Assignment 1

John Dawood

Version: October 18, 2024

https://github.com/JDaw2024/ser321-summer24-C-jdawood1

Part I.

Linux, Setup

1. Command line tasks

Linux System, AWS host server:

1. `mkdir cli_assignment`
2. `cd cli_assignment`
3. `touch stuff.txt (Use man touch to read more about the command)`
4. `cat > stuff.txt (Type some text <Enter>, then press Ctrl+D to save)`
5. `wc -wl stuff.txt`
6. `cat >> stuff.txt (Append more text <Enter>, then press Ctrl+D to save)`
7. `mkdir draft`
8. `mv stuff.txt draft/`
9. `cd draft && touch .secret.txt`
10. `cp -r ../draft ../final`
11. `mv ../draft ../draft.remove`
12. `mv ../draft.remove ../final/`
13. `cd .. && ls -la`
14. `zcat NASA_access_log_Aug95.gz`
15. `gunzip NASA_access_log_Aug95.gz`
16. `mv NASA_access_log_Aug95 logs.txt`
17. **[From Mac Terminal]:** `scp -i ~/Desktop/keypair.pem ~/Desktop/marks.csv ec2-user@00.000.00.000:/home/ec2-user/cli_assignment/`
18. `head -n 100 logs.txt`
19. `head -n 100 logs.txt > logs_top_100.txt`
20. `tail -n 100 logs.txt`
21. `tail -n 100 logs.txt > logs_bottom_100.txt`
22. `cat logs_top_100.txt logs_bottom_100.txt > logs_snapshot.txt`
23. `echo "asurite: This is a great assignment $(date)" >> logs_snapshot.txt`
24. `less logs.txt`
25. **[From Mac Terminal]:** `scp -i ~/Desktop/keypair.pem ~/Desktop/marks.csv ec2-user@00.000.00.000:/home/ec2-user/cli_assignment/` **[Then]:** `cut -d '%' -f1 marks.csv | tail -n +2`
26. `cut -d '%' -f4 marks.csv | sort`
27. `awk -F '%' '{sum+=$3; count++} END {print sum/count}' marks.csv`
28. `awk -F '%' '{sum+=$3; count++} END {print sum/count}' marks.csv > done.txt`
29. `mv done.txt final/`
30. `mv final/done.txt final/average.txt`

2. Some Setup and Examples

2.1. Setup a GitHub repo to submit your assignments

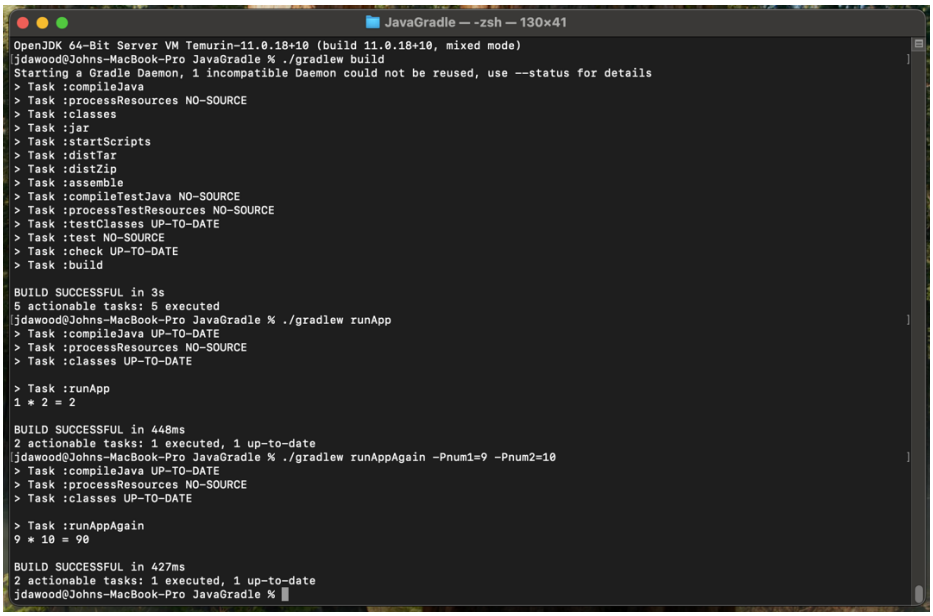
https://github.com/JDaw2024/ser321-summer24-C-jdawood1

