Exercise 3: Using Wireshark to understand basic HTTP request/response messages (2.5 marks, include in your report)

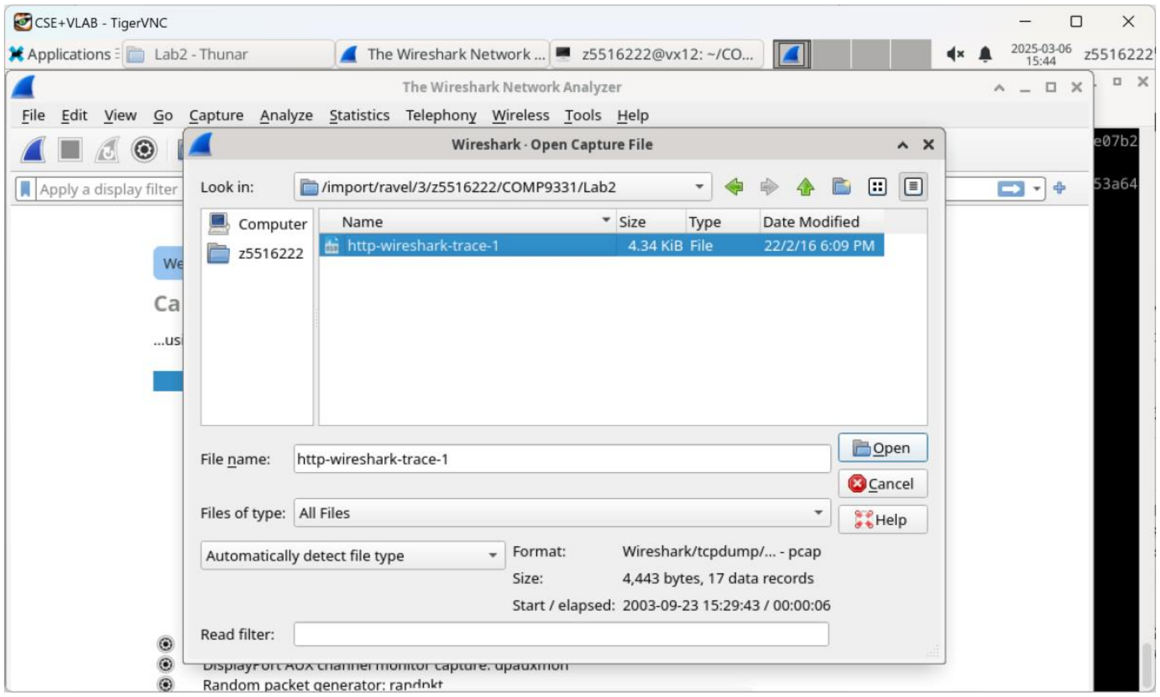
We will not be running Wireshark on a live network connection (You are strongly encouraged to try this on your native machine. Pointers are provided at the end of this exercise). The CSE network administrators do not permit live traffic monitoring for security reasons. Instead, for all our lab exercises,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

NOTE: IT IS NOT POSSIBLE TO RUN WIRESHARK VIA SSH. IT IS A RESOURCE-INTENSIVE PROGRAM, SLOW DOWN THE CSE LOGIN SERVERS. IF YOU WANT TO WORK REMOTELY, THEN MAKE SURE YOU VLAB. WIRESHARK IS AVAILABLE ON ALL LAB MACHINES AND IN VLAB. HOWEVER, IT CANNOT BE COMMAND LINE. INSTEAD, GO TO THE APPLICATION MENU AND SELECT "INTERNET" AND "WIRESH ALSO DOWNLOAD AND INSTALL WIRESHARK ON YOUR PERSONAL MACHINE.

The following indicates the steps involved:

Step 1: Start Wireshark natively on your machine or through VLAB, as noted above.

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


CSE+VLAB - TigerVNC
Applications
Lab2 - Thunar
The Wireshark Network ...
z5516222@vx12: ~/CO...
2025-03-06 15:44
z5516222
The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter
Wireshark · Open Capture File
Computer
z5516222
Open
Cancel
Help

Look in: /import/ravel/3/z5516222/COMP9331/Lab2

Name	Size	Type	Date Modified
http-wireshark-trace-1	4.34 KiB	File	22/2/16 6:09 PM

