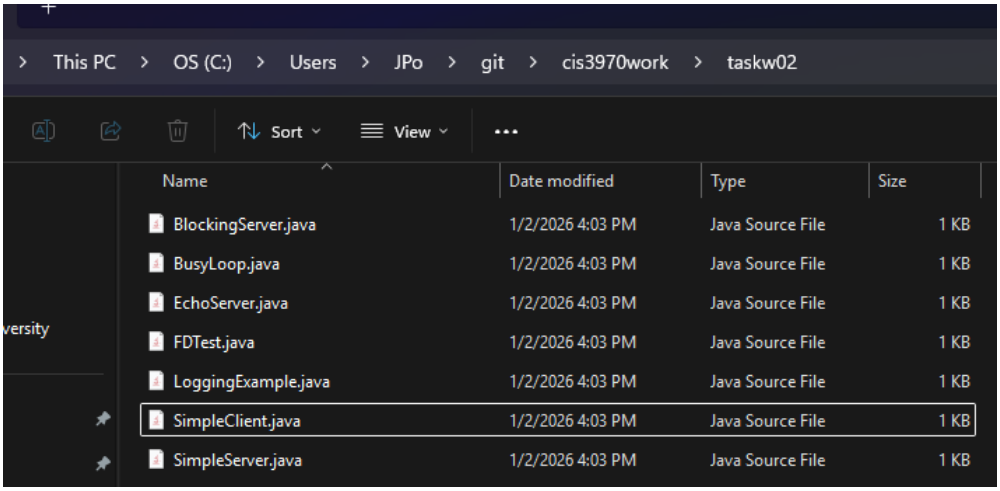
Task W02

For this task, we will explore common Linux networking tools to inspect and begin to understand what happens during communication between programs. I encourage you to review the Java code, but do not worry if you do not understand it now, because we will learn more about it later in the semester.

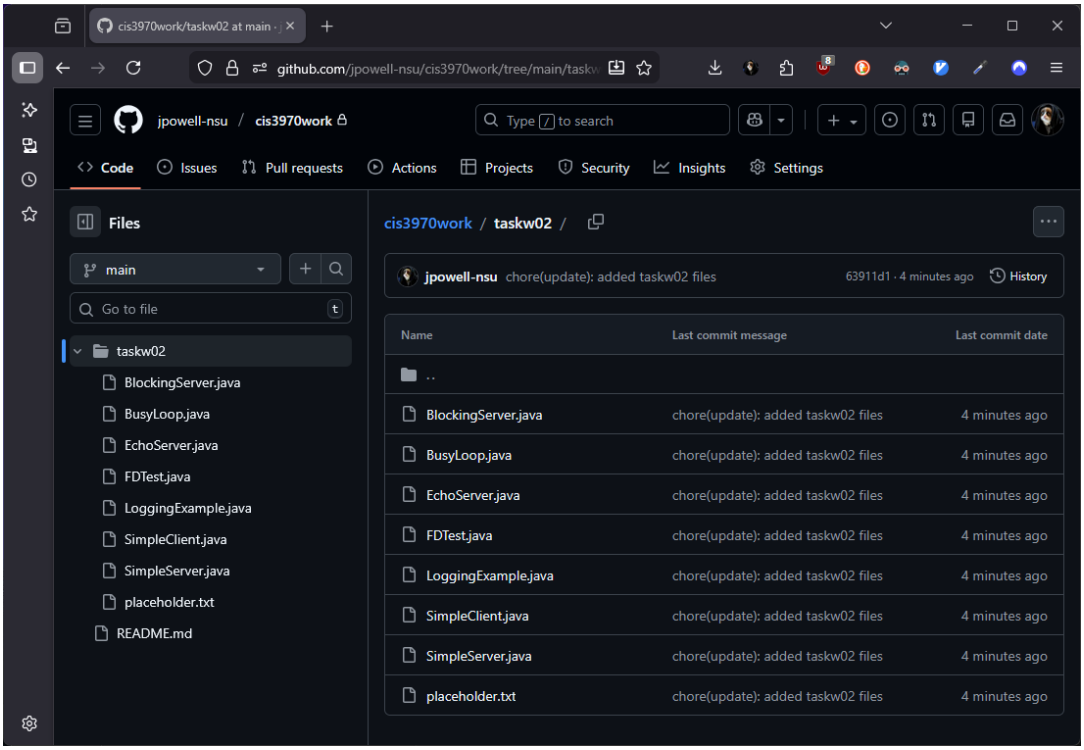
Set Up

First, we must set up our experiment environment. There are several ways to approach this, but these steps describe how I set it up. (Note: ignore the “placeholder.txt” file in these images. You will not have it.)

