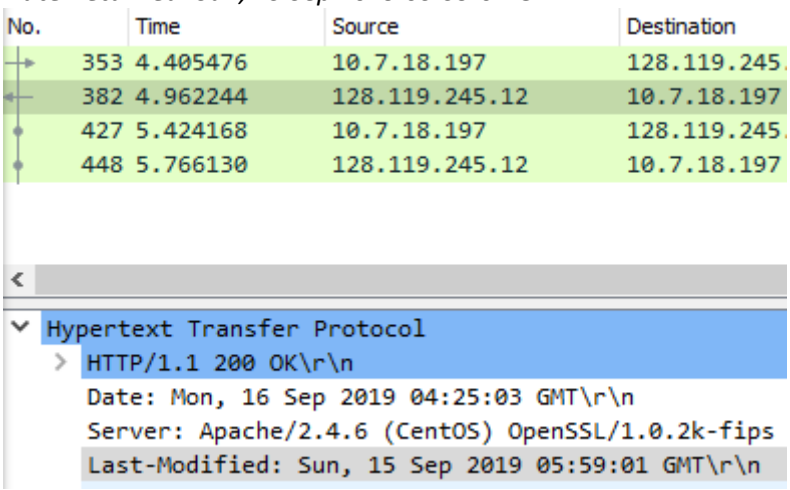
1.2 What is the status code returned from the server to your browser?

200 is the code returned, which means data was fetched successfully.



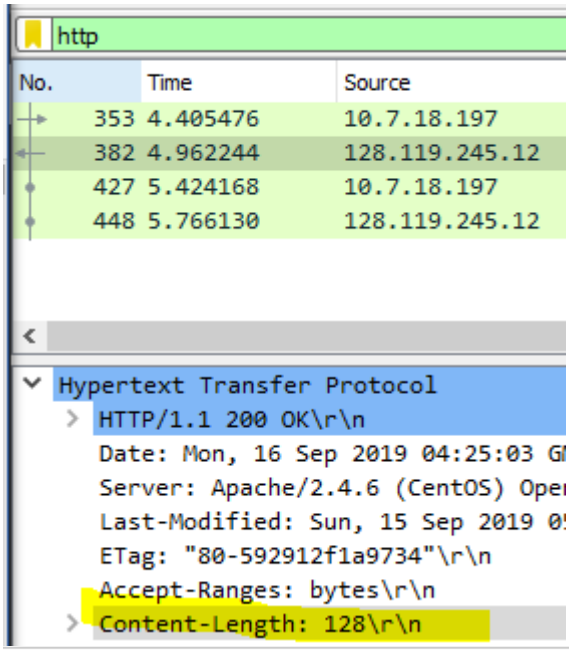
1.3 When was the HTML file that you are retrieving last modified at the server?

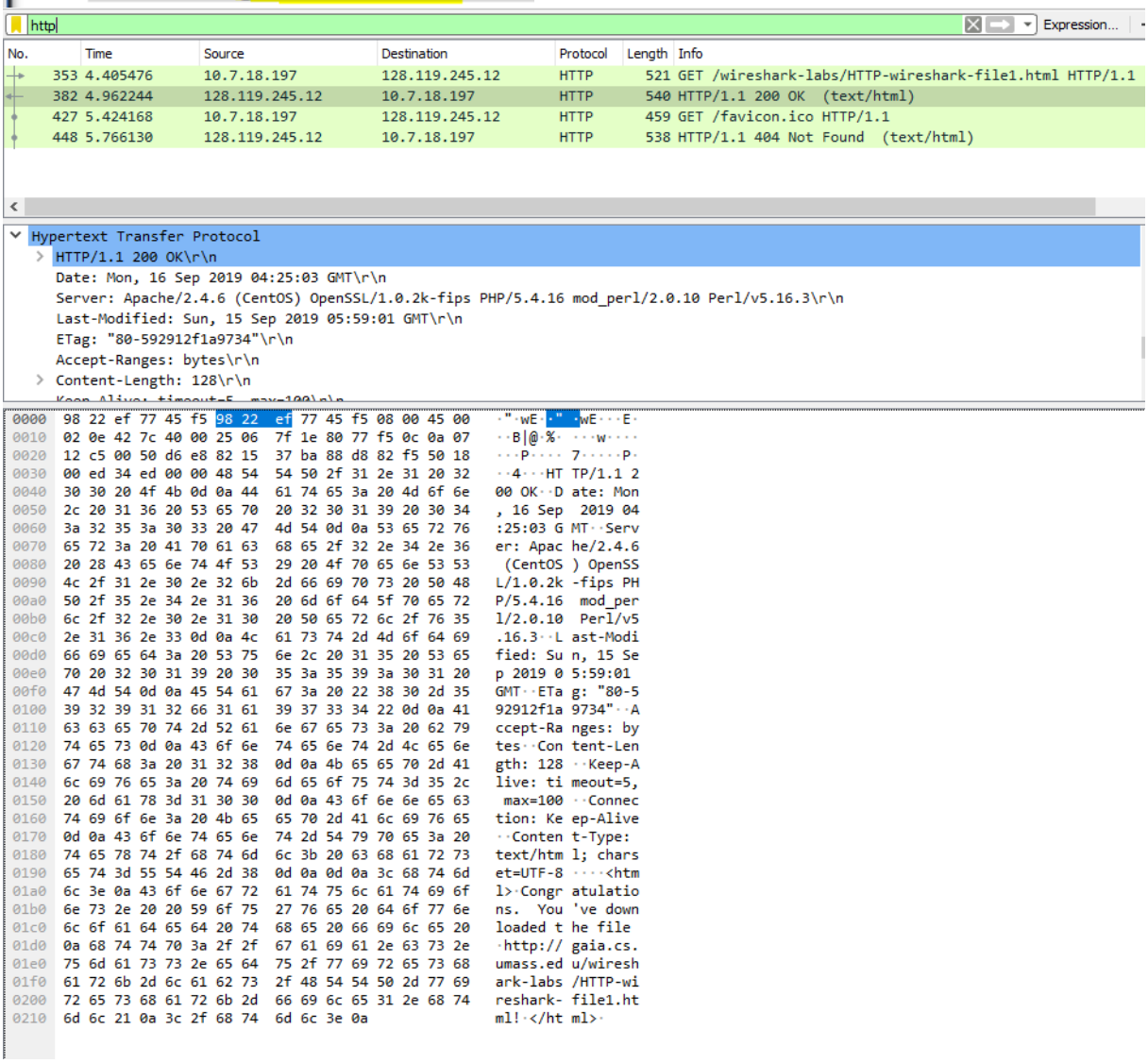
Date Returned: *Sun, 15 Sep 2019 05:59:01 GMT*



1.4 How many bytes of content are being returned to your browser?

Bytes Returned: *128*





Exercise 02: *The HTTP CONDITIONAL GET/response interaction*

Aim of this exercise: We will now learn about a variant of the HTTP GET request message that we've

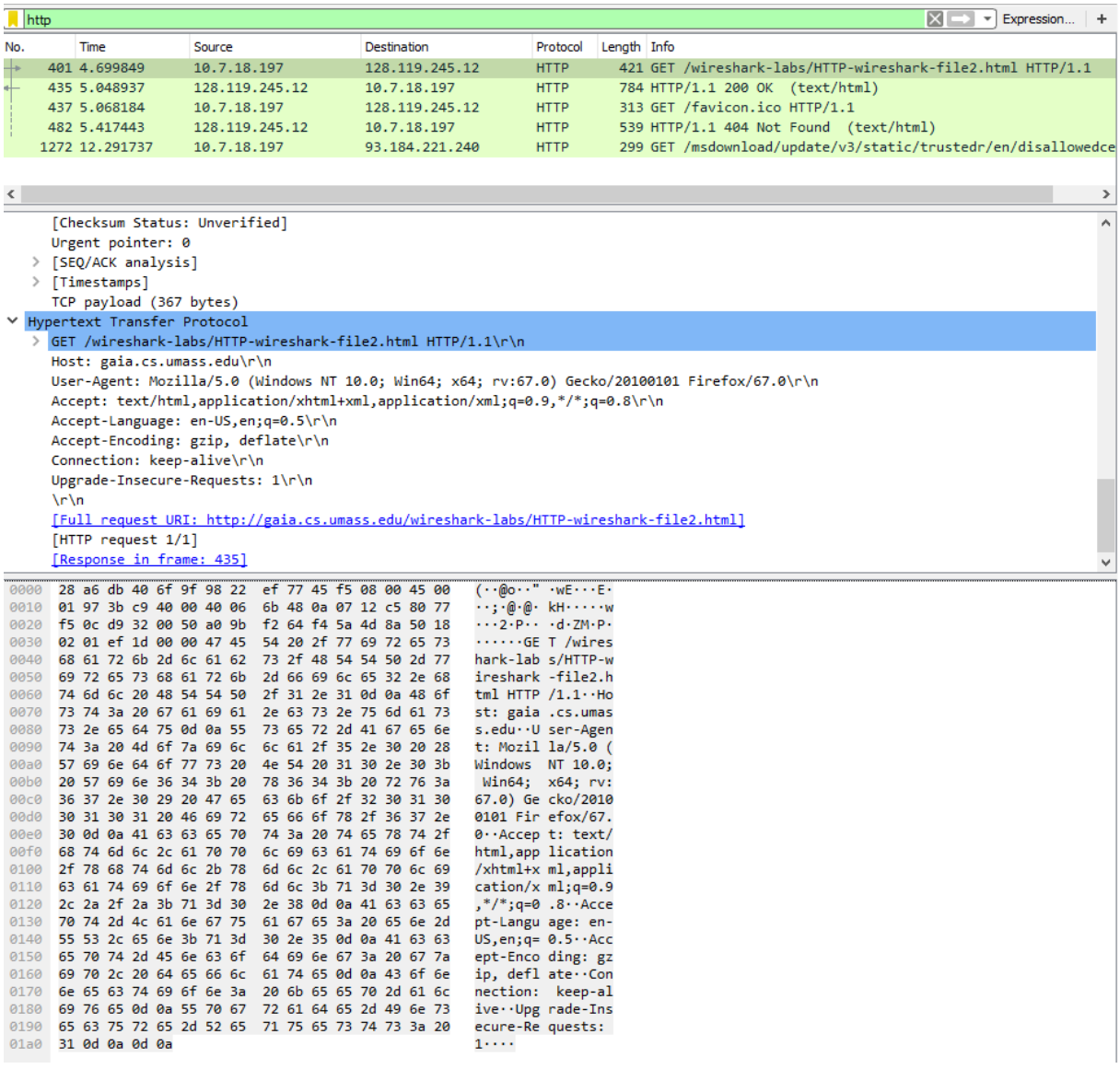
seen earlier. We will note how the HTTP CONDITIONAL GET request, and the reply to such a request differs from a simple HTTP GET request. Before performing the steps below, make sure your browser's cache is empty. (To do this under Firefox, select *Tools->Clear Recent History* and check the Cache box, or for Internet Explorer, select *Tools->Internet Options->Delete File;* these actions will remove cached files from your browser's cache.)

The following indicate the steps for this experiment:

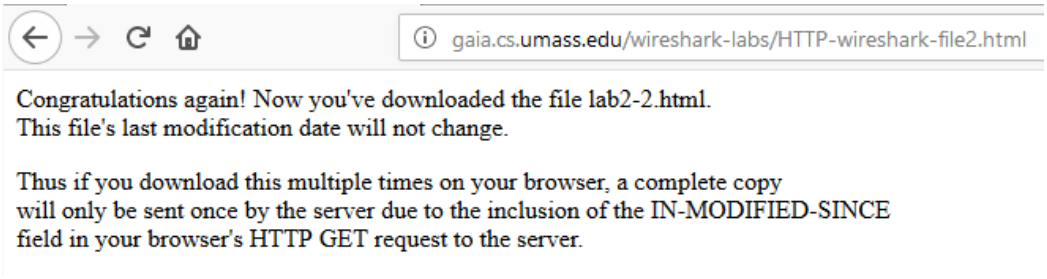
- Start up your web browser, and make sure your browser's cache is cleared, as discussed above.

- Start up the Wireshark packet sniffer

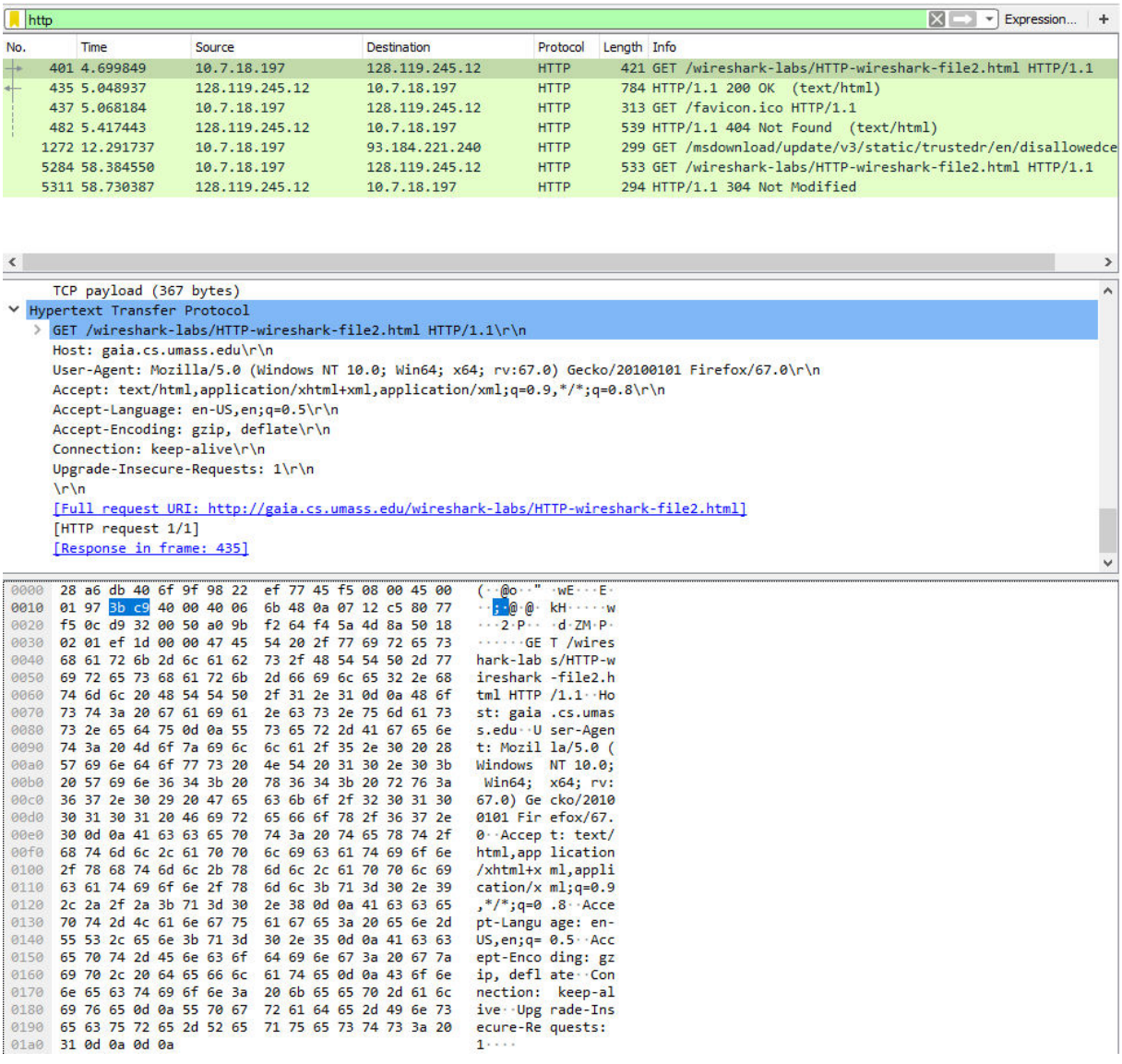
- Enter the following URL into your browser
  http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html



Your browser should display a very simple five-line HTML file.



- **Quickly enter the same URL into your browser again (or simply select the refresh button on your browser). Stop Wireshark packet capture, and enter "http" in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window.**



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

```
Hypertext Transfer Protocol
  GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1\r\n
  Host: gaia.cs.umass.edu\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:67.0) Gecko/20100101 Firefox/67.0\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
  Accept-Language: en-US,en;q=0.5\r\n
  Accept-Encoding: gzip, deflate\r\n
  Connection: keep-alive\r\n
  Upgrade-Insecure-Requests: 1\r\n
  \r\n
  [Full request URI: http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html]
  [HTTP request 1/1]
  [Response in frame: 435]
```

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

2.1 Inspect the contents of the first and 2nd HTTP GET requests from the browser to the server. Do you see "IF-MODIFIED-SINCE" and "IF-NONE-MATCH" lines in these HTTP GET message? Why?

- Yes.
- As this page was saved in the cache. Browser asked the server through HTTP Get request, if the page is not modified (through **If-Modified-Since** flag) since the given date, then do not send all the objects. Intead, send the header bits only and a response of 304.
- **If-None-Match** checks if the resource is there or not, if it is there then the condition of if-modified-since is checked.
- Value of if-none-match is also known as **ETag**, which is used to make sure the objects of the requested page are changed or not since the given date.

No.	Time	Source	Destination
401	4.699849	10.7.18.197	128.119.245.12
435	5.048937	128.119.245.12	10.7.18.197
437	5.068184	10.7.18.197	128.119.245.12
482	5.417443	128.119.245.12	10.7.18.197
1272	12.291737	10.7.18.197	93.184.221.240
5284	58.384550	10.7.18.197	128.119.245.12
5311	58.730387	128.119.245.12	10.7.18.197

```
Host: gaia.cs.umass.edu\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; r
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-US,en;q=0.5\r\n
Accept-Encoding: gzip, deflate\r\n
Connection: keep-alive\r\n
Upgrade-Insecure-Requests: 1\r\n
If-Modified-Since: Sun, 15 Sep 2019 05:59:01 GMT\r\n
If-None-Match: "173-592912f1a8b7c"\r\n
```

2.2 What is the difference in first and second response received? What is the last modified time in the first response message?

Last modified date in 1st response: Sun, 15 Sep 2019 05:59:01 GMT
Last modified date in 2nd response: Sun, 15 Sep 2019 05:59:01 GMT

Response of first GET request was 200, which means all the objects were fetched succesfully from the server.
Response of the second GET request was 304, which means only the header bits were sent and object files were fetche from the cache because of the if-modified-since and if-none-match flags.

No.	Time	Source	Destination	Protocol	Length	Info
401	4.699849	10.7.18.197	128.119.245.12	HTTP	421	GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1
435	5.048937	128.119.245.12	10.7.18.197	HTTP	784	HTTP/1.1 200 OK (text/html)
437	5.068184	10.7.18.197	128.119.245.12	HTTP	313	GET /favicon.ico HTTP/1.1
482	5.417443	128.119.245.12	10.7.18.197	HTTP	539	HTTP/1.1 404 Not Found (text/html)
1272	12.291737	10.7.18.197	93.184.221.240	HTTP	299	GET /msdownload/update/v3/static/trustedr/en/disallowedce
5284	58.384550	10.7.18.197	128.119.245.12	HTTP	533	GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1
5311	58.730387	128.119.245.12	10.7.18.197	HTTP	294	HTTP/1.1 304 Not Modified

2.3 What is the HTTP status code and phrase returned from the server in response to the first and second HTTP GET? Did the server explicitly return the contents of the file? Explain.

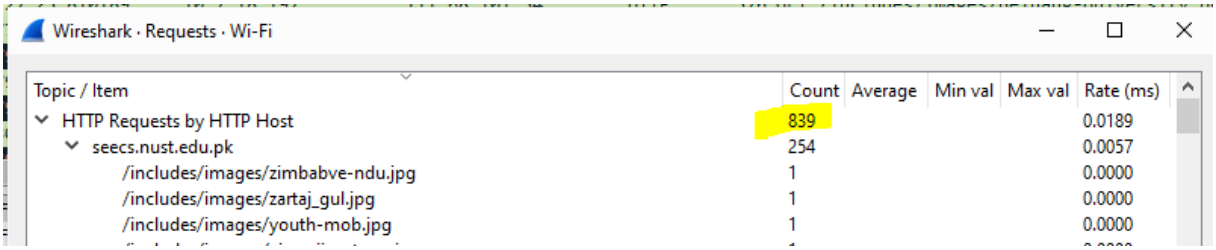
Status code: 200 – all objects were fetched successfully
Status code description: OK
Yes, the contents of file were explicity returned as indicated by the status code 200 and file content in the sniffed packets.

```
5311 58.730387    128.119.245.12    10.7.18.197    HTTP    294 HTTP/1.1 304 Not Modified
```

```
Transmission Control Protocol, Src Port: 80, Dst Port: 55602, Seq: 1, Ack: 368, Len: 730
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
    [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
    Response Version: HTTP/1.1
    Status Code: 200
    [Status Code Description: OK]
    Response Phrase: OK
  Date: Mon, 16 Sep 2019 04:50:58 GMT\r\n
  Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips PHP/5.4.16 mod_perl/2.0.10 Perl/v5.16.3\r\n
  Last-Modified: Sun, 15 Sep 2019 05:59:01 GMT\r\n
  ETag: "173-592912f1a8b7c"\r\n
  Accept-Ranges: bytes\r\n
  Content-Length: 371\r\n
    [Content length: 371]
  Keep-Alive: timeout=5, max=100\r\n
  Connection: Keep-Alive\r\n
  Content-Type: text/html; charset=UTF-8\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.349088000 seconds]
  [Request in frame: 401]
  [Request URI: http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html]
  File Data: 371 bytes
Line-based text data: text/html (10 lines)
  \n
  <html>\n
  \n
  Congratulations again!  Now you've downloaded the file lab2-2.html. <br>\n
  This file's last modification date will not change.  <p>\n
  Thus  if you download this multiple times on your browser, a complete copy <br>\n
  will only be sent once by the server due to the inclusion of the IN-MODIFIED-SINCE<br>\n
  field in your browser's HTTP GET request to the server.\n
  \n
  </html>\n
```

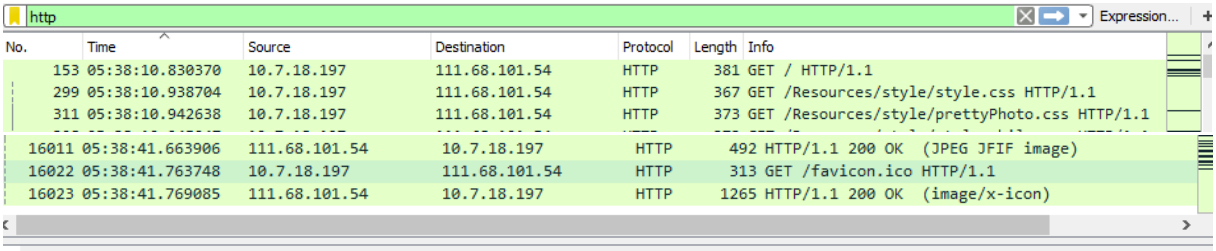
2.4 Empty your browser cache again and open the webpage www.seecs.edu.pk and capture the GET and OK response messages. How many total objects does the server return?

254 objects were returned from the host.
839 objects were returned by hot and the sites directed by host.



2.5 What is the page load time (PLT) for the interaction in 2.4?

Time of first GET request: 05:38:10.830370
Time of last response request: 05:38:41.769085
Loading time = 41.77 – 10.83 = 30.94



Name: M. Hasnain Naeem **Regn. No.: 212728**

Lab 4: Implement simple Client / Server applications using UDP and TCP

1.0 Objectives:

After this lab, the students should be able to

- Explain the concepts of client server communication
- Setup client/server communication
- Use the sockets interface of C programming language
- Implement simple Client / Server applications using UDP and TCP

2.0 Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *To obtain credit for this lab, you are supposed to complete the lab tasks and provide the source codes and the screen shot of your output in this document (please use red font color) and upload the completed document to your course's LMS site.*
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.*

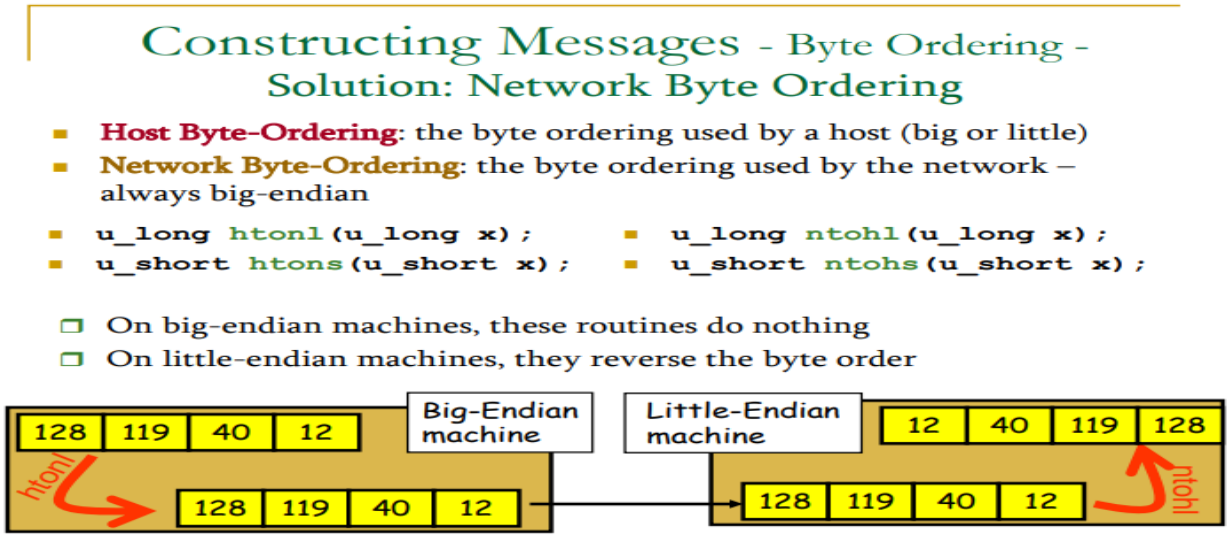
3.0 Background:

4.0 Client-Server Socket Programming

We introduce UDP and TCP socket programming by way of a simple UDP application and a simple TCP application both implemented in Python.

We'll use the following simple 'Echo' client-server application to demonstrate socket programming for both UDP and TCP:

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data .
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.



What is the difference between TCP & UDP protocols of TCP/IP protocol suite.

The 2 types of traffic in the network are based on TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). Following are the differences between the two

- TCP is connection Oriented protocol, hence a connection need to be established (using 3-way handshaking) before data is transmitted using TCP. UDP is Connectionless protocol and no connection need to be established. The packets are sent directly over the network.

- Because connection need to be established, TCP data transfer takes more time (3-way handshaking is done for establishing connection and then for removing the connection) than data transferred using UDP.