File name: http-wireshark-trace-1

Files of type: All Files

Automatically detect file type

Format: Wireshark/tcpdump/... - pcap

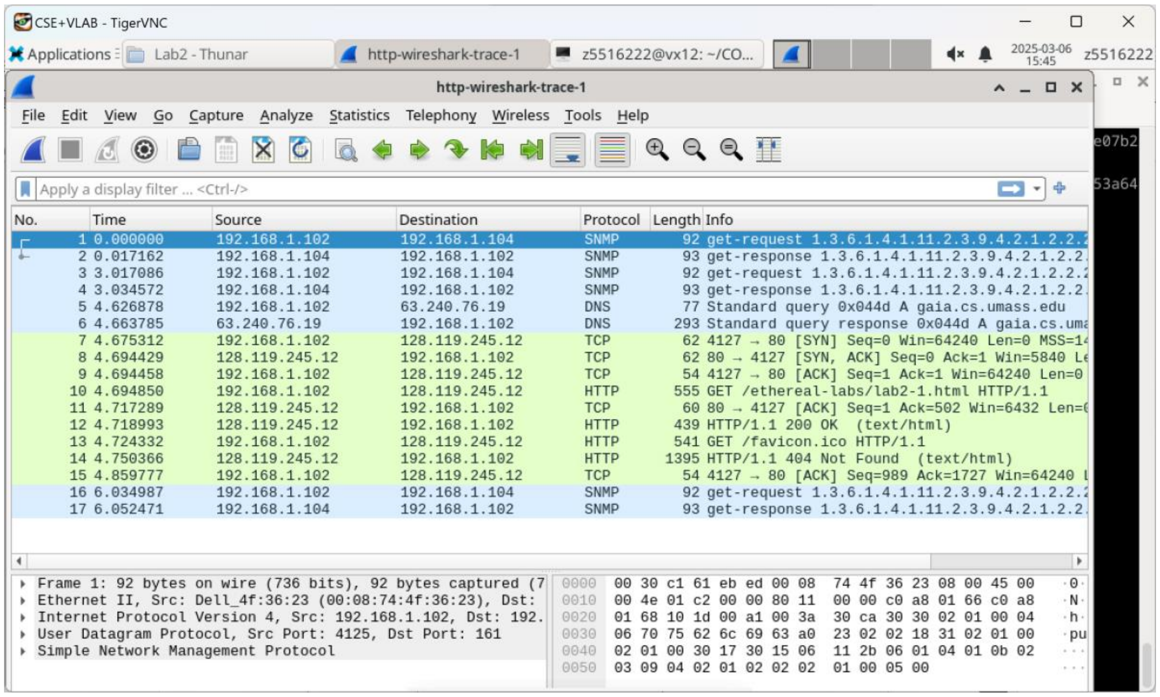
Size: 4,443 bytes, 17 data records

Start / elapsed: 2003-09-23 15:29:43 / 00:00:06

Read filter:

DisplayPort AUX channel monitor capture: dpauxmon

Random packet generator: randpkt

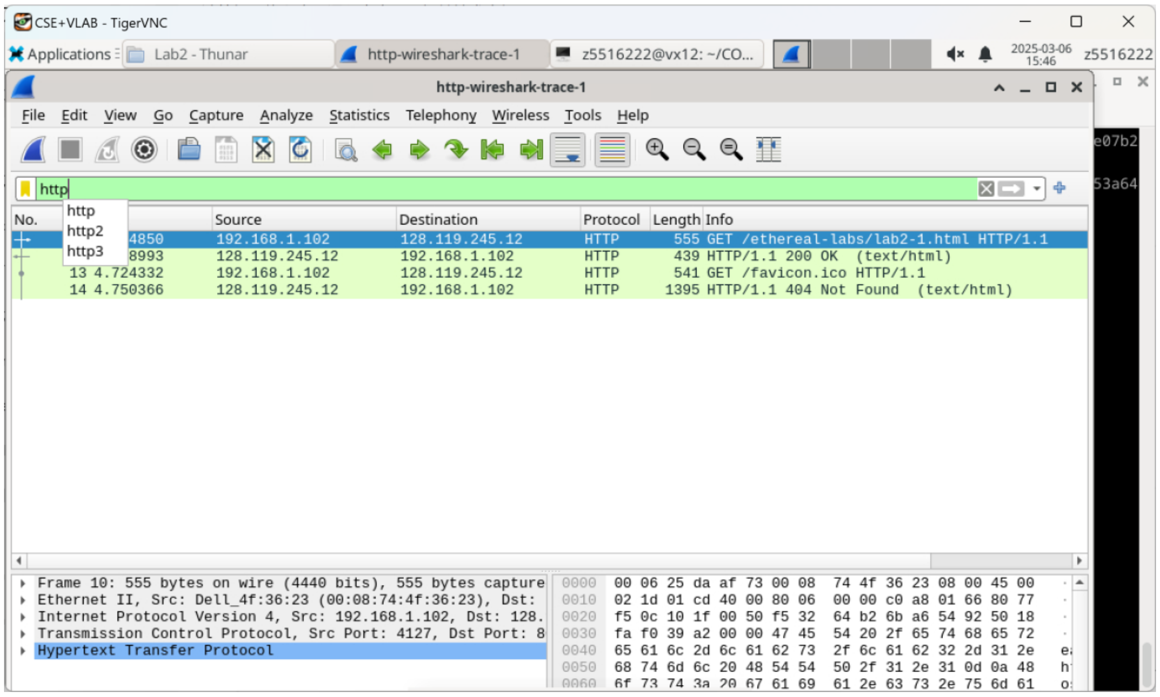


```
No.  Time       Source          Destination     Protocol Length Info
  1 0.000000   192.168.1.102   192.168.1.104   SNMP       92 get-request 1.3.6.1.4.1.11.2.3.9.4.2.1.2.2.
  2 0.017162   192.168.1.104   192.168.1.102   SNMP       93 get-response 1.3.6.1.4.1.11.2.3.9.4.2.1.2.2.
  3 3.017086   192.168.1.102   192.168.1.104   SNMP       92 get-request 1.3.6.1.4.1.11.2.3.9.4.2.1.2.2.
  4 3.034572   192.168.1.104   192.168.1.102   SNMP       93 get-response 1.3.6.1.4.1.11.2.3.9.4.2.1.2.2.
  5 4.626878   192.168.1.102   63.240.76.19    DNS        77 Standard query 0x044d A gaia.cs.umass.edu
  6 4.663785   63.240.76.19    192.168.1.102   DNS       293 Standard query response 0x044d A gaia.cs.uma
  7 4.675312   192.168.1.102   128.119.245.12  TCP        62 4127 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=14
  8 4.694429   128.119.245.12  192.168.1.102   TCP        62 80 → 4127 [SYN, ACK] Seq=0 Ack=1 Win=5840 Le
  9 4.694458   192.168.1.102   128.119.245.12  TCP        54 4127 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
 10 4.694850   192.168.1.102   128.119.245.12  HTTP      555 GET /ethereal-labs/lab2-1.html HTTP/1.1
 11 4.717289   128.119.245.12  192.168.1.102   TCP        60 80 → 4127 [ACK] Seq=1 Ack=502 Win=6432 Len=
 12 4.718993   128.119.245.12  192.168.1.102   HTTP      439 HTTP/1.1 200 OK  (text/html)
 13 4.724332   192.168.1.102   128.119.245.12  HTTP      541 GET /favicon.ico HTTP/1.1
 14 4.750366   128.119.245.12  192.168.1.102   HTTP     1395 HTTP/1.1 404 Not Found  (text/html)
 15 4.859777   192.168.1.102   128.119.245.12  TCP        54 4127 → 80 [ACK] Seq=989 Ack=1727 Win=64240 L
 16 6.034987   192.168.1.102   192.168.1.104   SNMP       92 get-request 1.3.6.1.4.1.11.2.3.9.4.2.1.2.2.
 17 6.052471   192.168.1.104   192.168.1.102   SNMP       93 get-response 1.3.6.1.4.1.11.2.3.9.4.2.1.2.2.
```

```
› Frame 1: 92 bytes on wire (736 bits), 92 bytes captured (7
› Ethernet II, Src: Dell_4f:36:23 (00:08:74:4f:36:23), Dst:
› Internet Protocol Version 4, Src: 192.168.1.102, Dst: 192.
› User Datagram Protocol, Src Port: 4125, Dst Port: 161
› Simple Network Management Protocol
```

```
0000  00 30 c1 61 eb ed 00 08  74 4f 36 23 08 00 45 00   ·0·
0010  00 4e 01 c2 00 00 80 11  00 00 c0 a8 01 66 c0 a8   ·N·
0020  01 68 10 1d 00 a1 00 3a  30 ca 30 30 02 01 00 04   ·h·
0030  06 70 75 62 6c 69 63 a0  23 02 02 18 31 02 01 00   ·pu
0040  02 01 00 30 17 30 15 06  11 2b 06 01 04 01 0b 02   ···
0050  03 09 04 02 01 02 02 02  01 00 05 00               ···
```

