Midterm Review II

Edward Agyeman(Eka3yt)

LOGISTICS

- Collab Online Exam(24 Hours - 12am exam day to 12am following day)
- **FORMAT?**: 10 Questions(4 or 5 wireshark questions, Coding questioning with sockets) (Short answers, multiple choice, and coding question, download and analyze trace)
- Open notes(Slides, Notes, Textbook, NO INTERNET OR PEER HELP)
- Exam should take 4 hours max, designed to be done in an 1 hour and 20 minutes
- Graham will be able during our scheduled class time to answer questions about the exam

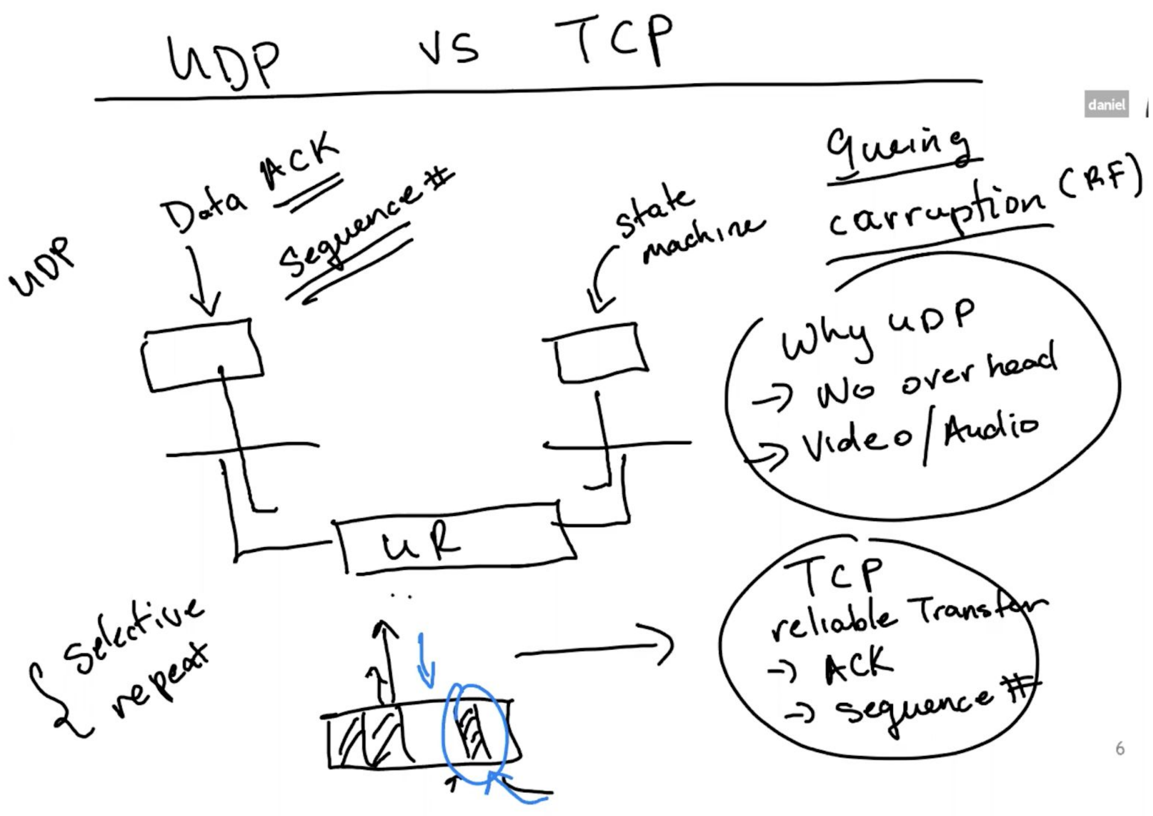
TOPICS

- Focused on Transport Layer and HTTP protocol
    - Get familiar with socket programming
    - HTTP 1.1
    - State Machines
    - Reliable Transports
    - Utilization, effects of windowing on utilization
    - Pipelining
    - On/Off - how we would know when to send off signal
    - Estimating RTT, Exponential weighting strategy
    - Three way handshake(looking at wireshark trace)
    - fin/message TCP
    - Sequence & ACK numbers, how they relate to one another(look at telnet example from slides)

