Matthew Lowber
SER 321
Assignment 1
https://github.com/mlowber/ser321-spring25B-mlowber

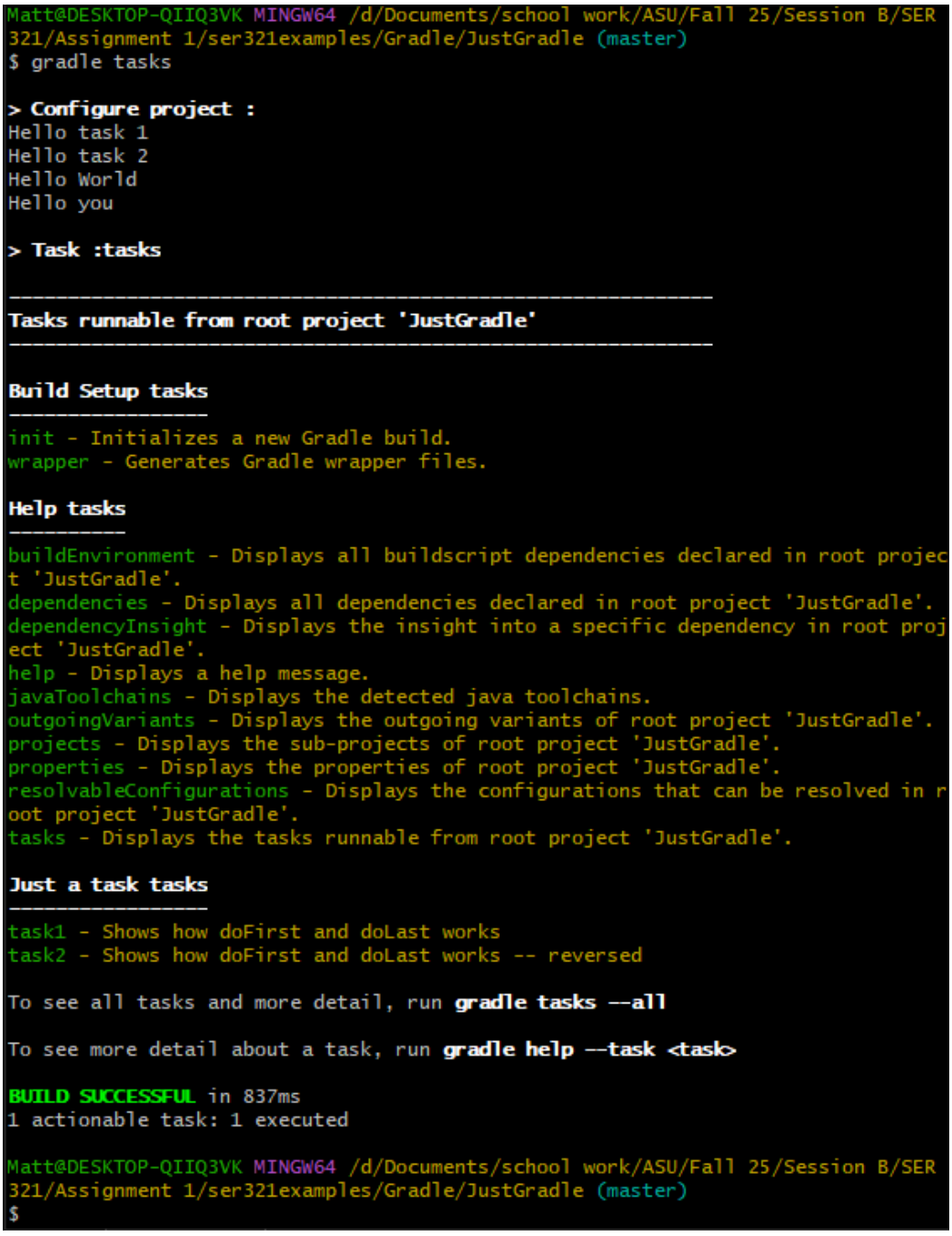
Part I

1) System: Windows 10 (using GitBash)
    1. mkdir cli_assignment
    2. cd cli_assignment
    3. touch stuff.txt
    4. cat > stuff.txt
    5. wc stuff.txt
    6. cat >> stuff.txt
    7. mkdir draft
    8. mv stuff.txt draft/
    9. cd draft
       touch .secret.txt (added '.' in front of file name to make it hidden on linux)
       attrib +h .secret.txt (added 'hidden' attribute to file to make it hidden in windows)
    10. cd ..
        cp -r draft final
    11. mv draft draft.remove
    12. mv draft.remove final/
    13. ls -l
    14. zcat ../NASA_access_log_Aug95.gz | head
    15. gunzip ../NASA_access_log_Aug95.gz