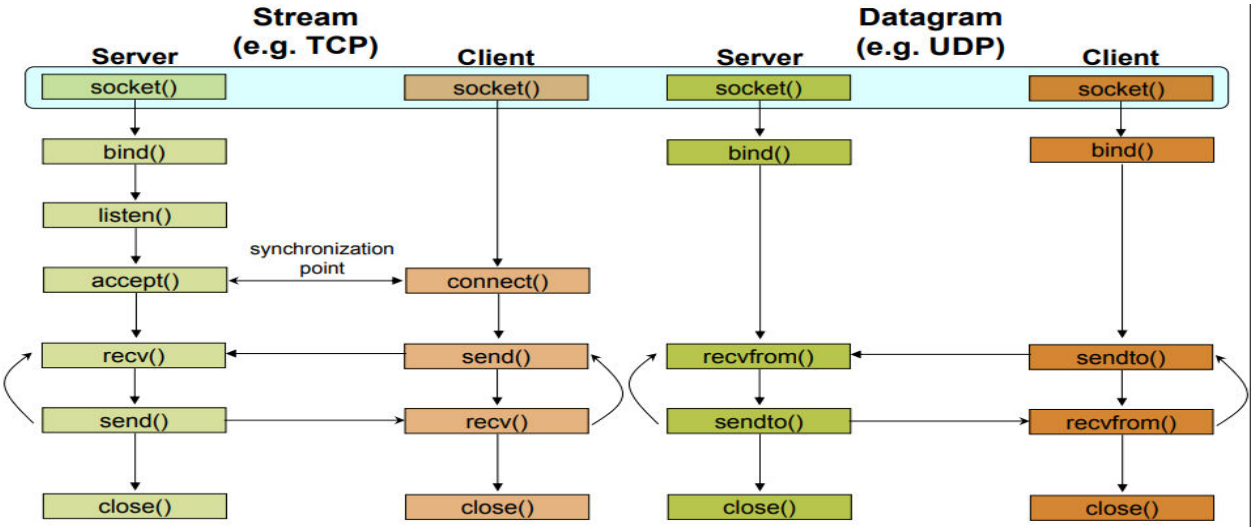
- Connection in the TCP is established to make the transfer reliable (acknowledgement based). Hence data transfer using TCP is reliable and UDP is non-reliable (sender does no know, for sure, if the packet has actually reached the receiver or not).

- Header Size of a TCP packet is bigger than the UDP header.

- TCP does the error checking also, UDP does not have an option for Error checking.

- Packets are ordered in case of TCP (i.e they are received in the same order as they are sent). 7. Application layer protocols like HTTP, FTP, Telnet, etc. uses TCP to transmit data whereas UDP is used by protocols like VoIP, DHCP, SNMP, etc.

Figure 2 TCP vs UDP Flow



4.1 Socket programming using UDP

Figure 2 highlights the main socket-related activity of the client and server that communicate over the UDP transport service.

Now let's take a look at the client-server program pair for a UDP implementation of this simple application. We'll begin with the UDP client, which will send a simple application-level message to the server. In order for the server to be able to receive and reply to the client's message, it must be ready and running— that is, it must be running as a process before the client sends its message.

The client program is called UDPClient.c, and the server program is called UDPServer.c. In order to emphasize the key issues, we intentionally provide code that is minimal. "Good code" would certainly have a few more auxiliary lines, in particular for handling error cases.

Socket API defines a generic data type for addresses:

Particular form of the sockaddr used for TCP/IP addresses:
Important: sockaddr_in can be casted to a sockaddr

```
struct sockaddr {
unsigned short sa_family; /* Address family (e.g. AF_INET) */
char sa_data[14]; /* Family-specific address information */
}
struct in_addr {
unsigned long s_addr; /* Internet address (32 bits) */
}
struct sockaddr_in {
unsigned short sin_family; /* Internet protocol (AF_INET) */
unsigned short sin_port; /* Address port (16 bits) */
struct in_addr sin_addr; /* Internet address (32 bits) */
char sin_zero[8]; /* Not used */
}
```

Socket creation in C

int sockid = socket(family, type, protocol);
sockid: socket descriptor, an integer (like a file-handle)
family: integer, communication domain, e.g.,
PF_INET, IPv4 protocols, Internet addresses (typically used)
PF_UNIX, Local communication, File addresses
type: communication type
SOCK_STREAM - reliable, 2-way, connection-based service
SOCK_DGRAM - unreliable, connectionless, messages of maximum length
protocol: specifies protocol „ IPPROTO_TCP IPPROTO_UDP usually set to 0 (i.e., use default protocol), upon failure returns -1
NOTE: socket call does not specify where data will be coming from,nor where it will be going to – it just creates the interface!

Closing a socket

status = close(sockid);
sockid: the file descriptor (socket being closed)
status: 0 if successful, -1 if error

Associates and reserves a port for use by the socket

int status = bind(sockid, &addrport, size);
sockid: integer, socket descriptor
addrport: struct sockaddr, the (IP) address and port of the machine for TCP/IP server, internet address is usually set to INADDR_ANY, i.e.,chooses any incoming interface
size: the size (in bytes) of the addrport structure
status: upon failure -1 is returned

Exchanging data with datagram socket

int count = sendto(sockid, msg, msgLen, flags, &foreignAddr, addrlen);
msg: const void[], message to be transmitted
msgLen: integer, length of message (in bytes) to transmit
flags: integer, special options, usually just 0
foreignAddr: struct sockaddr, address of the destination
addrLen: sizeof(foreignAddr)

int count = recvfrom(sockid, recvBuf, bufLen, flags, &clientAddr, addrlen);
recvBuf: void[], stores received bytes
bufLen: # bytes received
flags: integer, special options, usually just 0
clientAddr: struct sockaddr, address of the client
addrLen: sizeof(clientAddr)
Note:Calls are blocking , returns only after data is sent / received

Let's put everything together

UDPClient.c

```
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

int main() {
        const char* server_name = "localhost";
        const int server_port = 8877;

        struct sockaddr_in server_address;
        memset(&server_address, 0, sizeof(server_address));
        server_address.sin_family = AF_INET;

        // creates binary representation of server name
        // and stores it as sin_addr
        inet_pton(AF_INET, server_name, &server_address.sin_addr);

        // htons: port in network order format
        server_address.sin_port = htons(server_port);

        // open socket
        int sock;
        if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
                printf("could not create socket\n");
                return 1;
        }

        // data that will be sent to the server
        const char* data_to_send = " Hi BSCS 7";

        // send data
        int len =
           sendto(sock, data_to_send, strlen(data_to_send), 0,
                (struct sockaddr*)&server_address, sizeof(server_address));

        // received echoed data back
        char buffer[100];
        recvfrom(sock, buffer, len, 0, NULL, NULL);

        buffer[len] = '\0';
```

```
        printf("recieved: '%s'\n", buffer);

        // close the socket
        close(sock);
        return 0;
}
```

==========

UDPServer.c

==========

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
        // port to start the server on
        int SERVER_PORT = 8877;
      socklen_t client_address_len;

        // socket address used for the server
        struct sockaddr_in server_address;
        memset(&server_address, 0, sizeof(server_address));
        server_address.sin_family = AF_INET;

        // htons: host to network short: transforms a value in host byte
        // ordering format to a short value in network byte ordering format
        server_address.sin_port = htons(SERVER_PORT);

        // htons: host to network long: same as htons but to long
        server_address.sin_addr.s_addr = htonl(INADDR_ANY);

        // create a UDP socket, creation returns -1 on failure
        int sock;
        if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
                printf("could not create socket\n");
                return 1;
        }

        // bind it to listen to the incoming connections on the created server
        // address, will return -1 on error
        if ((bind(sock, (struct sockaddr *)&server_address,
                sizeof(server_address))) < 0) {
                printf("could not bind socket\n");
                return 1;
```

```
        }

        // socket address used to store client address
        struct sockaddr_in client_address;


        // run indefinitely
        while (true) {
                char buffer[500];

                // read content into buffer from an incoming client
                int len = recvfrom(sock, buffer, sizeof(buffer), 0,
                                (struct sockaddr *)&client_address,
                                &client_address_len);

                // inet_ntoa prints user friendly representation of the
                // ip address
                buffer[len] = '\0';
                printf("received: '%s' from client %s\n", buffer,
                      inet_ntoa(client_address.sin_addr));

                // send same content back to the client ("echo")
                sendto(sock, buffer, len, 0, (struct sockaddr *)&client_address,
                      sizeof(client_address));
        }

        return 0;
}
==========================================
```

4.1.1 Lab Task 1: A useful Python UDP client/server application.

Modify the UDPClient program such that the UDPClient is able to calculate the Application level Round Trip Time (RTT) for the communication between the Client and the Server. The Client should also print the time when Request is send and time when the Reply is received in human readable form.

```
/*
   Lab 4 - Task 1
   Client Side Code
   Written by: M. Hasnain Naeem
   Dated: 30 September, 2019
*/

#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
```

```
#include <sys/socket.h>
#include <unistd.h>
#include <time.h>

int main() {
   const char* server_name = "localhost";
   const int server_port = 8877;

   struct sockaddr_in server_address;
   memset(&server_address, 0, sizeof(server_address));
   server_address.sin_family = AF_INET;
   server_address.sin_addr.s_addr = htonl(INADDR_ANY);

   // creates binary representation of server name
   // and stores it as sin_addr
   inet_pton(AF_INET, server_name, &server_address.sin_addr);

   // htons: port in network order format
   server_address.sin_port = htons(server_port);

   // open socket
   int sock;
   if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
      printf("Could not create socket!\n");
      return 1;
   }

   // data that will be sent to the server
   const char* data_to_send = "Hi to server from the client!";

   // printing time before sending the request
   time_t rawtime;
   struct tm * timeinfo;
   time( &rawtime );
   timeinfo = localtime( &rawtime );
   int sending_second = timeinfo->tm_sec;
   printf("Time just before sending to server: %02d:%02d:%02d\n", timeinfo->tm_hour, timeinfo-
>tm_min,
   timeinfo->tm_sec);
   // send data
   int len = sendto(sock, data_to_send, strlen(data_to_send), 0,
            (struct sockaddr*)&server_address, sizeof(server_address));

   sleep(2);
   // received echoed data back
   char buffer[100];
   recvfrom(sock, buffer, len, 0, NULL, NULL);
```

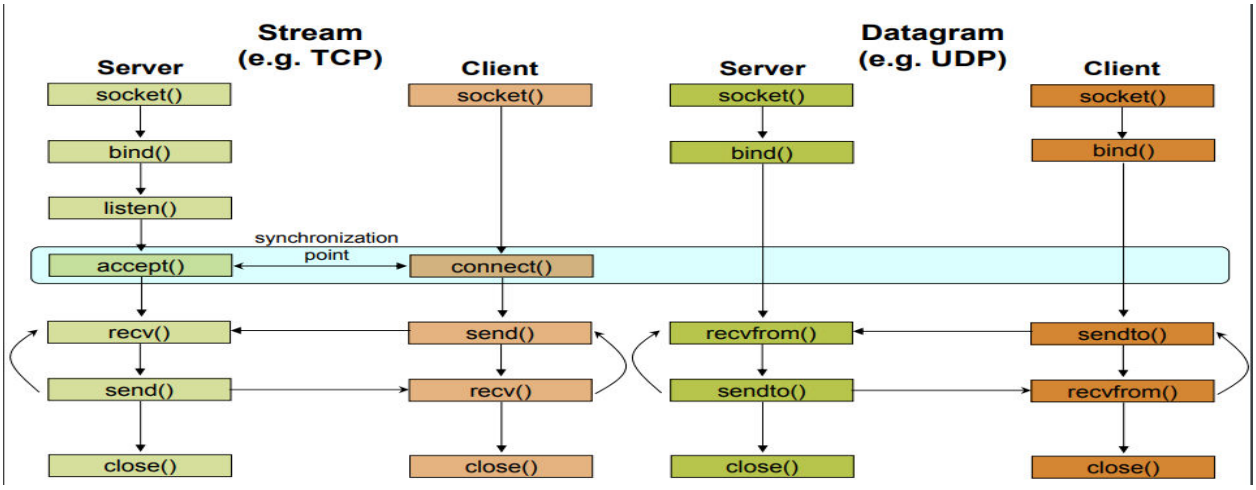
```
    buffer[len] = '\0';

    time( &rawtime );
    timeinfo = localtime( &rawtime );
    int recieving_second = timeinfo->tm_sec;
    printf("Time of recieving from the server: %02d:%02d:%02d\n", timeinfo->tm_hour, timeinfo->tm_min,
    timeinfo->tm_sec);
    printf("Recieved '%s' from server in %d seconds\n", buffer, recieving_second-sending_second);

    // close the socket
    close(sock);
    return 0;
}
```

```
hex00@ubuntu:~/Desktop/codes/cn/lab4$ ./UDPClient
Time just before sending to server: 00:04:00
Time of recieving from the server: 00:04:00
Recieved 'Hi to server from the client!' from server in 71.000000 ms.
```

4.2 Socket programming using TCP

We use the same simple client-server application to demonstrate socket programming with TCP: The client sends one line of data to the server, the server capitalizes the line and sends it back to the client. Figure 2 highlights the main socket-related activity of the client and server that communicate over the TCP transport service.

Let's now take a look at the lines that differ significantly from UDPServer and TCPClient.



Instructs TCP protocol implementation to listen for connections„

int status = listen(sockid, queueLimit);

sockid: integer, socket descriptor
queuelen: integer, # of active participants that can "wait" for a connection
status: 0 if listening, -1 if error

Note:listen() is non-blocking: returns immediately .The listening socket (sockid) is never used for sending and receiving is used by the server only as a way to get new sockets

The client establishes a connection with the server by calling connect()

int status = connect(sockid, &foreignAddr, addrlen);

sockid: integer, socket to be used in connection
foreignAddr: struct sockaddr: address of the passive participant
addrlen: integer, sizeof(name)
status: 0 if successful connect, -1 otherwise

Note:connect() is blocking

The server gets a socket for an incoming client connection by calling accept()

int s = accept(sockid, &clientAddr, &addrLen);

sockid: integer, the orig. socket (being listened on)
clientAddr: struct sockaddr, address of the active participant filled in upon return
addrLen: sizeof(clientAddr): value/result parameter must be set appropriately before call adjusted upon return

Note:accept() is blocking: waits for connection before returning ‰ dequeues the next connection on the queue for socket (sockid)

Sending and receiving data

int count = send(sockid, msg, msgLen, flags);

msg: const void[], message to be transmitted
msgLen: integer, length of message (in bytes) to transmit
flags: integer, special options, usually just 0
count: # bytes transmitted (-1 if error)

int count = recv(sockid, recvBuf, bufLen, flags);

recvBuf: void[], stores received bytes
bufLen: # bytes received
flags: integer, special options, usually just 0
count: # bytes received (-1 if error)

Note:Calls are blocking ‰ returns only afterdata is sent / received

Let's put everything together

TCPClient.c

```
#include <arpa/inet.h>
#include <stdio.h>
```

```
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

int main() {
	const char* server_name = "localhost";
	const int server_port = 8877;

	struct sockaddr_in server_address;
	memset(&server_address, 0, sizeof(server_address));
	server_address.sin_family = AF_INET;

	// creates binary representation of server name
	// and stores it as sin_addr
	// http://beej.us/guide/bgnet/output/html/multipage/inet_ntopman.html
	inet_pton(AF_INET, server_name, &server_address.sin_addr);

	// htons: port in network order format
	server_address.sin_port = htons(server_port);

	// open a stream socket
	int sock;
	if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
		printf("could not create socket\n");
		return 1;
	}

	// TCP is connection oriented, a reliable connection
	// **must** be established before any data is exchanged
	if (connect(sock, (struct sockaddr*)&server_address,
		 sizeof(server_address)) < 0) {
		printf("could not connect to server\n");
		return 1;
	}

	// send

	// data that will be sent to the server
	const char* data_to_send = "Gangadhar Hi Shaktimaan hai";
	send(sock, data_to_send, strlen(data_to_send), 0);

	// receive

	int n = 0;
	int len = 0, maxlen = 100;
```

```
        char buffer[maxlen];
        char* pbuffer = buffer;

        // will remain open until the server terminates the connection
        while ((n = recv(sock, pbuffer, maxlen, 0)) > 0) {
                pbuffer += n;
                maxlen -= n;
                len += n;

                buffer[len] = '\0';
                printf("received: '%s'\n", buffer);
        }

        // close the socket
        close(sock);
        return 0;
}
```

==========

TCPServer.c

```
include <arpa/inet.h>
#include <netinet/in.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/**
 * TCP Uses 2 types of sockets, the connection socket and the listen socket.
 * The Goal is to separate the connection phase from the data exchange phase.
 * */

int main(int argc, char *argv[]) {
        // port to start the server on
        int SERVER_PORT = 8877;
        socklen_t client_address_len;

        // socket address used for the server
        struct sockaddr_in server_address;
        memset(&server_address, 0, sizeof(server_address));
        server_address.sin_family = AF_INET;

        // htons: host to network short: transforms a value in host byte
        // ordering format to a short value in network byte ordering format
        server_address.sin_port = htons(SERVER_PORT);
```

```
// htonl: host to network long: same as htons but to long
server_address.sin_addr.s_addr = htonl(INADDR_ANY);

// create a TCP socket, creation returns -1 on failure
int listen_sock;
if ((listen_sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        printf("could not create listen socket\n");
        return 1;
}

// bind it to listen to the incoming connections on the created server
// address, will return -1 on error
if ((bind(listen_sock, (struct sockaddr *)&server_address,
        sizeof(server_address))) < 0) {
        printf("could not bind socket\n");
        return 1;
}

int wait_size = 16;  // maximum number of waiting clients, after which
                     // dropping begins
if (listen(listen_sock, wait_size) < 0) {
        printf("could not open socket for listening\n");
        return 1;
}

// socket address used to store client address
struct sockaddr_in client_address;


// run indefinitely
while (true) {
        // open a new socket to transmit data per connection
        int sock;
        if ((sock =
                accept(listen_sock, (struct sockaddr *)&client_address,
                    &client_address_len)) < 0) {
            printf("could not open a socket to accept data\n");
            return 1;
        }

        int n = 0;
        int len = 0, maxlen = 100;
        char buffer[maxlen];
        char *pbuffer = buffer;
```

```
            printf("client connected with ip address: %s\n",
                  inet_ntoa(client_address.sin_addr));

            // keep running as long as the client keeps the connection open
            while ((n = recv(sock, pbuffer, maxlen, 0)) > 0) {
                  pbuffer += n;
                  maxlen -= n;
                  len += n;

                  printf("received: '%s'\n", buffer);

                  // echo received content back
                  send(sock, buffer, len, 0);
            }

            close(sock);
      }

      close(listen_sock);
      return 0;
}
```

==========

4.2.1 Lab Task 2: A useful Python TCP client/server application.

Modify the TCPClient program such that the TCPClient is able to calculate the Application level Round Trip Time (RTT) for the communication between the Client and the Server. The Client should also print the time when connection request is send and time when the Reply (capitalized words) is received in human readable form.

```
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#include <time.h>

int main() {
   const char* server_name = "10.7.24.193";
   const int server_port = 9000;

   struct sockaddr_in server_address;
   memset(&server_address, 0, sizeof(server_address));
   server_address.sin_family = AF_INET;
```

```
// creates binary representation of server name
// and stores it as sin_addr
// http://beej.us/guide/bgnet/output/html/multipage/inet_ntopman.html
inet_pton(AF_INET, server_name, &server_address.sin_addr);

// htons: port in network order format
server_address.sin_port = htons(server_port);

// open a stream socket
int sock;
if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
   printf("could not create socket\n");
   return 1;
}

// TCP is connection oriented, a reliable connection
// **must** be established before any data is exchanged
if (connect(sock, (struct sockaddr*)&server_address,
          sizeof(server_address)) < 0) {
   printf("could not connect to server\n");
   return 1;
}

// send
// printing time before sending the request
time_t rawtime;
struct tm * timeinfo;
time( &rawtime );
timeinfo = localtime( &rawtime );
int sending_second = timeinfo->tm_sec;
printf("Time just before sending to server: %02d:%02d:%02d\n", timeinfo->tm_hour, timeinfo->tm_min,
timeinfo->tm_sec);
// data that will be sent to the server
const char* data_to_send = "Hello to server from client through TCP!";
send(sock, data_to_send, strlen(data_to_send), 0);

// receive

int n = 0;
int len = 0, maxlen = 100;
char buffer[maxlen];
char* pbuffer = buffer;

// will remain open until the server terminates the connection
while ((n = recv(sock, pbuffer, maxlen, 0)) > 0) {
```

```
        pbuffer += n;
        maxlen -= n;
        len += n;
        buffer[len] = '\0';

        time( &rawtime );
        timeinfo = localtime( &rawtime );
        int recieving_second = timeinfo->tm_sec;
        printf("Time of recieving from the server: %02d:%02d:%02d\n", timeinfo->tm_hour, timeinfo->tm_min,
        timeinfo->tm_sec);
        printf("Recieved '%s' from server in %d seconds\n", buffer, recieving_second-sending_second);

