**TCP Protocol using Wireshark**

Name: Varshini Maskapuri

Department: OMIS

Course No.:660

Course Name: Business Data Networks & Cybersecurity

Title: Lab Project2: TCP Protocol using Wireshark

Lab No.:2

Date: 09-25-2025

**Lab Summary:**

In this lab, we explored the behavior of the Transmission Control Protocol (TCP) by capturing and analyzing a file transfer using Wireshark. Specifically, we uploaded the text file alice.txt (Lewis Carroll’s Alice’s Adventures in Wonderland) to the server **gaia.cs.umass.edu** via an HTTP POST request over TCP. The packet capture enabled us to investigate critical TCP functionalities, including connection establishment (three-way handshake), sequence and acknowledgment numbers, flow control, congestion control, and retransmission detection.

Through the lab, we observed how TCP segments encapsulate application-layer data, how acknowledgments are generated, and how buffer sizes are managed to prevent overflow. Additionally, we measured round-trip times (RTTs) and analyzed the length of TCP segments to better understand throughput and performance characteristics.

This exercise reinforced the concepts of **reliable data transfer, flow control, congestion avoidance, and performance monitoring** within TCP connections, aligning theory with practical packet-level observations.

**Lab Objectives:**

* To capture and analyze a bulk TCP file transfer using Wireshark.
* To understand TCP’s three-way handshake process (SYN, SYN/ACK, ACK).
* To examine sequence and acknowledgment numbers in TCP segments.
* To investigate TCP’s flow control through advertised window size.
* To measure RTTs and evaluate throughput during file transfer.
* To detect retransmitted segments and understand their impact.

**Lab Implementation:**

**Step 1: Page displaying text of “Alice’s Adventures in Wonderland”**

In this step, we accessed the ASCII text version of Alice’s Adventures in Wonderland from [**http://gaia.cs.umass.edu/wireshark-labs/alice.txt**](http://gaia.cs.umass.edu/wireshark-labs/alice.txt). This file was saved locally on the computer to be later uploaded to the server. Capturing the transfer of this file provided the necessary dataset for analyzing TCP behavior such as sequence numbers, acknowledgments, and flow control.

**Purpose:** To prepare the test file (alice.txt) that will be transmitted over TCP during the HTTP POST operation, enabling a detailed packet-level study of TCP transmission.

A screenshot of a computer

AI-generated content may be incorrect.

**Step 2: Upload Interface for alice.txt**

In this step, we navigated to the **Upload page (TCP-wireshark-file1.html)** provided by gaia.cs.umass.edu. This page allows the user to browse and select the **alice.txt** file stored locally. The purpose of this interface is to initiate an **HTTP POST request** that transfers the file to the remote server over a TCP connection.

**Purpose:** To provide the mechanism for uploading the file, thereby triggering the TCP file transfer which can then be captured and analyzed in Wireshark.

A screenshot of a computer

AI-generated content may be incorrect.

**Step 3: Capturing the HTTP POST Request**

Using Wireshark, we captured the **HTTP POST request** that transferred alice.txt to the server gaia.cs.umass.edu. The capture shows key headers such as **Content-Length (152,351 bytes)**, **Content-Type (multipart/form-data)**, and the TCP segmentation of the payload. The POST message was reassembled from multiple TCP segments, demonstrating how TCP reliably handles large data transfers.

**Purpose:** To analyze how the application-layer POST request is broken into multiple TCP segments, providing insight into TCP’s reliable data transfer mechanisms.

A screenshot of a computer

AI-generated content may be incorrect.

**Step 4: Selecting alice.txt for Upload**

At this stage, the **alice.txt** file was selected from the local system using the upload interface. By choosing the file, the system prepared to initiate an **HTTP POST request** that would transmit the file to the destination server gaia.cs.umass.edu.

**Purpose:** To queue the file for transfer over a TCP connection, setting up the conditions for capturing the packet exchange in Wireshark for further TCP analysis.

A screenshot of a computer

AI-generated content may be incorrect.