Step 3: You will see many packets in the packet listing window. Since we are currently only interested in HTTP, we will filter out all the other packets by typing "http" in lowercase in the *Filter* field and pressing Enter. You should now see only HTTP packets in the packet-listing window.

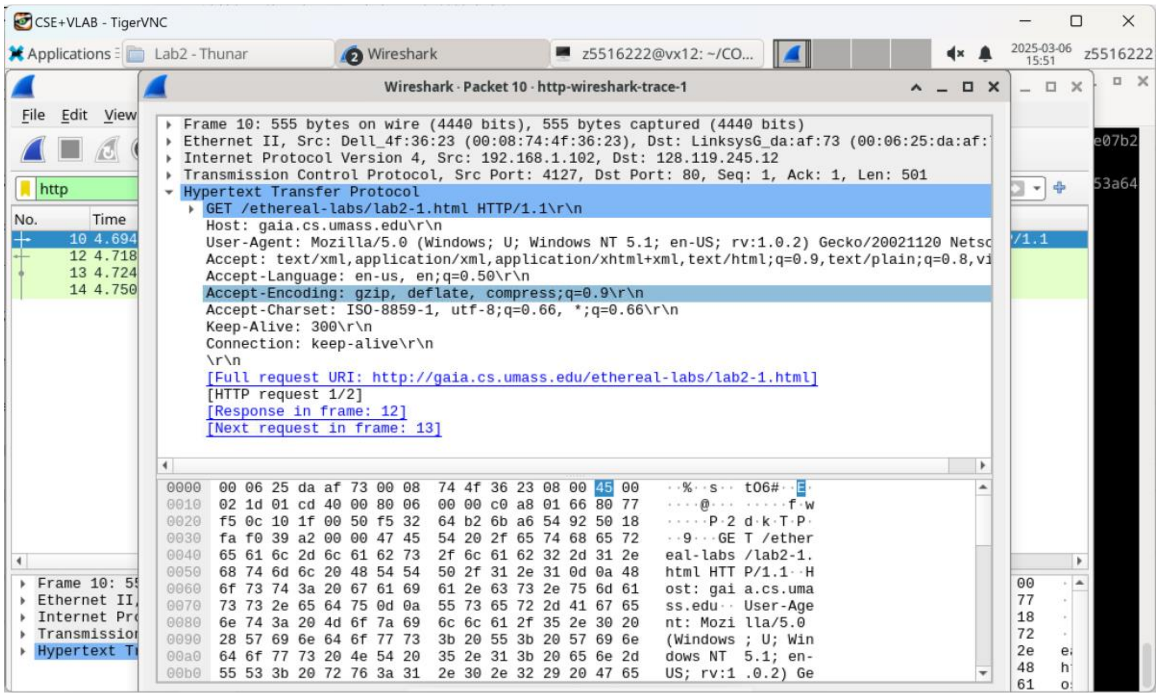


```
No.  Time       Source          Destination     Protocol Length Info
     4850      192.168.1.102   128.119.245.12  HTTP      555 GET /ethereal-labs/lab2-1.html HTTP/1.1
     8993      128.119.245.12  192.168.1.102   HTTP      439 HTTP/1.1 200 OK  (text/html)
 13 4.724332   192.168.1.102   128.119.245.12  HTTP      541 GET /favicon.ico HTTP/1.1
 14 4.750366   128.119.245.12  192.168.1.102   HTTP     1395 HTTP/1.1 404 Not Found  (text/html)
```

```
› Frame 10: 555 bytes on wire (4440 bits), 555 bytes capture
› Ethernet II, Src: Dell_4f:36:23 (00:08:74:4f:36:23), Dst:
› Internet Protocol Version 4, Src: 192.168.1.102, Dst: 128.
› Transmission Control Protocol, Src Port: 4127, Dst Port: 8
› Hypertext Transfer Protocol
```

```
0000  00 06 25 da af 73 00 08  74 4f 36 23 08 00 45 00   ·
0010  02 1d 01 cd 40 00 80 06  00 00 c0 a8 01 66 80 77   ·
0020  f5 0c 10 1f 00 50 f5 32  64 b2 6b a6 54 92 50 18   ·
0030  fa f0 39 a2 00 00 47 45  54 20 2f 65 74 68 65 72   ·
0040  65 61 6c 2d 6c 61 62 73  2f 6c 61 62 32 2d 31 2e   e
0050  68 74 6d 6c 20 48 54 54  50 2f 31 2e 31 0d 0a 48   h
0060  6f 73 74 3a 20 67 61 69  61 2e 63 73 2e 75 6d 61   o
```