    }

    // close the socket
    close(sock);
    return 0;
}
```

```
hex00@ubuntu:~/Desktop/codes/cn/lab4$ ./TCPClient
Time just before sending to server: 00:00:30
Time of recieving from the server: 00:00:30
Recieved 'Hello to server from client through TCP!' from server in 107.000000 ms
.
```

4.2.2 Lab Task 3: Compare the values of the RTT for both the UDP and TCP. Which one has got higher RTT? Why?

RRT using UDP: 71 ms
RRT using TCP: 107 ms

TCP has higher RRT, because it has three-way handshake, acknowledgment message from the receiver and in case of packet damage/lose, those are sent again.

UDP doesn't has connection establishment and it doesn't require acknowledgment from the receiver and damaged/lost packets are not retrieved from the sender.

4.2.3 Lab Task 4: What happens when your client (both UDP and TCP) tries to send data to a non-existent server?

UDP

No connection error occurs.

Client sends the request to server and keeps on waiting for the reply from server until the timeout criteria is met. No other error occurs because connection is not established and acknowledgment is not sent in UDP.

TCP

Connection error occurs when a client tries to send a packet to server which doesn't exist. It's because TCP requires connection establishment before sending the packets and acknowledgment from the server.

Conclusion:

TCP and UDP protocols are used in transport layer for communication between client and server through socket programming. Programmer can use utilize any of these two mechanisms for communication depending on the situation.

UDP can be used for broadcasting and time-sensitive scenarios where packet loss is tolerable. If packet loss is not tolerable then TCP is used.

1.1 Unix Socket – Functions Summary(From TutorialsPoint.com)

Here is a list of all the functions related to socket programming.

1.2 Port and Service Functions

Unix provides the following functions to fetch service name from the /etc/services file.

- **struct servent *getservbyname(char *name, char *proto)** − This call takes a service name and a protocol name and returns the corresponding port number for that service.
- **struct servent *getservbyport(int port, char *proto)** − This call takes a port number and a protocol name and returns the corresponding service name.

1.3 Byte Ordering Functions

- **unsigned short htons (unsigned short hostshort)** − This function converts 16-bit (2-byte) quantities from host byte order to network byte order.
- **unsigned long htonl (unsigned long hostlong)** − This function converts 32-bit (4-byte) quantities from host byte order to network byte order.
- **unsigned short ntohs (unsigned short netshort)** − This function converts 16-bit (2-byte) quantities from network byte order to host byte order.
- **unsigned long ntohl (unsigned long netlong)** − This function converts 32-bit quantities from network byte order to host byte order.

1.4 IP Address Functions

- **int inet_aton (const char *strptr, struct in_addr *addrptr)** − This function call converts the specified string, in the Internet standard dot notation, to a network address, and stores the address in the structure provided. The

converted address will be in Network Byte Order (bytes ordered from left to right). It returns 1 if the string is valid and 0 on error.

- **in_addr_t inet_addr (const char *strptr)** − This function call converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). It returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.

- **char *inet_ntoa (struct in_addr inaddr)** − This function call converts the specified Internet host address to a string in the Internet standard dot notation.

1.5 Socket Helper Functions

- **void bzero (void *s, int nbyte)** − The bzero function places nbyte null bytes in the string s. This function will be used to set all the socket structures with null values.

- **int bcmp (const void *s1, const void *s2, int nbyte)** − The bcmp function compares the byte string s1 against the byte string s2. Both the strings are assumed to be nbyte bytes long.

- **void bcopy (const void *s1, void *s2, int nbyte)** − The bcopy function copies nbyte bytes from the string s1 to the string s2. Overlapping strings are handled correctly.

- **void *memset(void *s, int c, int nbyte)** − The memset function is also used to set structure variables in the same way as bzero.

Refernces

Tuturiopoint.com

https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf

Department of Computer Science

EE353: Computer Networks

Class: BSCS7AB

Lab#5 : Domain Name System(DNS)

Date: 14 OCT 2019

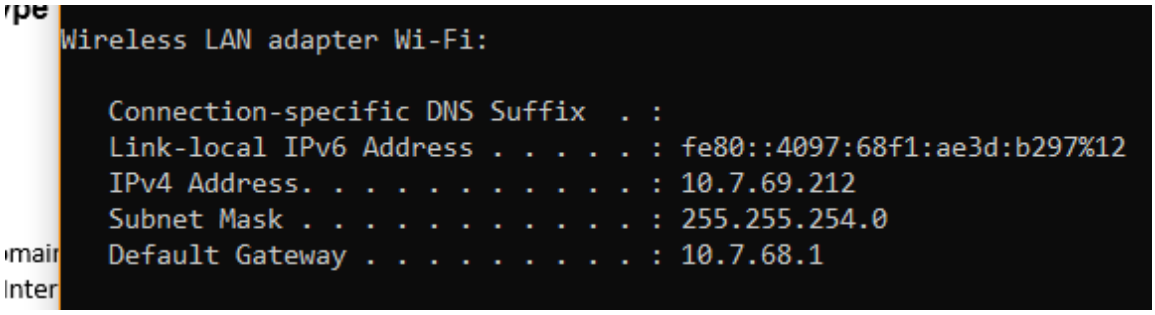
Lab Engineer: Kaleem Ullah

Instructor: Dr. Muhammad Zeeshan

Lab Title: Domain Name System(DNS)

Name: M. Hasnain Naeem Regn. No.: 212728

IP Screen Shot:



Objective of this lab:

As described in Section 2.5 of the textbook, the Domain Name System (DNS) translates hostnames to IP addresses, fulfilling a critical role in the Internet infrastructure. In this lab, we'll take a closer look at the client side of DNS. Recall that the client's role in the DNS is relatively simple – a client sends a *query* to its local DNS server, and receives a *response* back. Much can go on "under the covers," invisible to the DNS clients, as the hierarchical DNS servers communicate with each other to either recursively or iteratively resolve the client's DNS query. From the DNS client's standpoint, however, the protocol is quite simple – a query is formulated to the local DNS server and a response is received from that server.

Before beginning this lab, you'll probably want to review DNS by reading Section 2.5 of the textbook. In particular, you may want to review the material on **local DNS servers**, **DNS caching**, **DNS records and messages**, and the **TYPE field** in the DNS record.

Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *You are supposed to provide the answers to the questions listed at the end of this document and upload the completed report to your course's LMS site.*
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.*
- *Complete the lab half an hour before the lab ends.*
- *At the end of the lab, an online Quiz will be conducted to evaluate your understanding.*

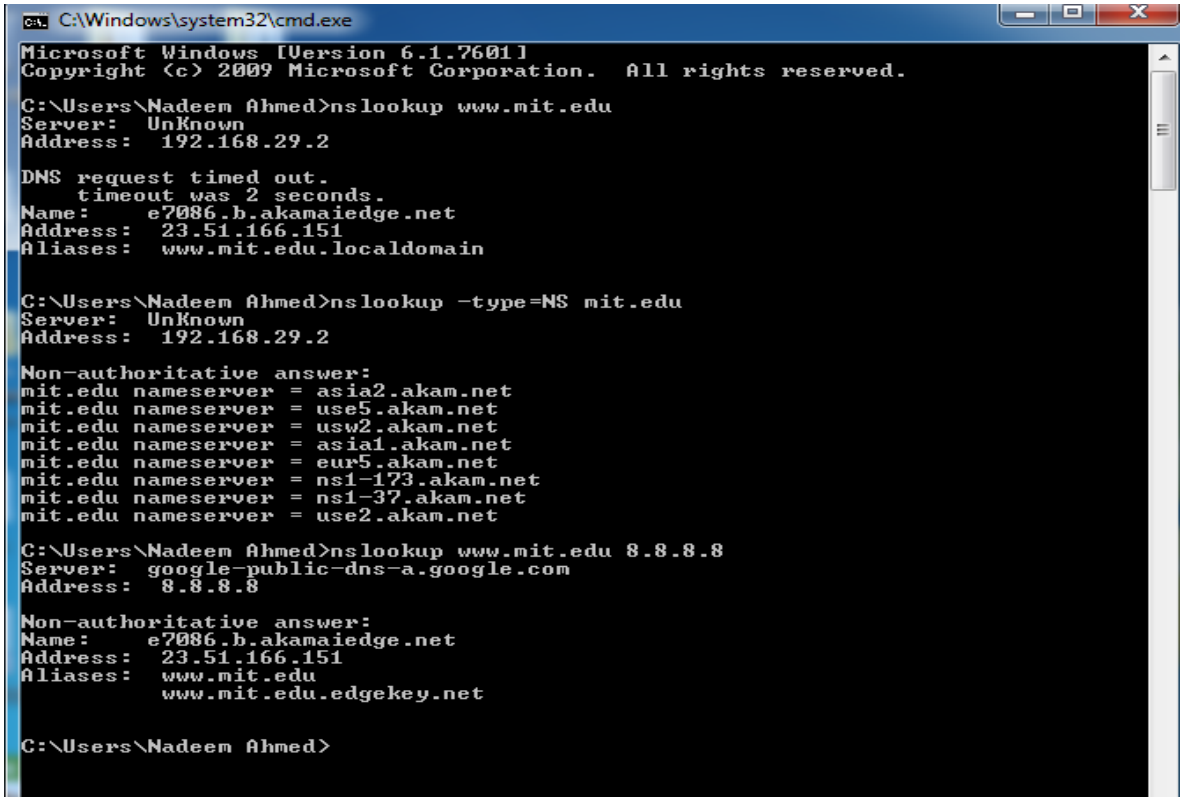
1. Introduction to DNS

There are two ways to identify a host -- a hostname and an IP address. People prefer the more mnemonic hostname identifier, while routers prefer fixed-length, hierarchically-structured IP addresses. In order to reconcile these different preferences, we need a directory service that translates hostnames to IP addresses. This is the main task of the Internet's **Domain Name System (DNS)**. The DNS is (i) a distributed database implemented in a hierarchy of **name servers** and (ii) an application-layer protocol that allows hosts and name servers to communicate in order to provide the translation service.

2. Introduction to NSLOOKUP

In this lab, we'll make extensive use of the *nslookup* tool, which is available in most Linux/Unix and Microsoft platforms today. To run *nslookup* in Linux/Unix, you just type the *nslookup* command on the command line. To run it in Windows, open the Command Prompt and run *nslookup* on the command line.

In it is most basic operation, *nslookup* tool allows the host running the tool to query any specified DNS server for a DNS record. The queried DNS server can be a root DNS server, a top-level-domain DNS server, an authoritative DNS server, or an intermediate DNS server (see the textbook for definitions of these terms). To accomplish this task, *nslookup* sends a DNS query to the specified DNS server, receives a DNS reply from that same DNS server, and displays the result.



The above screenshot shows the results of three independent *nslookup* commands (displayed

in the Windows Command Prompt). In this example, the client host is using Windows 7 and is connected to an ISP using ADSL modem. When running *nslookup*, if no DNS server is specified, then *nslookup* sends the query to the default DNS server, which in this case is 192.168.29.2. Consider the first command:

i. **nslookup www.mit.edu**

In words, this command is saying "please send me the IP address for the host www.mit.edu". As shown in the screenshot, the response from this command provides two pieces of information: (1) the name and IP address of the local DNS server that provides the answer; and (2) the answer itself, which is the host name and IP address (of a CDN node hosting the site) www.mit.edu. Although the response came from the local DNS server, it is quite possible that this local DNS server iteratively contacted several other DNS servers to get the answer, as described in Section 2.5 of the textbook.

Now consider the second command:

ii. **nslookup –type=NS mit.edu**

In this example, we have provided the option "-type=NS" and the domain "mit.edu". This causes *nslookup* to send a query for a type-NS record to the default local DNS server. In words, the query is saying, "please send me the host names of the authoritative DNS servers for mit.edu". (When the –type option is not used, *nslookup* uses the default, which is to query for type A records.) The answer, displayed in the above screenshot, first indicates the DNS server that is providing the answer (which is the default local DNS server) along with MIT nameservers (again hosted on Akamai CDN). Each of these servers is indeed an authoritative DNS server for the hosts on the MIT campus. However, *nslookup* also indicates that the answer is "non - authoritative," meaning that this answer came from the cache of some server rather than from an authoritative MIT DNS server. Note that the answer does not include the IP addresses of the authoritative DNS servers for MIT.

Now finally consider the third command:

iii. **nslookup www.mit.edu 8.8.8.8**

In this example, we indicate that we want the query sent to the Google public DNS server (well known IP 8.8.8.8) rather than to the default local DNS server (192.168.29.2). Thus, the query and reply transaction takes place directly between our querying host and Google public DNS server. In this example, the Google DNS server provided the IP address of the host www.mit.edu that is similar to the response in first nslookup.

Now that we have gone through a few illustrative examples, you are perhaps wondering about the general syntax of *nslookup* commands. The syntax is:

nslookup –option1 –option2 host-to-find dns-server

In general, *nslookup* can be run with zero, one, two or more options. And as we have seen in the above examples, the dns-server is optional as well; if it is not supplied, the query is sent to the default local DNS server.

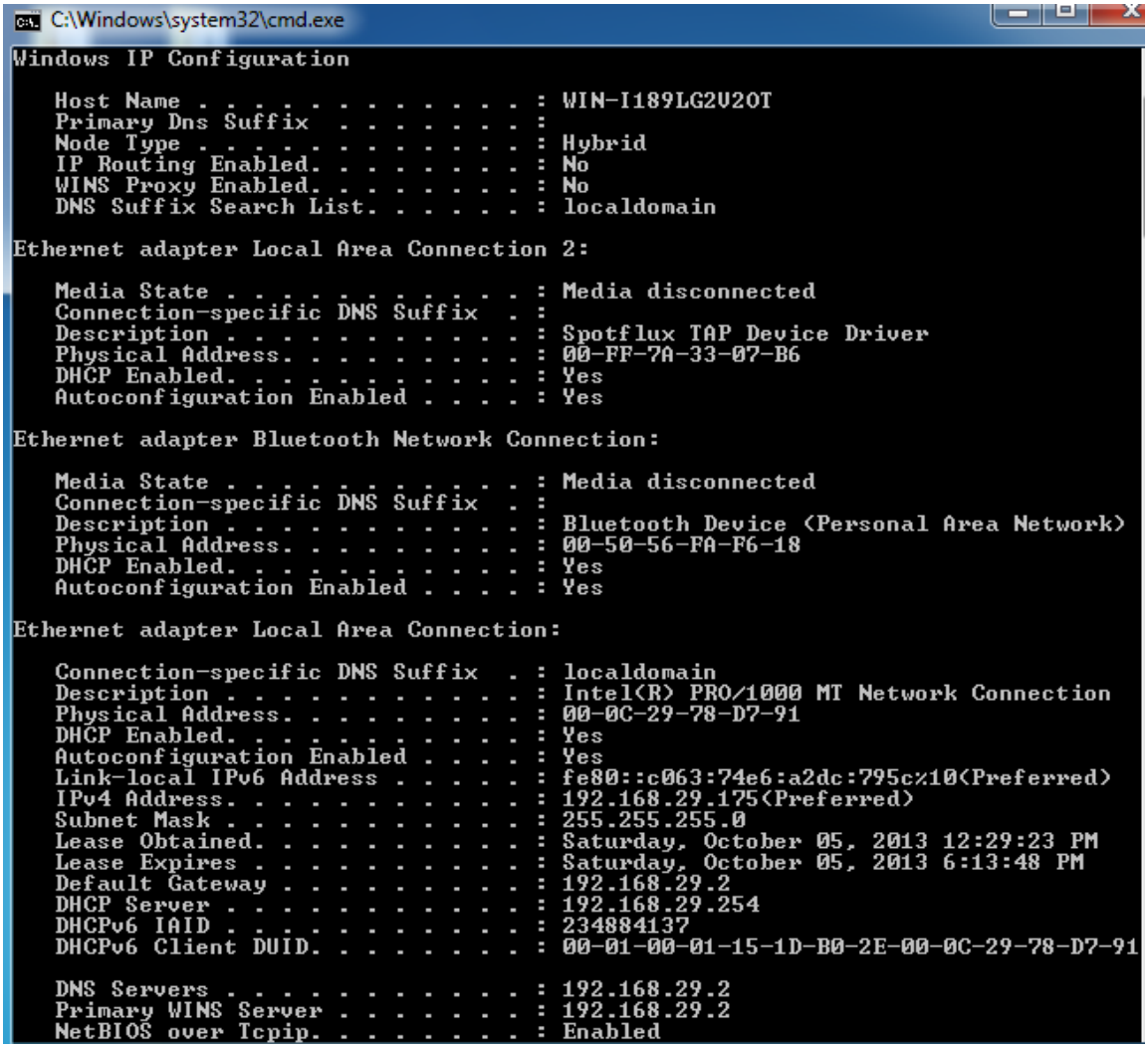
You can also run the nslookup in interactive mode. Simply type nslookup and press enter. Now at the > prompt, type help to see the list of all options available.

3. Introduction to IPCONFIG

ipconfig (for Windows) and *ifconfig* (for Linux/Unix) are among the most useful little utilities in your host, especially for debugging network issues. Here we'll only describe *ipconfig*, although the Linux/Unix *ifconfig* is very similar. *ipconfig* can be used to show your current TCP/IP information, including your address, DNS server addresses, adapter type and so on. For example, if you can get all this information about your host simply by entering:

i. Ipconfig /all

into the Command Prompt, as shown in the following screenshot.



ipconfig is also very useful for managing the DNS information stored in your host. In Section 2.5 we learned that a host can cache DNS records it recently obtained. To see these cached records, after the prompt C:\> provide the following command:

ii. ipconfig /displaydns

Each entry shows the remaining Time to Live (TTL) in seconds. To clear the cache, enter:

iii. ipconfig /flushdns

Flushing the DNS cache clears all entries.

Steps for performing this lab:

Exercise 01: nslookup

Now that we have provided an overview of *nslookup*, it is time for you to test drive it yourself. You have to use 8.8.8.8 as your local DNS server for all these exercises. Do the following (and write down the results):

1.1 Run *nslookup* to obtain the IP address of the Web server hosting www.seecs.nust.edu.pk.

```
C:\Users\G3NZ>nslookup -type=NS www.seecs.nust.edu.pk
Server:  dns.google
Address:  8.8.8.8

Non-authoritative answer:
www.seecs.nust.edu.pk   canonical name = seecs.nust.edu.pk
seecs.nust.edu.pk       nameserver = ns1.nust.edu.pk
seecs.nust.edu.pk       nameserver = ns2.nust.edu.pk
```

1.2 Run *nslookup* to determine the authoritative DNS servers for domain seecs.edu.pk. Provide both the names of these DNS servers and also the IP addresses of one of the DNS servers.

Nameservers point to the actual servers hosting the content of website. So, here authoritative DNS servers are actually the name servers. Which are:

```
C:\Users\G3NZ>nslookup -type=ns seecs.edu.pk
Server:  dns.google
Address:  8.8.8.8

Non-authoritative answer:
seecs.edu.pk    nameserver = ns1.seecs.edu.pk
seecs.edu.pk    nameserver = ns3.seecs.edu.pk
seecs.edu.pk    nameserver = ns2.seecs.edu.pk
```

IP addresses of nameservers are:

1. Name: ns1.seecs.edu.pk
   Address: 111.68.101.5
2. Name: ns2.seecs.edu.pk
   Address: 111.68.101.6
3. Name: ns3.seecs.edu.pk
   Address: 111.68.101.7

1.3 Run nslookup to determine the mail servers for seecs.edu.pk**. Provide both the names of these Mail servers and also the IP address of one of these Mail servers.**

Names:

- ASPMX.L.GOOGLE.COM
- ALT1.ASPMX.L.GOOGLE.COM

- ALT2.ASPMX.L.GOOGLE.COM
- ASPMX2.GOOGLEMAIL.COM
- ASPMX3.GOOGLEMAIL.COM
- ASPMX4.GOOGLEMAIL.COM
- ASPMX5.GOOGLEMAIL.COM

IP Address of ALT2.ASPMX.L.GOOGLE.COM: 209.85.233.27

```
C:\Users\G3NZ>nslookup -type=mx seecs.edu.pk
Server:  dns.google
Address:  8.8.8.8

Non-authoritative answer:
seecs.edu.pk    MX preference = 10, mail exchanger = ASPMX2.GOOGLEMAIL.COM
seecs.edu.pk    MX preference = 10, mail exchanger = ASPMX3.GOOGLEMAIL.COM
seecs.edu.pk    MX preference = 20, mail exchanger = ASPMX4.GOOGLEMAIL.COM
seecs.edu.pk    MX preference = 20, mail exchanger = ASPMX5.GOOGLEMAIL.COM
seecs.edu.pk    MX preference = 1, mail exchanger = ASPMX.L.GOOGLE.COM
seecs.edu.pk    MX preference = 5, mail exchanger = ALT1.ASPMX.L.GOOGLE.COM
seecs.edu.pk    MX preference = 5, mail exchanger = ALT2.ASPMX.L.GOOGLE.COM
```

```
C:\Users\G3NZ>nslookup ASPMX2.GOOGLEMAIL.COM
Server:  dns.google
Address:  8.8.8.8

Non-authoritative answer:
Name:    ASPMX2.GOOGLEMAIL.COM
Addresses:  2a00:1450:4010:c03::1a
          209.85.233.27
```

1.4 Query the public DNS service provided by Google at 8.8.8.8 to query for the IPv6 address of www.seecs.edu.pk. Provide the IPv6 address. Note how this address is different from the IPv4 addresses that you were getting for the previous questions.

```
C:\Users\G3NZ>nslookup -query=AAAA www.seecs.edu.pk
Server:  dns.google
Address:  8.8.8.8

Non-authoritative answer:
Name:    www.seecs.edu.pk
Address:  2400:fc00:8070::30
```

IPv6 Address: 2400:fc00:8070::30
Difference: Ipv4 uses 32 bit address and Ipv6 address uses 128 bit address.

Exercise 02: Tracing DNS with Wireshark (while using browser)

Now that we are familiar with *nslookup* and *ipconfig*, we're ready to get down to some serious business. Let's first capture the DNS packets that are generated by ordinary Web-surfing activity.

- Use *ipconfig* to empty the DNS cache in your host.
- Open your browser and empty your browser cache. (With Internet Explorer, go to Tools menu and select Internet Options; then in the General tab select Delete Files.)
- Open Wireshark and enter "ip.addr == your_IP_address" into the filter, where you obtain your_IP_address with ipconfig. This filter removes all packets that neither

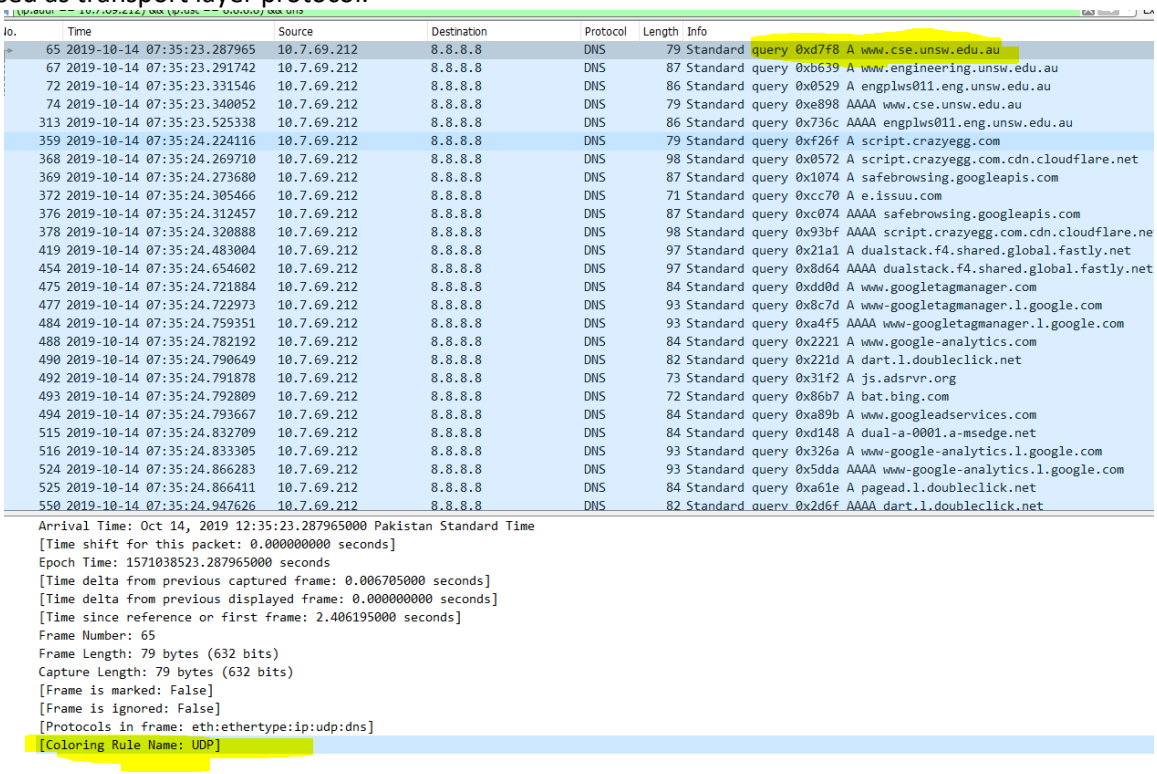
originate nor are destined to your host.

```
Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::4097:68f1:ae3d:b297%12
   IPv4 Address. . . . . . . . . . . : 10.7.69.212
   Subnet Mask . . . . . . . . . . . : 255.255.254.0
   Default Gateway . . . . . . . . . : 10.7.68.1
```

- Start packet capture in Wireshark.
- With your browser, visit the Web page: http://www.cse.unsw.edu.au
- Stop packet capture.

2.1 Locate the DNS query and response messages. Are these sent over UDP or TCP (i.e., what transport layer protocol is being used)?

UDP is used as transport layer protocol.



No.	Time	Source	Destination	Protocol	Length	Info
65	2019-10-14 07:35:23.287965	10.7.69.212	8.8.8.8	DNS	79	Standard query 0xd7f8 A www.cse.unsw.edu.au
67	2019-10-14 07:35:23.291742	10.7.69.212	8.8.8.8	DNS	87	Standard query 0xb639 A www.engineering.unsw.edu.au
72	2019-10-14 07:35:23.331546	10.7.69.212	8.8.8.8	DNS	86	Standard query 0x0529 A engplws011.eng.unsw.edu.au
74	2019-10-14 07:35:23.340052	10.7.69.212	8.8.8.8	DNS	79	Standard query 0xe898 AAAA www.cse.unsw.edu.au
313	2019-10-14 07:35:23.525338	10.7.69.212	8.8.8.8	DNS	86	Standard query 0x736c AAAA engplws011.eng.unsw.edu.au
359	2019-10-14 07:35:24.224116	10.7.69.212	8.8.8.8	DNS	79	Standard query 0xf26f A script.crazyegg.com
368	2019-10-14 07:35:24.269710	10.7.69.212	8.8.8.8	DNS	98	Standard query 0x0572 A script.crazyegg.com.cdn.cloudflare.net
369	2019-10-14 07:35:24.273680	10.7.69.212	8.8.8.8	DNS	87	Standard query 0x1074 A safebrowsing.googleapis.com
372	2019-10-14 07:35:24.305466	10.7.69.212	8.8.8.8	DNS	71	Standard query 0xcc70 A e.issuu.com
376	2019-10-14 07:35:24.312457	10.7.69.212	8.8.8.8	DNS	87	Standard query 0xc074 AAAA safebrowsing.googleapis.com
378	2019-10-14 07:35:24.320888	10.7.69.212	8.8.8.8	DNS	98	Standard query 0x93bf AAAA script.crazyegg.com.cdn.cloudflare.ne
419	2019-10-14 07:35:24.483004	10.7.69.212	8.8.8.8	DNS	97	Standard query 0x21a1 A dualstack.f4.shared.global.fastly.net
454	2019-10-14 07:35:24.654602	10.7.69.212	8.8.8.8	DNS	97	Standard query 0x8d64 AAAA dualstack.f4.shared.global.fastly.net
475	2019-10-14 07:35:24.721884	10.7.69.212	8.8.8.8	DNS	84	Standard query 0xdd0d A www.googletagmanager.com
477	2019-10-14 07:35:24.722973	10.7.69.212	8.8.8.8	DNS	93	Standard query 0x8c7d A www-googletagmanager.l.google.com
484	2019-10-14 07:35:24.759351	10.7.69.212	8.8.8.8	DNS	93	Standard query 0xa4f5 AAAA www-googletagmanager.l.google.com
488	2019-10-14 07:35:24.782192	10.7.69.212	8.8.8.8	DNS	84	Standard query 0x2221 A www.google-analytics.com
490	2019-10-14 07:35:24.790649	10.7.69.212	8.8.8.8	DNS	82	Standard query 0x221d A dart.l.doubleclick.net
492	2019-10-14 07:35:24.791878	10.7.69.212	8.8.8.8	DNS	73	Standard query 0x31f2 A js.adsrvr.org
493	2019-10-14 07:35:24.792809	10.7.69.212	8.8.8.8	DNS	72	Standard query 0x86b7 A bat.bing.com
494	2019-10-14 07:35:24.793667	10.7.69.212	8.8.8.8	DNS	84	Standard query 0xa89b A www.googleadservices.com
515	2019-10-14 07:35:24.832709	10.7.69.212	8.8.8.8	DNS	84	Standard query 0xd148 A dual-a-0001.a-msedge.net
516	2019-10-14 07:35:24.833305	10.7.69.212	8.8.8.8	DNS	93	Standard query 0x326a A www-google-analytics.l.google.com
524	2019-10-14 07:35:24.866283	10.7.69.212	8.8.8.8	DNS	93	Standard query 0x5dda AAAA www-google-analytics.l.google.com
525	2019-10-14 07:35:24.866411	10.7.69.212	8.8.8.8	DNS	84	Standard query 0xa61e A pagead.l.doubleclick.net
550	2019-10-14 07:35:24.947626	10.7.69.212	8.8.8.8	DNS	82	Standard query 0x2d6f AAAA dart.l.doubleclick.net

```
Arrival Time: Oct 14, 2019 12:35:23.287965000 Pakistan Standard Time
[Time shift for this packet: 0.000000000 seconds]
Epoch Time: 1571038523.287965000 seconds
[Time delta from previous captured frame: 0.006705000 seconds]
[Time delta from previous displayed frame: 0.000000000 seconds]
[Time since reference or first frame: 2.406195000 seconds]
Frame Number: 65
Frame Length: 79 bytes (632 bits)
Capture Length: 79 bytes (632 bits)
[Frame is marked: False]
[Frame is ignored: False]
[Protocols in frame: eth:ethertype:ip:udp:dns]
[Coloring Rule Name: UDP]
```

2.2 To what IP address is the DNS query message sent? Use ipconfig to determine the IP address of your local DNS server. Are these two IP addresses the same?

Local DNS Server IP: 8.8.8.8
Query DNS Server IP: 8.8.8.8
Yes, both are same.