    16. mv ../NASA_access_log_Aug95 logs.txt
    17. 'logs.txt' already renamed into 'cli_assignment/' during step 16
    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 "mlowber: This is a great assignment $(date)" >> logs_snapshot.txt
    24. less logs.txt
    25. cut -d '%' -f 1 ../marks.csv | tail -n +2
    26. cut -d '%' -f 4 ../marks.csv | tail -n +2 | sort -n
    27. cut -d '%' -f 3 ../marks.csv | tail -n +2 | awk '{sum += $1} END {print sum/NR}'
    28. cut -d '%' -f 3 ../marks.csv | tail -n +2 | awk '{sum += $1} END {print sum/NR}' > done.txt
    29. mv done.txt final/
    30. mv final/done.txt final/average.txt

2.1) https://github.com/mlowber/ser321-spring25B-mlowber

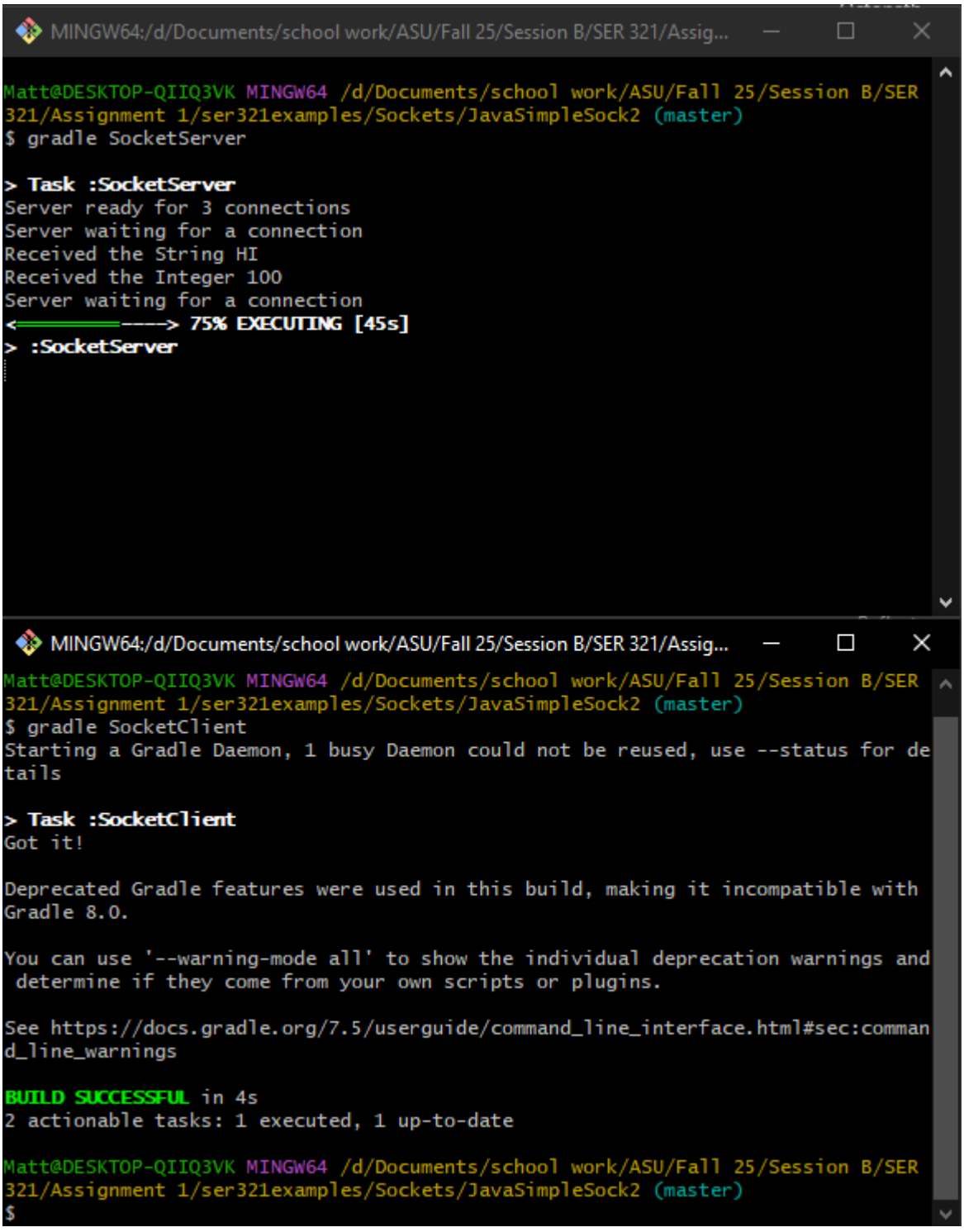
2.2)