2.2. Running examples

Example 1: JavaGradle

Directory: Gradle/JavaGradle

- **Command to Run**: `./ gradlew runApp`
- **Explanation**: This example runs the Multiply Java class, which multiplies two numbers (1 * 2 in this case) and prints the result.
    - **Screenshot**:

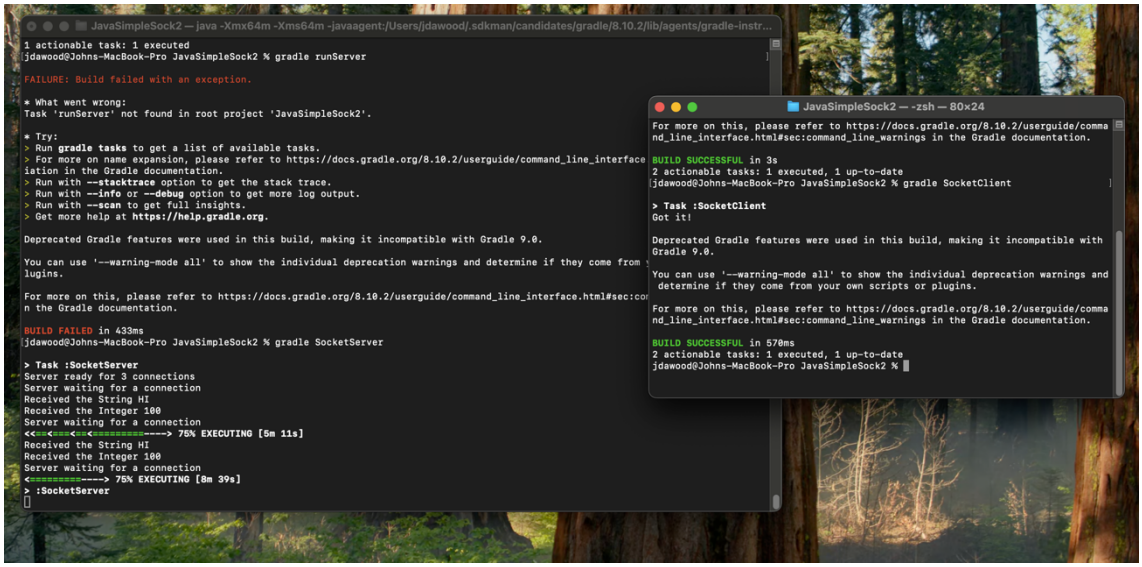


Screenshot of the terminal showing the output.

Example 2: JavaSimpleSock2

Directory: Sockets/JavaSimpleSock2

- **Command to Run**:
    - **Server**: `gradle SocketServer`
    - **Client**: `gradle SocketClient`
- **Explanation**: This example demonstrates a simple TCP connection between a server and a client using Java Sockets. The server listens on port 8888 and waits for connections. The client connects to the server, sending a message ("HI") and an integer (100). The server receives these values and sends back an acknowledgment ("Got it!") to the client. The client then displays the server's response.
- **Screenshot**:



Terminal output of both the server and client running successfully.