```
Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   Description . . . . . . . . . . . : Qualcomm Atheros QCA61x4A Wireless Network Adapter
   Physical Address. . . . . . . . . : 98-22-EF-77-45-F5
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes
   Link-local IPv6 Address . . . . . : fe80::4097:68f1:ae3d:b297%12(Preferred)
   IPv4 Address. . . . . . . . . . . : 10.7.69.212(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.254.0
   Lease Obtained. . . . . . . . . . : Monday, October 14, 2019 11:27:24 AM
   Lease Expires . . . . . . . . . . : Monday, October 14, 2019 1:28:29 PM
   Default Gateway . . . . . . . . . : 10.7.68.1
   DHCP Server . . . . . . . . . . . : 10.7.68.1
   DHCPv6 IAID . . . . . . . . . . . : 160965359
   DHCPv6 Client DUID. . . . . . . . : 00-01-00-01-24-7C-F2-2C-98-29-A6-47-A7-E4
   DNS Servers . . . . . . . . . . . : 8.8.8.8
                                       9.9.9.9
   NetBIOS over Tcpip. . . . . . . . : Enabled
```

No.	Time	Source	Destination	Protocol	Length	Info
65	2019-10-14 07:35:23.287965	10.7.69.212	8.8.8.8	DNS	79	Standard query 0xd7f8 A www.cse.unsw.edu.au

2.3 *What* is contained in the flag field of your DNS request and response? Explain each set bit.

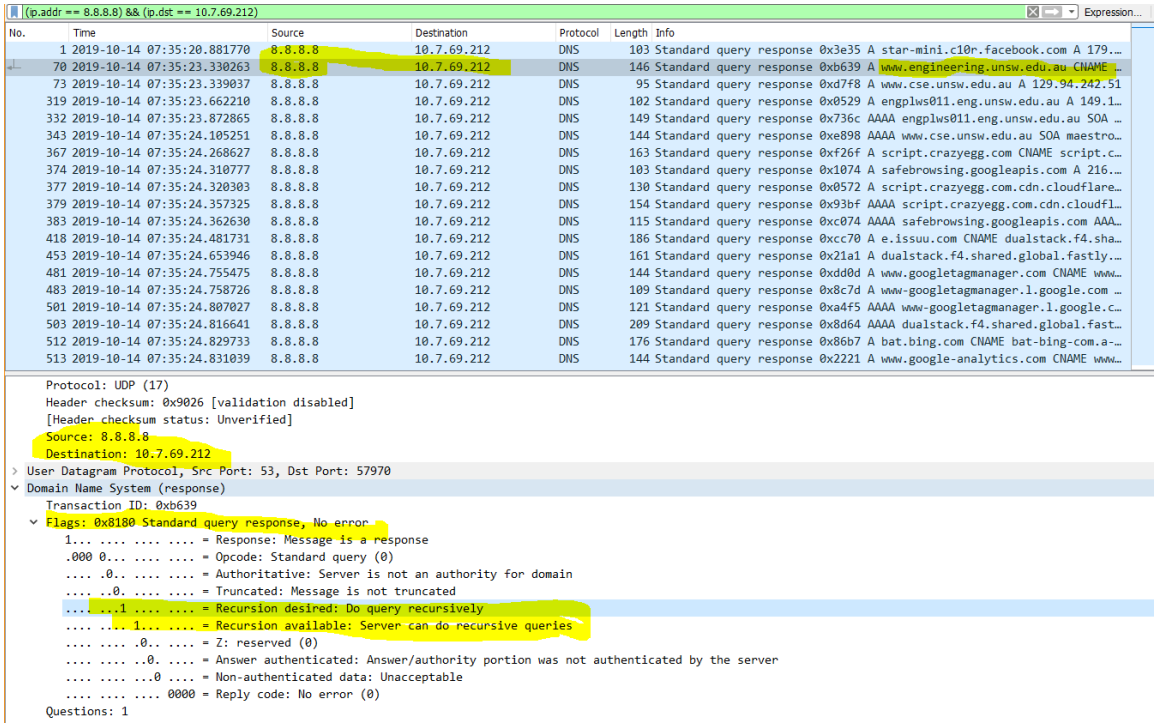
REQUEST:

Query Used: (ip.addr == 10.7.69.212) && (ip.dst == 8.8.8.8)

```
    [Header checksum status: Unverified]
    Source: 10.7.69.212
    Destination: 8.8.8.8
> User Datagram Protocol, Src Port: 56955, Dst Port: 53
v Domain Name System (query)
    Transaction ID: 0xd7f8
  v Flags: 0x0100 Standard query
      0... .... .... .... = Response: Message is a query
      .000 0... .... .... = Opcode: Standard query (0)
      .... ..0. .... .... = Truncated: Message is not truncated
      .... ...1 .... .... = Recursion desired: Do query recursively
      .... .... .0.. .... = Z: reserved (0)
      .... .... ...0 .... = Non-authenticated data: Unacceptable
    Questions: 1
```

RESPONSE:

Query Used: (ip.addr == 8.8.8.8) && (ip.dst == 10.7.69.212)

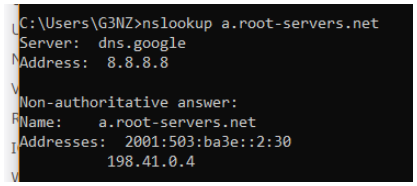


Flag Bit Explanation:

- 0th Bit: Indicates if the message is Request (0) or Response(1)
- 1-4th Bit: Query opcode
- 5th Bit: Indicates if the server is Authoritative (1) or Non-authoritative (0). It is meaningless in request header. In our case, we are receiving the packets from a Non-authoritative server with IP: 8.8.8.8, owned by Google.
- 6th Bit: Indicates if the message is truncated (1) or not truncated (0).
- 7th Bit: Indicates that the Recursion is desired or not for DNS query processing. In our case, it was sent in Request object that recursion is desired. Also, in Response object, query processing through recursion is desired,
- 8th Bit: Indicates if the Recursion is available for the query processing or not. In our case, Response flag object indicates that it is available.
- 9th Bit: 0 if reserved and 1 if not.
- 10th Bit: Answer authenticated if 1, otherwise it is unauthenticated.
- 11th Bit: Specifies whether to accept unauthenticated data or not. 0 means, unauthenticated data should be unacceptable.

2.4 Find out the IP address of at-least one ROOT name server.

IP address of a.root-servers.net: 198.41.0.4



Tried to get to the root-server through nameservers of www.cse.unsw.edu.au, but query was refused by its hosting server beethoven.orchestra.cse.unsw.edu.au with IP 129.94.242.33.

```
C:\Users\G3NZ>nslookup -type=ns www.cse.unsw.edu.au
Server:  dns.google
Address:  8.8.8.8

cse.unsw.edu.au
        primary name server = maestro.orchestra.cse.unsw.edu.au
        responsible mail addr = hostmaster.cse.unsw.edu.au
        serial  = 2019101401
        refresh = 2000 (33 mins 20 secs)
        retry   = 300 (5 mins)
        expire  = 1209600 (14 days)
        default TTL = 900 (15 mins)

C:\Users\G3NZ>nslookup -type=ns  maestro.orchestra.cse.unsw.edu.au
Server:  dns.google
Address:  8.8.8.8

orchestra.cse.unsw.edu.au
        primary name server = maestro.orchestra.cse.unsw.edu.au
        responsible mail addr = hostmaster.cse.unsw.edu.au
        serial  = 2019101401
        refresh = 2000 (33 mins 20 secs)
        retry   = 300 (5 mins)
        expire  = 1209600 (14 days)
        default TTL = 900 (15 mins)
```

```
C:\Users\G3NZ>nslookup -type=ns edu 129.94.242.33
DNS request timed out.
    timeout was 2 seconds.
Server:  UnKnown
Address:  129.94.242.33

*** UnKnown can't find edu: Query refused
```

2.5 Find out the fully qualified domain name for IP address 149.171.96.2

FQDN of given IP: oracle.unsw.edu.au

```
C:\Users\G3NZ>nslookup  149.171.96.2
Server:  dns.google
Address:  8.8.8.8

Name:    oracle.unsw.edu.au
Address:  149.171.96.2
```

Conclusion:

Website URLs are resolved through a DNS server hierarchy. We can inspect the DNS servers used for a specific URL and its IP address through various tools to get familiar with the traceroute of a URL.

For DNS packets, we can snif the packets used for communication with the DNS servers.

Computer Networks – Lab 6

OSAMA MOHAMMED AFZAL	237529
MUHAMMAD HASNAIN NAEEM	212728

Running and Compilation:

To run it, open up two shell windows (either both on the same machine, or on different machines). In one, run it like this:

./a.out receive filename_to_create.txt 9999

Then, in the other, run it like this:

./a.out send filename_to_send.txt 127.0.0.1:9999

(if the receiver is on a different machine than the sender, replace **127.0.0.1** with the IP address of the receiver machine).

If all is successful, you will see output in both windows indicating data is being transferred, and at the end, both programs will exit, and there will be a file **filename_to_create.txt** on the receiving machine that is identical to **filename_to_send.txt** is on the sending machine.

How to read and Audio /video File using FFmpeg in C

Audio:

https://batchloaf.wordpress.com/2017/02/10/a-simple-way-to-read-and-write-audio-and-video-files-in-c-using-ffmpeg/

Video:

https://batchloaf.wordpress.com/2017/02/12/a-simple-way-to-read-and-write-audio-and-video-files-in-c-using-ffmpeg-part-2-video/

Conclusion & Takeaways:

In this lab, we learned and demonstrated how socket programming can be used to send files through TCP/UDP protocols. Moreover, we can utilize the FFMPEG library to perform operations on each frame of video file being sent. Plus, we can perform operations on the audio such as changing number of channels, sampling rate etc.
To summarize, we looked into performing real-time operations on files whilst sending them from server to client.

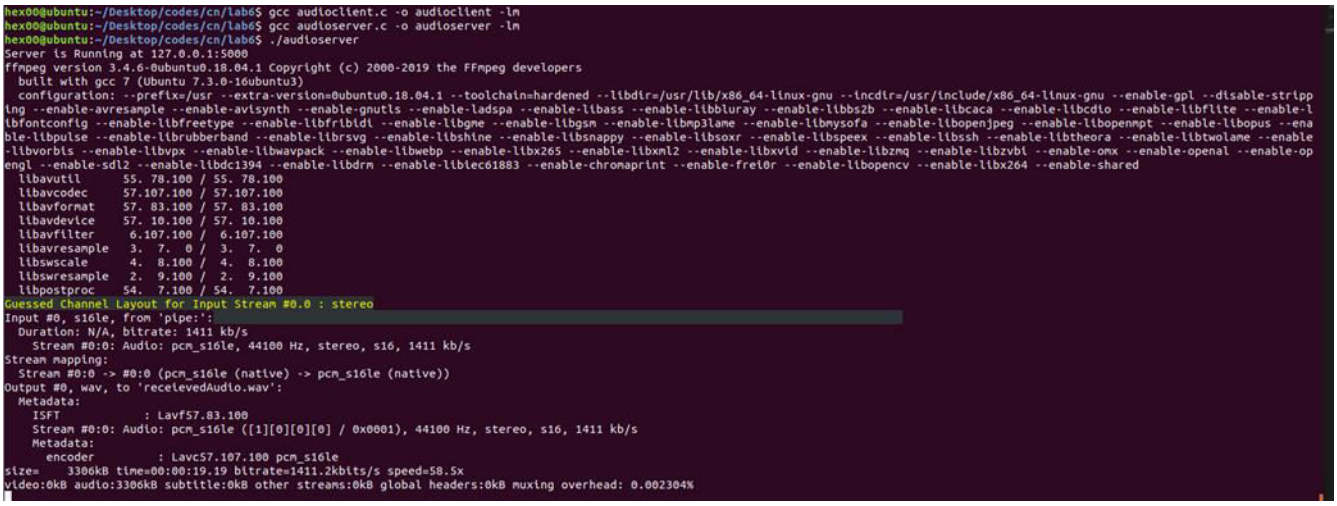
Task 1

Audioserver.c

```
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


int main(void)
{
  // variables decleration
        int SenderSock = 0; //socket descriptor
        int connectionSock = 0;
```

```
        int bytesRead = 0;
        struct sockaddr_in serv_addr;
        int16_t bufferread[256]; //for reading file content

        // socket creation
        SenderSock = socket(AF_INET, SOCK_STREAM, 0);
        printf("Server is Running at 127.0.0.1:5000\n");
        memset(&serv_addr, '0', sizeof(serv_addr)); //filling block of memory with 0
        memset(bufferread, '0', sizeof(bufferread)); //filling 256 memory locations with 0
    serv_addr.sin_port = htons(5000);
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

        // binding to local host
        bind(SenderSock, (struct sockaddr*)&serv_addr,sizeof(serv_addr));

        // waiting for the sender
        if(listen(SenderSock, 10) == -1)
        {
        printf("No response from sender\n");
        }

        // sending the file
        while(1)
        {
```

```
        connectionSock = accept(SenderSock, (struct sockaddr*)NULL ,NULL);
//connection established between sender and receiver

        FILE *audioFile; // file pointer to our file

        // reading 2 audio channels.

        audioFile = popen("ffmpeg -y -f s16le -ar 44100 -ac 2 -i - receievedAudio.wav",
"w");

        // reading file from buffer and writing to a new file

        while((bytesRead = read(connectionSock, bufferread, 256)) > 0) //reading
segments and checking if there are any segments

  {

        fwrite(bufferread, 1, bytesRead, audioFile); //writing segments to a file
recievedAudio.wav

        if (bytesRead < 256) // because only last segment would be less than 256

        {

        printf("End of file\n");

        break;

        }

  }



        fclose(audioFile); //closing file

        close(connectionSock); //closing socket

        }

        return 0;

}
```



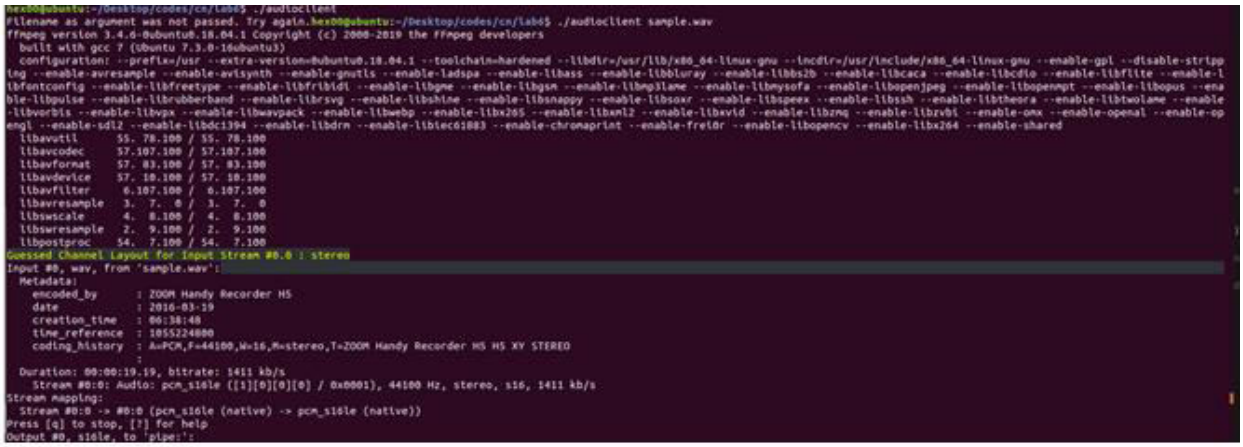
Audioclient.c

```
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


int main(int argc, char** argv)
{
   if(argc < 2){
```

```
        printf("Filename as argument was not passed. Try again.");
        return -1;
}
        //variable decleration
int receiverSock = 0;
struct sockaddr_in serv_addr;
int16_t bufferread[256]={0}; //for reading file content

// socket creation
receiverSock = socket(AF_INET, SOCK_STREAM, 0);
serv_addr.sin_port = htons(5000);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

// initiating connection on socket
if(connect(receiverSock, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
{
        printf("Failed to connect to server \n"); //could not connect
        return 1;
}

// Open WAV file with FFmpeg and read raw samples via the pipe.
FILE *audioFile;
char init_command[] = "ffmpeg -i ";
char end_command[] = " -f s16le -ar 44100 -ac 2 -";
```

```
char command[100];
strcpy(command, init_command);
strcat(command, argv[1]);
strcat(command, end_command);
audioFile = popen(command, "r");

if(audioFile==NULL)
{
	printf("failed to open file \n"); //video file does not exists
	return 1;
}

while(1)
	{
	// reading from a video file
	int nread = fread(bufferread,1, 256, audioFile);  //segments upto length of 256
	printf("Bytes read %d \n", nread);    //reading file in segments

	// creating segments and sending them
	write(receiverSock, bufferread, nread); //writing bytes to buffer
	if(nread > 0)
	{
		printf("Sending file to receiver \n");

	}
```

```
        if (nread < 256) // because only last segment would be less than 256
        {
                if (feof(audioFile))//end of file
                printf("End of file\n");
                break;
        }
        }

  pclose(audioFile);
        close(receiverSock); //closing socket
        return 0;
}
```



Design a GNU C program in Linux environment, in which server transfer Audio file to Client .

Task2:

Design a GNU C program in Linux environment, in which server transfer video file to Client.

Task 2

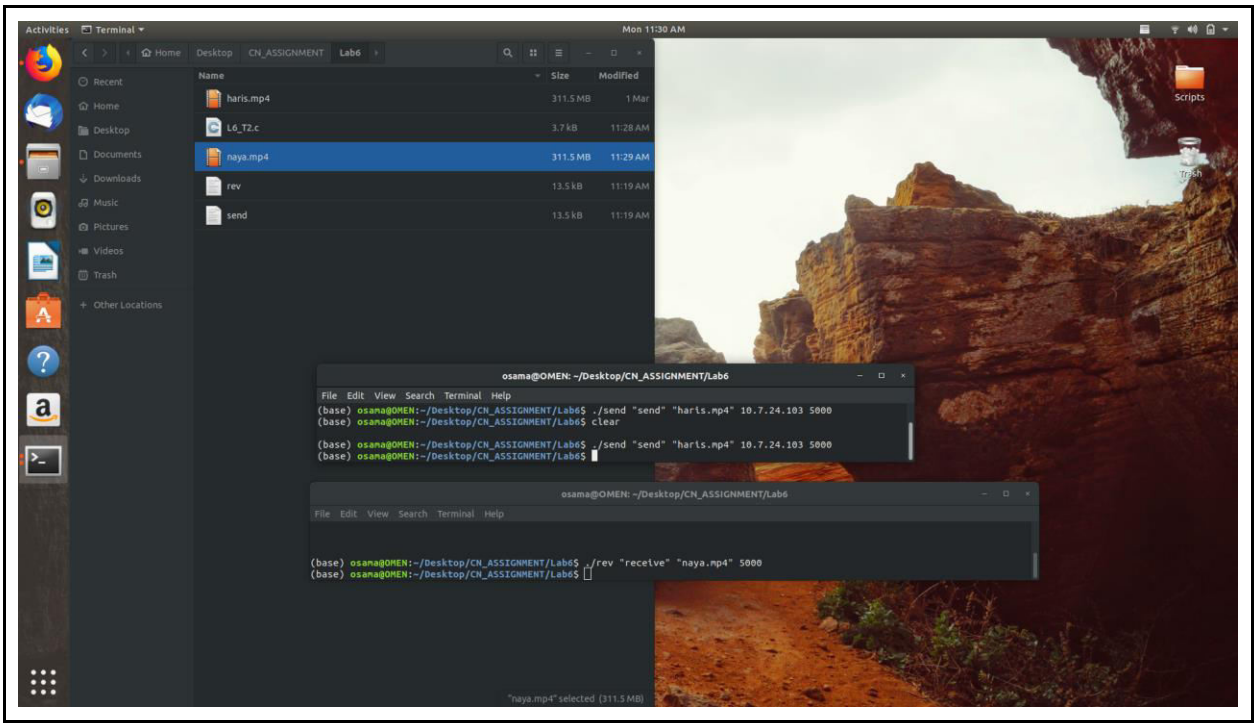
```
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include<fcntl.h>
#include <netinet/in.h>
#include <string.h>

#define SEND "send"
#define RECEIVE "receive"

int isSend;

int validate(int argc, char *argv[]){

    char* method, filename, IP_ADDR;
    int PORT;
    int isValid = 0;

    if (argc < 4){
        printf("\nInsufficient Arguments");
        exit(0);
    }
    else{
        method = argv[1];
        if(strcmp(method,SEND) == 0  ){
            isSend  = 1;
            isValid = 1;
        }
        else if(strcmp(method,RECEIVE) == 0){
```

```
            isSend = 0;
            isValid = 1;
        }
        else{
            isSend = -1;
            printf("Method specified does not exist");
            exit(-1);
        }
        isValid = 1;
    }
    return isValid;
}

void sendC(char *args[]){
    int sock, bytes_rec;
    struct sockaddr_in server;
    char buffer[5000] = {0};
    double buff[1] = {0};
    int addlen = sizeof(server), opt = 1;

    FILE *fp;
    fp = fopen(args[2],"rb");

    if (fp == NULL) {
        perror("fopen()");
        exit(EXIT_FAILURE);
    }

    //Creating a socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
                                                &opt, sizeof(opt)))
    {
        perror("setsockopt");
        exit(EXIT_FAILURE);
```

```
    }

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(atoi(args[4]));

    //binding Socket to the port
    if(bind(sock,(struct sockaddr*)&server,sizeof(server))<0){

        perror("Failed to bind");
        exit(EXIT_FAILURE);
    }
    if(listen(sock,10)<0){
        perror("listen");
        exit(EXIT_FAILURE);
    }

    if((sock = accept(sock,(struct
sockaddr*)&server,(socklen_t*)&addlen)) <0){
        perror("accept");
        exit(EXIT_FAILURE);
    }


    while(1){
        double bytes_read = fread(buffer,1,sizeof(buffer),fp);
        if (bytes_read == 0)
            break;

        void *p = buffer;

        while(bytes_read > 0){

            int bytes_written = send(sock,p,bytes_read,0);
            bytes_read -= bytes_written;
            p += bytes_written;
        }
    }
```

```
}

int receive(char *args[]){
    FILE *fp;
    fp = fopen(args[2],"wb");
    if (fp == NULL) {
        perror("fopen()");
        exit(EXIT_FAILURE);
    }

    int sock = 0, n;
    struct sockaddr_in serv_addr;
    char buffer[5000] = {0};
    char* pbuffer = buffer;
    double size_rec = 0.0;


    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(atoi(args[3]));

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, "10.7.24.193", &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
< 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }
```

```
    // will remain open until the server terminates the connection
    while ((n = recv(sock, pbuffer, 5000, 0)) > 0) {

        fwrite(buffer,n,1,fp);
        bzero(buffer,sizeof(buffer));
    }
    fclose(fp);
    close(sock);
}

int main(int argc, char *argv[]){


    if(validate(argc,argv)){

        if(isSend){
            sendC(argv);
        }
        else{
            receive(argv);
        }
    }
    else{
        exit(0);
    }

}
```



README BEFORE RUNNING:

- File takes 4 arguments
  - Method (send or receive) : STRING
  - Filename + extension : STRING
  - IP address : INT (only if you are server else dont give)
  - PORT : INT
- The above arguments are space seperated.

National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computer Science

EE353: Computer Networks

Name: M. Hasnain Naeem

Reg #: 212728

Class: BSCS-7B

Lab 10: Socket Programming (Remote Procedure Calls)

Date: 28th Oct, 2019

Lab Engineer: Engr. Kaleem Ullah

Instructor: Dr. Muhammad Zeeshan

Lab 10: Socket Programming for Linux

Introduction

Remote Procedure Calls

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. RPC makes coding easy in distributed systems' applications since the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. That is, the programs are coded as if the procedure call were a normal (local) procedure call, without explicitly coding the details for the remote interaction.

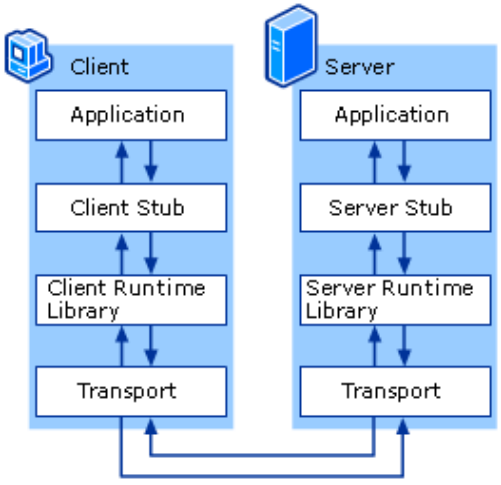
Message passing in RPC

The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.

- The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
- The client's local operating system sends the message from the client machine to the server machine.
- The local operating system on the server machine passes the incoming packets to the server stub.
- The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.
- Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

Steps involved in developing a RPC application developing:

- Specify the protocol for client server communication
- Develop the client program
- Develop the server program



Step1 : Specify the protocol for client server communication

An interface description language (IDL) to let various platforms call the RPC. The IDL files can then be used to generate code to interface between the client and servers. So, in IDL file, we define the program structure like below. Save this IDL file with .x extension.

calculate.x

```
struct inputs{
float num1;
float num2;
char operator;
};

program CALCULATE_PROG{
version CALCULATE_VER{
float ADD(inputs)=1;
float SUB(inputs)=2;
float MUL(inputs)=3;
float DIV(inputs)=4;
}=1;
}=0x2fffffff;
```

inputs – Name of the data structure. Through this data structure the parameters are send to the server for computations.

CALCULATE_PROG — Name of the program

CALCULATE_VER — Name of the program version

ADD(inputs) — This is a remote method which is calling locally and the parameters are passed to that remote method through the inputs structure which contains 2 operands(2 numbers) and an operator.

A remote procedure is uniquely identified by the triple: (**program number, version number, procedure number)**. Therefore, you need to give any numbers you like to the program, to the version and to the procedures as in the above sample IDL file.

Then compile your IDL file using **rpcgen** protocol compiler. The protocol compiler reads the definition of the IDL file and automatically generates client and server stubs. First you need to check whether the **rpcbind** has been installed in your machine. Type this command to check it and if the execution of the command gives you a long list , then your machine has already installed **rpcbind** package and you can work on with the **rpc** commands.

Try this command if it is already installed

$ rpcinfo

Otherwise, you have to install the **rpcbind** package by following below command and check whether it has been actually installed by typing the $ rpcinfo command again.

$ sudo apt-get install rpcbind

Now your PC is ready to execute rpc commands. The next step is to compile your IDL file using rpcgen command.

$ rpcgen -a -C calculate.x

This command generated 7 additional files. Keep in mind that, you can edit only the 'calculate_client.c' and 'calculate_server.c' files among from the created files according to your need. Don't do changes to other files.

- calculate_client.c —-> client program (editable file)
- **calculate_server.c** —> server program (editable file)
- **calculate_cln.c** —> client stub
- **calculate_svc.c** —> server stub
- **calculate_xdr.c** —> XDR(External Data Representation) filters
- **calculate.h** —> header file needed for any XDR filters
- **Makefile.calculate** —> compile all the source files by using this file

Step 2 – Develop the client program

edit the code in the **calculate_client.c** file like the way it accepts the client's inputs from the keyboard and outputs the remote procedures (add,sub,mul,div) invoking results to the client .

calculate_client.c

```
#include "calculate.h"

float calculate_prog_1(char *host,float n1,float n2,char opr,CLIENT
*clnt)
{
float *result_1;
inputs add_1_arg;
float *result_2;
inputs sub_1_arg;
float *result_3;
inputs mul_1_arg;
float *result_4;
inputs div_1_arg;

if(opr=='+'){
add_1_arg.num1=n1;
add_1_arg.num2=n2;
add_1_arg.operator=opr;
result_1 = add_1(&add_1_arg, clnt);
if (result_1 == (float *) NULL) {
clnt_perror (clnt, "call failed");
}
return *result_1;
}
else if(opr=='-'){
sub_1_arg.num1=n1;
sub_1_arg.num2=n2;
sub_1_arg.operator=opr;
result_2 = sub_1(&sub_1_arg, clnt);
if (result_2 == (float *) NULL) {
clnt_perror (clnt, "call failed");
```

```
}
return *result_2;
}
else if(opr=='*'){
mul_1_arg.num1=n1;
mul_1_arg.num2=n2;
mul_1_arg.operator=opr;
result_3 = mul_1(&mul_1_arg, clnt);
if (result_3 == (float *) NULL) {
clnt_perror (clnt, "call failed");
}
return *result_3;
}
else if(opr=='/'){
div_1_arg.num1=n1;
div_1_arg.num2=n2;
div_1_arg.operator=opr;
if(n2 == 0){
printf("Division by zero is not valid.\n");
exit(0);
}else{
result_4 = div_1(&div_1_arg, clnt);
if (result_4 == (float *) NULL) {
clnt_perror (clnt, "call failed");
}
return *result_4;
}
}
}

int main (int argc, char *argv[])
{
char *host;
float a,b;
char op;
CLIENT *clnt;
if (argc < 2) {
printf ("usage: %s server_host\n", argv[0]);
exit (1);
}
printf("Welcome to Quick Cal!!!\n");
printf("+ for Addition\n- for Substraction\n* for Multiplication\n/
for Division\n");
printf("Enter number 1 :\n");
scanf("%f",&a);
printf("Enter number 2 :\n");
scanf("%f",&b);
printf("Enter the Operator :\n");
scanf("%s",&op);
host = argv[1];
clnt = clnt_create (host, CALCULATE_PROG, CALCULATE_VER, "udp");
if (clnt == NULL) {
clnt_pcreateerror (host);
```

```
exit (1);
}
printf("The Answer = %f\n",calculate_prog_1 (host,a,b,op,clnt));
clnt_destroy (clnt);
exit (0);
}
```

Step 3 – Develop the server program

Similarly you can edit the **calculate_server.c** file while adding your own codes for the previously defined remote methods(add,sub,mul,div).

calculate_server.c

```
#include "calculate.h"

float * add_1_svc(inputs *argp, struct svc_req *rqstp)
{
static float result;
result = argp->num1+argp->num2;
printf("Got Request : Adding %f and %f\n",argp->num1,argp->num2);
printf("Sent Response : %f\n",result);
return &result;
}

float * sub_1_svc(inputs *argp, struct svc_req *rqstp)
{
static float result;
result = argp->num1-argp->num2;
printf("Got Request : substituting %f from %f\n",argp->num2,argp-
>num1);
printf("Sent Response : %f\n",result);
return &result;
}

float * mul_1_svc(inputs *argp, struct svc_req *rqstp)
{
static float result;
result = argp->num1*argp->num2;
printf("Got Request : Multiplying %f by %f\n",argp->num1,argp-
>num2);
printf("Sent Response : %f\n",result);
return &result;
}

float * div_1_svc(inputs *argp, struct svc_req *rqstp)
{
static float result;
result = argp->num1/argp->num2;
printf("Got Request : Dividing %f by %f\n",argp->num1,argp->num2);
printf("Sent Response : %f\n",result);
return &result;
}
```

After editing the client and the server code, you need to compile the files. As usual, it's mandatory to compile the files if u made any changes. Otherwise those changes won't apply at the execution time. Use this command for compilation of files.

$ make -f Makefile.calculate

This command will generate additional 2 executable files of the client and the server. (calculate_client and calculate_server)

To up the server,run the executable file of the server(calculate_server) which was created at the compile time. To do that, type the below command in a terminal.
$ sudo ./calculate_server

Take a new terminal. Then run the executable file of the client(calculate_client) which was created at the compile time.

$ sudo ./calculate_client localhost

Now the client program is ready to accept keyboard inputs from the client. So, you can enter 2 numbers and an operator which gives the output to the client by invoking the remote procedures at the server side which the client has requested.

Example of Passing Command line Argument in C

command line arguments are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly

```
#include <stdio.h>
int main( int argc, char *argv[] )  {
  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
```

```
  }
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
./a.out testing
```

The argument supplied is testing

When the above code is compiled and executed with a two arguments, it produces the following result.

$./a.out testing1 testing2

Too many arguments supplied.

When the above code is compiled and executed without passing any argument, it produces the following result.

$./a.out

One argument expected

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes

```
#include <stdio.h>

int main( int argc, char *argv[] )  {
  printf("Program name %s\n", argv[0]);
  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
  }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

$./a.out "testing1 testing2"

Program name ./a.out

The argument supplied is testing1 testing2

Task: Design and code a sender/receiver program, client side should be able to get command from user and send that command to server over a TCP connection. Server must receive, execute that command and reply results to sender and results must also be displayed on client side screen.It is required to code in GNU C on Linux operating system.