JustGradle



This example is a minimal gradle project with no java code. It's most likely used to explore how gradle works as a build system, define tasks manually, and learn gradle's DSL.

JavaSimpleSock2



This example is of a simple client-server interaction using java sockets. The server listens on a port and handles communication from a client. The client connects to the server, sends a string and an int, and receives a response.

FirstThread

```
Matt@DESKTOP-QIIQ3VK MINGW64 /d/Documents/school work/ASU/Fall 25/Session B/SER
321/Assignment 1/ser321examples/Threads/FirstThread (master)
$ gradle run

> Task :run
Hello from 0 loop=0
Hello from 2 loop=0
Hello from 4 loop=0
Hello from 3 loop=0
Hello from 1 loop=0
Hello from 0 loop=1
Hello from 0 loop=2
Hello from 0 loop=3
Hello from 0 loop=4
Hello from 1 loop=1
Hello from 2 loop=1
Hello from 1 loop=2
Hello from 3 loop=1
Hello from 1 loop=3
Hello from 4 loop=1
Hello from 2 loop=2
Hello from 1 loop=4
Hello from 3 loop=2
Hello from 2 loop=3
Hello from 4 loop=2
Hello from 2 loop=4
Hello from 3 loop=3
Hello from 4 loop=3
Hello from 3 loop=4
Hello from 4 loop=4

Deprecated Gradle features were used in this build, making it incompatible with
Gradle 8.0.

You can use '--warning-mode all' to show the individual deprecation warnings and
 determine if they come from your own scripts or plugins.

See https://docs.gradle.org/7.5/userguide/command_line_interface.html#sec:comman
d_line_warnings

BUILD SUCCESSFUL in 1s
2 actionable tasks: 2 executed

Matt@DESKTOP-QIIQ3VK MINGW64 /d/Documents/school work/ASU/Fall 25/Session B/SER
321/Assignment 1/ser321examples/Threads/FirstThread (master)
$
```

This example is of basic multithreading in java. The program creates multiple threads which print to the console and shows how threads can run concurrently and interleave their outputs.

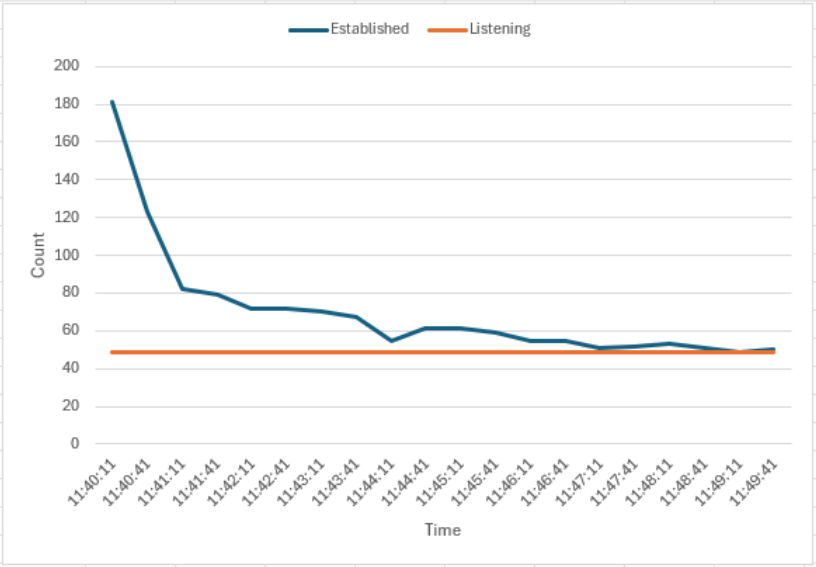
2.3) Folder uploaded to github directory