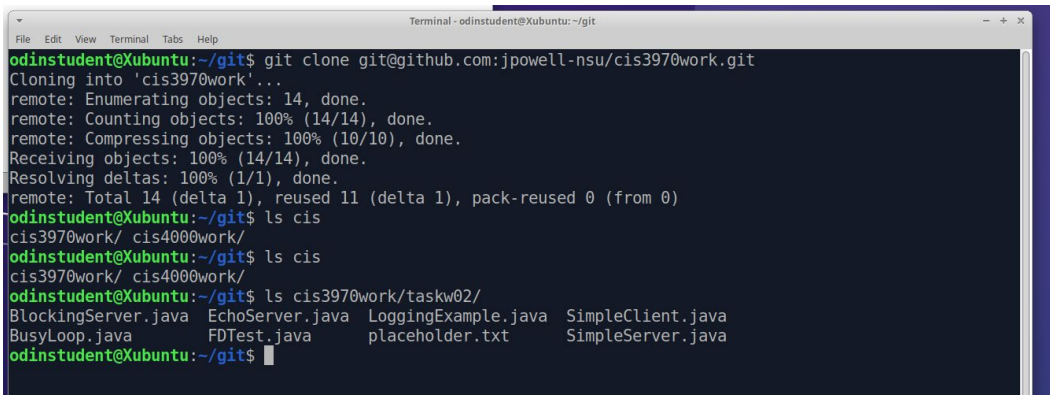
Get the Java files from the taskw02 folder in our class cis3970 repository and place them in a taskw02 folder in your cis3970 repository. I used Git Bash in Windows, as in the previous assignment.




```
JPo@JPO-DESKTOP MINGW64 ~/git/cis3970work (main)
$ ls taskw02/
BlockingServer.java  EchoServer.java  LoggingExample.java  SimpleServer.java
BusyLoop.java        FDTest.java      SimpleClient.java    placeholder.txt

JPo@JPO-DESKTOP MINGW64 ~/git/cis3970work (main)
$ git add .

JPo@JPO-DESKTOP MINGW64 ~/git/cis3970work (main)
$ git commit -m "chore(update): added taskw02 files"
[main 63911d1] chore(update): added taskw02 files
 8 files changed, 136 insertions(+)
 create mode 100644 taskw02/BlockingServer.java
 create mode 100644 taskw02/BusyLoop.java
 create mode 100644 taskw02/EchoServer.java
 create mode 100644 taskw02/FDTest.java
 create mode 100644 taskw02/LoggingExample.java
 create mode 100644 taskw02/SimpleClient.java
 create mode 100644 taskw02/SimpleServer.java
 create mode 100644 taskw02/placeholder.txt

JPo@JPO-DESKTOP MINGW64 ~/git/cis3970work (main)
$ git push
Enter passphrase for key '/c/Users/JPo/.ssh/id_rsa':
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 24 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 2.18 KiB | 2.18 MiB/s, done.
Total 11 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), done.
To github.com:jpowell-nsu/cis3970work.git
   f259d2e..63911d1  main -> main

JPo@JPO-DESKTOP MINGW64 ~/git/cis3970work (main)
$
```



In Linux, I cloned my repository, but if yours is already there from Lab W02, you can do “git pull” instead.

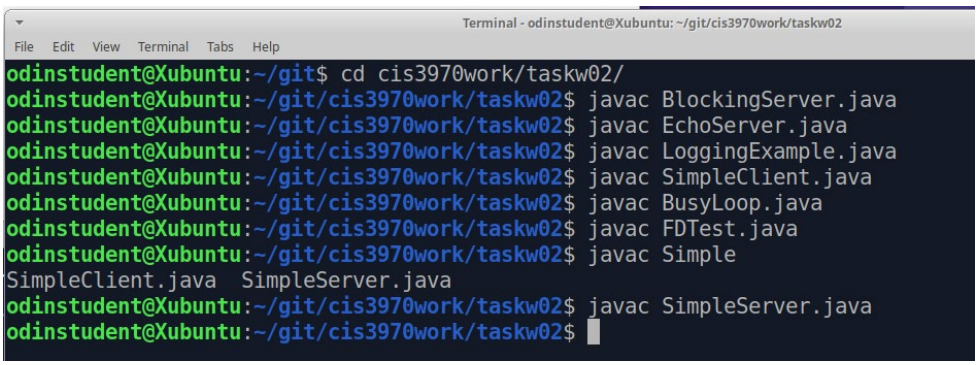


Note: There are other ways to do this, but ultimately it comes down to getting your files into Linux.

We need the Java compiler to perform command-line compilation and execution of Java programs.

Test if you have it by issuing the command:	javac
If it indicates you do not have it installed:	sudo apt install openjdk-25-jdk-headless

Once installed, compile the Java programs by navigating to the directory and running javac.



We also need to install Wireshark and htop (alternatively, you can use top).