**Step 5: Upload Confirmation**

After the file transfer, the browser displayed a **confirmation message** indicating that alice.txt was successfully uploaded to the server gaia.cs.umass.edu. This marked the completion of the HTTP POST process. At this point, the Wireshark packet capture could be stopped, since the required TCP transmission had been fully recorded.

**Purpose:** To validate that the file transfer was completed successfully and to provide a clear end point for the Wireshark capture, ensuring all relevant TCP segments were collected for analysis.

A screenshot of a computer

AI-generated content may be incorrect.

**Step 6: Capturing the HTTP POST Transmission**

This Wireshark capture shows the **HTTP POST request** used to upload the alice.txt file from the client (10.167.95.46) to the server (128.119.245.12) on port 80. The packet details reveal the HTTP headers, including the **Host (gaia.cs.umass.edu)**, **Content-Length (152,351 bytes)**, and **Content-Type (multipart/form-data)**, confirming the transfer of a large text file. Wireshark also indicates that the POST was reassembled from **123 TCP segments**, since the file exceeded a single packet’s size. The lower pane displays the raw data in hexadecimal and ASCII, showing both header information and file content. This capture illustrates how TCP segments work together to ensure reliable delivery of application-layer data.

**Purpose:**  
To observe how the alice.txt file is transmitted to the server via an HTTP POST request, segmented into multiple TCP packets. This step demonstrates the encapsulation of application-layer data within TCP, highlights important HTTP headers (such as content length and type), and shows how Wireshark reassembles the file transfer across several segments for reliable delivery.

A screenshot of a computer program

AI-generated content may be incorrect.



**Step 7: TCP Segmentation and Reassembly**

This capture shows the alice.txt upload as observed at the **TCP layer**. The client (10.167.95.46) communicates with the server (128.119.245.12) using port 80. The POST request is too large for a single TCP packet, so it is split into multiple **segments of 1250 bytes each**. Wireshark reassembles these segments into the original payload of **152,351 bytes**, ensuring complete delivery of the file. The middle pane details the HTTP POST request headers, while the right pane displays the raw data in hexadecimal and ASCII form.

A screenshot of a computer

AI-generated content may be incorrect.

**Purpose:**  
To analyze how TCP handles large data transfers by dividing the payload into segments, tracking sequence and acknowledgment numbers, and reliably reassembling them into the complete message. This step demonstrates the core role of TCP in ensuring ordered and error-free file delivery.

**Step 8: TCP Acknowledgments and Flow Control**

This capture shows the TCP segment details of the alice.txt transfer between the client (10.167.95.46) and the server (128.119.245.12) over port 80. Each segment carries a **payload of 1250 bytes**, and the TCP header provides important control information:

* **Sequence Number (raw): 3843237578** → Identifies the starting byte position of this segment in the overall data stream.
* **Acknowledgment Number (raw): 9123861** → Confirms successful receipt of all bytes up to this point by the receiver.
* **Flags: 0x010 (ACK)** → Indicates that this segment is an acknowledgment for previously received data.
* **Window Size: 65,280 bytes** → Shows the remaining buffer space available at the receiver, enabling flow control.

**Purpose:**  
To demonstrate how TCP ensures reliable file transfer using **sequence numbers to track byte positions, acknowledgment numbers for confirmation, and window size for flow control**. This mechanism guarantees ordered delivery and prevents data loss or buffer overflow during large transmissions.

A screenshot of a computer

AI-generated content may be incorrect.

**QUESTIONS**

1)What is the IP address and TCP port number used by the client computer (source) that is transferring the alice.txt file to gaia.cs.umass.edu?

Answer:

During the transfer of the alice.txt file to the server gaia.cs.umass.edu, the client computer initiated the connection using the IP address “**10.167.95.46”** and the TCP source port “**57480”**. These values are obtained by examining the details of the HTTP POST packet within Wireshark, where the IPv4 header identifies the source IP address and the TCP header specifies the port number used for the communication. This combination of IP address and dynamically assigned port uniquely identifies the client-side socket in the TCP connection, enabling reliable end-to-end data transfer between the client and the server.

A screenshot of a computer

AI-generated content may be incorrect.