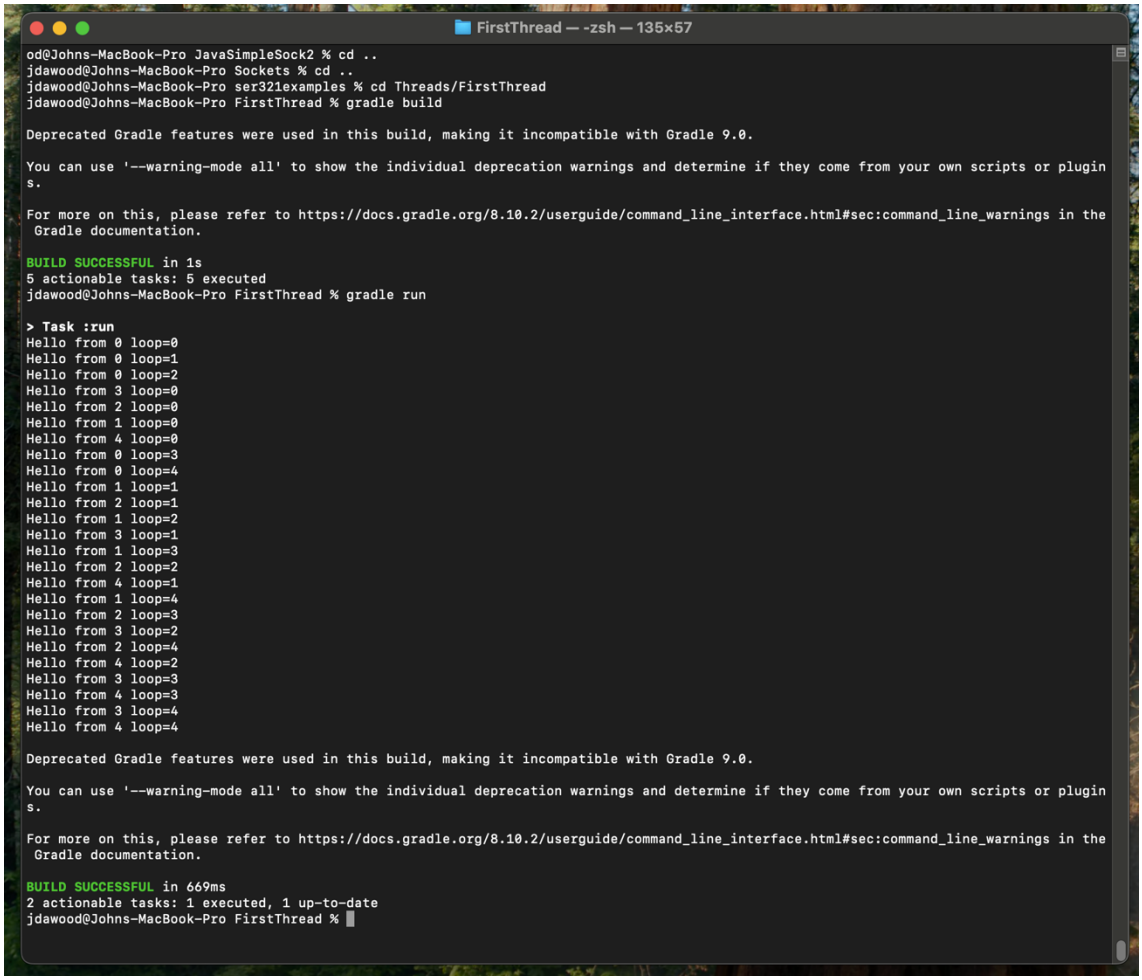
Example 3: FirstThread

Directory: Threads/FirstThread

- **Command to Run**: `gradle run`

- **Explanation:** This example demonstrates the creation of multiple threads where each thread prints a message along with its ID and loop iteration. The program runs five threads, and each one prints a message in a loop. The result shows interleaving output from the multiple threads executing concurrently.

- **Screenshot**:



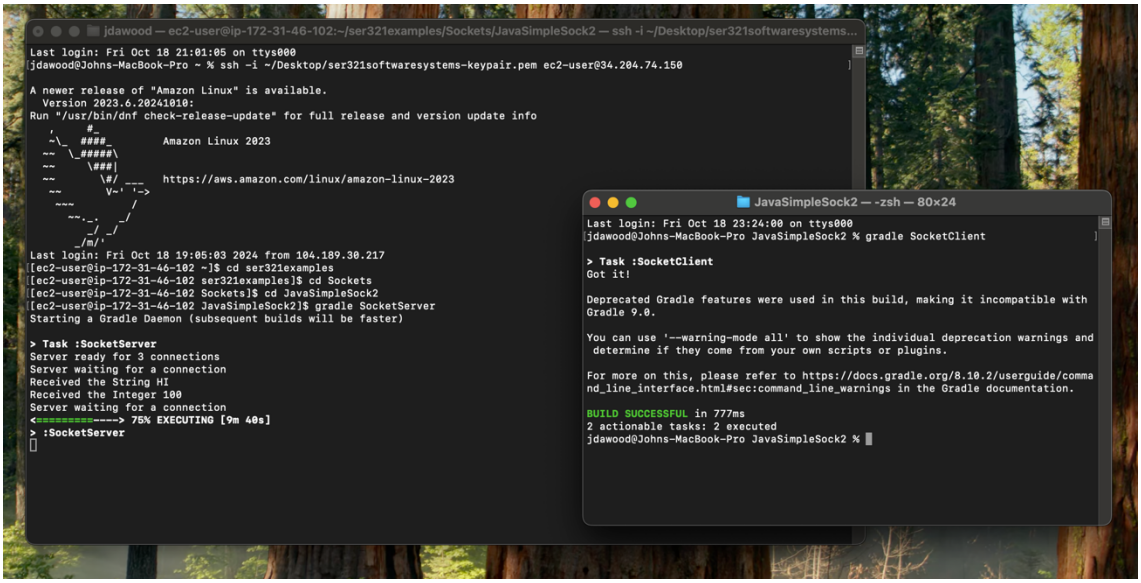
Terminal output of the threads running and their output