Specifications:

Client will get command input from user and send it to server
Server will execute the command remotely on server machine and will reply back the results.
Client will display the execution results to the user on client's screen
Server should be able to receive and run following commands.

Objectives:
- Remote execution of commands sent by client on the server.
- Communication through the TCP protocol between server and client.
- Returning results from the client to server.

Client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define PORT 5000
#define BUFFSIZE 128
#define RESULTSLEN 100000

int main(int argc, char *argv[])
{
   struct sockaddr_in server_details;
   struct hostent *he;
```

```
int socket_fd;
int bytes_read = 0;
long long total_bytes_read = 0;

char results[RESULTSLEN] = {0};
long long message_length = 0;
char command_buffer[BUFFSIZE] = {0};
char results_buffer[BUFFSIZE] = {0};
int command_len = 0;
char *server_address_str = NULL; // sender address

// used to store and read the results
FILE *results_file;
char readC;

if (argc < 2)
{
   // assume the server address
   server_address_str = "127.0.01";
}
else if (argc >= 2)
{
   server_address_str = argv[1];
}

if ((he = gethostbyname(server_address_str))==NULL)
{
   fprintf(stderr, "ERROR: Could not get the hostname.\n");
   exit(1);
}

if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0))== -1)
{
   fprintf(stderr, "Error: Coould not create the socket.\n");
   exit(1);
}

// configuring the server details
memset(&server_details, 0, sizeof(server_details));
server_details.sin_family = AF_INET;
server_details.sin_port = htons(PORT);
server_details.sin_addr = *((struct in_addr *)he->h_addr);

if (connect(socket_fd, (struct sockaddr *)&server_details, sizeof(struct sockaddr))<0)
{
   fprintf(stderr, "ERROR: Connection Failure.\n");
```

```
    exit(1);
  }

  // To send multiple commands
  int i = 0; // to iterate over the input
  while(1) {
    // comamand input
    printf("Enter comamnd to be executed on the server (-QUIT to stop):\n");
    fgets(command_buffer, BUFFSIZE - 1, stdin);
    command_len = strlen(command_buffer);

    // removing the newline character from input
    for(i = 0; i < command_len; i++)
    {
      if(command_buffer[i] == '\n')
      {
        command_buffer[i] = '\0';
        break;
      }
    }

    // function compares both strings to given length (returns 0 if equal)
    if (!strncmp(command_buffer, "-QUIT", i+1))
      break; // exit from program

    printf("Sending '%s' command to server.\n", command_buffer);
    // sending commmand
    if ((send(socket_fd, command_buffer, strlen(command_buffer), 0))== -1)
    {
      // deal with erros
      fprintf(stderr, "ERROR: Failed to send the command.\n");
      close(socket_fd);
      exit(1);
    }

    else
    {
      total_bytes_read = 0;
      bytes_read = 0;
      message_length = 0;

      recv(socket_fd, &message_length, 64, 0);
      message_length = ntohl(message_length);
      printf("Message Length: %lld\n", message_length);
      // keep on recieving the results until the recieved bytes < BUFFSIZE
      results_file = fopen("recieved_results.txt", "w");
```

```
        if (message_length != 0)
        {
          while(1)
          {
            bytes_read = recv(socket_fd, results_buffer, sizeof(results_buffer),0);
            printf("%d bytes read.\n", bytes_read);

            if ( bytes_read <= 0 )
            {
              printf("ERROR: Could not recieve results from the server. It can be due to closed connectio
n or some other reason.\n");
              //Break from the While
              break;
            }
            // break when the exiting message is recieved
            if (total_bytes_read + bytes_read >= message_length) {
              fwrite(results_buffer, 1, message_length - total_bytes_read, results_file);
              printf("Total bytes read = %lld\n", total_bytes_read+bytes_read);
              break;
            }
            total_bytes_read += bytes_read;
            // save results
            fwrite(results_buffer, 1, strlen(results_buffer), results_file);
            printf("%d bytes written into the file.\n", bytes_read);

          }
          // empty the buffer
          memset(results_buffer, 0, BUFFSIZE * (sizeof results_buffer[0]) ); // to prevent saving previousl
y stored data if read data < BUFFSIZE
          fclose(results_file);

          printf("\nResults recieved from the server:\n");
          printf("_____\n");
          // read results to console
          results_file = fopen("recieved_results.txt", "r");
          while((readC=fgetc(results_file))!=EOF){ // printing character by character
            printf("%c", readC);              // because file can be large
          }
          fclose(results_file);
        }
        else {
          printf("Command executed.");
        }
      }
      printf("\n_____\n\n");
    } // while loop ends, user typed "-QUIT"
```

```
    close(socket_fd);

}//End of main
```

Server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 5000
#define BACKLOG 10
#define RESULTSLEN 100000
#define BUFFSIZE 128
#define PATHSIZE 100

int main()
{
    // networking vars
    struct sockaddr_in server_info;
    struct sockaddr_in client_info;
    int status, socket_fd, client_fd, num;
    socklen_t size;

    char results[RESULTSLEN] = {0};
    char command_buffer[BUFFSIZE] = {0};
    char results_buffer[BUFFSIZE] = {0};
    int reuse_flag = 1; // true
    long long message_length = 0;

    // used to run commands and store results
    FILE *fp;
    FILE *results_file;                    // file pointer to temp results file

    if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0))== -1) {
        fprintf(stderr, "ERROR: Socket creation failure.\n");
        exit(1);
    }
```

```
  if (setsockopt(socket_fd, SOL_SOCKET, SO_REUSEADDR, &reuse_flag, sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
  }

  memset(&server_info, 0, sizeof(server_info));
  memset(&client_info, 0, sizeof(client_info));

  server_info.sin_family = AF_INET;
  server_info.sin_port = htons(PORT);
  server_info.sin_addr.s_addr = INADDR_ANY;

  if ((bind(socket_fd, (struct sockaddr *)&server_info, sizeof(struct sockaddr )))== -
1)   { //sizeof(struct sockaddr)
    fprintf(stderr, "ERROR: Binding Failure.\n");
    exit(1);
  }

  if ((listen(socket_fd, BACKLOG))== -1){
    fprintf(stderr, "ERROR: Listening Failure.\n");
    exit(1);
  }

  while(1) {

    size = sizeof(struct sockaddr_in);

    if ((client_fd = accept(socket_fd, (struct sockaddr *)&client_info, &size))==-1 ) {
      perror("accept");
      exit(1);
    }
    printf("Connection established with a client: %s.\n", inet_ntoa(client_info.sin_addr));

    while(1) { // Keep on dealing with the established connection
      message_length = 0;
      if ((num = recv(client_fd, command_buffer, BUFFSIZE, 0))== -1) {
            perror("recv");
            exit(1);
      }
      else if (num == 0) {
        printf("Connection closed\n");
        // Proceed to deal with other client
        break;
      }
      command_buffer[num] = '\0';
```

```
        printf("Command Received: %s\n", command_buffer);
        /* Begin Execution of Command and Store Results */
        // command contains the command string (a character array)
        // execute the command
        fp = popen(command_buffer, "r");

        // file to write results
        results_file = fopen("results.txt", "w");
        printf("Results are being sent to the client.\n");

        // store the results in buffer and then into the file
        while (fgets(results_buffer, BUFFSIZE, fp) != NULL) {
          fwrite(results_buffer, 1, strlen(results_buffer), results_file);
        } // data transferred
        fseek(results_file, 0L, SEEK_END);
        message_length = ftell(results_file);
        fclose(results_file);
        int converted_number = htonl(message_length);
        write(client_fd, &converted_number, sizeof(converted_number));
        printf("\nMessage Length: %lld\n", message_length);

        results_file = fopen("results.txt", "r");
        // read results into a buffer from the file then send to client
        while (fgets(results_buffer, BUFFSIZE, results_file) != NULL) {
          if ((write(client_fd, results_buffer, strlen(results_buffer)))== -1)
          {
            // throw error if results could not be sent.
            fprintf(stderr, "ERROR: Failed to send the results to client.\n");
            close(client_fd);
            break;
          }
        } // data transferred
        printf("Results sent to the client.\n\n");
        pclose(fp);  // closing command file
        fclose(results_file); // closing results file
      } // While loop end, connection ended
      //Close Connection Socket
      close(client_fd);
    } // While loop end, server Exits
    // Flow never reaches here
    close(socket_fd);
    return 0;
} //End of main
```

1-- netstat command with one argument

```
hex00@hex00:~/Desktop/codes/cn/lab7$ ./client
Enter comamnd to be executed on the server (-QUIT to stop):
netstat
Sending 'netstat' command to server.
Message Length: 80684
128 bytes read.
128 bytes written into the file.
128 bytes read.
128 bytes written into the file.
128 bytes read.
128 bytes written into the file.
128 bytes read.
128 bytes written into the file.
128 bytes read.
128 bytes written into the file.
128 bytes read.
128 bytes written into the file.
128 bytes read.
128 bytes written into the file.
```

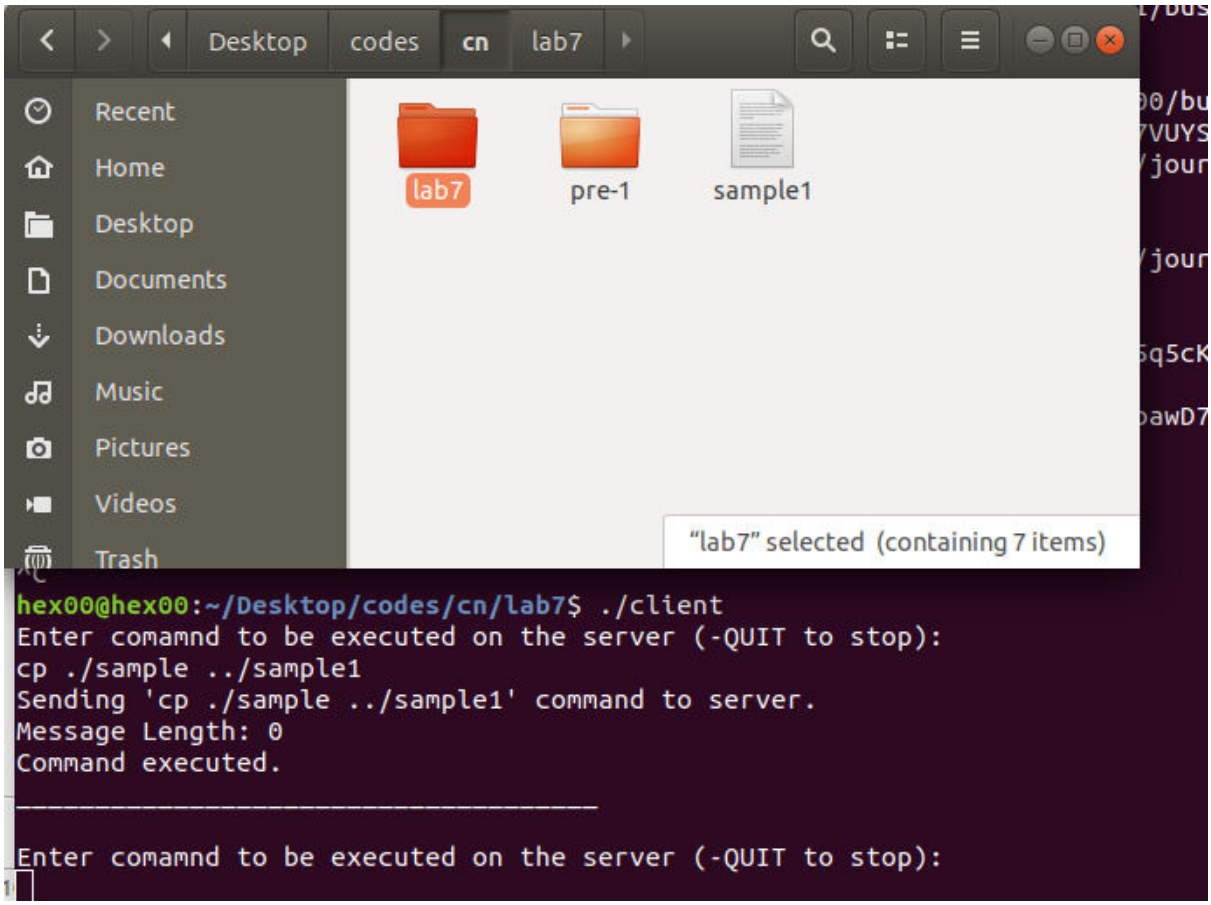
...

```
unix  3      [ ]         STREAM     CONNECTED     34129    @/tmp/dbus-O7VUYSWM
unix  3      [ ]         STREAM     CONNECTED     95603
unix  3      [ ]         STREAM     CONNECTED     95596
unix  3      [ ]         STREAM     CONNECTED     86960
unix  3      [ ]         STREAM     CONNECTED     36197    /run/user/1000/bus
unix  3      [ ]         STREAM     CONNECTED     33118
unix  3      [ ]         STREAM     CONNECTED     74876    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     32419    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     31938    @/tmp/.X11-unix/X1024
unix  3      [ ]         STREAM     CONNECTED     25293    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     19975    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     98082
unix  3      [ ]         STREAM     CONNECTED     71077
unix  3      [ ]         STREAM     CONNECTED     68206
unix  3      [ ]         STREAM     CONNECTED     26613
unix  3      [ ]         STREAM     CONNECTED     39024    @/tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     37646    @/tmp/dbus-a5q5cKKxNr
unix  3      [ ]         STREAM     CONNECTED     35478
unix  3      [ ]         STREAM     CONNECTED     30262    /run/user/121/bus
unix  3      [ ]         STREAM     CONNECTED     86157
unix  3      [ ]         STREAM     CONNECTED     42062
unix  3      [ ]         STREAM     CONNECTED     36202    /run/user/1000/bus
unix  3      [ ]         STREAM     CONNECTED     30250    @/tmp/dbus-O7VUYSWM
unix  3      [ ]         STREAM     CONNECTED     30795    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     43252
unix  3      [ ]         STREAM     CONNECTED     28844
unix  3      [ ]         STREAM     CONNECTED     25816    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     82864
unix  3      [ ]         STREAM     CONNECTED     32667
unix  3      [ ]         STREAM     CONNECTED     37971    @/tmp/dbus-a5q5cKKxNr
unix  3      [ ]         STREAM     CONNECTED     38209
unix  3      [ ]         STREAM     CONNECTED     30257    @/tmp/dbus-ioawD7XlkT

-----------------------------------

Enter comamnd to be executed on the server (-QUIT to stop):
```

Server Side:

```
hex00@hex00:~/Desktop/codes/cn/lab7$ ./server
Connection established with a client: 127.0.0.1.
Command Received: netstat
Results are being sent to the client.

Message Length: 80684
Results sent to the client.
```

2-- cp command with two arguments



3-- rm command with one argument

```
Enter comamnd to be executed on the server (-QUIT to stop):
rm ./sample
Sending 'rm ./sample' command to server.
Message Length: 0
Command executed.

-----------------------------------
```

4-- ls command without any argument

```
Enter comamnd to be executed on the server (-QUIT to stop):
ls
Sending 'ls' command to server.
Message Length: 72
72 bytes read.
Total bytes read = 72

Results recieved from the server:

-----------------------------------
client
client.c
recieved_results.txt
results.txt
sample
server
server.c


-----------------------------------
```

Conclusion:

In this lab, we learned and demonstrated the use of socket programming for Remote Procedure Calls. Conventionally, marshaling and stubs are used for RPC but other ways can also be used. For example, TCP can be used to transfer the commands in the form of packets from client to server. Server will execute the commands and return the results to the client.

Deliverables:

Objective, Source code with explanation, screenshots of output and a paragraph of Conclusion .

Department of Computer Science

EE353: Computer Networks

Submitted by: M. Hasnain Naeem

Registration No. : 212728

Class: BSCS-7B

Lab 8: *Analysis of FTP in Wireshark*

Date: 4th Nov, 2019

Lab Engineer: Engr. Kaleem Ullah

Instructor: Dr. Muhammad Zeeshan

Lab Title: *Analysis of FTP in Wireshark*

Objective of this lab:

In this lab, we will analyze the behavior of FTP in detail.

Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *You are supposed to provide the answers to the questions listed at the end of this document (substantiate your answers with screen shots of your Wireshark captures) and upload the completed report to your course's LMS site.*
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.*

Background:

FTP (File Transfer Protocol) is a simple application layer protocol (based on client/server network architecture). FTP is primarily used for transfer of files between the client and server.

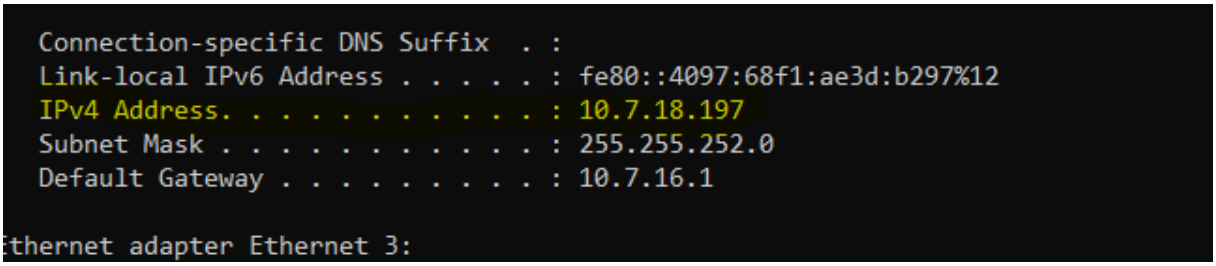
Pl go through the lecture slides to revise the following important concepts regarding FTP:

1. FTP uses out of band signaling
2. FTP uses two separate TCP connections, one for control and the other one for data
3. FTP control connection is persistent, while the data connection is non-persistent
4. FTP can work in either active or passive mode
5. There are several commands and responses available in FTP protocol

Objectives:

- *Getting familiar with FTP.*
- *Analyzing FTP packets using Wireshark.*

IP Address:



Steps for performing this lab:

There are 2 parts of this lab. A and B.

A. *Do the following:*

1. ***Start up the Wireshark software.***
2. ***Begin packet capture,*** *select the Capture pull down menu and select Options.*

3. ***Selecting the network interface on which packets would be captured:*** *You can use most of the default values in this window. The network interfaces (i.e., the physical connections) that your computer has to the network will be shown in the Interface pull down menu at the top of the Capture Options window. Click Start. Packet capture will now begin*

4. ***Open command prompt*** *and use command ftp [ftp.cdc.org](ftp.cdc.org)*

5. ***Use anonymous as username and guest as password***

6. ***Type 'exit'***

7. ***Stop the wireshark capture***

Questions:

1. *What other protocols does FTP require for its working?*

TCP transport layer protocol and IP network layer protocol.

2. *How many TCP connections are formed by FTP in this transaction? What is the source IP, source port No, destination IP and destination port No for the "Control connection" of FTP for this interaction?*

1 3-way-handshake connection is formed in this transaction to transfer control information. No connection is formed for the files.

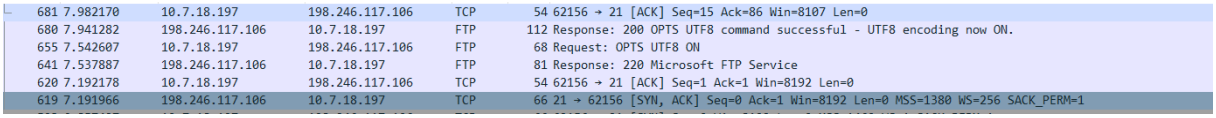
3. *What is the first response code and message received from the FTP server on the control connection?*

Response Code: 220
Message: Microsoft FTP service.

```
C:\Users\G3NZ>ftp ftp.cdc.gov
Connected to ftp.cdc.gov.
220 Microsoft FTP Service
```

4. *How many requests/responses are involved for authentication between the client and server? What response code and message does the server return when the authentication fails?*

6 connections are involved in authentication purpose.



Response code on authentication failure: 530
Message on authentication failure: User cannot log in.

```
Password:
530 User cannot log in.
Login failed.
ftp>
```

5. *What is the response code and message from server when the client sent 'QUIT'?*

Code: 221
Message: Goodbye.

```
ftp> QUIT
221 Goodbye.
```

B. *Do the following:*

1. ***Start up the Wireshark software.***
2. ***Begin packet capture,*** *select the Capture pull down menu and select Options.*
3. ***Selecting the network interface on which packets would be captured:*** *You can use most of the default values in this window. The network interfaces (i.e., the physical connections) that your computer has to the network will be shown in the Interface pull down menu at the top of the Capture Options window. Click Start. Packet capture will now begin*
4. ***Open winscp and change the file protocol to FTP. Enter*** *ftp.cdc.gov in the Host name.*
5. ***Use anonymous as username and guest as password***
6. ***Drag and drop 'Readme' file from the FTP server to your local drive.***

7. ***Drag and drop 'welcome.msg' file from the FTP server to your local drive.***

8. ***Type 'F10' to terminate the application.***

9. ***Stop the Wireshark capture.***

Questions:

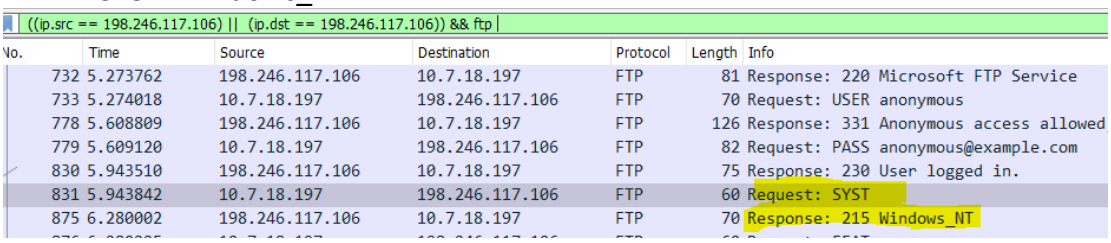
1. *Once the user is authenticated, the client asks for 'SYST' and 'FEAT'. What is being asked and what are the responses by the server?*

Purpose:

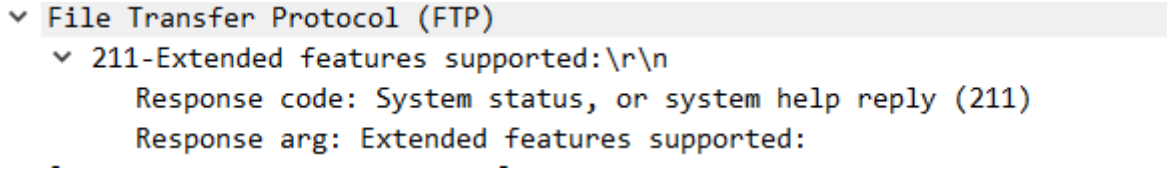
- ***SYST:*** *client is asking the server for its sytem type.*
- ***FEAT:*** *ask for the list of features implemented on server.*

Responses:

- ***SYST:*** *Windows_NT*

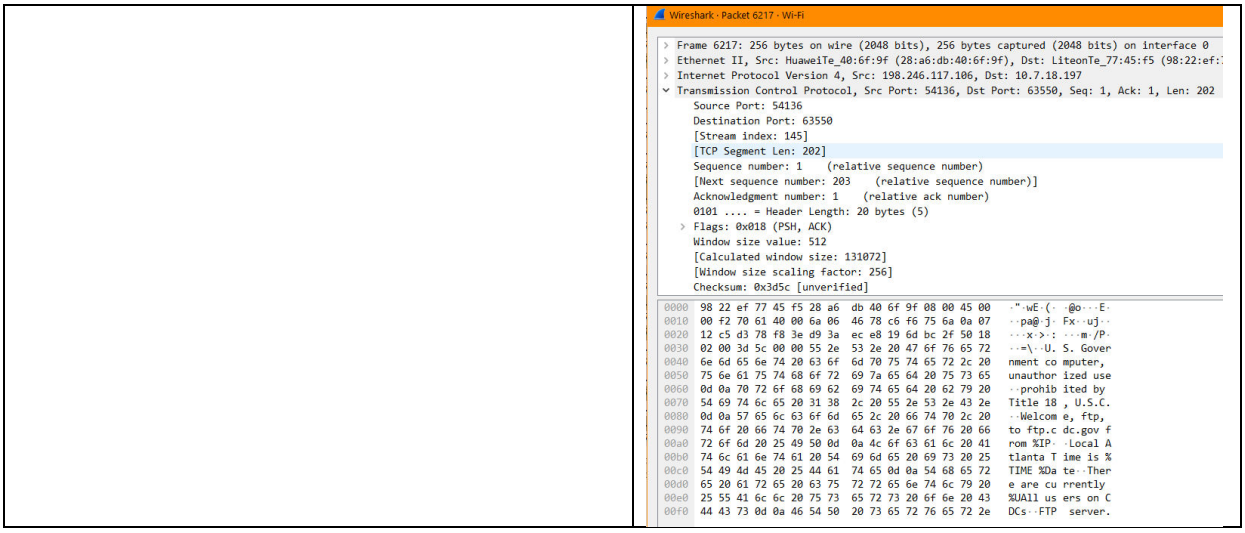


- ***FEAT:*** *Empty body.*



2. *How many TCP connections are formed by FTP in this transaction? What is the source IP, source port No, destination IP and destination port No for the "Control connection" and "Data connection" of FTP for this interaction?*

Control Connection	**Data Connection**
Source (Client): *10.7.18.197 : 63438* *Destination (Server):* *198.246.117.106 : 21* Transmission Control Protocol, Src Port: 63438, Dst Port: Source Port: 63438 Destination Port: 21 [Stream index: 16] [TCP Segment Len: 6] Sequence number: 45 (relative sequence number)	*Source (Server):* *198.246.117.106 : 54136* *Destination(Client):* *10.7.18.197 : 63550*



3. *Who does the 'passive open' for the data connection, client or server? Which mode the FTP is working in (ACTIVE, PASSIVE, EPSV, LPSV)? Why?*

Client does the passive open in FTP.