2)What is the IP address of gaia.cs.umass.edu? On what port number is it sending and receiving TCP segments for this connection?

Answer:

The server gaia.cs.umass.edu is identified by the IP address 128.119.245.12. From the Wireshark packet details, it is evident that the server is sending and receiving TCP segments over port 80, which is the standard port used for HTTP communication. This confirms that the transfer of the alice.txt file between the client and the server was conducted using the HTTP protocol over TCP port 80.

A screenshot of a computer program

AI-generated content may be incorrect.

3)What is the sequence number of the TCP SYN segment that is used to initiate the TCP connection between the client computer and gaia.cs.umass.edu? (Note: this is the “raw” sequence number carried in the TCP segment itself; it is NOT the packet # in the “No.” column in the Wireshark window. Remember there is no such thing as a “packet number” in TCP or UDP; as you know, there are sequence numbers in TCP and that’s what we’re after here. Also note that this is not the relative sequence number with respect to the starting sequence number of this TCP session.). What is it in this TCP segment that identifies the segment as a SYN segment?

Answer:

The TCP SYN segment that initiates the connection between the client computer and gaia.cs.umass.edu carries a **raw sequence number of 3834323577**, as shown in the TCP header under the Sequence Number (raw) field. This value represents the initial sequence number chosen by the client for the session. The segment is identified as a SYN segment by the **Flags field**, which is set to **0x002 (SYN)**. In the breakdown of the flags, the SYN bit is explicitly marked as “Set,” while other control bits such as ACK and FIN remain unset. This confirms that the packet is the initial connection request, beginning the three-way handshake process of TCP.

A screenshot of a computer

AI-generated content may be incorrect.

4) What is the sequence number of the SYN/ACK segment sent by gaia.cs.umass.edu to the client computer in reply to the SYN? What is it in the segment that identifies the segment as a SYN/ACK segment? What is the value of the Acknowledgement field in the SYN/ACK segment? How did gaia.cs.umass.edu determine that value?

Answer:

The **sequence number** of the SYN/ACK segment sent by gaia.cs.umass.edu to the client is **91283060**, as shown in the Sequence Number (raw) field of the TCP header. This segment is identified as a SYN/ACK segment because the **Flags field** is set to **0x012 (SYN, ACK)**, meaning that both the SYN and ACK control bits are set.

The **Acknowledgment field** in this SYN/ACK segment has a value of **3834323578**. This value was determined by gaia.cs.umass.edu by taking the initial sequence number sent by the client in its SYN segment (**3834323577**) and adding **1** to it, as required by the TCP three-way handshake process. This increment indicates that the server has successfully received the client’s SYN and is now acknowledging it while also proposing its own initial sequence number.

A screenshot of a computer

AI-generated content may be incorrect.

5)What is the sequence number of the TCP segment containing the header of the HTTP POST command? How many bytes of data are contained in the payload (data) field of this TCP segment? Did all of the data in the transferred file alice.txt fit into this single segment?

Answer:

The TCP segment carrying the HTTP POST header is found in **Packet 2389**, with a **raw sequence number of 3834476078**. This segment has a **payload size of 462 bytes**, which contains the POST command header. Since the total file size of alice.txt is **152,351 bytes**, it could not fit into a single segment. Instead, the file was divided across multiple **TCP segments**, each contributing payload data, and then reassembled by Wireshark into the complete file. This confirms TCP’s segmentation process, where the POST header is in the first payload while the rest of the file data is distributed over subsequent segments.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

6. Consider the TCP segment containing the HTTP “POST” as the first segment in the data transfer part of the TCP connection.

* At what time was the first segment (the one containing the HTTP POST) in the data-transfer part of the TCP connection sent?
* At what time was the ACK for this first data-containing segment received?
* What is the Round Trip Time (RTT)5 for this first data-containing segment?
* What is the RTT for the second data-carrying TCP segment and its ACK?