2.3. Understanding Gradle

https://github.com/JDaw2024/ser321-summer24-C-jdawood1
following a .zip file for canvas.

2.4. Set up your second system

Screencast for JavaSimpleSock2 Example:
https://youtu.be/K_VqU6Cep4Y



Part II.

Networking

3. Network traffic

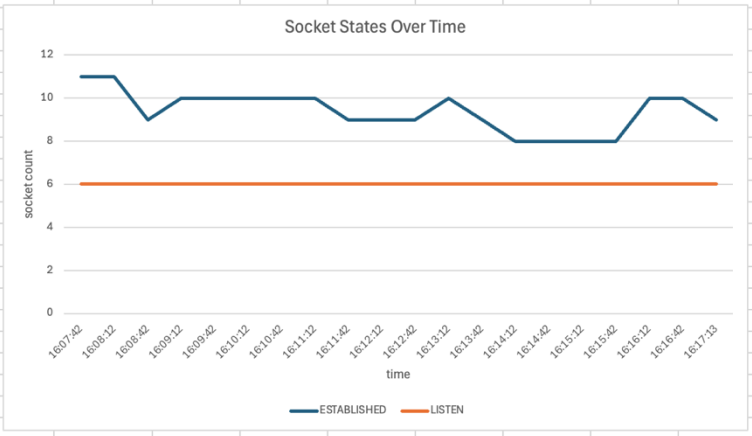
3.1. Understanding TCP network sockets

Command Script:

- nano monitor_socket_states.sh
  - #!/bin/bash
  -
  - # Log file to store the output
  - LOGFILE="socket_states.log"
  -
  - # Initialize the log file
  - echo "Time, ESTABLISHED, LISTEN" > $LOGFILE
  -
  - # Run for 20 iterations (i.e., 10 minutes with a 30-second interval)
  - for i in {1..20}
  - do
  - # Get the current time
  - CURRENT_TIME=$(date +"%T")
  -
  - # Count ESTABLISHED and LISTEN states using netstat
  - ESTABLISHED_COUNT=$(netstat -ant | grep ESTABLISHED | wc -l)
  - LISTEN_COUNT=$(netstat -ant | grep LISTEN | wc -l)
  -
  - # Log the data to the file
  - echo "$CURRENT_TIME, $ESTABLISHED_COUNT, $LISTEN_COUNT" >> $LOGFILE
  -
  - # Sleep for 30 seconds
  - sleep 30
  - done
  -
  - echo "Data collection complete. Results saved to $LOGFILE"
- chmod +x monitor_socket_states.sh
- ./monitor_socket_states.sh