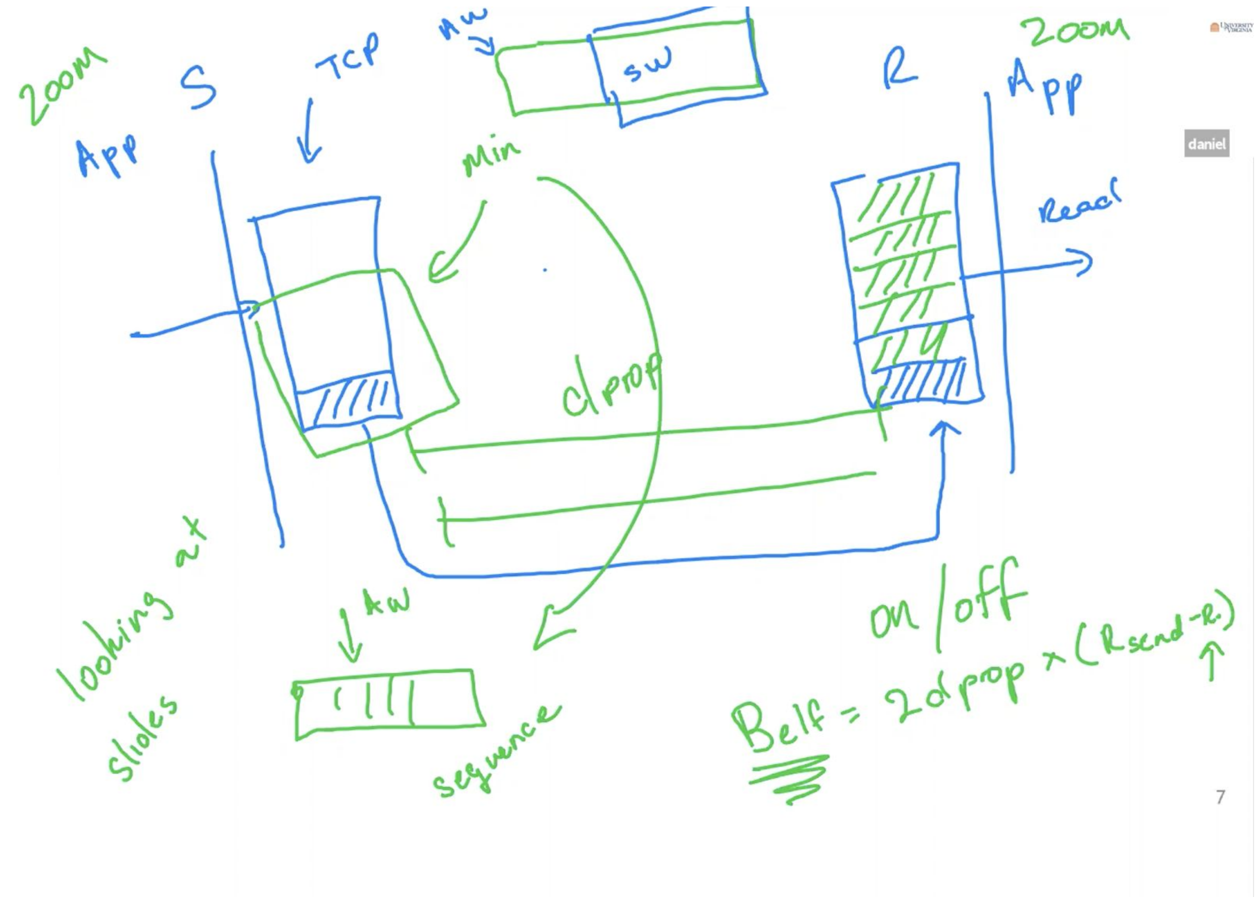
REVIEWED TOPICS

- **Given Example Coding Question**
  Write an application program that uses sockets to create a half duplex text messaging client?