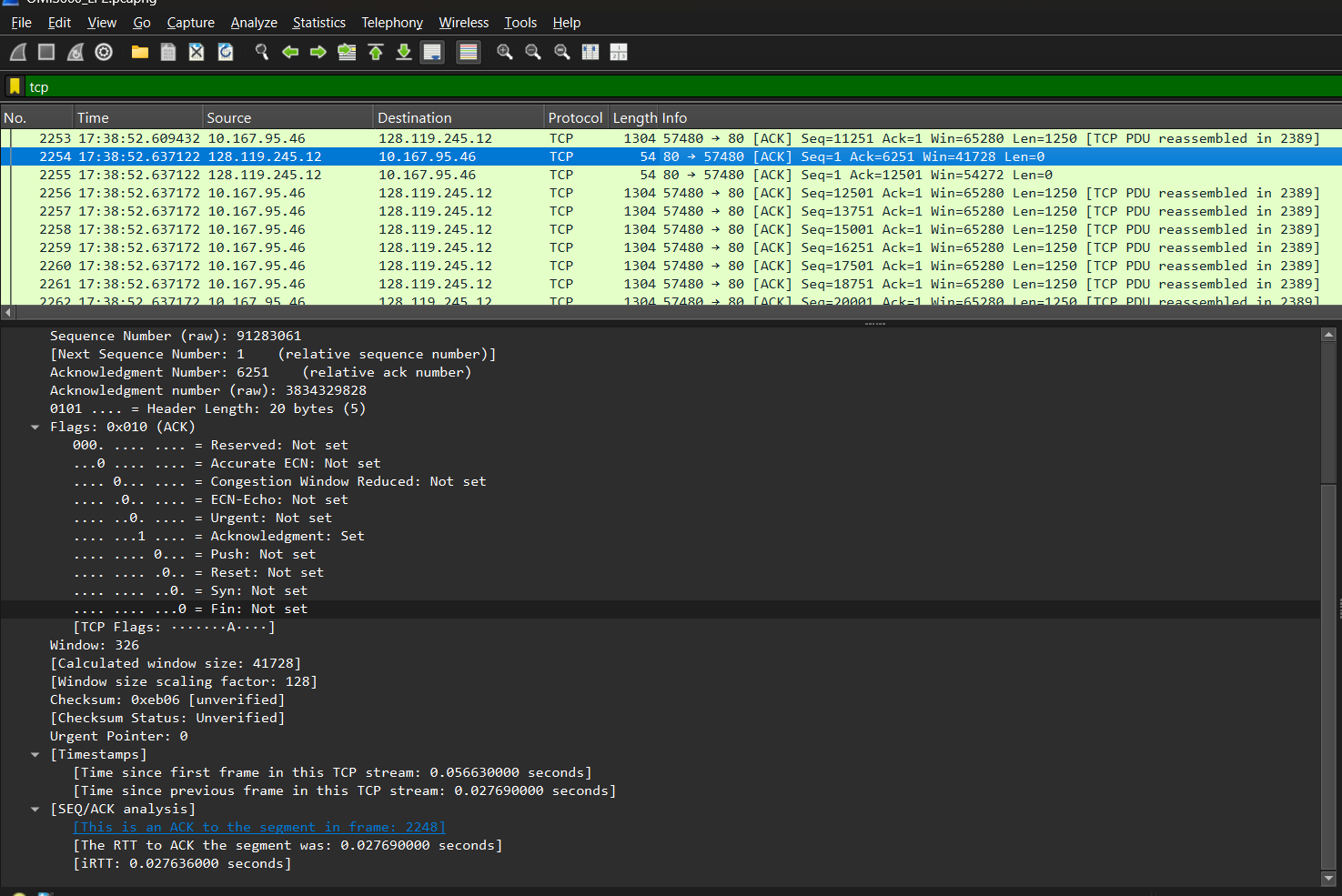
Answer:

The first HTTP POST segment was sent at 17:38:52.609432 and its ACK was received at 17:38:52.637122, giving an RTT of about 0.0277 seconds. The second data segment had the same RTT, as the ACK covered both segments.

A screenshot of a computer

AI-generated content may be incorrect.