2.4) https://youtu.be/-PtLxRaryWc

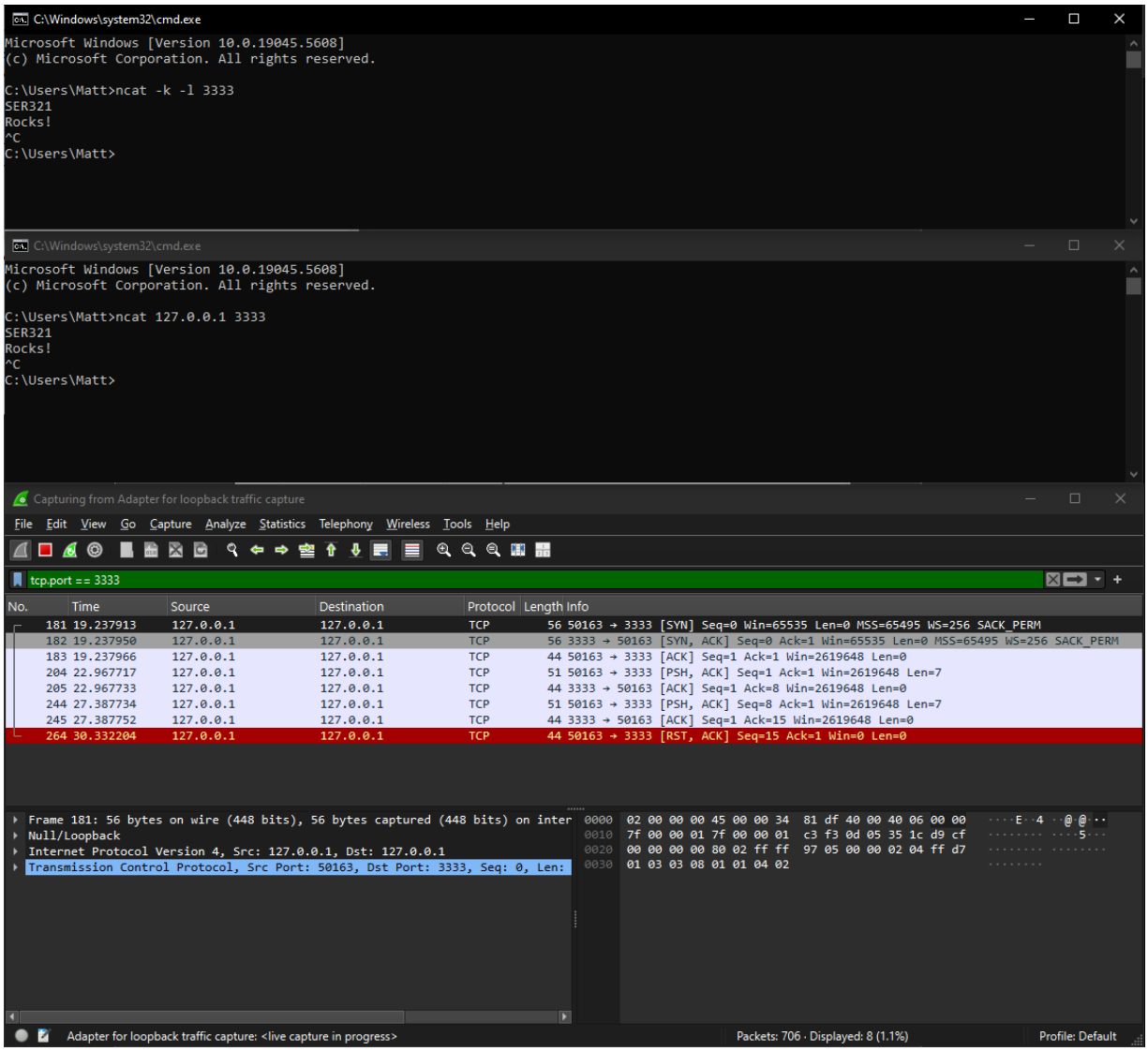
3.1)
PowerShell script

```
# monitor-netstat.ps1

$interval = 30
$duration = 600
$iterations = $duration / $interval
$outputFile = "netstat_log.csv"

"Time,Established,Listening" | Out-File -Encoding UTF8 $outputFile

for ($i = 0; $i -lt $iterations; $i++) {
  $time = Get-Date -Format "HH:mm:ss"

  $netstat = netstat -an

  $established = ($netstat | Select-String "ESTABLISHED").Count
  $listening = ($netstat | Select-String "LISTENING").Count

  "$time,$established,$listening" | Out-File -Encoding UTF8 -Append $outputFile

  Start-Sleep -Seconds $interval
}
```