sudo apt install wireshark-qt

sudo apt install htop

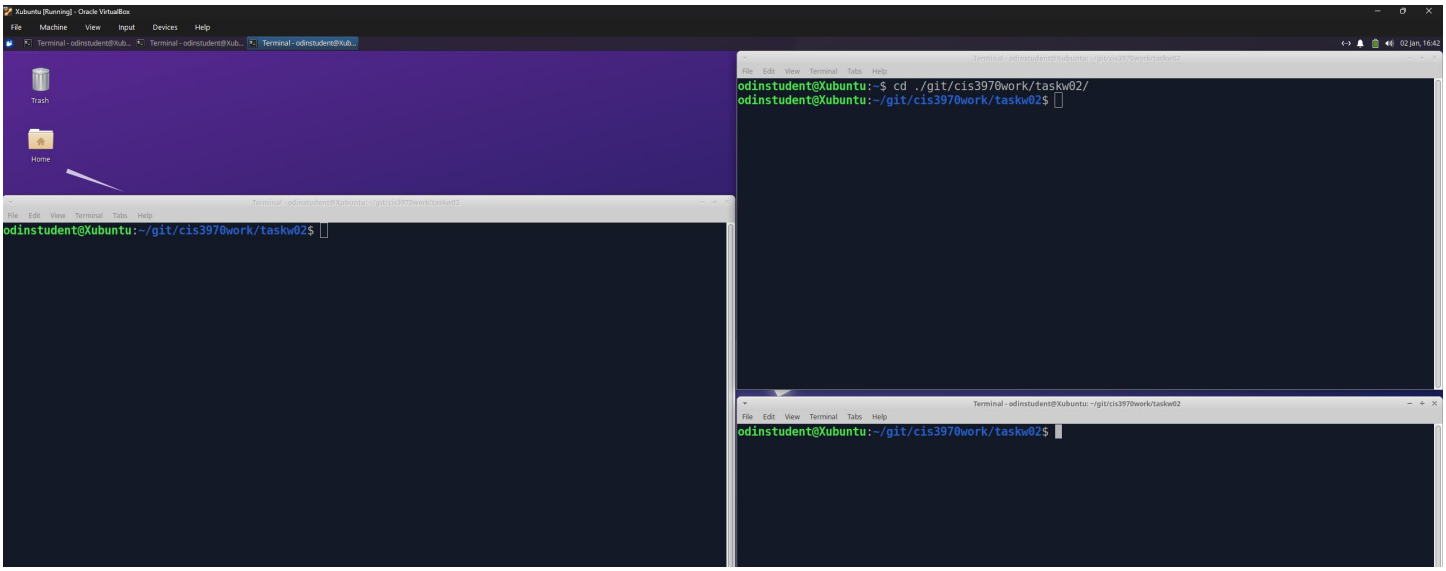
Now we are ready.

Part A - Seeing a Socket Program on the Network

Tools: netstat, ss, and lsof

Concept: A socket is an OS resource bound to a port.

For this, we will need three terminals, at least two of them in the taskw02 directory with your Java files.



In one of the taskw02 terminals, issue the command: java SimpleServer

In another terminal, issue the command: ss -ltn

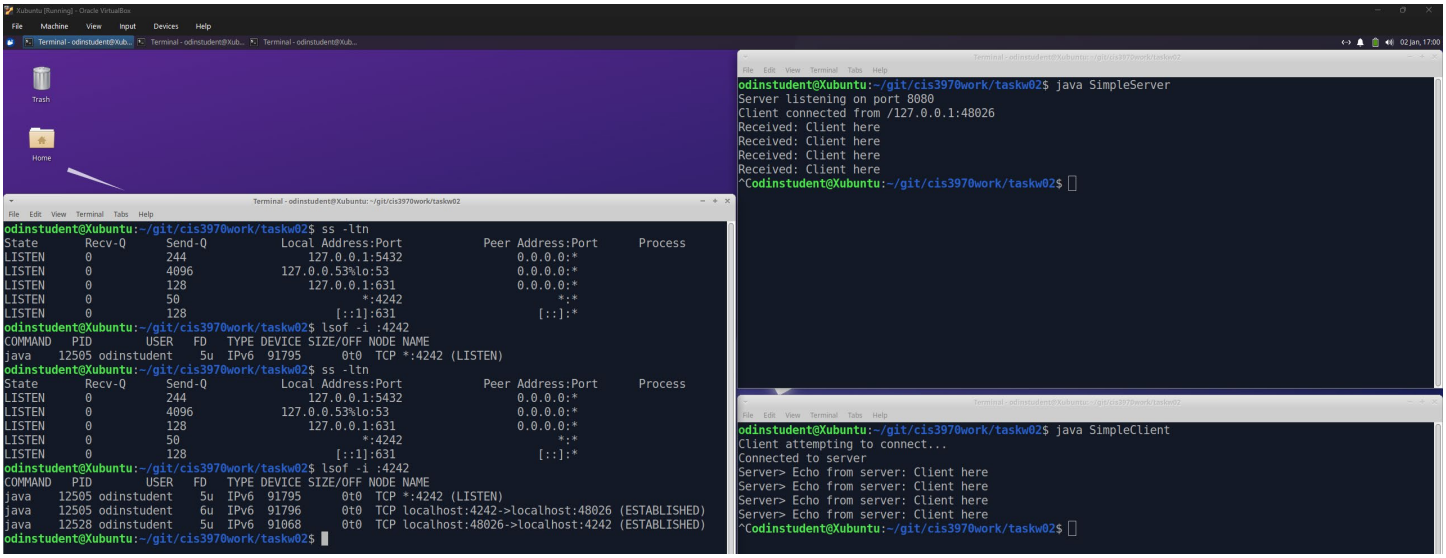
Then issue the command lsof -i :4242

Notice that port 4242 is now listening for connections and that Java is using it. Applications do not own the network; the OS does and allows them to use it.

Now in the third terminal, issue the command: java simpleClient

Leave it running for a little bit. Every few seconds, the client and server are exchanging messages. Now, reissue the ss and lsof commands and notice the changes. The lsof output shows they are communicating on a different port (port 48026 in my case). We will learn more about this later in the semester.

To stop the programs, you can hit Ctrl-c in the terminals. Take a screenshot and save it in the taskw02 folder. This will be a record of your completion of this step.



Part B - Talking to Java Without Writing Java

Tools: nc (netcat)

Concept: Separate protocol behavior from application code.