The first HTTP POST data segment was sent at 17:38:52.609432 (Frame 2244), and the ACK for this segment was received at 17:38:52.637122 (Frame 2254). This results in a Round-Trip Time (RTT) of approximately 0.0277 seconds. The same ACK also covers the second data segment, so the RTT for the second data-carrying segment is also about 0.0277 seconds.



The first TCP data segment carrying the HTTP POST was sent at 17:38:52.609432 (Frame 2244), and its acknowledgment was received at 17:38:52.637122 (Frame 2254). This gives a Round-Trip Time (RTT) of about 0.0277 seconds for the first data-containing segment. Since the same ACK cumulatively acknowledged the next segment as well, the RTT for the second data-carrying TCP segment is also approximately 0.0277 seconds.

A screenshot of a computer

AI-generated content may be incorrect.

The first TCP segment containing the HTTP POST was transmitted at 17:38:52.609432 (Frame 2244), and its acknowledgment arrived at 17:38:52.637122 (Frame 2254). This gives a Round-Trip Time (RTT) of about 0.0277 seconds. The second data-carrying TCP segment also had an RTT of approximately 0.0277 seconds, since the same ACK cumulatively acknowledged both the first and second data segments.

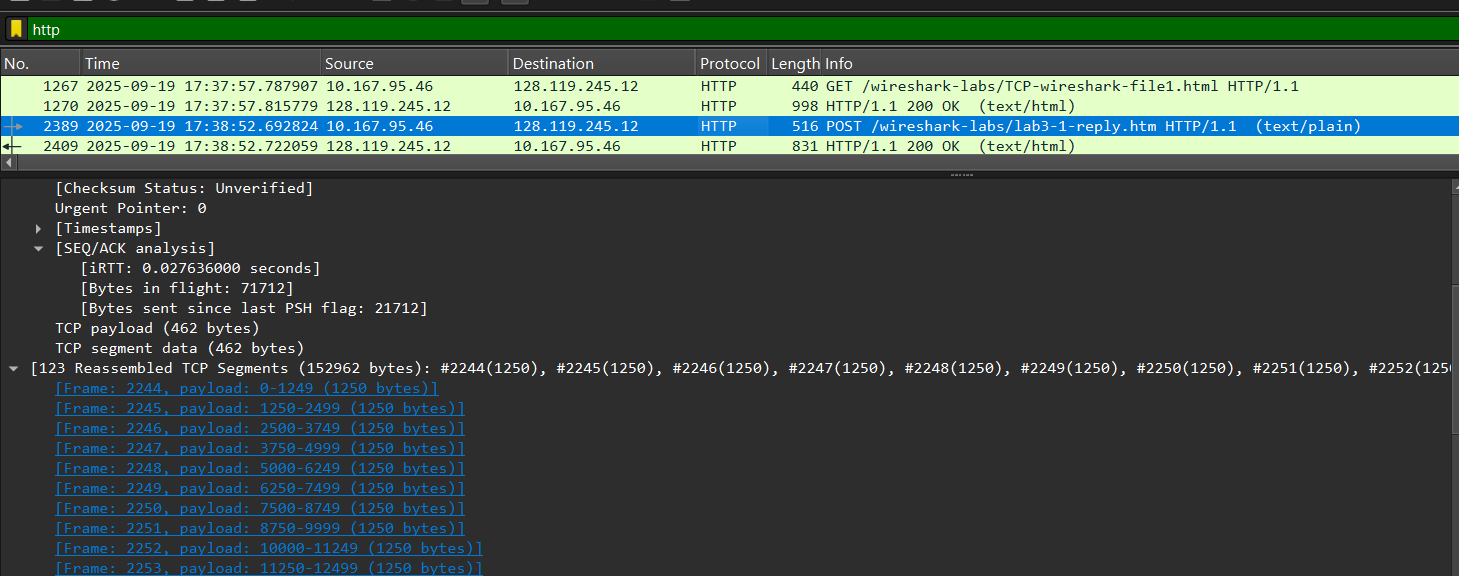
A screenshot of a computer program

AI-generated content may be incorrect.

7)What is the length (header plus payload) of each of the first four data-carrying TCP segments?