Time	Established	Listening
11:40:11	181	49
11:40:41	123	49
11:41:11	82	49
11:41:41	79	49
11:42:11	72	49
11:42:41	72	49
11:43:11	70	49
11:43:41	67	49
11:44:11	55	49
11:44:41	61	49
11:45:11	61	49
11:45:41	59	49
11:46:11	55	49
11:46:41	55	49
11:47:11	51	49
11:47:41	52	49
11:48:11	53	49
11:48:41	51	49
11:49:11	49	49
11:49:41	50	49

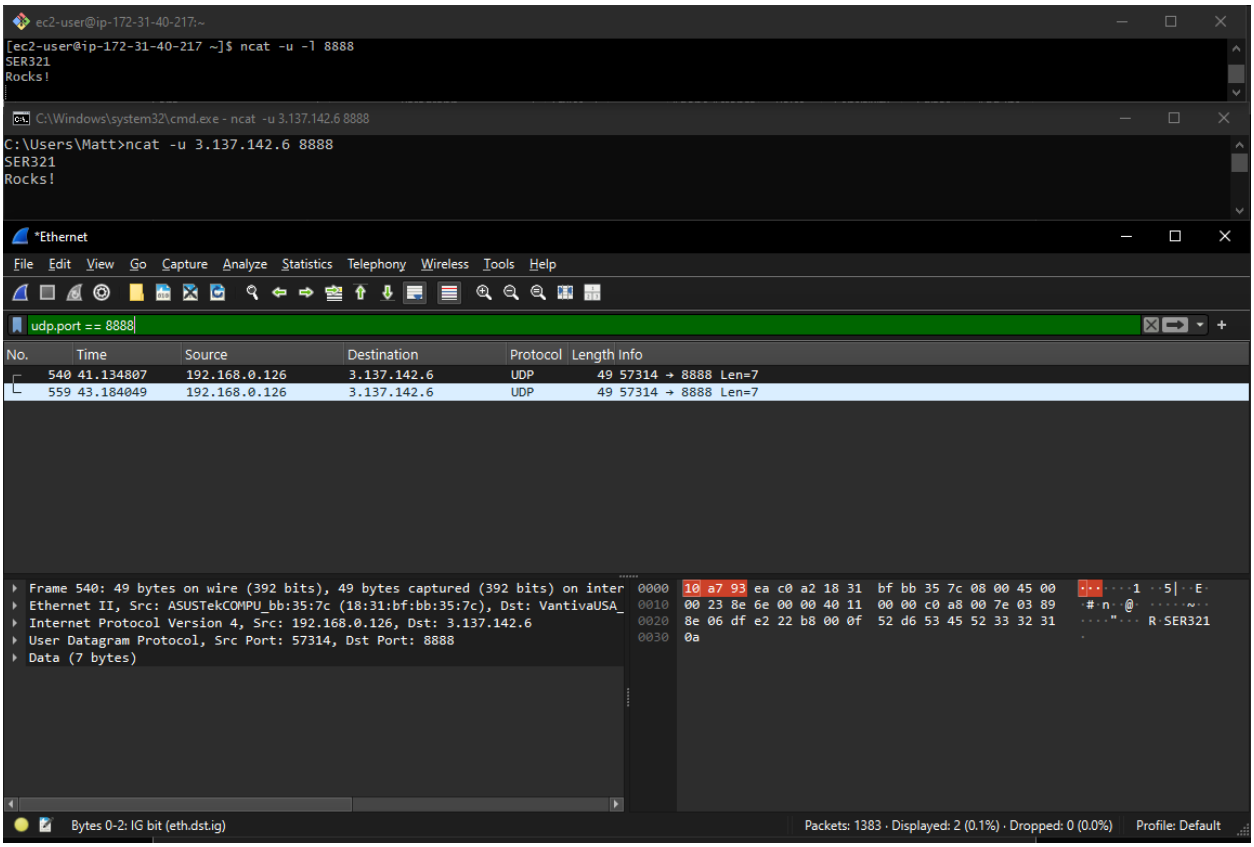


3.2) Step One (TCP)



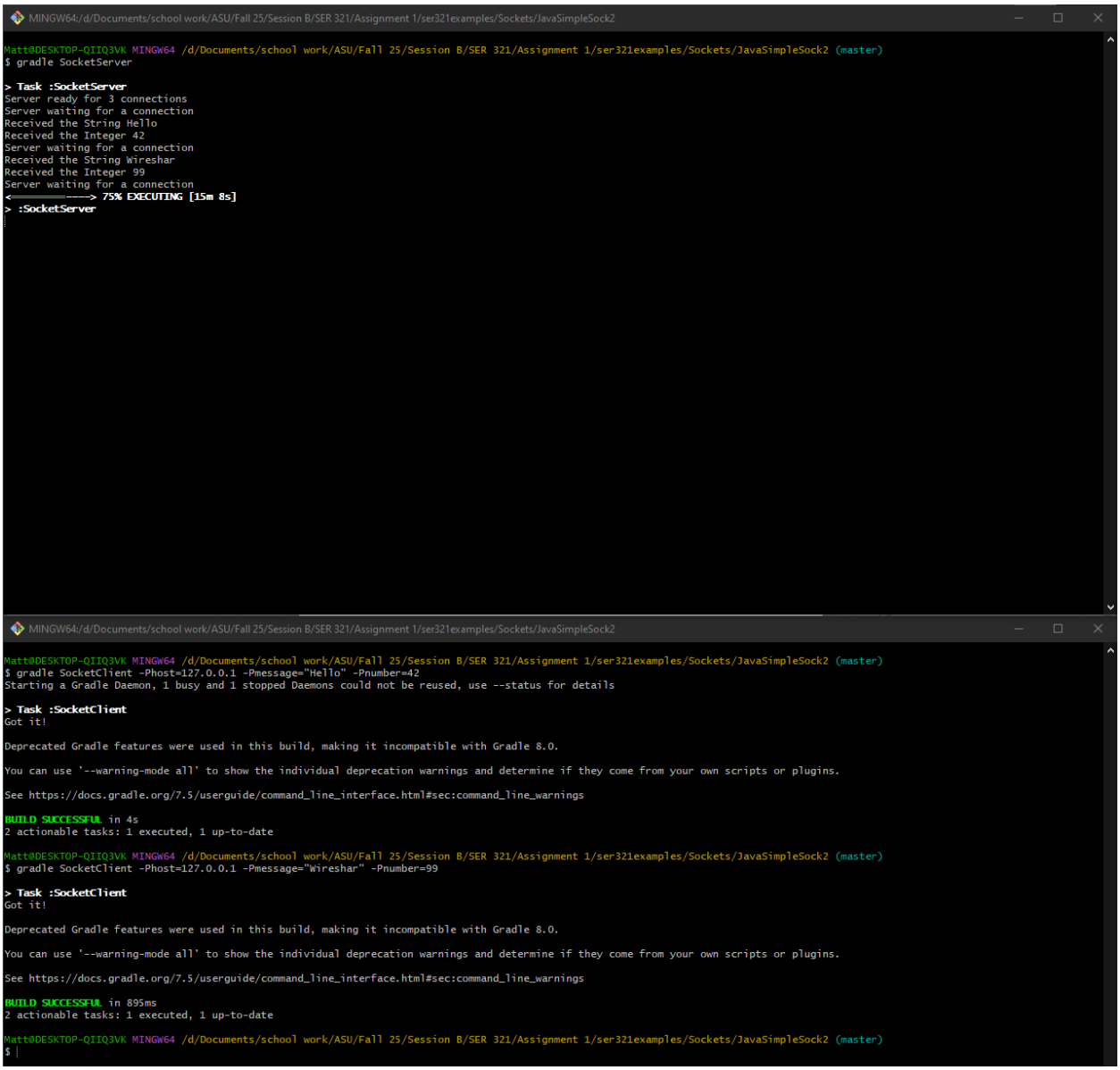
a) (terminal 1) ncat -k -l 3333
   starts a TCP server listening on port 3333, -k keeps it running after a connection ends, and -l puts it in listen mode
   (terminal 2) ncat 127.0.0.1 3333
   connects to the TCP server at 127.0.0.1 (local loopback) on port 3333, and whatever is typed into the terminal after the connection is sent to the server.
b) 4: 204, 205, 244, 245
c) 4, same as above. Two data packets and 2 acks.
d) 8
e) 16 bytes (6 characters + 2 bytes for \r\n = 8 bytes each)
f) 56+56+44+51+44+51+44+44 = 390 bytes
g) 390-16 = 374 bytes (~95.9%) of overhead