The great thing about sockets is that they are generalized. Two completely different programs written in different languages can exchange messages (of course, internally, they must know how to read each other's messages).

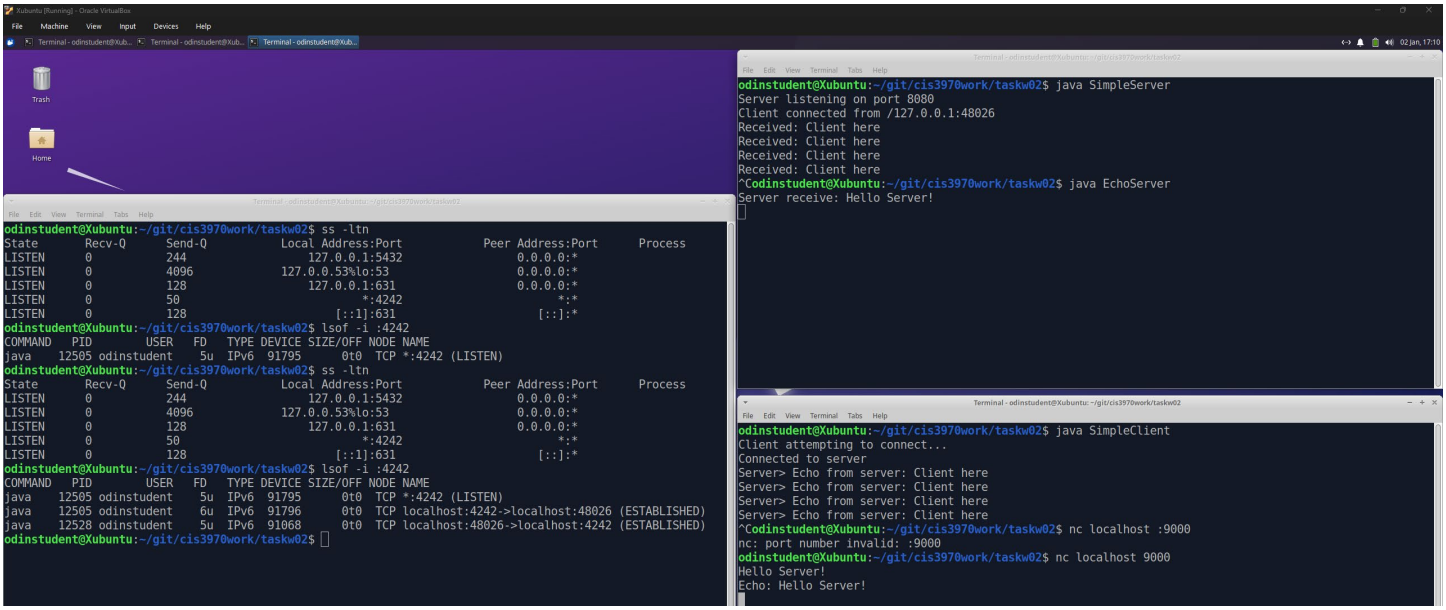
In one of the terminals, start the EchoServer: java EchoServer

In another terminal, issue the command: nc localhost 9000

Our echo server is listening on port 9000, and netcat lets us pass strings to it. Try typing in "Hello Server!" and hitting enter. The server received the message, displayed it, and echoed it back to the client.

The key idea is that connections and protocols can be tested without complete client code, because just about anything can send messages over sockets.

To stop the program, you can hit Ctrl-c as before. Take a screenshot and save it in the taskw02 folder. This will be a record of your completion of this step.



Part C - Blocking, Threads, and "Why Is It Hanging?"