Answer:

In a TCP connection, each data-carrying segment is composed of a header and a payload, with the total size influenced by the Maximum Segment Size (MSS) and current network conditions. In this capture, the first four data-carrying TCP segments (frames 2244, 2245, 2246, and 2247) each have a payload of **1250 bytes**. When the TCP header (20 bytes) and IP header (20 bytes) are included, the total length of each of these segments is **1290 bytes**. This confirms that the sender is transmitting data efficiently using the available MSS size for each of the initial segments.



1)Length of Frame 2244: 1250 bytes

2)Length of Frame 2245: 1250 bytes

3)Length of Frame 2246: 1250 bytes

4)Length of Frame 2247: 1250 bytes

8)What is the minimum amount of available buffer space advertised to the client by gaia.cs.umass.edu among these first four data-carrying TCP segments7? Does the lack of receiver buffer space ever throttle the sender for these first four data-carrying segments?

Answer: From the first four **server→client ACKs** that follow your POST (the ACKs to frames 2244–2247), Wireshark reports an advertised receive window of **65,280 bytes** (window field scaled). This is the **minimum** window value among those four ACKs. Because your sender only has at most ~**5,000 bytes in flight** after the first four data segments (4 × 1,250-byte payloads), the in-flight data is far below the 65,280-byte receive window.

A screenshot of a computer program

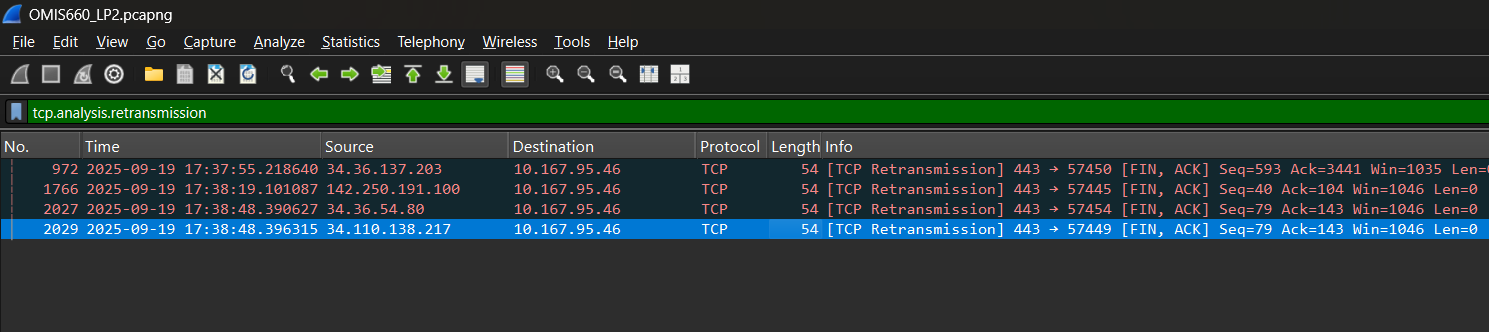
AI-generated content may be incorrect.

**Conclusion:** the receiver’s buffer space does **not** throttle the sender during these first four data-carrying segments.

9)Are there any retransmitted segments in the trace file? What did you check for (in the trace) in order to answer this question?

Answer:

Yes, there were few retransmitted segments in the trace file. When filtering it with TCP or simply just scrolling through the captured packets, I could see some retransmitted packets highlighted in red colour and “TCP Retransmission”.



**CONCLUSION:** This lab offered a comprehensive analysis of the Transmission Control Protocol (TCP) using Wireshark, providing practical insights into its key mechanisms and functions. We examined TCP’s reliable communication process, beginning with the three-way handshake that establishes synchronization between endpoints. The analysis demonstrated how large HTTP POST messages, such as the Alice in Wonderland file, are segmented into smaller TCP packets for efficient transmission. Sequence and acknowledgment numbers were critical in tracking these segments, ensuring accuracy and data integrity. Additionally, we observed TCP’s congestion control strategies, including slow start and congestion avoidance, which optimize throughput while preventing network overload.