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



```
10 4.694
12 4.718
13 4.724
14 4.750
```

```
▸ Frame 10: 555 bytes on wire (4440 bits), 555 bytes captured (4440 bits)
▸ Ethernet II, Src: Dell_4f:36:23 (00:08:74:4f:36:23), Dst: LinksysG_da:af:73 (00:06:25:da:af:
▸ Internet Protocol Version 4, Src: 192.168.1.102, Dst: 128.119.245.12
▸ Transmission Control Protocol, Src Port: 4127, Dst Port: 80, Seq: 1, Ack: 1, Len: 501
▾ Hypertext Transfer Protocol
  ▸ GET /ethereal-labs/lab2-1.html HTTP/1.1\r\n
    Host: gaia.cs.umass.edu\r\n
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0.2) Gecko/20021120 Netsc
    Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,vi
    Accept-Language: en-us, en;q=0.50\r\n
    Accept-Encoding: gzip, deflate, compress;q=0.9\r\n
    Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66\r\n
    Keep-Alive: 300\r\n
    Connection: keep-alive\r\n
    \r\n
    [Full request URI: http://gaia.cs.umass.edu/ethereal-labs/lab2-1.html]
    [HTTP request 1/2]
    [Response in frame: 12]
    [Next request in frame: 13]
```

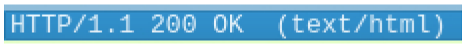
```
0000  00 06 25 da af 73 00 08  74 4f 36 23 08 00 45 00   ··%··s·· tO6#··E·
0010  02 1d 01 cd 40 00 80 06  00 00 c0 a8 01 66 80 77   ····@··· ·····f·w
0020  f5 0c 10 1f 00 50 f5 32  64 b2 6b a6 54 92 50 18   ·····P·2 d·k·T·P·
0030  fa f0 39 a2 00 00 47 45  54 20 2f 65 74 68 65 72   ··9···GE T /ether
0040  65 61 6c 2d 6c 61 62 73  2f 6c 61 62 32 2d 31 2e   eal-labs /lab2-1.
0050  68 74 6d 6c 20 48 54 54  50 2f 31 2e 31 0d 0a 48   html HTT P/1.1··H
0060  6f 73 74 3a 20 67 61 69  61 2e 63 73 2e 75 6d 61   ost: gai a.cs.uma
0070  73 73 2e 65 64 75 0d 0a  55 73 65 72 2d 41 67 65   ss.edu·· User-Age
0080  6e 74 3a 20 4d 6f 7a 69  6c 6c 61 2f 35 2e 30 20   nt: Mozi lla/5.0 
0090  28 57 69 6e 64 6f 77 73  3b 20 55 3b 20 57 69 6e   (Windows ; U; Win
00a0  64 6f 77 73 20 4e 54 20  35 2e 31 3b 20 65 6e 2d   dows NT  5.1; en-
00b0  55 53 3b 20 72 76 3a 31  2e 30 2e 32 29 20 47 65   US; rv:1 .0.2) Ge
```

```
▸ Frame 10: 55
▸ Ethernet II,
▸ Internet Pro
▸ Transmission
▸ Hypertext Tr
```

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

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

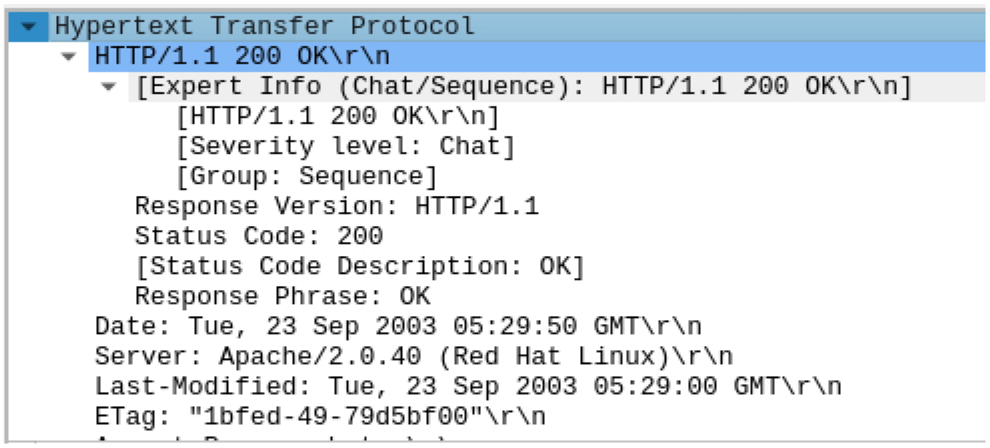
Question 1: What is the status code and phrase returned from the server to the client browser?



```
HTTP/1.1 200 OK  (text/html)
```

Answer 1: "HTTP/1.1 200 OK".