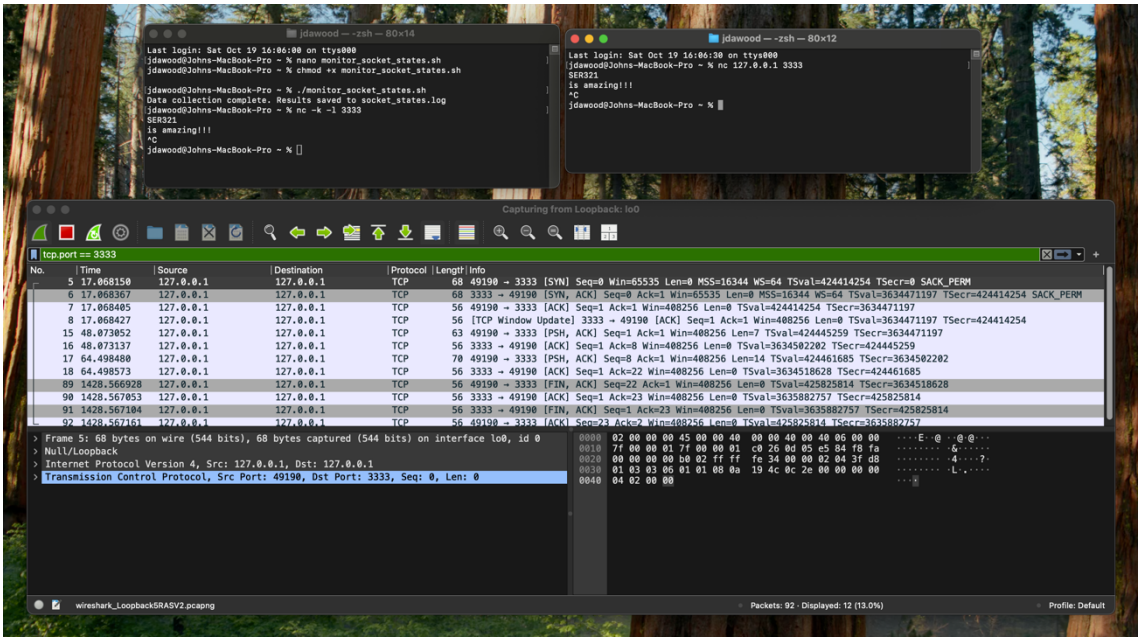
Socket States Over Time

	A	B	C
1	Time	ESTABLISHED	LISTEN
2	16:07:42	11	6
3	16:08:12	11	6
4	16:08:42	9	6
5	16:09:12	10	6
6	16:09:42	10	6
7	16:10:12	10	6
8	16:10:42	10	6
9	16:11:12	10	6
10	16:11:42	9	6
11	16:12:12	9	6
12	16:12:42	9	6
13	16:13:12	10	6
14	16:13:42	9	6
15	16:14:12	8	6
16	16:14:42	8	6
17	16:15:12	8	6
18	16:15:42	8	6
19	16:16:12	10	6
20	16:16:42	10	6
21	16:17:13	9	6
22			
23			



3.2. Sniffing TCP/UDP traffic

Step 1 (TCP)



1. Questions and Answers 1

   a) Explain both the commands you used in detail. What did they actually do?

   - **nc -k -l 3333**: This starts a server listening on port 3333 and keeps the connection open (-k flag) for further communication.
   - **nc 127.0.0.1 3333**: This connects to the server running on localhost (127.0.0.1) on port 3333 and sends any input typed into the terminal.

   b) How many packets were send back and forth so the client/server could send/receive these two lines? A total of four Data-packets.

   TCP communication:

   - **Data Packet 1** (Client to Server): Sends the first line "SER321".
   - **ACK** (Server to Client): Acknowledges receipt of "SER321".
   - **Data Packet 2** (Client to Server): Sends the second line "is amazing!!!".
   - **ACK** (Server to Client): Acknowledges receipt of "is amazing!!!".

   c) How many packets were needed back and forth to capture the whole "process" (starting the communication, ending the communication, sending the lines)? A Total of 12 packets

   - **3-way handshake**:
     - **SYN** (Client to Server)
     - **SYN-ACK** (Server to Client)
     - **ACK** (Client to Server)
     - **ACK** (Server to Client, the window size is updated)
   - **Data Transfer**:
     - **PSH-ACK** (Client to Server: "SER321")
     - **ACK** (Server to Client)
     - **PSH-ACK** (Client to Server: "is amazing!!!")
     - **ACK** (Server to Client)
   - **Connection Termination**:
     - **FIN-ACK** (Client to Server)
     - **ACK** (Server to Client)
     - **FIN-ACK** (Client to Server)
     - **ACK** (Server to Client)

   d) How many bytes is the data (only the data) that was sent from client to server?