3.2) Step 2 (UDP)

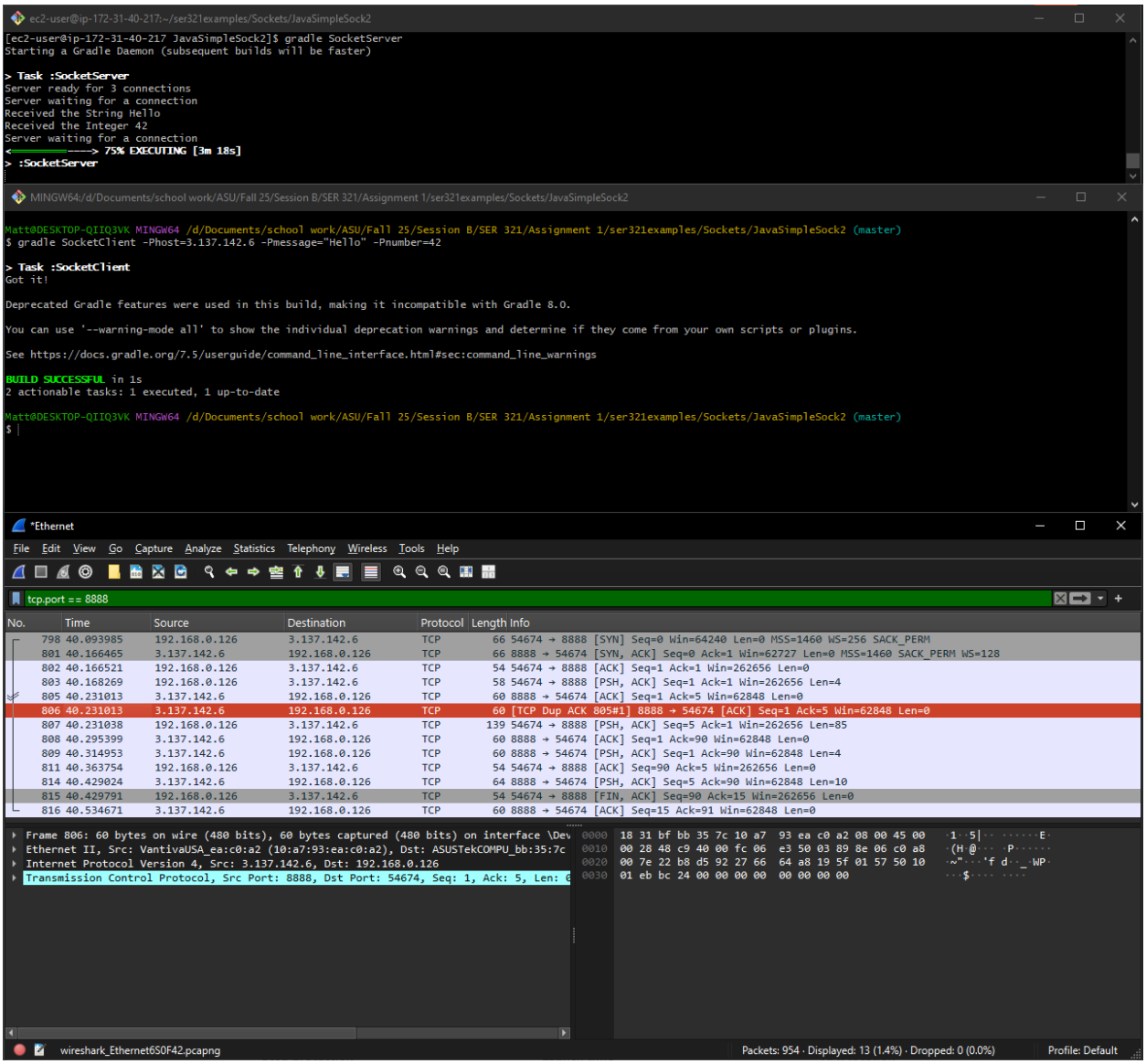


a) (on EC2) ncat -u -l 8888
   listens for incoming UDP messages on port 8888, -u enables UDP mode, -l makes it a listener, and prints any data received just like the TCP example
   (on local) ncat -u 3.137.142.6 8888
   connects to UDP server at the EC2 public IP address on port 8888, -u enables UDP mode, and sends the packets as before
b) 2: 540, 559
c) 2 packets
d) UDP is connectionless, there is no handshake/teardown. 2 packets sent/captured.
e) 49+49 = 98 bytes
f) 16 bytes, same as before
g) 98-16 = 82 bytes (~84%) of overhead
h) UDP has lower overhead than TCP because it doesn't require a handshake (no syn/syn-ack/ack) and doesn't include acknowledgements or sequence numbers, but it also does not guarantee delivery or correct ordering. TCP sends extra packets for connection, acknowledgements, and teardown. As a comparison, for 16 bytes of data: TCP used 8 packets and 390 bytes, UDP used 2 packets and 98 bytes. Relative overhead was much higher in TCP due to additional headers and reliability features.

3.3.1) https://youtu.be/E9KEU5R3Kig



3.3.2)



To run the server on AWS, I ssh'd into my EC2 instance, navigated to JavaSimpleSock2, and ran: gradle SocketServer

On my local machine, I ran the client and connected to the EC2 server using:

gradle SocketClient -Phost=[my EC2 public address] -Pmessage="Hello" -Pnumber=42

For wireshark, I switched form the loopback adapter to my ethernet adapter to capture the network traffic and applied the 'tcp.port = 8888' filter to isolate packets for this connection.

As for differences, instead of communicating over 127.0.0.1 or localhost, the connection is now over the internet connecting to a public IP, packet latency was slightly higher in wireshark, and port access had to be manually configured in AWS (unlike localhost). Otherwise, the client/server interaction remained the same.