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



```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
    [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      [HTTP/1.1 200 OK\r\n]
      [Severity level: Chat]
      [Group: Sequence]
    Response Version: HTTP/1.1
    Status Code: 200
    [Status Code Description: OK]
    Response Phrase: OK
  Date: Tue, 23 Sep 2003 05:29:50 GMT\r\n
  Server: Apache/2.0.40 (Red Hat Linux)\r\n
  Last-Modified: Tue, 23 Sep 2003 05:29:00 GMT\r\n
  ETag: "1bfed-49-79d5bf00"\r\n
```

Answer 2:

Date: Tue, 23 Sep 2003 05:29:50 GMT

Last-Modified: Tue, 23 Sep 2003 05:29:50 GMT

Difference:

Last-Modified: Reflects the file's modification time on the server.

Date: Reflects the time the server sent the response.

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

Answer 3: The connection is persistent. This can be inferred from the request header "Connection: keep-alive" in the image.

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

Answer 4: TCP payload 385 bytes.

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

Answer 5: The data is the HTML content of the webpage.

Note: Students are strongly encouraged to use Wireshark to capture real network traffic on their own machines. Check https://www.wireshark.org/download.html for details. Once you have Wireshark installed, do the following. Clear your web browser's cache (Firefox->Tools->Clear Recent History). Launch the Wireshark tool by typing Wireshark in the command line. Start Wireshark capture by clicking on capture -> interfaces -> click Start on the interface eth0. Run the Web browser and enter a URL for a website (e.g. www.bbc.co.uk). Stop capturing packets when the web page is fully loaded. Examine the captured trace and answer the questions above. This is just for you to try in your own time. You do not have to include this in your report.

Exercise 4: Using Wireshark to understand the HTTP CONDITIONAL GET/response interaction (2.5 marks, include in your report)

For this particular experiment, download the second trace file: http-wireshark-trace-2

The following indicates the steps for this experiment:

Step 1: Start Wireshark natively on your machine or through VLAB, as noted above.

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

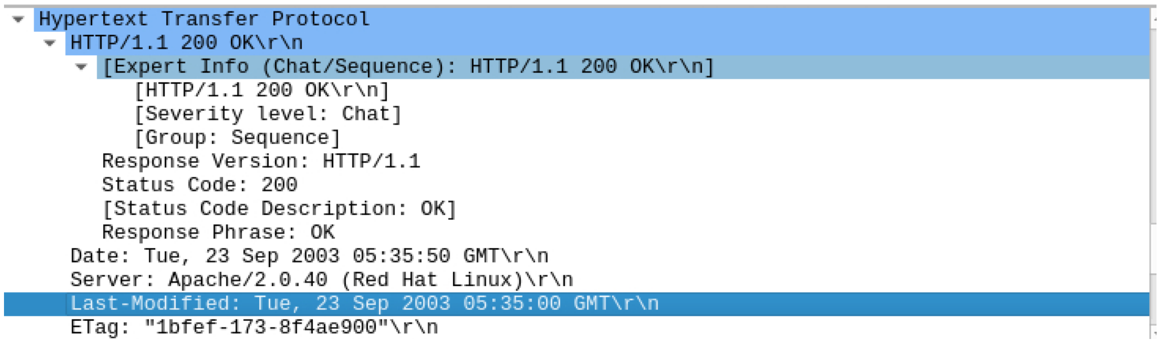
Step 3: Filter out all the non-HTTP packets and focus on the HTTP header information in the packet-header detail window.

We will focus on the first two GET requests and the corresponding responses (the first 4 HTTP messages).

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

Answer 1: No, the first HTTP GET request does not contain an "IF-MODIFIED-SINCE" line.

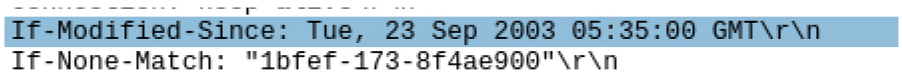
Question 2: Does the HTTP response from the server indicate the last time the requested file was modified?



```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
    [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      [HTTP/1.1 200 OK\r\n]
      [Severity level: Chat]
      [Group: Sequence]
    Response Version: HTTP/1.1
    Status Code: 200
    [Status Code Description: OK]
    Response Phrase: OK
  Date: Tue, 23 Sep 2003 05:35:50 GMT\r\n
  Server: Apache/2.0.40 (Red Hat Linux)\r\n
  Last-Modified: Tue, 23 Sep 2003 05:35:00 GMT\r\n
  ETag: "1bfef-173-8f4ae900"\r\n
```

Answer 2: Yes, the HTTP response from the server includes a "Last-Modified" header, indicating the last time the requested file was modified.

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



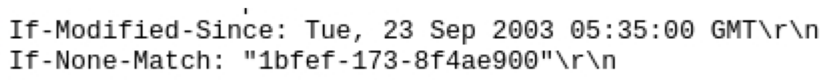
```
If-Modified-Since: Tue, 23 Sep 2003 05:35:00 GMT\r\n
If-None-Match: "1bfef-173-8f4ae900"\r\n
```

Answer 3:

If-Modified-Since: Tue, 23 Sep 2003 05:35:00 GMT

If-None-Match: "1bfef-173-8f4ae900"

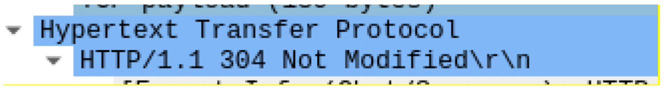
Question 4: What is the HTTP status code and phrase returned from the server in response to this second HTTP GET? Did the server explicitly return the file's contents? Explain.