- **7 + 14 = 21 bytes** of actual data were sent from the client to the server:
    - "SER321" has **7 bytes**
    - "is amazing!!!" has **14 bytes**.

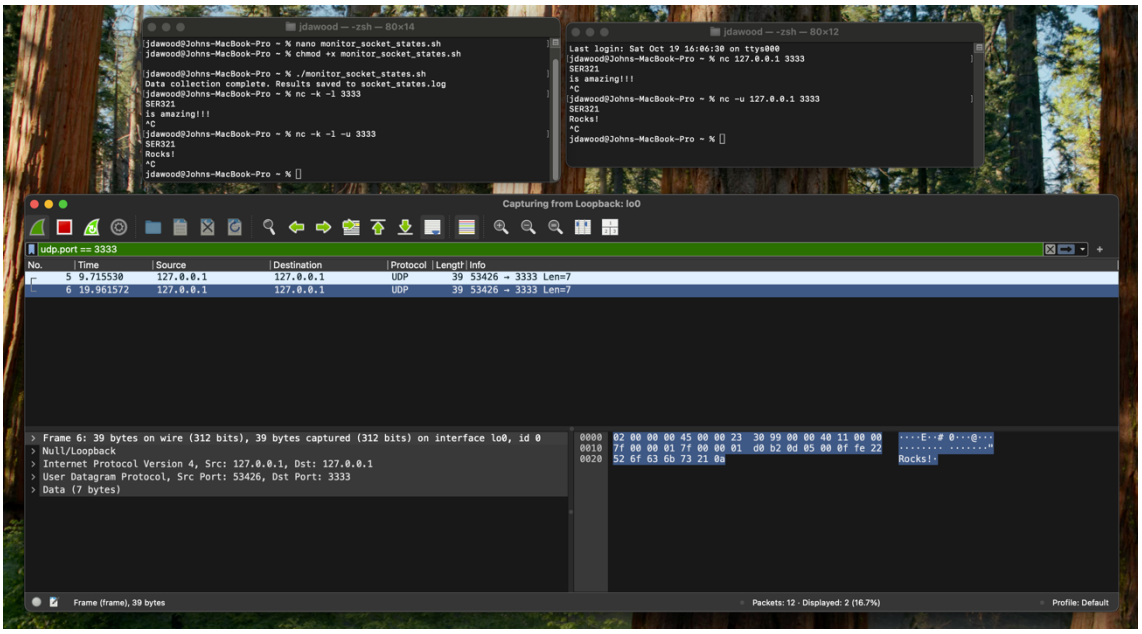
e) How many total bytes went over the wire (back and forth) for the whole process?

- Length = 68 + 68 + 56 + 56 + 63 + 56 + 70 + 56 + 56 + 56 + 56 + 56 = 717 bytes

f) How much overhead was there. Basically how many bytes was the whole process compared to the actually data that we did send.

- The overhead is the total bytes minus:
    - 717 bytes (total) - 21 bytes (data) = 696 bytes overhead

Step 2 (UDP)



2. Questions and Answers 2.

a) Explain both the commands you used in detail. What did they actually do?

- **nc -k -l -u 3333**: The server listens on UDP port 3333 (-u), allowing multiple connections (-k). It waits for incoming messages from clients.
- **nc -u 127.0.0.1 3333**: The client sends data to the server running on **localhost (127.0.0.1)** via UDP (-u) on port 3333.

b) How many packets were needed to capture those 2 lines? **Just 2 Packets.**

c) How many packets were needed to capture the whole "process" (starting the communication, ending the communication)? **Just 2 Packets.**

d) How many bytes is the data (only the data) that was sent?

- The data sent was:
  - "SER321": 7 **bytes**
  - "Rocks!": 7 **bytes, so Total 14 bytes**

e) How many total bytes went over the wire? **Length = 39 + 39 = 78 bytes**

f) Basically how many bytes was the whole process compared to the actually data that we did send? **78 – 14 = 64 bytes**