3.3.3) Running the client on AWS and the server on my local machine would likely not work—at least not the same way as in 3.3.2. The main issue is that my local machine is behind a NAT which blocks unsolicited inbound traffic from the internet as a security feature. In 3.3.2 it worked because AWS servers are publicly addressable and I opened port 8888 in the security group. To make the reverse work, I would need to port forward 8888 on my home router to my local machine, probably configure a firewall exception, use my public IP address using ipconfig, and use that public IP in the AWS client's -Phost=... argument. The setup is more complex and less secure by default, which is why it would be typically avoided.

3.3.4) Mostly answered in 3.3.3 by coincidence. In a typical home network, my computer has a private IP address (192.168.x.x) assigned by my router. This address is only valid inside my local network and is not directly reachable via the internet. When I run a client locally to connect to an AWS server, the connection works easily because AWS assigns the server a public IP address, my router allows outgoing connections so traffic can leave the network and reach AWS, and the server responds to that connection and the router lets the response back in.

However, the reverse is not allowed by default. So if a client on AWS tries to connect to my local server, the router doesn't know which local device should handle the request and blocks the incoming connection for security. To make it work, I would need to set up port forwarding as explained in 3.3.3, possibly configure a static IP for my computer, adjust firewall settings to allow external traffic, and share my public IP with the AWS client—all of which is more complicated and introduces a lot of security risks, which is why cloud servers like AWS are often used instead.