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

Answer 4: Yes.

Question 5: What is the value of the Etag field in the 2nd response message, and how is it used? Is the Etag value the same as in the 1 st response?


```
Hypertext Transfer Protocol
  HTTP/1.1 304 Not Modified\r\n
```

Answer 5: HTTP/1.1 304 Not Modified. It is not the same as the 1st response.

Exercise 5: Ping Client (5 marks, submit source code as a separate file, include sample output in the report)

Objective:

In this exercise, you will implement a Ping client with enhanced functionality using UDP, similar to the standard Ping tool. You will calculate Round-Trip Time (RTT) and report additional statistics such as the percentage of packets acknowledged, total transmission time, and jitter. The server, `PingServer.java` , is provided and simulates network conditions by introducing packet loss and delays.

Ping Server (Provided):

You are given the code for the Ping server (`PingServer.java`). The server listens for UDP packets and responds with the same data it receives. However, the server may drop packets or delay responses based on predefined parameters.

You need to first compile the Ping Server using the following.

```
javac PingServer.java
```

Then you should be able to run it using the following. Make sure to run the server in a seperate terminal before starting the client.

```
java PingServer <whatever port you choose>
```

Your Task: Implementing Ping Client

You will implement a client (`PingClient.java, PingClient.c, or PingClient.py`) that performs the following tasks:

1. Send 15 ping requests to the server.
2. Each ping request should include:
    - The keyword `PING` .
    - A sequence number starting from a random number between 10,000 and 20,000.
    - A timestamp (any timestamp format should suffice) indicating when the message was sent.
3. Wait up to 600 ms for a response from the server. If no response is received within this time, record it as a timeout (packet lost).
4. Report the following statistics:
    - RTT for each successful ping.
    - **Minimum RTT** , **Maximum RTT** , and **Average RTT** .

   - **Percentage of packets acknowledged** (i.e., received a response).
   - **Total transmission time** (from the first sent packet to the last received response or timeout).
   - **Jitter** (variance in RTT between successive packets).

Example Command to Run the Client:

Java

```
$javac PingClient.java

$java PingClient <host> <port>
```

C

```
$gcc -o PingClient PingClient.c

$./PingClient <host> <port>
```

Python

```
$python3 PingClient.py <host> <port>
```

Where `<host>` is the IP address of the server (use 127.0.0.1 for localhost) and <port> is the port number on which the server is listening.

Detailed Reporting:

The client must report the following at the end of execution:

- **RTT** for each ping request (or "timeout" if no response).
- **Minimum RTT, Maximum RTT** , and **Average RTT** for the successful pings.
- Percentage of **packets acknowledged** (i.e., how many responses were received out of 15 pings).
- **Total transmission time** (the time from sending the first ping to receiving the last response).
- **Jitter** , which measures the variance in RTT values between successive pings.
  (Jitter = Sum of (|RTT(n) - RTT(n-1)|) / (Number of packets received - 1))

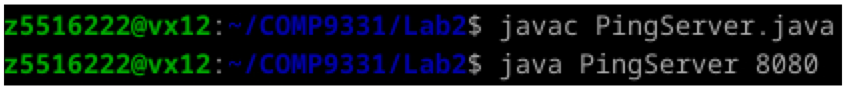
```
PING to 127.0.0.1, seq=10215, rtt=120 ms

PING to 127.0.0.1, seq=10216, rtt=200 ms

PING to 127.0.0.1, seq=10217, rtt=timeout

PING to 127.0.0.1, seq=10218, rtt=150 ms

...

Total packets sent: 15

Packets acknowledged: 12

Packet loss: 20%
```

```
Minimum RTT: 50 ms, Maximum RTT: 200 ms, Average RTT: 120 ms

Total transmission time: 2000 ms

Jitter: 25 ms
```

Marking Criteria (5 Marks):

1. **Correct implementation of the ping protocol and handling of packet loss (2.5 marks):**
   - This includes correct handling of packet transmission, timeout, and managing packet loss as per the server's behavior.
2. **Accurate calculation and display of RTT for each successful ping (1.5 marks):**
   - RTT should be calculated accurately for every ping request that receives a response.
3. **Proper reporting of packet acknowledgment percentage, total transmission time, and jitter (1 mark):**
   - Includes accurate reporting of additional statistics like packet acknowledgment percentage, total transmission time, and jitter.

Answer:



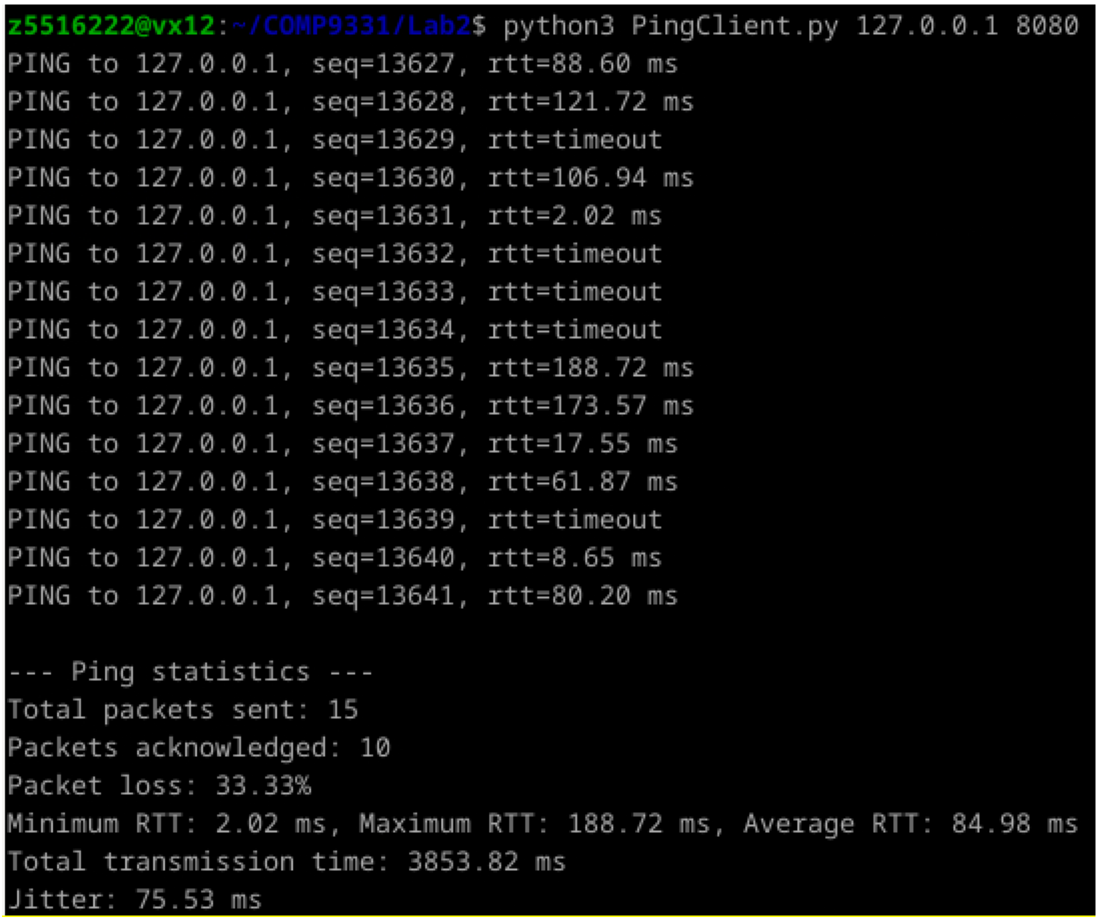
```
z5516222@vx12:~/COMP9331/Lab2$ javac PingServer.java
z5516222@vx12:~/COMP9331/Lab2$ java PingServer 8080
```

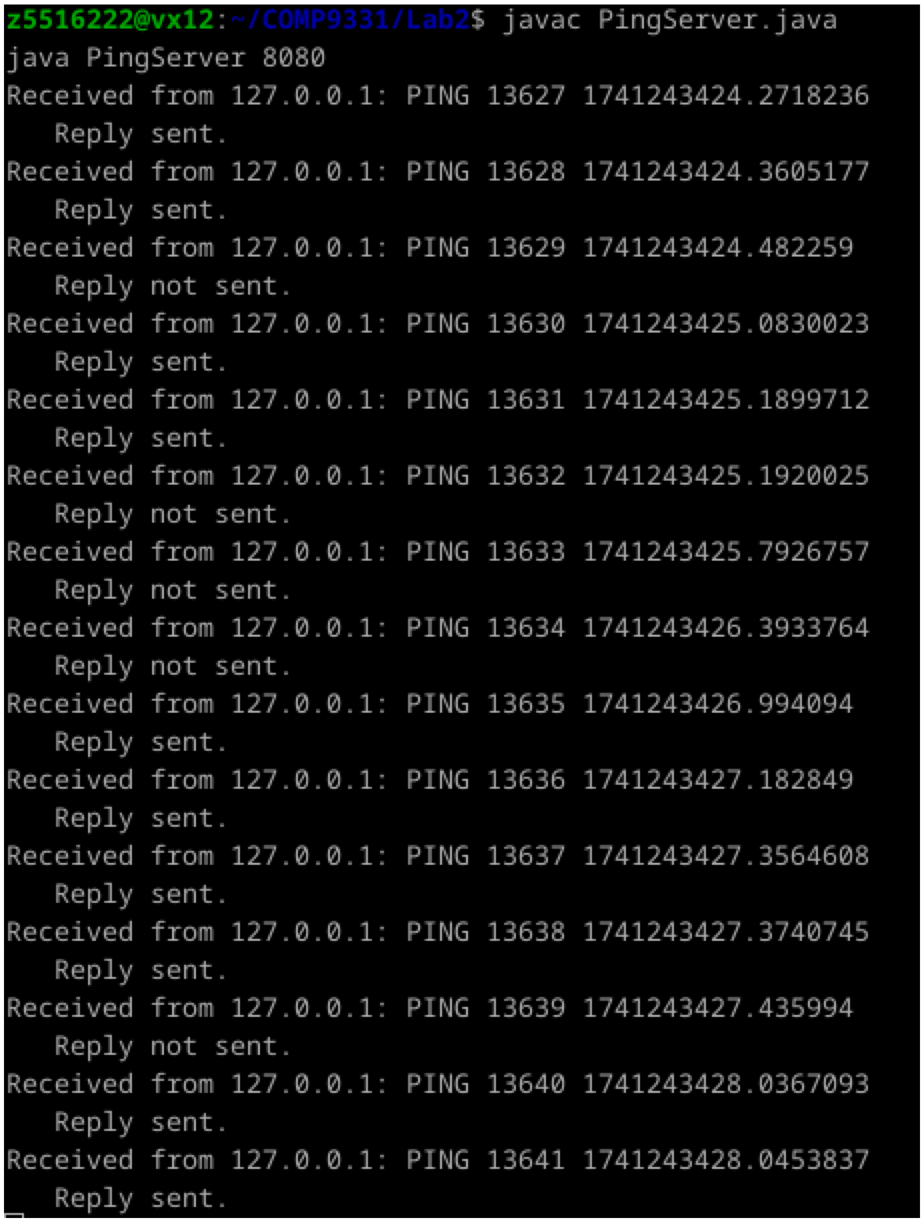
Code of PingClient.py:

```
import socket

import time

import random

import sys



if len(sys.argv) != 3:

    print("Usage: python3 PingClient.py <host> <port>")

    sys.exit(1)



host = sys.argv[1]

port = int(sys.argv[2])
```