Tools: jstack, jps

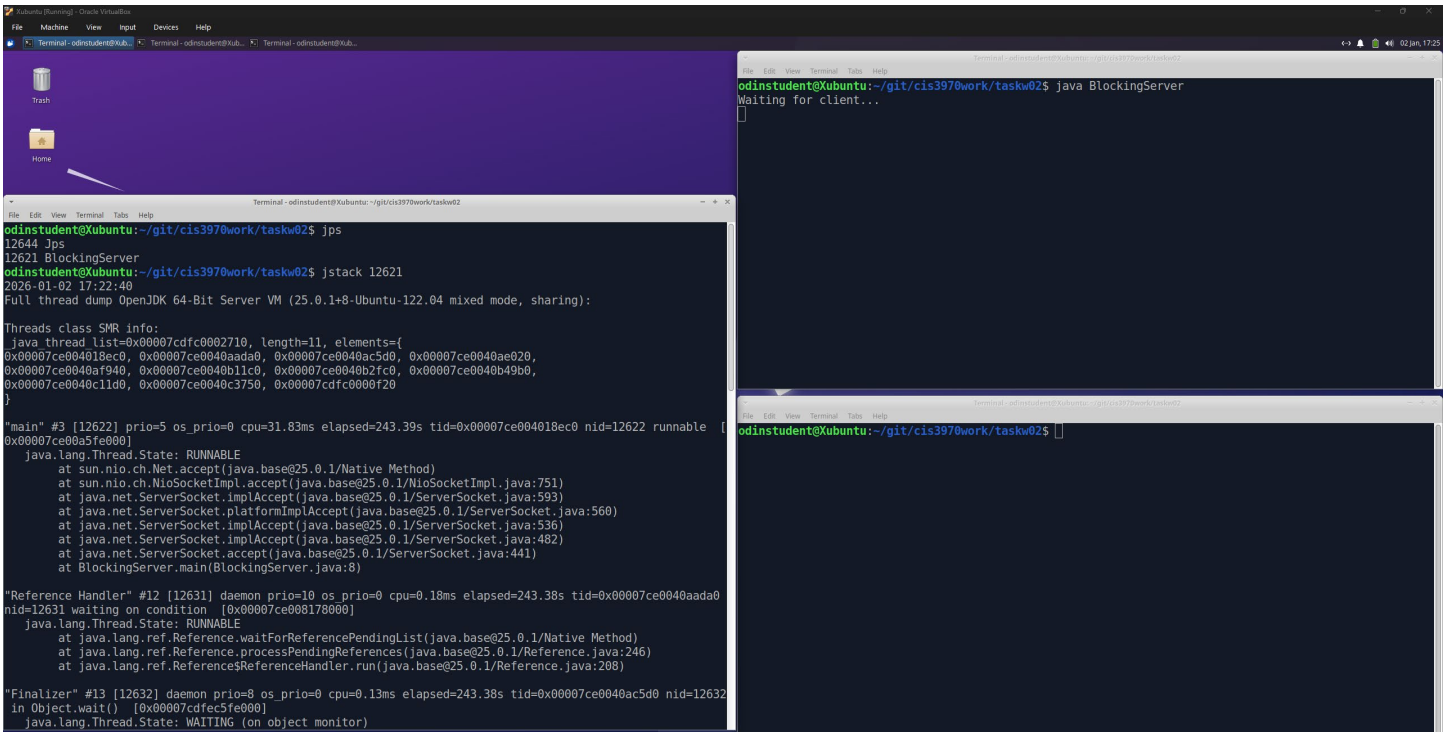
Concept: Blocking I/O causes threads to wait and can be hard to detect during debugging.

In one of the terminals, issue the command: java BlockingServer

In another, get the Java process ID number for the server: jps

Then do a thread dump and identify the blocked thread: jstack <PID>

Scroll through the information. You can see that the thread is in a runnable state and is waiting from the attached listener. A "stuck" program may simply be waiting.



To stop the program, you can hit Ctrl-c like before. Take a screenshot and save it in the taskw02 folder.

Part D - Watching Java Consume System Resources

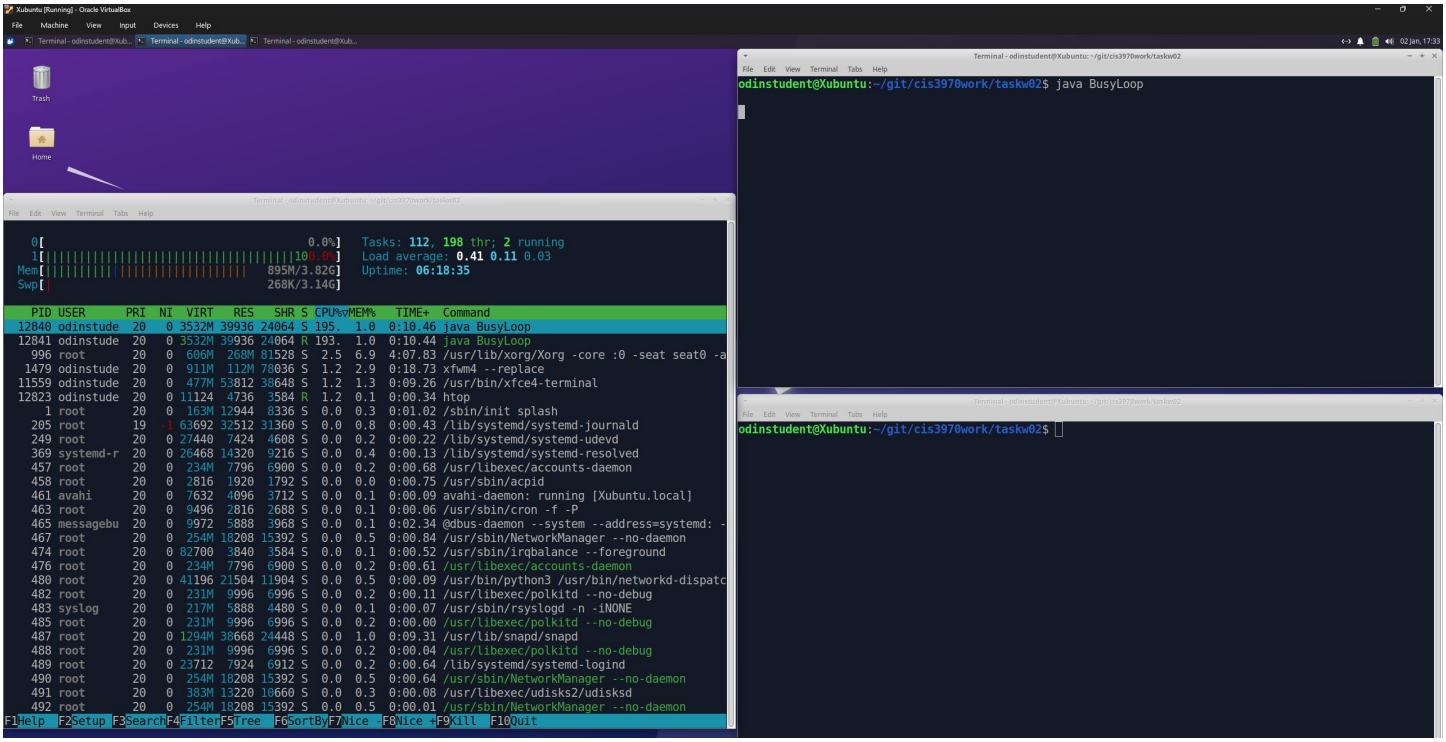
Tool: top, htop

Concept: Programs compete for CPU and memory.