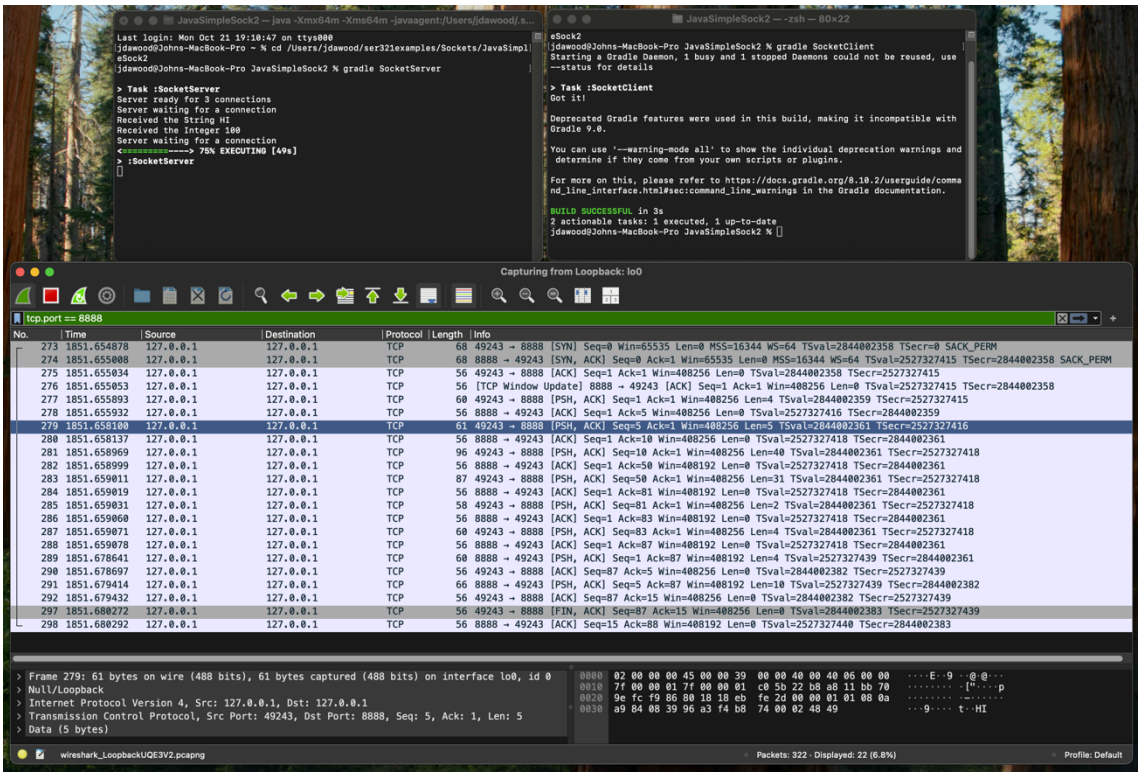
g) What is the difference in relative overhead between UDP and TCP and why? Specifically, what kind of information was exchanged in TCP that was not exchanged in UDP? Show the relative parts of the packet traces.
   - **UDP** does not establish a connection or handle acknowledgments, so there is no handshake like in **TCP**. This means less overhead because **TCP** packets include information for establishing and maintaining the connection (e.g., SYN, ACK, FIN flags)
   - In contrast, **UDP** only sends data without ensuring it arrives or confirming delivery.
   - Wireshark analysis:
     - Look at the relative size of the TCP and UDP packet captures. Compare the headers in each protocol and identify the extra fields in TCP (e.g., sequence numbers, acknowledgment numbers) that are absent in UDP.

3.3. Running client servers in different ways

In this section I want you to run the JavaSimpleSock2 example from the repo. You will also need to have Wireshark open to capture the traffic between client and server. This is similar to the setup you already did but here we also want to look through Wireshark.