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client_socket.settimeout(0.6)  # 设置超时时间为 600ms

total_packets = 15
seq_start = random.randint(10000, 20000)
rtt_list = []
packets_acknowledged = 0
start_time = time.time()

for i in range(total_packets):
    seq_num = seq_start + i
    send_time = time.time()
    message = f"PING {seq_num} {send_time}"
    client_socket.sendto(message.encode(), (host, port))

    try:
        response, server_address = client_socket.recvfrom(1024)
        recv_time = time.time()
        rtt = (recv_time - send_time) * 1000  # 转换为毫秒
        rtt_list.append(rtt)
        packets_acknowledged += 1
        print(f"PING to {host}, seq={seq_num}, rtt={rtt:.2f} ms")
    except socket.timeout:
        print(f"PING to {host}, seq={seq_num}, rtt=timeout")
```

```
end_time = time.time()

total_transmission_time = (end_time - start_time) * 1000  # 转换为毫秒

packet_loss = (total_packets - packets_acknowledged) / total_packets * 100


min_rtt = min(rtt_list) if rtt_list else 0

max_rtt = max(rtt_list) if rtt_list else 0

avg_rtt = sum(rtt_list) / len(rtt_list) if rtt_list else 0


jitter = 0

if len(rtt_list) > 1:

    jitter = sum(abs(rtt_list[i] - rtt_list[i - 1]) for i in range(1, len(rtt_list))) / (len(rtt_list) - 1)


print("\n--- Ping statistics ---")

print(f"Total packets sent: {total_packets}")

print(f"Packets acknowledged: {packets_acknowledged}")

print(f"Packet loss: {packet_loss:.2f}%")

print(f"Minimum RTT: {min_rtt:.2f} ms, Maximum RTT: {max_rtt:.2f} ms, Average RTT:
{avg_rtt:.2f} ms")

print(f"Total transmission time: {total_transmission_time:.2f} ms")

print(f"Jitter: {jitter:.2f} ms")

client_socket.close()
```


```
z5516222@vx12:~/COMP9331/Lab2$ python3 PingClient.py 127.0.0.1 8080
PING to 127.0.0.1, seq=13627, rtt=88.60 ms
PING to 127.0.0.1, seq=13628, rtt=121.72 ms
PING to 127.0.0.1, seq=13629, rtt=timeout
PING to 127.0.0.1, seq=13630, rtt=106.94 ms
PING to 127.0.0.1, seq=13631, rtt=2.02 ms
PING to 127.0.0.1, seq=13632, rtt=timeout
PING to 127.0.0.1, seq=13633, rtt=timeout
PING to 127.0.0.1, seq=13634, rtt=timeout
PING to 127.0.0.1, seq=13635, rtt=188.72 ms
PING to 127.0.0.1, seq=13636, rtt=173.57 ms
PING to 127.0.0.1, seq=13637, rtt=17.55 ms
PING to 127.0.0.1, seq=13638, rtt=61.87 ms
PING to 127.0.0.1, seq=13639, rtt=timeout
PING to 127.0.0.1, seq=13640, rtt=8.65 ms
PING to 127.0.0.1, seq=13641, rtt=80.20 ms

--- Ping statistics ---
Total packets sent: 15
Packets acknowledged: 10
Packet loss: 33.33%
Minimum RTT: 2.02 ms, Maximum RTT: 188.72 ms, Average RTT: 84.98 ms
Total transmission time: 3853.82 ms
Jitter: 75.53 ms
```



```
z5516222@vx12:~/COMP9331/Lab2$ javac PingServer.java
java PingServer 8080
Received from 127.0.0.1: PING 13627 1741243424.2718236
   Reply sent.
Received from 127.0.0.1: PING 13628 1741243424.3605177
   Reply sent.
Received from 127.0.0.1: PING 13629 1741243424.482259
   Reply not sent.
Received from 127.0.0.1: PING 13630 1741243425.0830023
   Reply sent.
Received from 127.0.0.1: PING 13631 1741243425.1899712
   Reply sent.
Received from 127.0.0.1: PING 13632 1741243425.1920025
   Reply not sent.
Received from 127.0.0.1: PING 13633 1741243425.7926757
   Reply not sent.
Received from 127.0.0.1: PING 13634 1741243426.3933764
   Reply not sent.
Received from 127.0.0.1: PING 13635 1741243426.994094
   Reply sent.
Received from 127.0.0.1: PING 13636 1741243427.182849
   Reply sent.
Received from 127.0.0.1: PING 13637 1741243427.3564608
   Reply sent.
Received from 127.0.0.1: PING 13638 1741243427.3740745
   Reply sent.
Received from 127.0.0.1: PING 13639 1741243427.435994
   Reply not sent.
Received from 127.0.0.1: PING 13640 1741243428.0367093
   Reply sent.
Received from 127.0.0.1: PING 13641 1741243428.0453837
   Reply sent.
```