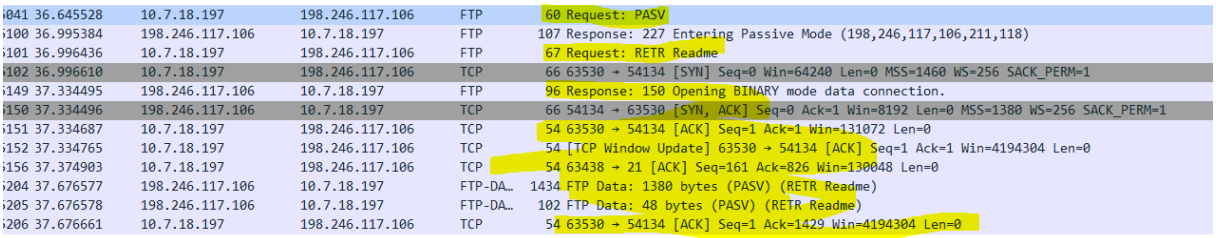
FTP is working in Passive mode because the client is requesting for the list of files. Moreover, it requests for files when we drag "readme" and "welcome.msg" files.

((ip.src == 198.246.117.106) || (ip.dst == 198.246.117.106))

Destination	Protocol	Length	Info
198.246.117.106	TCP	54	63438 → 21 [ACK] Seq=71 Ack=344 Win=
198.246.117.106	FTP	59	Request: PWD
10.7.18.197	FTP	85	Response: 257 "/" is current directo
198.246.117.106	FTP	61	Request: CWD /
10.7.18.197	FTP	83	Response: 250 CWD command successful
198.246.117.106	FTP	59	Request: PWD
10.7.18.197	FTP	85	Response: 257 "/" is current directo
198.246.117.106	FTP	62	Request: TYPE A
10.7.18.197	FTP	74	Response: 200 Type set to A.
198.246.117.106	FTP	60	Request: PASV

4. *What happens when you drag and drop 'Readme'? List the conversation between the client and server (request code/message and response code/message).*

- *Client requests to set the mode to "PASV"*
- *Server sets the mode to passive.*
- *Client asks requests for the "Readme" file using the RETR command.*
- *Server opens a binary mode data connection (TCP).*
- *Client acknowledges the connection.*
- *Then the data transfer begins.*
- *Client acknowledges the end of file.*



5041	36.645528	10.7.18.197	198.246.117.106	FTP	60 Request: PASV
5100	36.995384	198.246.117.106	10.7.18.197	FTP	107 Response: 227 Entering Passive Mode (198,246,117,106,211,118)
5101	36.996436	10.7.18.197	198.246.117.106	FTP	67 Request: RETR Readme
5102	36.996610	10.7.18.197	198.246.117.106	TCP	66 63530 → 54134 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
5149	37.334495	198.246.117.106	10.7.18.197	FTP	96 Response: 150 Opening BINARY mode data connection.
5150	37.334496	198.246.117.106	10.7.18.197	TCP	66 54134 → 63530 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1380 WS=256 SACK_PERM=1
5151	37.334687	10.7.18.197	198.246.117.106	TCP	54 63530 → 54134 [ACK] Seq=1 Ack=1 Win=131072 Len=0
5152	37.334765	10.7.18.197	198.246.117.106	TCP	54 [TCP Window Update] 63530 → 54134 [ACK] Seq=1 Ack=1 Win=4194304 Len=0
5156	37.374903	10.7.18.197	198.246.117.106	TCP	54 63438 → 21 [ACK] Seq=161 Ack=826 Win=130048 Len=0
5204	37.676577	198.246.117.106	10.7.18.197	FTP-DA…	1434 FTP Data: 1380 bytes (PASV) (RETR Readme)
5205	37.676578	198.246.117.106	10.7.18.197	FTP-DA…	102 FTP Data: 48 bytes (PASV) (RETR Readme)
5206	37.676661	10.7.18.197	198.246.117.106	TCP	54 63530 → 54134 [ACK] Seq=1 Ack=1429 Win=4194304 Len=0

5. *Which connection is closed when you type "Quit"?*

- *Both, control and data connections are closed when "Quit" is sent to the server. Hence, all the connections between server and client terminate, which ends the session.*

Conclusion:

FTP application layer protocol utilize the TCP transport layer protocol and IP network layer protocol to transfer files from one machine to another. We can analyze the FTP packets sent from the client to the server using Wireshark. We can look up the control information sent by client and server, plus, the data transferred between them.

Department of Computer Science

EE353: Computer Networks

Name: M. Hasnain Naeem

Reg #: 212728

Class: BSCS7AB

Lab 8: Introduction to the Network Layer

Date: 11 NOV 2019

Lab Engineer: Kaleem Ullah

Instructor: Dr. Muhammad Zeeshan

Name:
Regn No:
Current IP address:

Lab Title: Introduction to the Network Layer

1.0 Objective of this lab:

In this lab, we'll investigate the IP protocol, focusing on the IP datagram. We'll do so by analyzing a trace of IP datagrams sent and received by the execution of the traceroute program. We'll investigate the various fields in the IP datagram, and study IP fragmentation in detail.

2.0 Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *You are supposed to provide the answers to the questions listed in this document (attach screen shots) and upload the completed report to your course's LMS site.*
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work. Your submitted work should be written by yourself.*

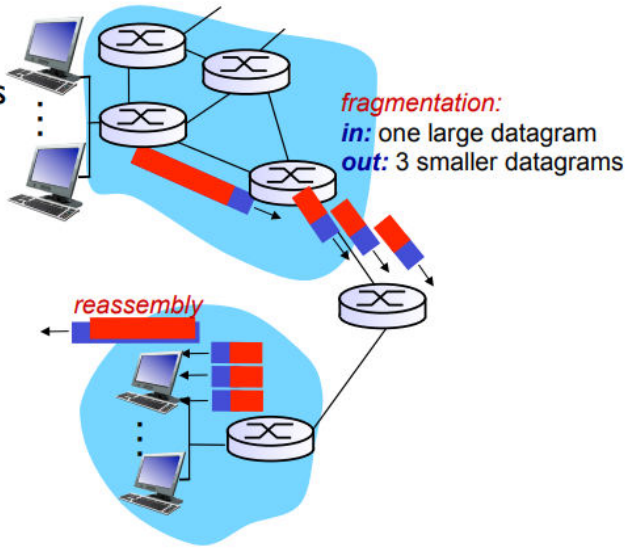
4. IP fragmentation

4.1 Background:

In this part of the lab, we will try to find out what happens when IP fragments a datagram by increasing the size of a datagram until fragmentation occurs. We will ping with options (use the –s option on MacOS, -l on windows) to set the size of data to be carried in the ICMP echo request message. Note that the default packet size is 64 bytes in MacOS and 32 Bytes in Windows. Once you have send a series of packets with the increasing data sizes, IP will start fragmenting packets that it cannot handle.

IP fragmentation, reassembly

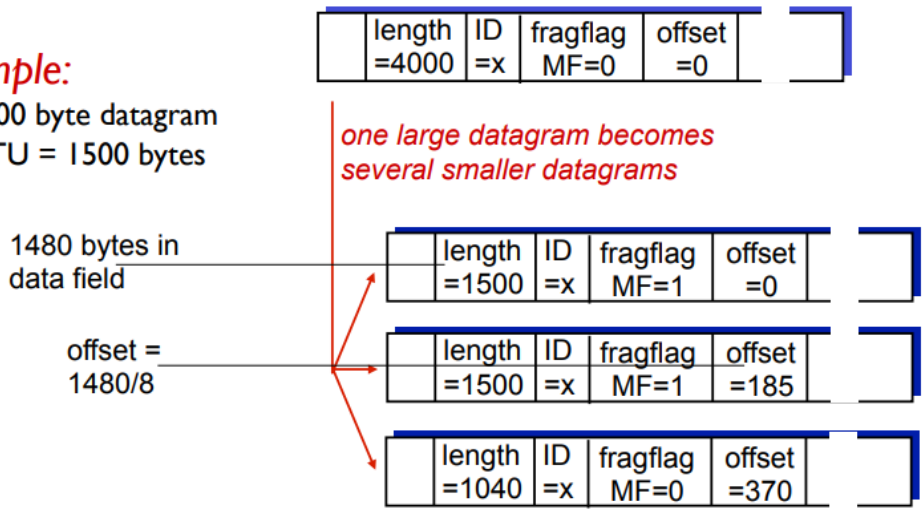
- Large IP datagram divided ("fragmented")
  - one datagram becomes several datagrams
  - "reassembled" only at final destination
  - IP header bits used to identify, order related fragments



IP fragmentation, reassembly

example:

- 4000 byte datagram
- MTU = 1500 bytes



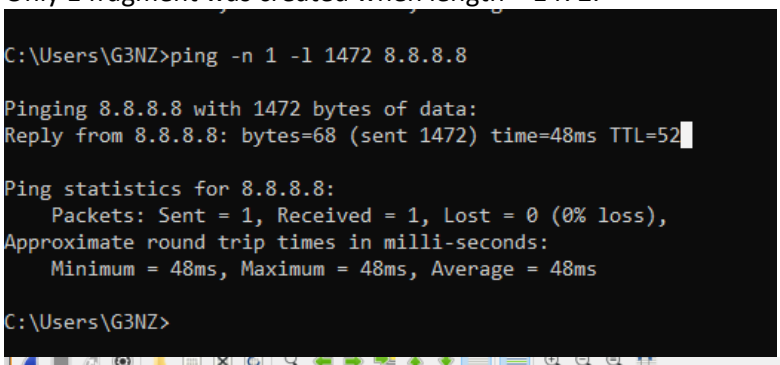
4.2 Steps for performing this part of the lab:

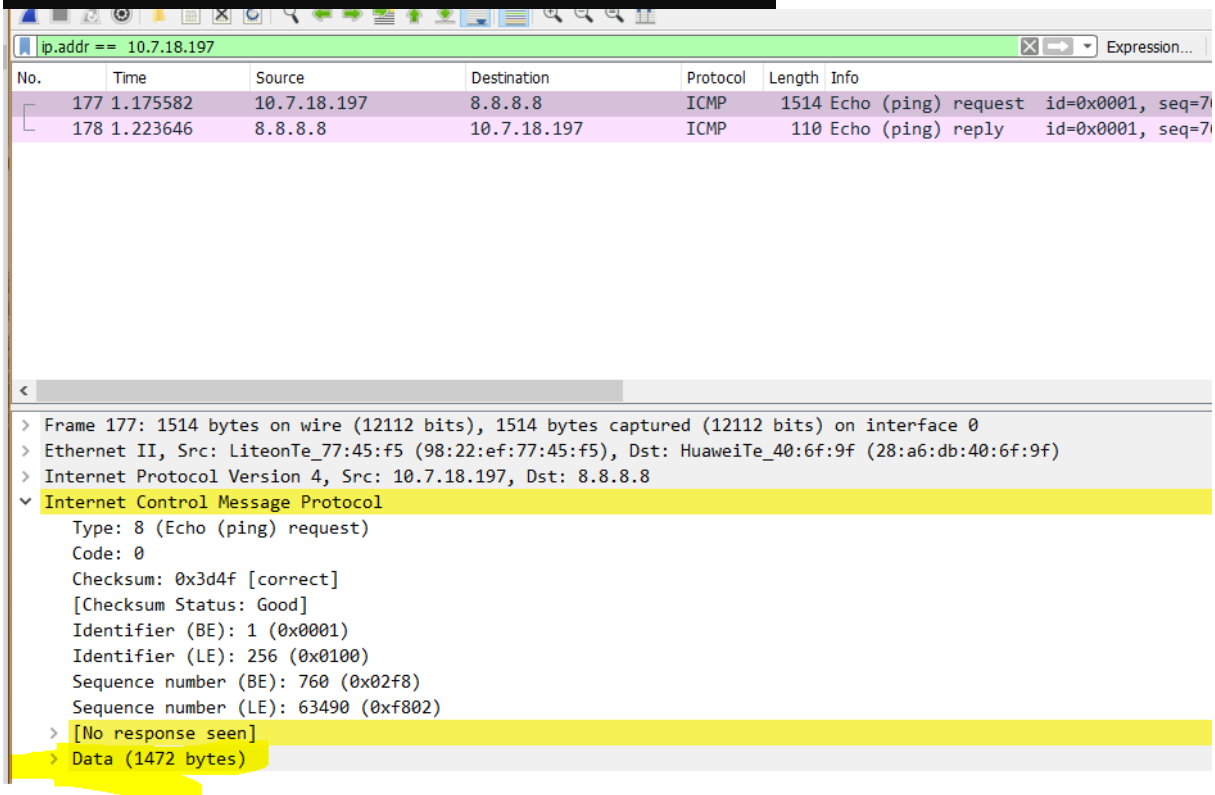
- Startup Wireshark and begin packet capture (Capture->Start) and then press OK on the Wireshark Packet Capture Options screen (we'll not need to select any options here).
- Startup ping with default values to the target destination as 8.8.8.8
- Stop the wireshark capture when you receive at least 6 replies from target destination.
- Next, repeat the above steps by sending a set of ICMP requests with a length of 2000.
- Finally, repeat again with length value set as 3500

Now answer the following questions.

1. **Which datagram size has caused fragmentation and why? Which host/router has fragmented the original datagram? How many fragments have been created in this case?**
   Any size greater than 1472 bytes is causing fragmentation. Because, MTU = 1500 bytes. TCP uses 20 bytes as header size and ICMP uses 8 bytes for header. So, 1500 – 20 – 8 = 1472.

   Only 1 fragment was created when length = 1472.





2. **Did the reply from the destination 8.8.8.8. for 3500 byte datagram also get fragmented? Why and why not?**

Yes.
Because MTU = 1500. So, 3500 byte data must be transferred in 3 chunks of size: 1472, 1480, 548.
First fragment uses 8 byte ICMP header and 20 byte TCP header. So, total size of 1ST fragment = 1500.
Second fragment uses 20 byte TCP header. So, total size of 2nd fragment = 1500.
Third fragment uses 20 byte TCP header. So, total size of 3rd fragment = 568.

```
C:\Users\G3NZ>ping -n 1 -l 3500 8.8.8.8

Pinging 8.8.8.8 with 3500 bytes of data:
Request timed out.

Ping statistics for 8.8.8.8:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),

C:\Users\G3NZ>
```

ip.addr == 10.7.18.197 Expression... +

No.	Time	Source	Destination	Protocol	Length	Info
282	1.821772	10.7.18.197	8.8.8.8	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, of
283	1.821772	10.7.18.197	8.8.8.8	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, of
284	1.821774	10.7.18.197	8.8.8.8	ICMP	582	Echo (ping) request id=0x0001, seq=764/

```
Data (1480 bytes)
    Data: 0800dbe7000102fc6162636465666768696a6b6c6d6e
    [Length: 1480]
```

```
Data (1480 bytes)
    Data: 6162636465666768696a6b6c6d6e6f707172737475767761…
    [Length: 1480]
```

```
ata (3500 bytes)
    Data: 6162636465666768696a6b6c6d6e6f707172737475767761…
```

3. **Give the ID, length, MF and offset values for all the fragments of any single datagram with size 3500 bytes?**

For 1st fragment:
ID: 0xd260
Length = 1500
MF: 1
Fragment offset: 0

```
    Total Length: 1500
    Identification: 0xd260 (53856)
  ▾ Flags: 0x2000, More fragments
      0... .... .... .... = Reserved bit: Not set
      .0.. .... .... .... = Don't fragment: Not set
      ..1. .... .... .... = More fragments: Set
    ...0 0000 0000 0000 = Fragment offset: 0
```

For 2nd fragment:
ID: 0xd260
Length = 1500
MF: 1
Fragment offset: 1480

```
Total Length: 1500
Identification: 0xd260 (53856)
Flags: 0x20b9, More fragments
    0... .... .... .... = Reserved bit: Not set
    .0.. .... .... .... = Don't fragment: Not set
    ..1. .... .... .... = More fragments: Set
...0 0101 1100 1000 = Fragment offset: 1480
```

For 3rd fragment:
ID: 0xd260
Length = 568
MF: 1
Fragment offset: 2960

```
Total Length: 568
Identification: 0xd260 (53856)
Flags: 0x0172
    0... .... .... .... = Reserved bit: Not set
    .0.. .... .... .... = Don't fragment: Not set
    ..0. .... .... .... = More fragments: Not set
...0 1011 1001 0000 = Fragment offset: 2960
```

4. **Has fragmentation of fragments occurred when datagram of size 3500 has been used? Why and why not?**

It may occur. If the MTU for each router/host is different, then it will occur.
It is possible that fragment traversing through the network may have the size less than MTU of next router. So, that fragment will be fragmented.

5. **What is the least MTU value along the complete path to destination? Which is the bottleneck link?**

Continuing with assumption, due to timeout issue in the lab, that least MTU will be the IPv4 allowed least.
576 bytes is the minimum supported value in IP infrastructure. So, the bottleneck will be 576 bytes, which can occur anywhere in the network.

Conclusion:

IP datagrams can be analyzed over Wireshark. Analysis conclude that IP datagrams can be fragmented depending on the specifications of a router or host. We can analyze the MTU values in a network by checking the sizes of fragments (and how those fragments are fragmented).

Department of Computer Science

EE353: Computer Networks

Class: BSCS-7B

Lab 10: ***Tracing the path to a destination***

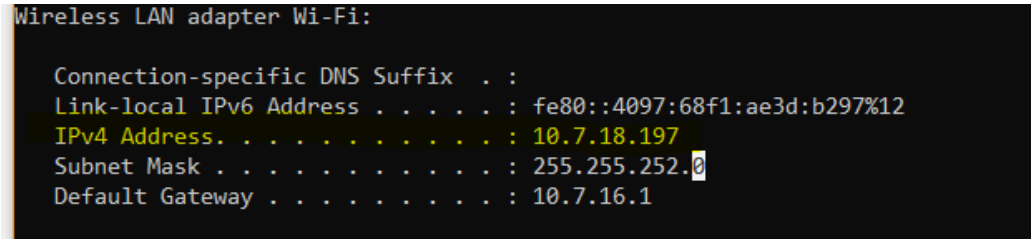
Date: 25th Nov, 2019

Lab Engineer: Engr. Kaleem Ullah

Instructor: Dr. Muhammad Zeeshan

Lab 10: *Tracing the path to a destination*

Name : M. Hasnain Naeem
REG# : 212728
IP Screen Shot:



Lab Title: *Tracing the path to a destination*

1.0 Objective of this lab:

In this lab, we'll explore several networking tools to trace the path followed by packets to a particular destination.

2.0 Instructions:

- Read carefully before starting the lab.
- These exercises are to be done individually.
- You are supposed to provide the answers to the questions listed at the end of this document, paste the screenshots of your working and upload the completed report to your course's LMS site.
- Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.

Traceroute

Background:

Trace Route (called traceroute under UNIX and tracert under Windows) is a very useful tool for analyzing network behavior and isolating problems. It determines the path that your TCP/IP packets take to a given destination, entered as an IP address or domain name. The results are fairly straightforward.

Here is a completed trace from www.cs.cf.ac.uk to www.stairways.com:

```
Hop Min Avg     Max         Ip             Name
1 0.001 0.002   0.004   131.251.1.42    mr1-e0.cf.ac.uk
2 0.010 0.034   0.076   194.83.178.17   c7000.cf.welshman.net.uk
3 0.006 0.008   0.010   146.97.252.93   welshnet.bristol-core.ja.net
4 0.015 0.017   0.019   146.97.252.190  ext-gw6.ja.net
5 0.012 0.017   0.023   193.63.94.95    us-gw3.ja.net
6 0.083 0.093   0.108   193.62.157.18   ny-pop.ja.net
7 0.088 0.090   0.094   207.45.196.141  if-8-2.core1.newyork.teleglobe.net
8 0.076 0.077   0.079   207.45.223.110  if-10-0.bb8.newyork.teleglobe.net
9 0.081 0.084   0.088   207.45.198.74   ix-8-0-1.bb8.newyork.teleglobe.net
```

```
10 0.083 0.087 0.092   152.63.22.218   518.at-6-0-0.xr1.nyc9.alter.net
11 0.083 0.087 0.090   152.63.20.66    181.at-2-0-0.tr1.nyc8.alter.net
12 0.192 0.200 0.206   152.63.5.214    124.at-6-0-0.tr1.por3.alter.net
13 0.189 0.192 0.198   152.63.104.253  297.atm6-0.xr1.sea1.alter.net
14 0.204 0.208 0.211   146.188.200.41  195.atm7-0.gw1.sea1.alter.net
15 0.163 0.164 0.166   137.39.136.6    ixa-gw.customer.alter.net
16 0.226 0.233 0.237   63.237.224.54
17 0.233 0.250 0.273   199.254.168.243
```

Hop

-- Gives the order in which the TCP/IP packets progress from machine to machine, called the 'distance' (in hops) from the originating machine.

Result

-- Received/Sent packets, or other information (see below). Assuming all is well, these numbers should match - if more packets are sent than received, there may be a problem.

Min, Avg, & Max

-- The Minimum, Average and Maximum round trip time in seconds that the packets took to go to and return from that machine.

IP & Name

-- The IP address and domain name of the remote machine which is conveying your TCP/IP packets. There are other possible values which can appear in the Result column, identifying network problems with the trace:

How Trace Route Works: TTLs

TTL stands for Time To Live. When a TCP packet is sent, its TTL is set, which is the number of routers (hops) it can pass through before the packet is discarded. As the packet passes through a router the TTL is decremented until, when the TTL reaches zero, the packet is destroyed and an ICMP "time exceeded" message is returned. The return message's TTL is set by the terminating router when it creates the packet, and decremented normally.

Trace Route works by setting the TTL for a packet to 1, sending it towards the requested destination host, and listening for the reply. When the initiating machine receives a "time exceeded" response, it examines the packet to determine where the packet came from - this identifies the machine one hop away. Then the tracing machine generates a new packet with TTL 2, and uses the response to determine the machine 2 hops away, and so on.

Unfortunately not all TCP stacks behave correctly. Some TCP stacks set the TTL for the ICMP "time exceeded" message to that of the message being killed. So if the TTL is 0, the packet will be killed by the next machine to which it is passed. This can have two effects on a trace. If the computer is an intermediate machine in the trace, the entry will remain blank. No information is returned to the machine conducting the trace because the "time exceeded" message never makes it back. If the machine you are doing a trace to has this bug in its TCP stack, return packets won't reach the originating machine unless the TTL is high enough to cover the round trip. So Trace Route will show a number of failed connections equal to n (the number of hops to the destination machine) minus 1.

Lab 10: *Tracing the path to a destination*

Steps for performing this lab:

1. **Open the command prompt application**
2. **Start up the Wireshark packet sniffer.**
3. **Begin packet capture.**
4. Type "tracert www.usyd.edu.au"(or traceroute) in command prompt and press enter.

Now answer the following questions:

1. **What are the IP address of the host www.usyd.edu.au and the IP of your machine?**
   IP of www.usyd.edu.au: 129.78.5.11

```
C:\Users\G3NZ>ping  www.usyd.edu.au

Pinging rp0.ucc.usyd.edu.au [129.78.5.11] with 32 bytes of data:
```

   IP of my machine: 10.7.16.1

```
Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::4097:68f1:ae3d:b297%12
   IPv4 Address. . . . . . . . . . . : 10.7.18.197
   Subnet Mask . . . . . . . . . . . : 255.255.252.0
   Default Gateway . . . . . . . . . : 10.7.16.1
```

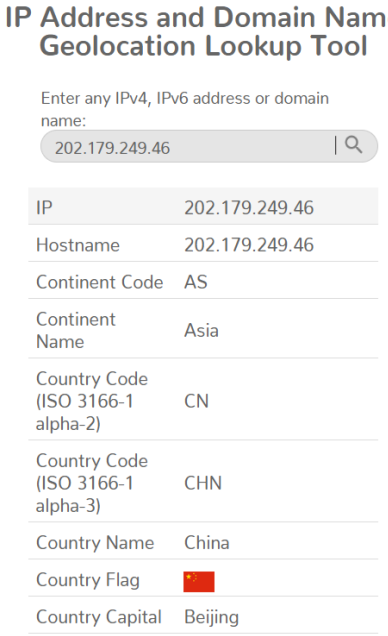
23	0.985691	10.7.29.182	10.7.31.255	UDP	305 54915 → 54915 Len=263
24	1.147925	10.7.29.187	255.255.255.255	DB-LSP…	222 Dropbox LAN sync Discovery Protocc
25	1.148008	10.7.29.187	255.255.255.255	DB-LSP…	222 Dropbox LAN sync Discovery Protocc
26	1.148362	10.7.29.187	10.7.31.255	DB-LSP…	222 Dropbox LAN sync Discovery Protocc
27	1.149213	10.7.29.187	255.255.255.255	DB-LSP…	222 Dropbox LAN sync Discovery Protocc
28	1.250898	10.7.18.197	129.78.5.11	ICMP	106 Echo (ping) request id=0x0001, se
29	1.256894	10.7.16.1	10.7.18.197	ICMP	70 Time-to-live exceeded (Time to liv
30	1.258733	10.7.18.197	129.78.5.11	ICMP	106 Echo (ping) request id=0x0001, se
31	1.268696	10.7.16.1	10.7.18.197	ICMP	70 Time-to-live exceeded (Time to liv
32	1.270275	10.7.18.197	129.78.5.11	ICMP	106 Echo (ping) request id=0x0001, se
33	1.277051	10.7.16.1	10.7.18.197	ICMP	70 Time-to-live exceeded (Time to liv
34	1.310048	10.7.54.80	10.7.55.255	NBNS	92 Name query NB ISATAP<00>
35	1.310681	10.7.31.198	10.7.31.255	NBNS	92 Name query NB WPAD<00>
36	1.311444	Apple_6d:a5:ae	Broadcast	RARP	56 Who is ff:ff:ff:ff:ff:ff? Tell 7c:

2. **How many hops is the destination host away from your machine?**
   13 hops.