Note: This one will eat up resources and slow everything down. Be ready to hit Ctrl-c on the Java program terminal.

In one of the terminals (htop or top): htop

In one of the terminals: java BusyLoop



CPU 1 maxed out in mine. Hit Ctrl-c to kill the program. The lesson: bad code affects the system. Use q to exit htop.

Take a screenshot and save it in the taskw02 folder.

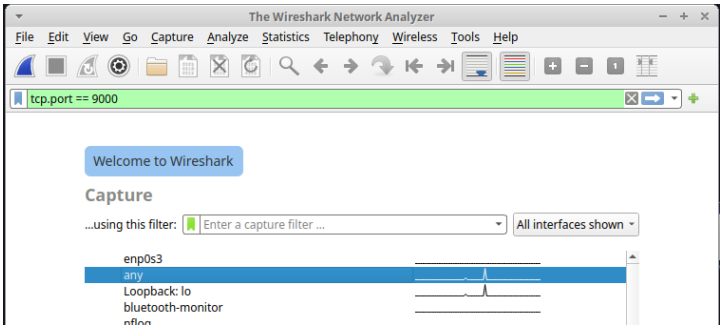
Part E - Network Traffic Is Real and Visible

Tool: wireshark

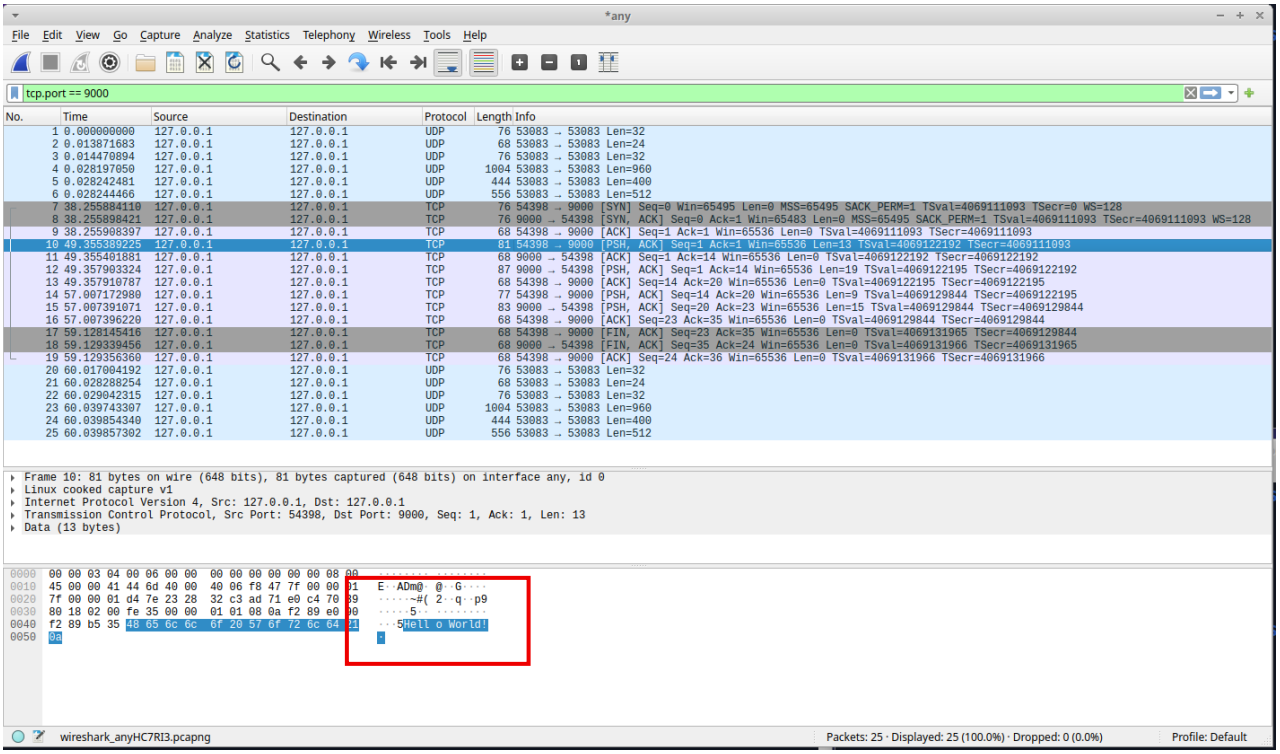
Concept: Java sockets generate real packets.

Start Wireshark (you will likely need sudo at the command line): sudo wireshark

Enter “tcp.port == 9000” as the filter and select “any” as the interface. Click the shark fin to start capturing packets.



Now start the EchoServer, use netcat, and exchange some messages as before. Stop the EchoServer and stop Wireshark collection. You can see TCP handshake information, packets, and even data!



Your code generates network traffic... And maybe encryption is needed?

Take a screenshot and save it in the taskw02 folder.