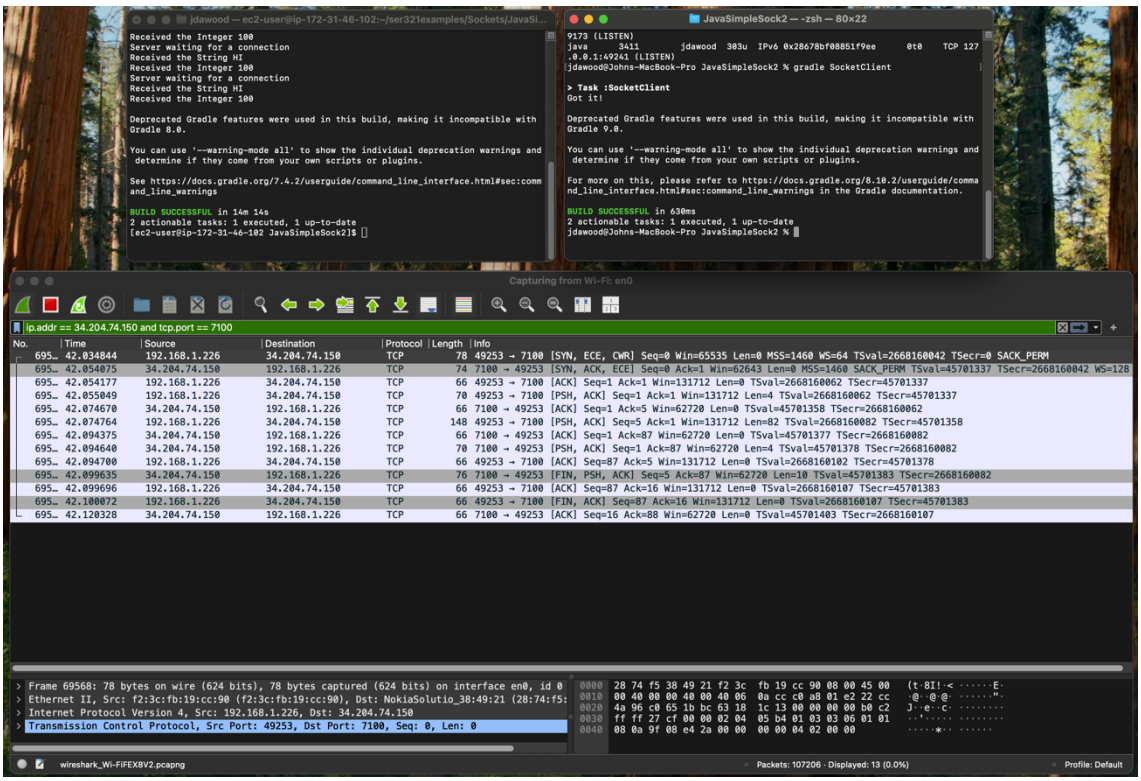
3.3.1. Running things locally

Brief Video Example:
https://youtu.be/viCqIPCliww



3.3.2. Server on AWS

When running the server locally, the client connected to localhost (127.0.0.1), and Wireshark captured traffic using the loopback interface (lo0). After moving the server to AWS, the client needed to connect to the AWS public IP (34.204.74.150) over port 7100, and Wireshark was updated to capture traffic on the Wi-Fi interface (en0) with a filter for the AWS IP and port. The Gradle call or client code was modified to use the AWS IP instead of localhost, and the network traffic now showed TCP handshakes over the Internet, increasing latency compared to local communication.



3.3.3. Client on AWS

Running the server locally on your home computer and the client on AWS introduces some challenges compared to the setup in 3.3.2 where the server was on AWS. Unlike a cloud server, your local computer is typically behind a router or firewall, meaning it does not have a public IP address that is directly accessible from the Internet. To allow the AWS client to connect to your home server, you would need to configure **port forwarding** on your router to expose the server's port to the public Internet. Additionally, firewall settings on your local machine or network could block incoming traffic from the AWS client. In contrast, with AWS, security groups can easily be configured to allow incoming traffic, and the AWS instance has a public IP by default. The difference is mainly in network accessibility, as home computers are not as easily reachable from external networks without additional configuration.

3.3.4. Client on AWS 2

The key difference is that AWS servers have public IP addresses, making them easily accessible from anywhere, including clients on a local network. In contrast, devices on a local network, like a server running at home, use private IP addresses assigned by a router, which are not directly reachable from outside the network. The router acts as a gateway and uses **NAT (Network Address Translation)** to manage traffic between the local network and the internet. To allow an external client (such as one on AWS) to reach a server on your local network, you need to configure **port forwarding** on the router to direct incoming traffic to the server's private IP address. The "issue" with running a server locally is that it's not inherently exposed to the public internet, so without configuring port forwarding or using a service like a VPN, external access is blocked. This setup is more secure but less convenient than having a publicly accessible AWS server.