Lab 10: *Tracing the path to a destination*

```
C:\Users\G3NZ>tracert www.usyd.edu.au

Tracing route to rp0.ucc.usyd.edu.au [129.78.5.11]
over a maximum of 30 hops:

  1     5 ms     4 ms     4 ms  10.7.16.1
  2     *        *        *     Request timed out.
  3     5 ms     3 ms     2 ms  10.31.254.25
  4     9 ms     5 ms     5 ms  172.31.254.25
  5    34 ms    33 ms    35 ms  202.179.249.46
  6   109 ms   109 ms   112 ms  202.179.249.45
  7   113 ms   112 ms   219 ms  202.179.249.42
  8   180 ms   110 ms   109 ms  202.179.249.62
  9   201 ms   201 ms   201 ms  et-7-3-0.pe1.nsw.brwy.aarnet.net.au [113.197.15.232]
 10   199 ms   199 ms   199 ms  gw1.vl216.ae11.pe1.brwy-pe1.aarnet.net.au [138.44.5.47]
 11     *        *        *     Request timed out.
 12     *        *        *     Request timed out.
 13   199 ms   199 ms   198 ms  lists.medsci.usyd.edu.au [129.78.5.11]

Trace complete.
```

3. **How many hops are between your machine and the NUST gateway router?**
   2 or 3. First one for the router inside the hostel. Second one gives timeout. Third one should be default gateway for the NUST. 4th should be ISP within Pakistan. (Information of these IPs is not available through lookups).
   Because, 5th hop reaches to IP: 202.179.249.46 and location of this IP is china.



IP Address and Domain Name Geolocation Lookup Tool

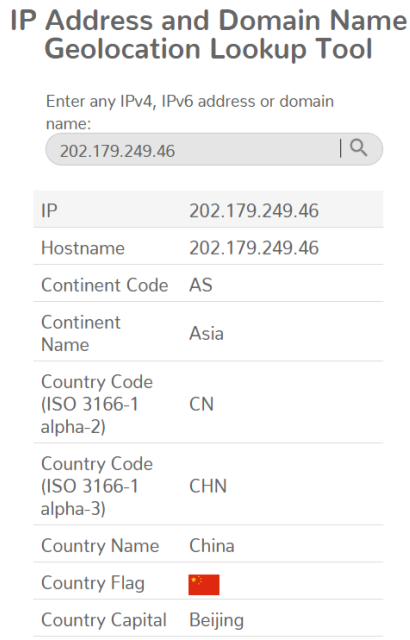
Enter any IPv4, IPv6 address or domain name:

202.179.249.46

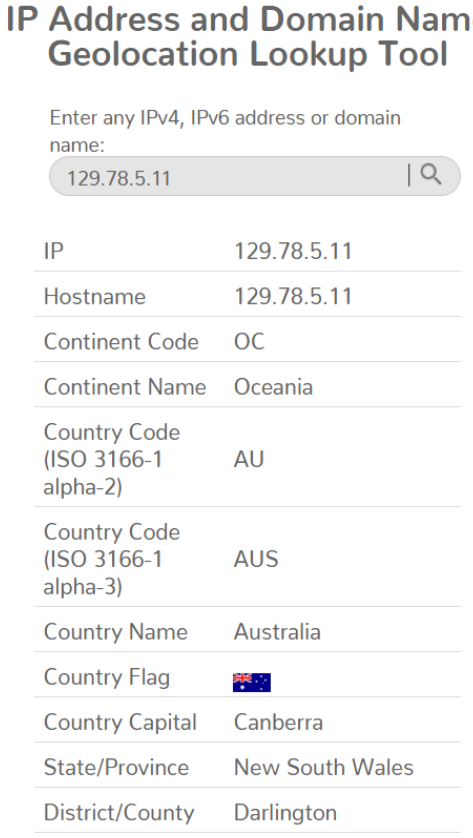
IP	202.179.249.46
Hostname	202.179.249.46
Continent Code	AS
Continent Name	Asia
Country Code (ISO 3166-1 alpha-2)	CN
Country Code (ISO 3166-1 alpha-3)	CHN
Country Name	China
Country Flag	
Country Capital	Beijing

4. **How many routers does these packets visit in Pakistan?**
   4 routers. Because 5th hop IP has location of China.

Lab 10: *Tracing the path to a destination*



5. **Where is the website www.usyd.edu.au hosted (city and country)?**
   Sydney, Australia is the location of website.
   According to the last hop, which gave IP address: 129.78.5.1



6. **How many cities your packets have actually visited? List all these cities along with the name of country in the order these have been visited?**
   Locations of IP addresses in hops are:
   - Islamabad, Pakistan
   - Beijing, China
   - Beijing, China

- Beijing, China
- Beijing, China
- Sydney, Australia
- Sydney, Australia
- Sydney Australia

So, it traveled total 3 cities.

7. **Comment if you observe any abnormal/wayward path followed by the traffic from your machine to the destination (It may be useful to roughly draw the path followed by the traffic on a map).**
   There is no abnormality in path followed. Although, traffic went to China and then it was sent to Australia.
8. **Does the generated traffic always follow the same path to this destination?**
   No. Switches and routers may send them to different routes depending on the traffic.
9. **How many routers in the path are working in "safe mode" (not replying to any query)?**
   3 routers (traceroute command gave the timeout error)
10. **Which hop is the longest in the path to the destination?**
    9^{th} hop which ended at IP: 113.197.15.232. Its average RTT is 201 ms. All other hops have less than 200 ms RTT.

Conclusion:

We can examine the routers through which the traffic passes by using special commands which limit the hops. Each hop returns the IP address of the router (if not in safe mode) and we can find the locations of routers from their IP addresses.
This can help in various network analysis tasks.

Refrence : http://users.cs.cf.ac.uk/Dave.Marshall/Internet/node76.html
http://users.cs.cf.ac.uk/Dave.Marshall/Internet/node77.html

Department of Computer Science

EE353: Computer Networks

Class: BSCS-7AB

Lab 11: *NAT*

Date: 2th Dec, 2019

Lab Engineer: Engr. Kaleem Ullah

Instructor: Dr. Zeeshan

Lab 11: *Wireshark—NAT*

Name: M. Hasnain Naeem
Reg. No: 212728

1.0 NAT Network Address Translation:

The Internet is expanding at an exponential rate. As the amount of information and resources increases, it is becoming a requirement for even the smallest businesses and homes to connect to the Internet. Network Address Translation (NAT) is a method of connecting multiple computers to the Internet (or any other IP network) using one IP address. This allows home users and small businesses to connect their network to the Internet cheaply and efficiently.

In computer networking, **network address translation** (**NAT**) is the process of modifying IP address information in IP packet headers while in transit across a traffic routing device.

The simplest type of NAT provides a one to one translation of IP addresses. RFC 2663 refers to this type of NAT as **basic NAT**. It is often also referred to as **one-to-one NAT**. In this type of NAT only the IP addresses, IP header checksum and any higher level checksums that include the IP address need to be changed. The rest of the packet can be left untouched (at least for basic TCP/UDP functionality, some higher level protocols may need further translation). Basic NATs can be used when there is a requirement to interconnect two IP networks with incompatible addressing.

The impetus towards increasing use of NAT comes from a number of factors:

- A world shortage of IP addresses
- Security needs
- Ease and flexibility of network administration

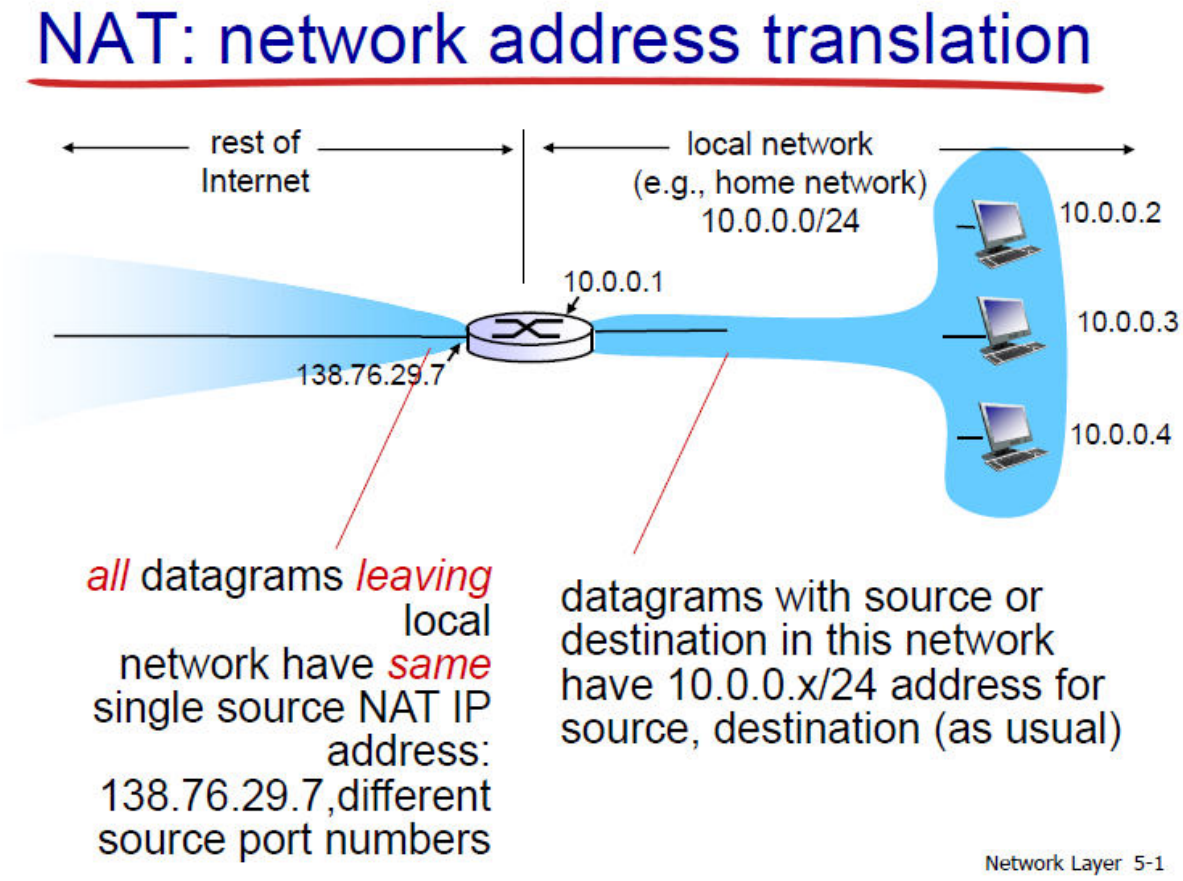
IP Addresses

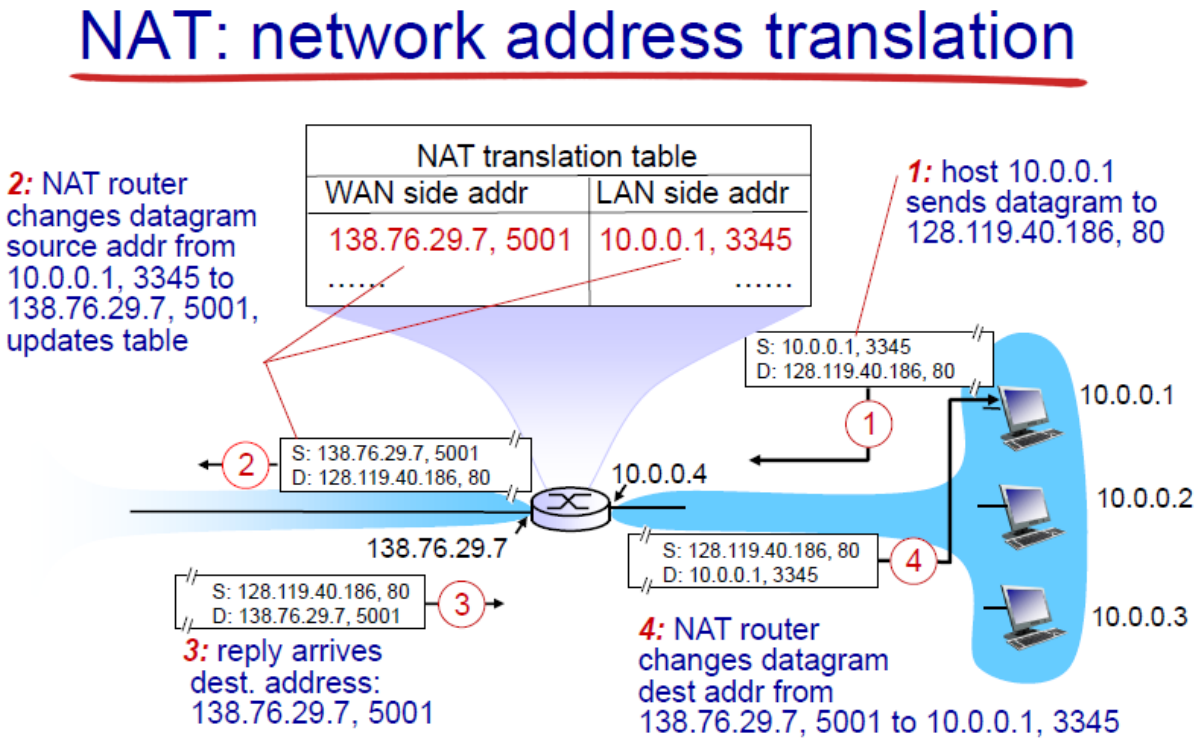
Because IP addresses are a scarce resource, most Internet Service Providers (ISPs) will only allocate one address to a single customer. In majority of cases this address is assigned dynamically, so every time a client connects to the ISP a different address will be provided. Big companies can buy more addresses, but for small businesses and home users the cost of doing so is prohibitive. Because such users are given only one IP address, they can have only one computer connected to the Internet at one time. When IP addressing first came out, everyone thought that there were plenty of addresses to cover any need. Theoretically, you could have 4,294,967,296 unique addresses (2^{32}). The actual number of available addresses is smaller (somewhere between 3.2 and 3.3 billion) because of the way that the addresses are separated into classes, and because some addresses are set aside for multicasting, testing or other special uses. This is where NAT comes to the rescue. Network Address Translation allows a single device, such as a router, to act as an agent between the Internet (or "public network") and a local (or "private") network. This means that only a single, unique IP address is required to represent an entire group of computers. With an NAT gateway running on this single computer, it is possible to share that single address between multiple local computers and connect them all at the same time. The outside world is unaware of this division and thinks that only one computer is connected.

Security Considerations

To combat the security problem, a number of firewall products are available. They are placed between the user and the Internet and verify all traffic before allowing it to pass through. This means, for example, that no unauthorized user would be allowed to access the company's file or email server. The problem with firewall solutions is that they are expensive and difficult to set up and maintain, putting them out of reach for home and small business users.

NAT automatically provides firewall-style protection without any special set-up. That is because it only allows connections that are originated on the inside network. This means, for example, that an internal client can connect to an outside FTP server, but an outside client will not be able to connect to an internal FTP server because it would have to originate the connection, and NAT will not allow that.





Steps for performing this part of lab:

In this lab, we'll capture packets from a simple web request from a client PC in a home network to a www.google.com server. Within a home network, the home network router typically provides a NAT service.

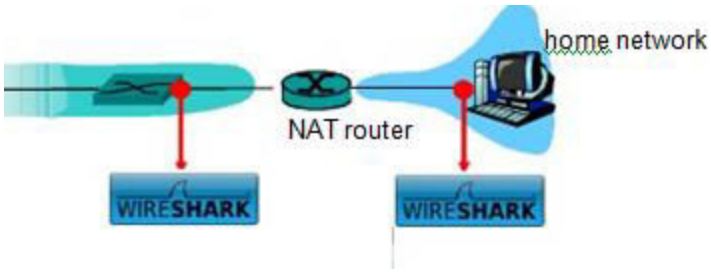


Figure 5: NAT trace collection scenario

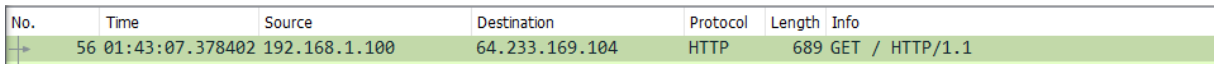
Figure 5 shows our Wireshark trace-collection scenario. As in our other Wireshark labs, we collect a Wireshark trace on the client PC in our home network. This file is called NAT_home_side. Because we are also interested in the packets being sent by the NAT router into the ISP, we'll collect a second trace file at a PC (not shown) tapping into the link from the home router into the ISP network, as shown in Figure 1. (The hub device shown on the ISP side of the router is used to tap into the link between the NAT router and the first hop router in the ISP). Client-to-server packets captured by Wireshark at this point will have

undergone NAT translation. The Wireshark trace file captured on the ISP side of the home router is called NAT_ISP_side.

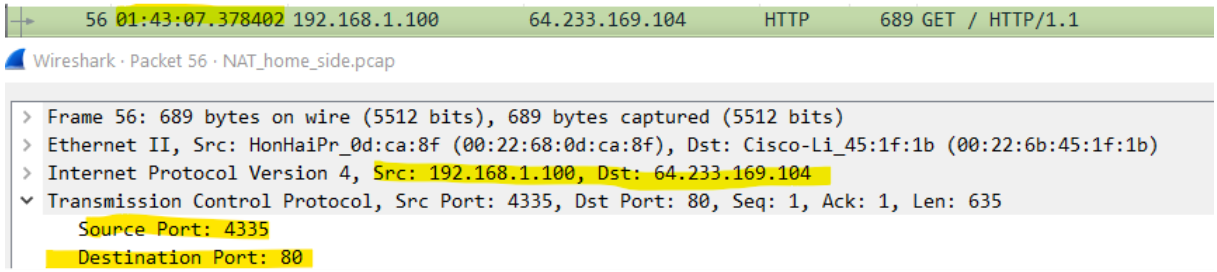
Open the NAT_home_side file and answer the following questions. You might find it useful to use a Wireshark filter so that only frames containing HTTP messages are displayed from the trace file. The main Google server that will serve up the main Google web page has IP address 64.233.169.104. In order to display only those frames containing HTTP messages that are sent to/from this Google server, enter the expression "http && ip.addr == 64.233.169.104" (without quotes) into the Filter field in Wireshark.

Questions:

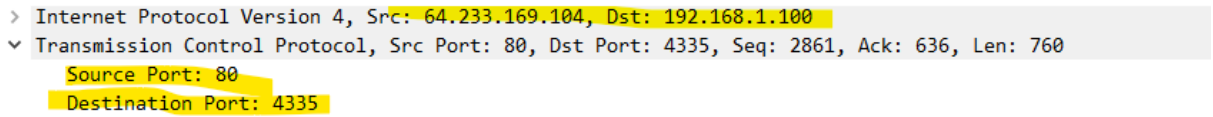
1. What is the IP address of the client?
   Client IP: 192.168.1.100



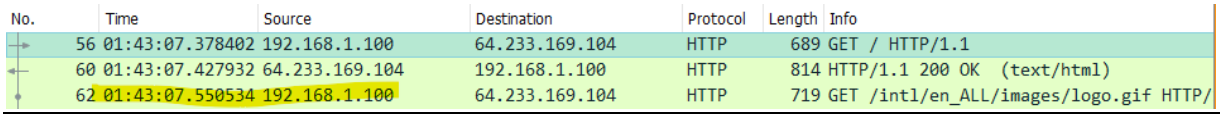
2. Consider now the HTTP GET sent from the client to the Google server (IP address 64.233.169.104) at time 02:43:07.378402. What are the source and destination IP addresses and TCP source and destination ports on the IP datagram carrying this HTTP GET?
   Source IP & Port: 192.168.1.100**:***4335*
   Destination IP & Port: 64.233.169.104**:***80*



3. At what time is the corresponding 200 OK HTTP message received from the Google server? What are the source and destination IP addresses and TCP source and destination ports on the IP datagram carrying this HTTP 200 OK message?
   Source IP & Port: 64.233.169.104**:***80*
   Destination IP & Port: 192.168.1.100**:***4335*



4. Recall that before a GET command can be sent to an HTTP server, TCP must first set up a connection using the three-way SYN/ACK handshake. At what time is the TCP connection ready?
   TCP connection will be ready after 3-way handshake when both client and server have acknowledged their presence to each other.
   In this case, TCP connection is ready at time: 01:43:07.550534



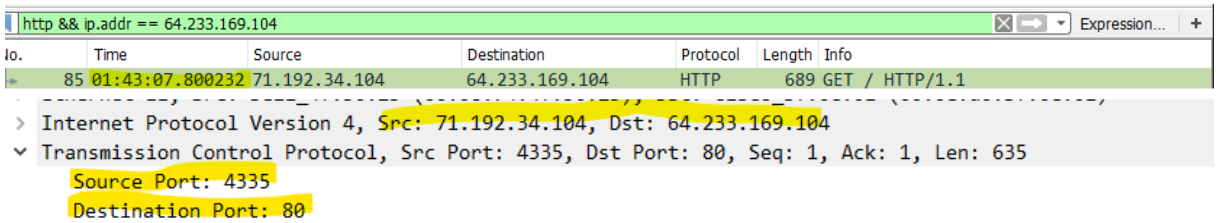
No.	Time	Source	Destination	Protocol	Length	Info
56	01:43:07.378402	192.168.1.100	64.233.169.104	HTTP	689	GET / HTTP/1.1
60	01:43:07.427932	64.233.169.104	192.168.1.100	HTTP	814	HTTP/1.1 200 OK (text/html)
62	01:43:07.550534	192.168.1.100	64.233.169.104	HTTP	719	GET /intl/en_ALL/images/logo.gif HTTP/

In the following we'll focus on the two HTTP messages (GET and 200 OK) identified above. Our goal below will be to locate these two HTTP messages in the trace file (NAT_ISP_side) captured on the link between the router and the ISP. Because these captured frames will have already been forwarded through the NAT router, some of the IP address and port numbers will have been changed as a result of NAT translation.

Open the NAT_ISP_side. Note that the time stamps in this file and in NAT_home_side are not synchronized since the packet captures at the two locations shown in Figure 1 were not started simultaneously.

5. In the NAT_ISP_side trace file, find the first HTTP GET message that was sent from the client to the Google server. At what time does this message appear in the NAT_ISP_side trace file? What are the source and destination IP addresses and TCP source and destination ports on the IP datagram carrying this HTTP GET?

   Time of appearance: 01.43.07.800232
   Source IP & Port: 71.192.34.104.104**:***4335*
   Destination IP & Port: 64.233.169.104**:***80*



6. Compare these values with the corresponding values in the NAT_home_side file and comment whether NAT or NAPT is being used at the NAT router.

   NAT is being used. Because, IP and Port are the same on client and ISP side NAT (4334 and 80). NAPT changes the Port along with the IP. But, NAT changes only the IP address of client computers.

7. In the NAT_ISP_side trace file, at what time is the 200 OK HTTP message received from the Google server? What are the source and destination IP addresses and TCP source and destination ports on the IP datagram carrying this HTTP 200 OK message?

Receiving Time: 01:43:07.848634
Source IP & Port: 64.233.169.104:*80*
Destination IP & Port: 71.192.34.104:4335

```
> Internet Protocol Version 4, Src: 64.233.169.104, Dst: 71.192.34.104
v Transmission Control Protocol, Src Port: 80, Dst Port: 4335, Seq: 2861, Ack: 636, Len: 760
    Source Port: 80
    Destination Port: 4335
```

8. Locate the TCP connection(s) made for this HTTP transaction. How many TCP connections have been made? Does the TCP connection addresses also get modified while passing through the NAT router?

   Assuming transaction means opening www.google.com
   Although there are 7 OK messages but only 1 TCP connection is made. Because, HTTP/1.1 is, by default, persistent.

   Yes, TCP source connection address gets modified when it passes through the NAT router.

No.	Time	Source	Destination	Protocol	Length	Info
Number	6 01:43:07.378402	192.168.1.100	64.233.169.104	HTTP	689	GET / HTTP/1.1
	60 01:43:07.427932	64.233.169.104	192.168.1.100	HTTP	814	HTTP/1.1 200 OK (text/html)
	62 01:43:07.550534	192.168.1.100	64.233.169.104	HTTP	719	GET /intl/en_ALL/images/logo.gif HTTP/
	73 01:43:07.618586	64.233.169.104	192.168.1.100	HTTP	226	HTTP/1.1 200 OK (GIF89a)
	75 01:43:07.639320	192.168.1.100	64.233.169.104	HTTP	809	GET /extern_js/f/CgJlbhICdXMrMAo4NUAIL
	92 01:43:07.717784	64.233.169.104	192.168.1.100	HTTP	648	HTTP/1.1 200 OK (text/javascript)
	94 01:43:07.761459	192.168.1.100	64.233.169.104	HTTP	695	GET /extern_chrome/ee36edbd3c16a1c5.js
	100 01:43:07.806488	64.233.169.104	192.168.1.100	HTTP	870	HTTP/1.1 200 OK (text/html)
	107 01:43:07.921971	192.168.1.100	64.233.169.104	HTTP	712	GET /images/nav_logo7.png HTTP/1.1
	112 01:43:07.951496	192.168.1.100	64.233.169.104	HTTP	806	GET /csi?v=3&s=webhp&action=&tran=unde
	119 01:43:07.954921	64.233.169.104	192.168.1.100	HTTP	1359	HTTP/1.1 200 OK (PNG)
	122 01:43:07.978625	192.168.1.100	64.233.169.104	HTTP	670	GET /favicon.ico HTTP/1.1
	124 01:43:08.006918	64.233.169.104	192.168.1.100	HTTP	269	HTTP/1.1 204 No Content
	127 01:43:08.032636	64.233.169.104	192.168.1.100	HTTP	1204	HTTP/1.1 200 OK (image/x-icon)

9. List at least two entries that exist in the NAT table of the router.

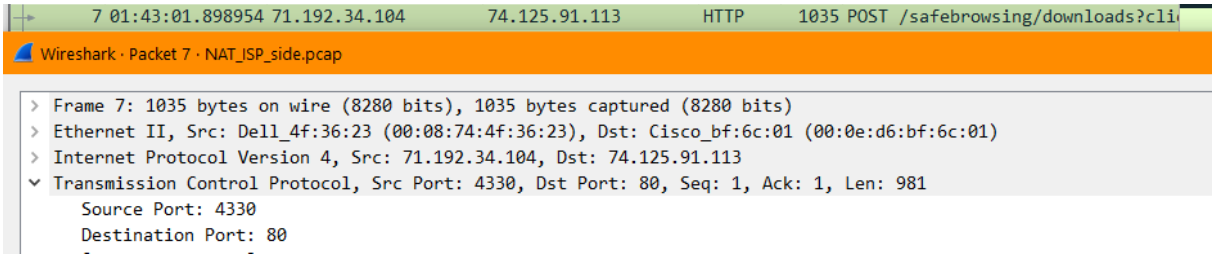
WAN Side Address	LAN Side Address
71.192.34.104.10 : 4335	192.168.1.100 : 4335
71.192.34.104.10 : 4330	192.168.1.100 : 4330

Using another packet to get 2nd entry for the NAT router table.