Part F - Firewalls Can Break Correct Code

Tool: iptables

Concept: The environment can block applications.

In one of the terminals, let us set up the Linux firewall to block input from port 9000:

```
sudo iptables  -I INPUT -p tcp -i lo --dport 9000 -j DROP
```

This command causes the firewall to drop any packets on that port, thus preventing them from reaching any application. There is a similar REJECT option instead of DROP; however, REJECT sends a notification back to the sender. Drop makes it less noticeable to an attacker. You can see the firewall rules, including the one you just added. Yours should be near the top of the INPUT chain.

```
sudo iptables -L --line-numbers
```

Start up the EchoServer again and use netcat to send messages like before. Notice nothing happens.

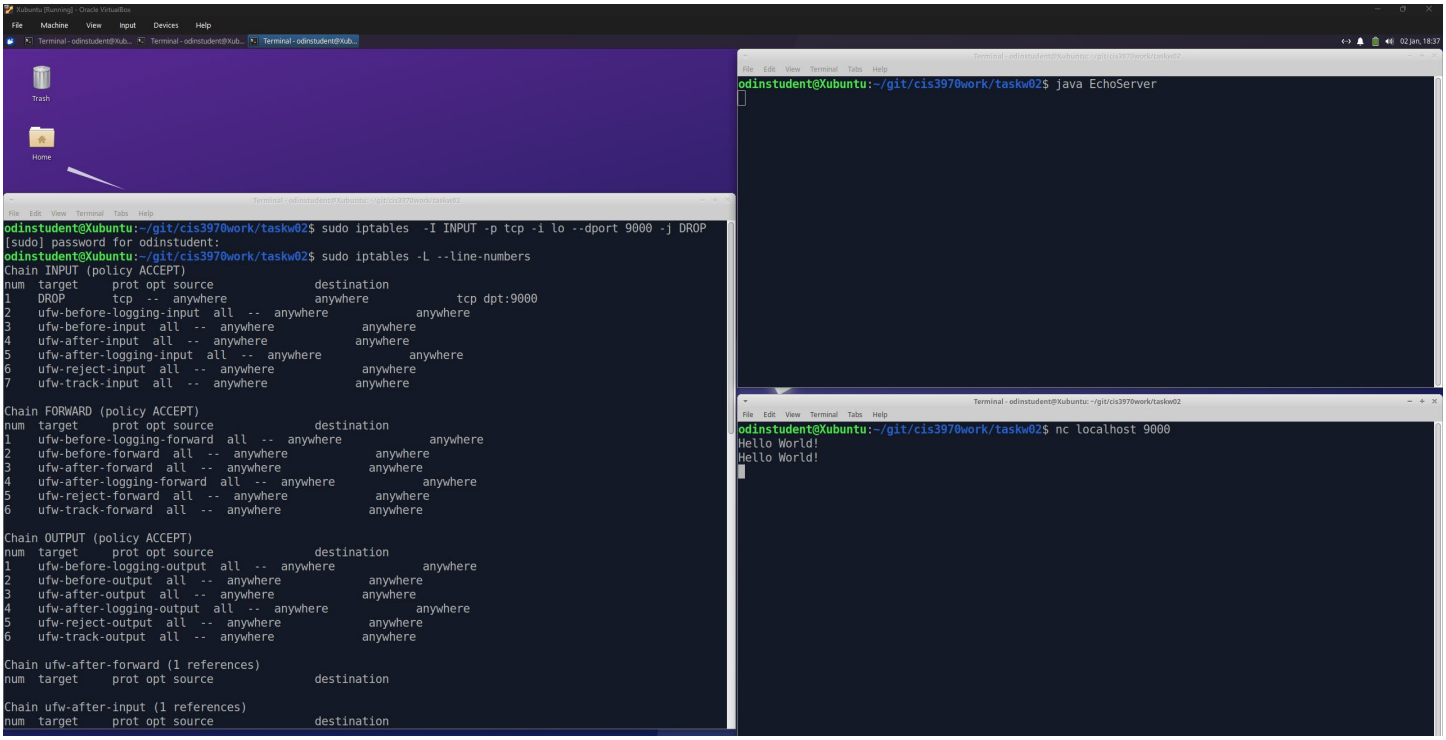
Let us remove the rule.

Find out which line it is in the rules: sudo iptables -L --line-numbers

Remove the rule (1 in my case): sudo iptables -D INPUT 1

It should work again. This means that the applications we write may need to have ports opened to allow traffic.

Take a screenshot and save it in the taskw02 folder.