Information of packet is given below.
Client Side

```
    7 01:43:01.477175 192.168.1.100    74.125.91.113    HTTP    1035 POST /safebrowsing/downloads?cli
Wireshark · Packet 7 · NAT_home_side.pcap

> Frame 7: 1035 bytes on wire (8280 bits), 1035 bytes captured (8280 bits)
> Ethernet II, Src: HonHaiPr_0d:ca:8f (00:22:68:0d:ca:8f), Dst: Cisco-Li_45:1f:1b (00:22:6b:45:1f:1b)
> Internet Protocol Version 4, Src: 192.168.1.100, Dst: 74.125.91.113
v Transmission Control Protocol, Src Port: 4330, Dst Port: 80, Seq: 1, Ack: 1, Len: 981
    Source Port: 4330
    Destination Port: 80
```

ISP Side



Conclusion

IP address assignment and translation are very important aspects in computer networks. Due to limited number of IP addresses in IPV4, NAT mechanism is used to entertain multiple computers through a router.

In this lab, we analyzed the working of NAT by inspecting the packets from ISP and client side.

Department of Computer Science

EE353: Computer Networks

Name: M. Hasnain naeem

Reg #: 212728

Class: BSCS-7B

Lab 12: *Analysis of UDP and TCP packets in Wireshark*

Date: 9th DEC 2019

Lab Engineer: Kaleem Ullah

Instructor: Dr. Zeeshan

Lab Title: *Analysis of UDP and TCP packets in Wireshark*

Objective of this lab:

In this lab, we will analyze the behavior of UDP in detail, determining the number of fields in UDP header, the value in the UDP header fields, and maximum number of bytes in UDP payload, source & destination port numbers etc.

we'll also investigate the behavior of TCP in detail. We'll do so by analyzing TCP segments sent and received from your computer to a remote server. We'll study TCP's use of sequence and acknowledgement numbers for providing reliable data transfer.

Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *You are supposed to provide the answers to the questions listed at the end of this document and upload the completed report to your course's LMS site.*
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.*

Background:

1. ***Introduction to UDP:***
   UDP (User Datagram Protocol) is a simple transport layer protocol for client/server network applications based on Internet Protocol (IP). UDP is the main alternative to TCP and one of the oldest network protocols in existence, introduced in 1980. UDP is often used in videoconferencing applications or computer games specially tuned for real-time performance. To achieve higher performance, the protocol allows individual packets to be dropped (with no retries) and UDP packets to be received in a different order than they were sent as dictated by the application.
2. ***UDP Datagrams:***
   UDP network traffic is organized in the form of datagrams. A datagram comprises one message unit. The first eight (8) bytes of a datagram contain header information and the remaining bytes contain message data.

   A UDP datagram header consists of four (4) fields of two bytes each: Source port number, Destination port number, Datagram size and checksum

   a. **UDP port number:**UDP port numbers allow different applications to maintain their own channels for data similar to TCP. UDP port headers are two bytes long.

b. **Datagram size:**The UDP datagram size is a count of the total number of bytes contained in header and data sections. As the header length is a fixed size, this field effectively tracks the length of the variable-sized data portion (sometimes called payload). The size of datagrams varies depending on the operating environment but has a maximum of 65535 bytes.

c. **Checksum**: UDP checksums protect message data from tampering. The checksum value represents an encoding of the datagram data calculated first by the sender and later by the receiver. Should an individual datagram be tampered with or get corrupted during transmission, the UDP protocol detects a checksum calculation mismatch. In UDP, check-summing is optional as opposed to TCP where checksums are mandatory.

Steps for performing this lab:

Do the following:

1. ***Download*** *files UDPCient.py and UDPServer.py from your LMS site.*

2. ***Edit*** *these files. In UDPClient.py TheserverIP address; use one of your neighbor and the message; as your name. In UDPServer.py use your own IP address*

3. ***Start up the Wireshark software.***

4. ***Begin packet capture,*** *select the Capture pull down menu and select Options.*

5. ***Selecting the network interface on which packets would be captured:*** *You can use most of the default values in this window. The network interfaces (i.e., the physical connections) that your computer has to the network will be shown in the Interface pull down menu at the top of the Capture Options window. Click Start. Packet capture will now begin*

6. ***Run your UDPServer and UDPClient.***

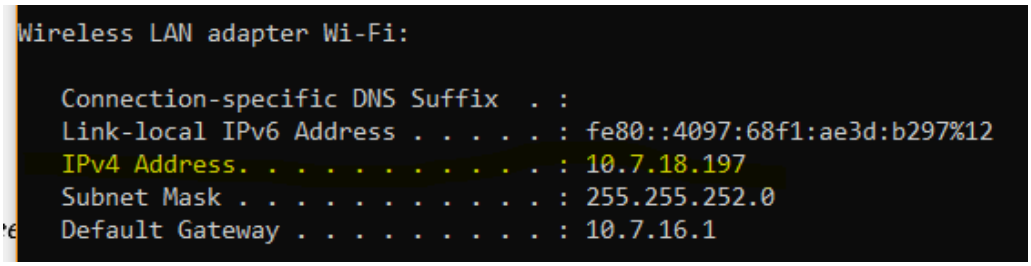
7. ***Stopping the capture and inspecting captured packets:*** *After you have received a welcome message, stop Wireshark packet capture*

8. ***Filtering:*** *Filter the UDP packets.*

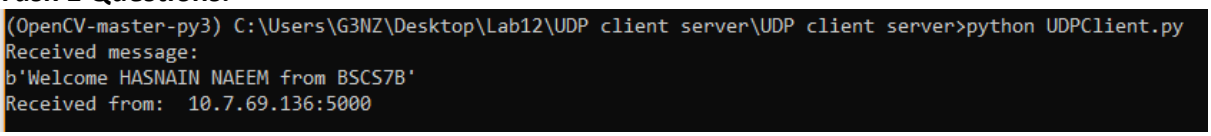
7. ***Details of a packet:*** *Select theUDP messages shown in the packet-listing window and analyze by looking into the detail of packets pane and answer the questions given at the end of this document.*

8. ***Obtaining credit for this lab:*** *Now, please proceed to the questions section to answer the questions. You must note down your answers,along with screen shots in this file itself. Please note that you must upload this file (after duly filling in the answers) through the appropriate link at your LMS to obtain credit. Please clarify with your instructor/ lab engineer if you have any queries.*

IP Information:



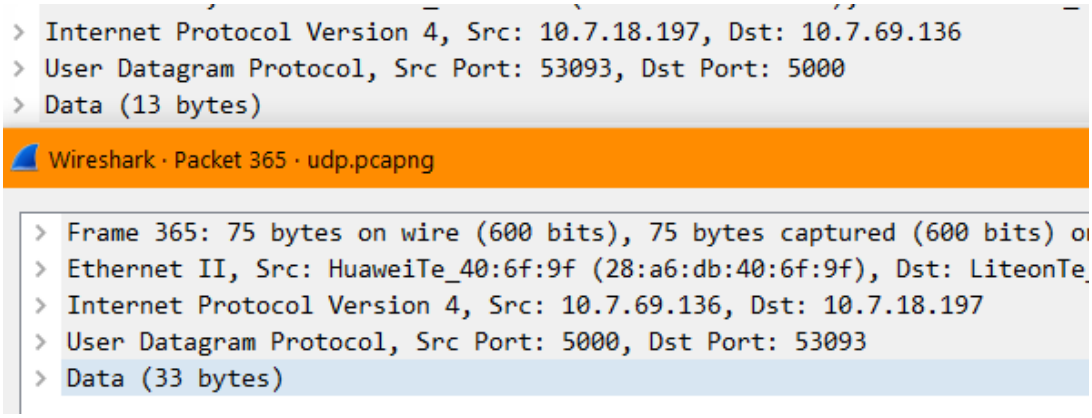
Task 1 Questions:



1. *Select one UDP packet and determine the* ***Source IP, Source port No, Destination IP and Destination port No*** *of that UDP packet.*
   Server IP & Port: 10.7.69.136:5000
   Client IP & Port: 10.7.18.197:53093

2. *Select one UDP packet and determine how many* ***fields*** *are there in the UDP header. List* ***the name of these fields****.*
   There are 4 fields. Namely:
   - Source port
   - Destination port
   - Length
   - Checksum
3. *From the packet content field, determine* ***the length (in bytes) of each of the UDP header*** *fields.*
   - Source port: 2 bytes
   - Destination port: 2 bytes
   - Length: 2 bytes
   - Checksum: 2 bytes

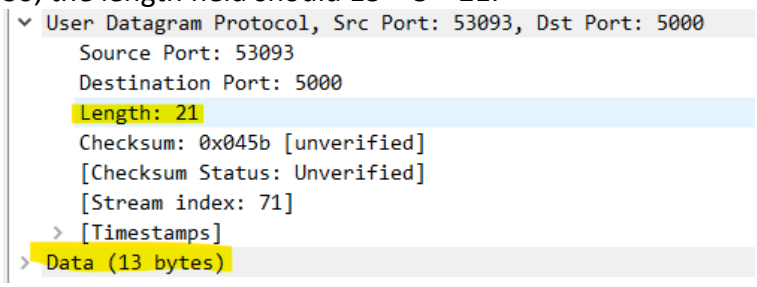
4. ***Examine the pair of UDP packets*** *in which your host sends the first packet and the second packet is a reply to the first packet. Describe the relationship between the port numbers in the two packets.*
   In first packet, source port was that of client and destination port was that of server.
   In second packet, positions were switched – server's port was in the source field and client's port was in the destination field.



5. *Analyze the UDP packet and answer that the **value in the Length field** is the length of what? Verify your claim with your captured UDP packet.*

   Value in the length field = length of the data + header size (8 bytes)

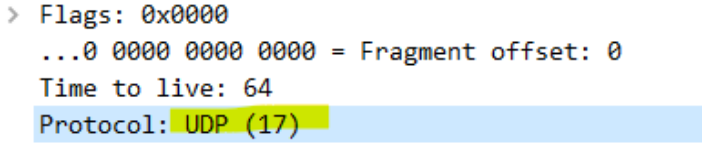
   Verification of claim:
   In the client's packet, data length = 13.
   So, the length field should 13 + 8 = 21.



6. *What is the **maximum number of bytes** that can be included in a UDP payload? Why this is the maximum?*
   65,535 bytes is the theoretical limit of the UDP payload which includes the 8 bytes of header.
   This theoretical limit is because the "length" field in the header can contain only 2 bytes. So, maximum value which it can hold is 2^{16}-1 = 65,535.

7. *What is the **largest possible source port number**?*
   Largest possible port number = 2^{16} - 1 = 65,535. Because, "source port" field can hold only 16 bits.

8. *What is the **protocol number for UDP**? Give your answer in both hexadecimal and decimal notation.*
   Protocol number: 11d OR 0x11.



9. *Which fields are included in calculating the UDP checksum?*
   Following are included the calculation of UDP checksum:
   - Header fields
     - Source port
     - Destination port
     - Length
   - Data in the packet

---

1. The Transmission Control Protocol (TCP) Introduction:

TCP provides connections between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.

TCP also provides **reliability**. When TCP sends data to the other end, it requires an acknowledgment in return. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time. After some number of retransmissions, TCP will give up, with the total amount of time spent trying to send data typically between 4 and 10 minutes (depending on the implementation).

TCP contains algorithms to estimate the **round-trip time** (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgment. For example, the RTT on a LAN can be milliseconds while across a WAN it can be in seconds. Furthermore, TCP continuously estimates the RTT of a given connection, because the RTT is affected by variations in the network traffic.

TCP also sequences the data by associating a **sequence number** with every byte that it sends. For example, assume an application writes 2,048 bytes to a TCP socket, causing TCP to send two segments, the first containing the data with sequence numbers 1–1,024 and the second containing the data with sequence numbers 1,025–2,048. (A segment is the unit of data that TCP passes to IP.) If the segments arrive out of order, the receiving TCP will reorder the two segments based on their sequence numbers before passing the data to the receiving application. If TCP receives duplicate data from its peer (say the peer thought a segment was lost and retransmitted it, when it wasn't really lost, the network was just overloaded), it can detect that the data has been duplicated (from the sequence numbers), and discard the duplicate data.

In contrast to TCP, there is no reliability provided by UDP. UDP itself does not provide anything like acknowledgments, sequence numbers, RTT estimation, timeouts, or retransmissions. If a UDP datagram is duplicated in the network, two copies can be delivered to the receiving host. Also, if a UDP client sends two datagrams to the same destination, they can be reordered by the network and arrive out of order.

TCP provides **flow control**. TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer at any one time. This is called the advertised window. At any time, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receive buffer. The window changes dynamically over time: As data is received from the sender, the window size decreases, but as the receiving application reads data from the buffer, the window size increases. It is possible for the window to reach 0: when TCP's receive buffer for a socket is full and it must wait for the application to read data from the buffer before it can take any more data from the peer.

Finally, a TCP connection is **full-duplex**. This means that an application can send and receive data in both directions on a given connection at any time. This means that TCP must keep track of state information such as sequence numbers and window sizes for each direction of data flow: sending and receiving. After a full-duplex connection is established, it can be turned into a simplex connection if desired.

2. TCP Connection Establishment and Termination

2.1 Three-Way Handshake:

Following scenario occurs when a TCP connection is established:

1. The server must be prepared to accept an incoming connection. This is normally done by calling socket, bind, and listen and is called a passive open.
2. The client issues an active open by calling connect. This causes the client TCP to send a "synchronize" (SYN) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options (which we will talk about shortly).
3. The server must acknowledge (ACK) the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the client's SYN in a single segment.
4. The client must acknowledge the server's SYN.

The minimum number of packets required for this exchange is three; hence, this is called TCP's three-way handshake. We show the three segments in Figure 1

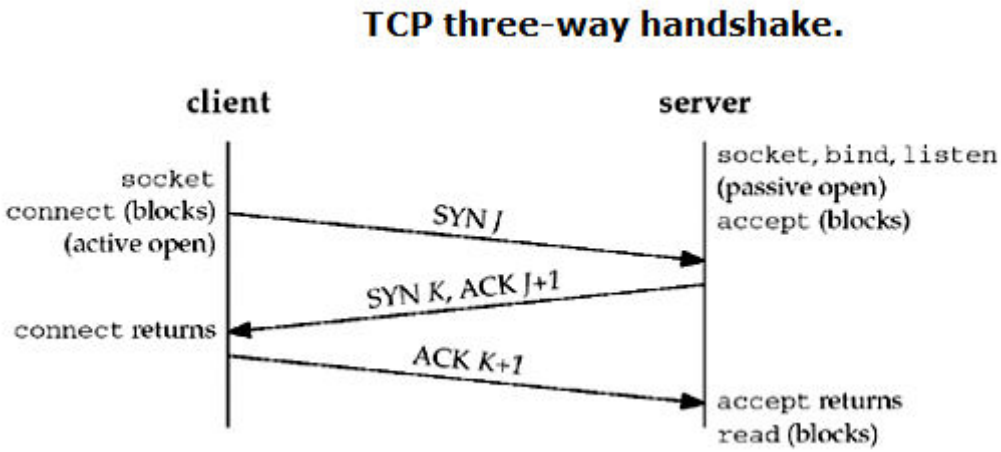


Figure 1

TCP three-way handshake.

We show the client's initial sequence number as J and the server's initial sequence number as K. The acknowledgment number in an ACK is the next expected sequence number for the end sending the ACK. Since a SYN occupies one byte of the sequence number space, the acknowledgment number in the ACK of each SYN is the initial sequence number plus one. Similarly, the ACK of each FIN is the sequence number of the FIN plus one.

An everyday analogy for establishing a TCP connection is the telephone system. The **socket** function is the equivalent of having a telephone to use. **bind** is telling other people your telephone number so that they can call you. **listen** is turning on the ringer so that you will hear when an incoming call arrives. **connect** requires that we know the other person's phone number and dial it. **accept** is when the person being called answers the phone. Having the client's identity returned by accept (where the identify is the client's IP address and port number) is similar to having the caller ID feature show the caller's phone number. One difference, however, is that accept returns the client's identity only after the connection has been established, whereas the caller ID feature shows the caller's phone number before we choose whether to answer the phone or not.

2.2. TCP Connection Termination

While it takes three segments to establish a connection, it takes four to terminate a connection.

1. One application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file (after any data that may have already been queued for the application to receive), since the receipt of the FIN means the application will not receive any additional data on the connection.
3. Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.

4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.

Since a FIN and an ACK are required in each direction, four segments are normally required. We use the qualifier "normally" because in some scenarios, the FIN in Step 1 is sent with data. Also, the segments in Steps 2 and 3 are both from the end performing the passive close and could be combined into one segment. We show these packets in Figure 2.

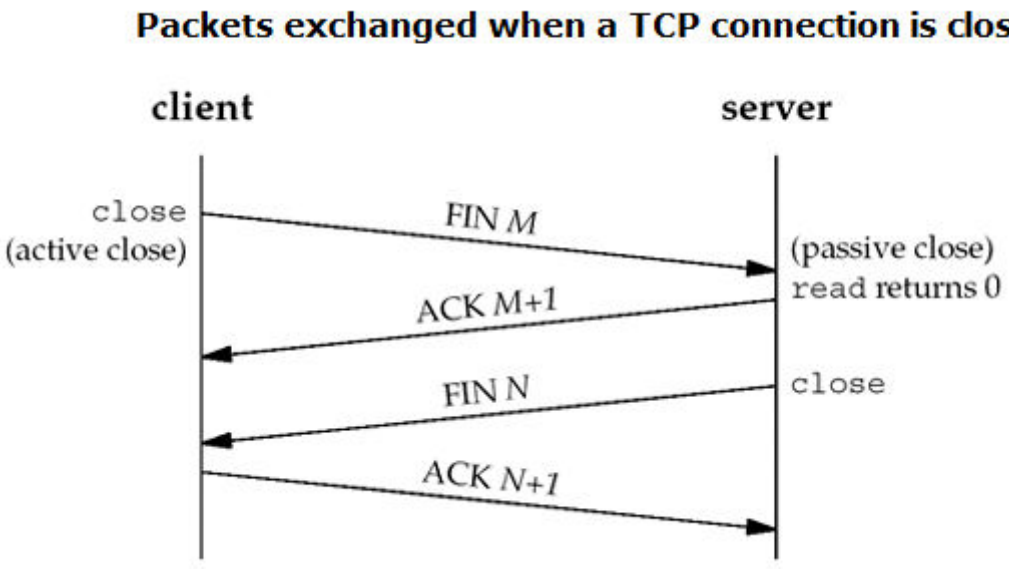


Figure 2 Packets exchanged when a TCP connection is closed.

A FIN occupies one byte of sequence number space just like a SYN. Therefore, the ACK of each FIN is the sequence number of the FIN plus one.

Between Steps 2 and 3 it is possible for data to flow from the end doing the passive close to the end doing the active close. This is called a half open connection.

The sending of each FIN occurs when a socket is closed. We indicated that the application calls close for this to happen, but realize that when a Unix process terminates, either voluntarily (calling exit or having the main function return) or involuntarily (receiving a signal that terminates the process), all open descriptors are closed, which will also cause a FIN to be sent on any TCP connection that is still open.

Although we show the client in Figure 2 performing the active close, either end—the client or the server—can perform the active close. Often the client performs the active close, but with some protocols (notably HTTP), the server performs the active close.

Steps for obtaining credit for this lab.

Lab 12: Analysis of UDP and TCP packets in Wireshark

Do the following:

1. ***Download*** *files TCPCient.py and TCPServer.py from your LMS site.*

2. ***Edit*** *the client file to change two things. The serverIP address and the message; as your name.*

3. ***Start up the Wireshark software.***

4. ***Begin packet capture,*** *select the Capture pull down menu and select Options.*

5. ***Selecting the network interface on which packets would be captured:*** *You can use most of the default values in this window. The network interfaces (i.e., the physical connections) that your computer has to the network will be shown in the Interface pull down menu at the top of the Capture Options window. Click Start. Packet capture will now begin*

6. ***Run your TCPServer and your edited TCPClient.***

7. ***Stopping the capture and inspecting captured packets:*** *After you have received a message, stop Wireshark packet capture*

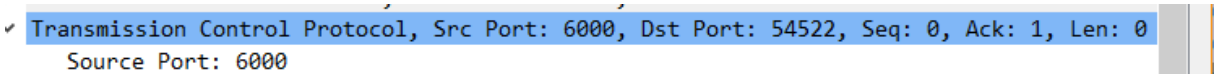
8. ***Filtering:*** *Filter the TCP packets.*

9. ***Details of a packet:*** *Select the TCP messages shown in the packet-listing window and analyze by looking into the detail of packets pane and answer the questions given at the end of this document.*

Task 2 Questions:

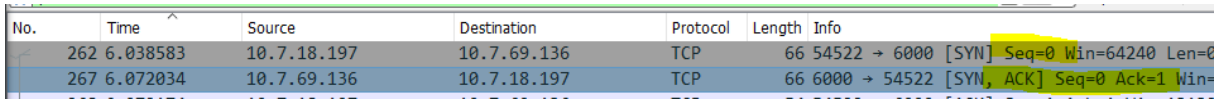
```
(OpenCV-master-py3) C:\Users\G3NZ\Desktop\Lab12\tcp clientserver>python TCPClient.py
b'Welcome: Hasnain Naeem Mon Dec  9 10:25:33 2019'
```

1. **What are the IP addresses and TCP port numbers used by the client and the server?**
   Server IP & Port: 10.7.69.136:6000
   Client IP & Port: 10.7.18.197:54522

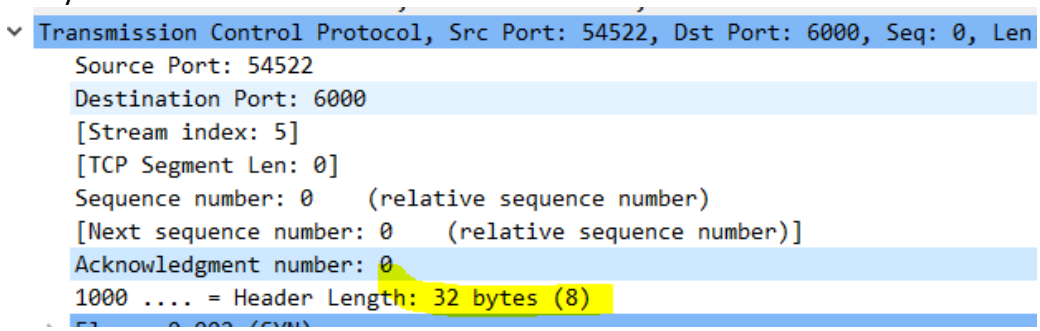


2. **What are the SEQ and ACK Nos of the TCP SYN segment that is used to initiate the TCP connection between the client and server?**
   Seq #: 0
   Ack #: 1

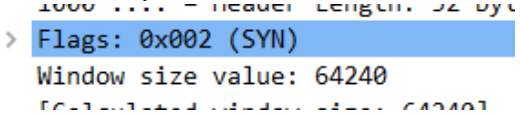
Lab 12: Analysis of UDP and TCP packets in Wireshark



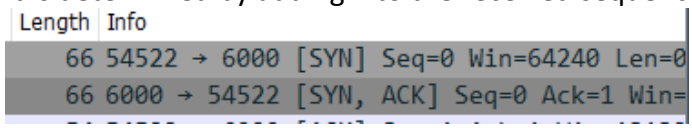
3. **What is the header length of TCP used for this connection?**
   32 bytes.



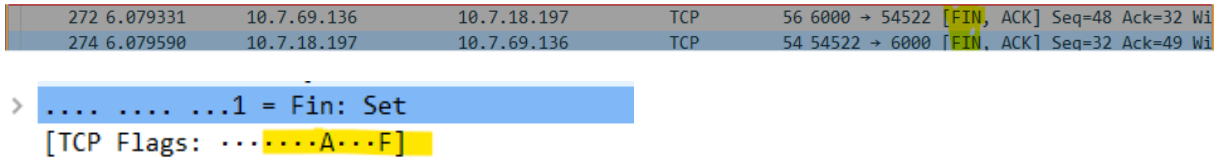
4. **What is the minimum amount of available buffer space advertised by the client and the server?**
   64240 bytes.



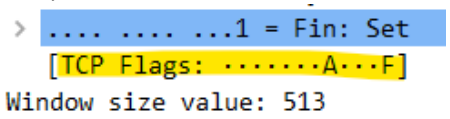
5. **What is the sequence number of the SYNACK segment sent by server to the client computer in reply to the SYN? What is the value of the ACK field in the SYNACK segment? How did the server determine that value?**
   SYNACK segment ACK number: 1
   It is determined by adding 1 to the received sequence number, which was 0.



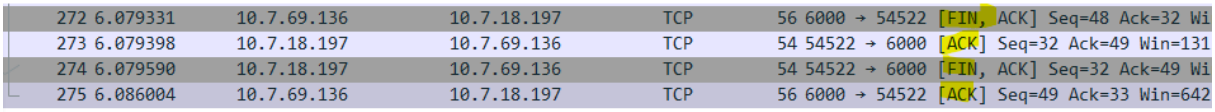
6. **Who has done the active close? Client or the server? How you have determined this?**
   Initially client closed the connection.



   Then, server also closed its tunnel.



7. **What type of closure has been performed? 3 segment (FIN/FIN-ACK/ACK) or four segments (FIN/ACK/FIN/ACK) or simultaneous close?**
   Four segment closure (FIN/ACK/FIN/ACK).



8. **What are SEQ and ACK Nos for all the segments used for the connection closure?**

   Segment # 1 (client to server)

   Seq #: 48

   Ack #: 32

   Segment # 2 (server to client)

   Seq #: 32

   Ack #: 49

   Segment # 3 (server to client)

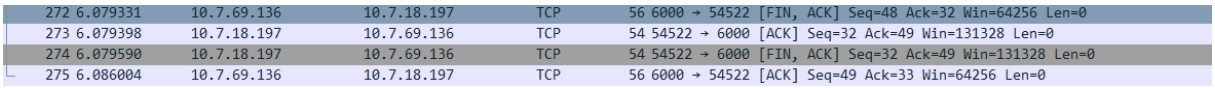
   Seq #: 32

   Ack #: 49

   Segment # 4 (server to client)

   Seq #: 49

   Ack #: 33



9. **How many bytes in total have been transferred from the client to the server and from the server to the client during the whole connection?**

   As the sequence number started from 0. So, it is safe to determine number of bytes transferred from the Ack numbers in the closing segments.

   Number of byte transferred = Ack number – 1

   Bytes transferred are:

   - **Client to server:** 32 (excluding the header bytes)
   - **Server to client:** 48 (excluding the header bytes)

Conclusion:

UDP and TCP protocols are used in the protocol layer. UDP has limited header size and functionality. TCP provides reliable data transferred but it involves more header size and time overhead. Moreover, Wireshark can be used to analyze both TCP & UDP packets.