Part G - Debugging Without a GUI

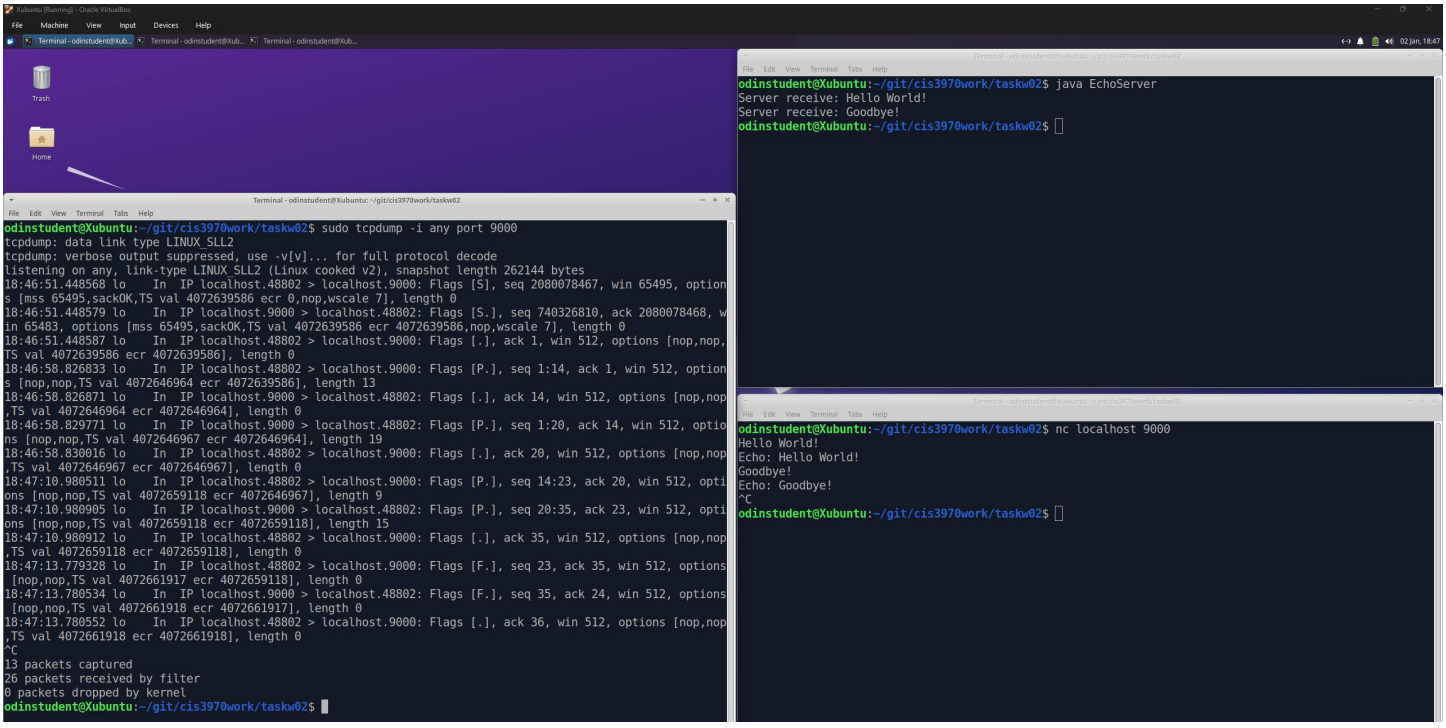
Tool: tcpdump

Concept: Network debugging works on servers, too.

In one of the terminals, start the tcpdump: sudo tcpdump -i any port 9000

Restart the EchoServer and rerun netcat to see what happens.

You can see everything the protocol did to get the messages communicated.



Take a screenshot and save it in the taskw02 folder.

Part H - Logging Beats Print Statements

Tool: Java logging + Linux output

Concept: Logs create a timeline of execution.

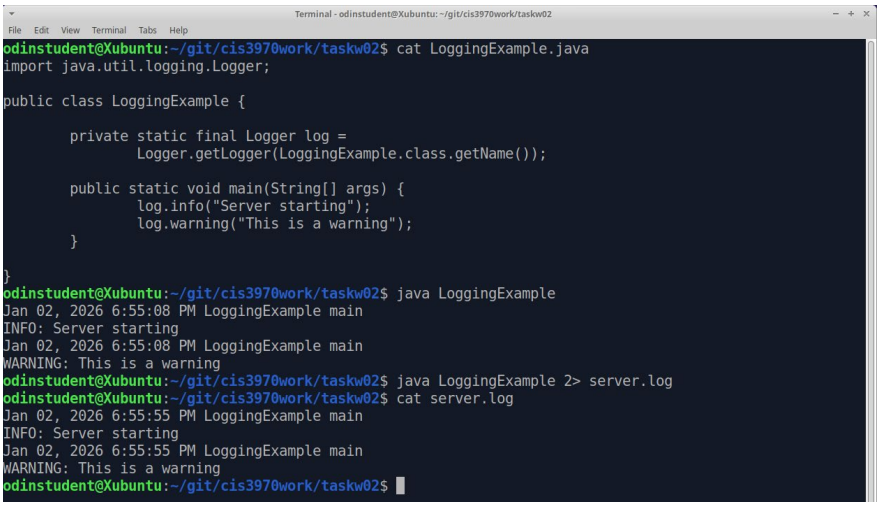
LoggingExample.java demonstrates the Java Logger, a great way to log activity generated instead of using print statements.

View the code: cat LoggingExample.java

Run the code: java LoggingExample

Redirect it to a file: java LoggingExample 2> server.log

View the log file: cat server.log



Take a screenshot and save it in the taskw02 folder.

Part I - Encryption Is a Layer, Not Magic

Tool: openssl

Concept: TLS wraps sockets

Issue the command: openssl s_client -connect example.com:443

Inspect the chain and identify the handshake steps. Secure sockets, which will be explored later this semester, are still sockets, but wrapped for security.

Take a screenshot and save it in the taskw02 folder.

Submission

When complete, remember to commit all the screenshots in the taskw02 folder to your git repository. Then return to Moodle and put “done” in the submission textbox.