UDP VS. TCP



- Fundamental idea with transport layer protocols is an unreliable network, which drops packets
- Reasons for drop: queuing, corruption(RF), Collisions, etc
- UDP-send data over unreliable network and not care if packets get lost
    - Why UDP? No overhead, and data like video and audio are lost tolerance, which leverages the unreliable nature of the network- great performance and better than waiting for recovered packets
- TCP- building a mechanism(ex. State Machine) that takes data and uses acknowledgements and sequence numbers, you can guarantee reliable, in-order delivery of packets



- (Application layer) Having a sender and a receiver, the sender receives a packet and buffers it. When ready, the sender sends the packets to the receiver
- **Problems** that can arise:
    - application layer of receiver does read quickly, leaving the the buffer of the receiver to be overwhelmed, causing packets to be dropped
      Solution(s):
      1. On/Off message: tells sender not to send anymore packets, done with a Belf/BLeft(space left on the buffer) = 2Dprop*(Rsend - Rreceive)
      2. Window: having a window size and a sequence number(look at slides)

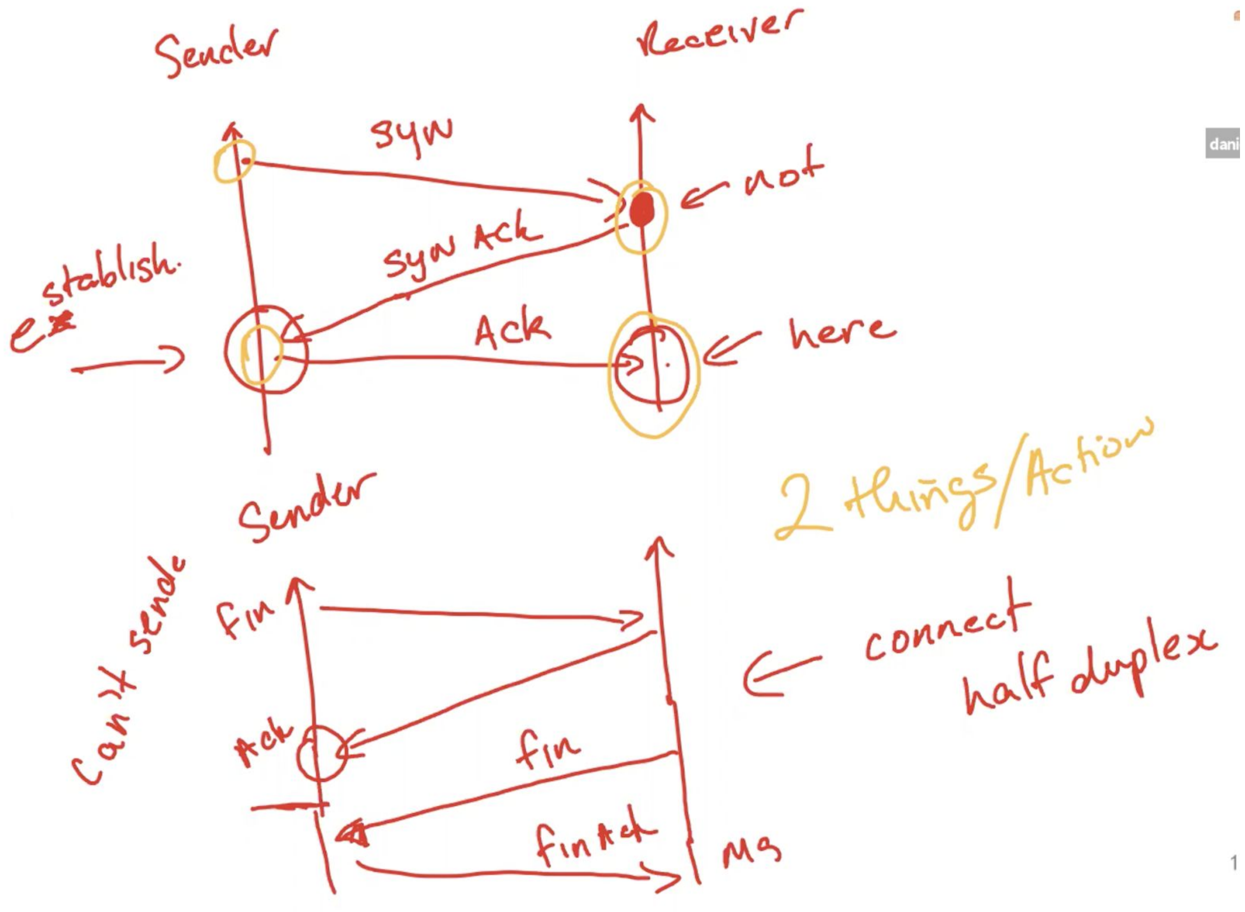
<u>Computation Style Questions</u>

- Utilization
- How much space is left in Buffer
- Calculate window size in relation to utilization
- Estimate RTT

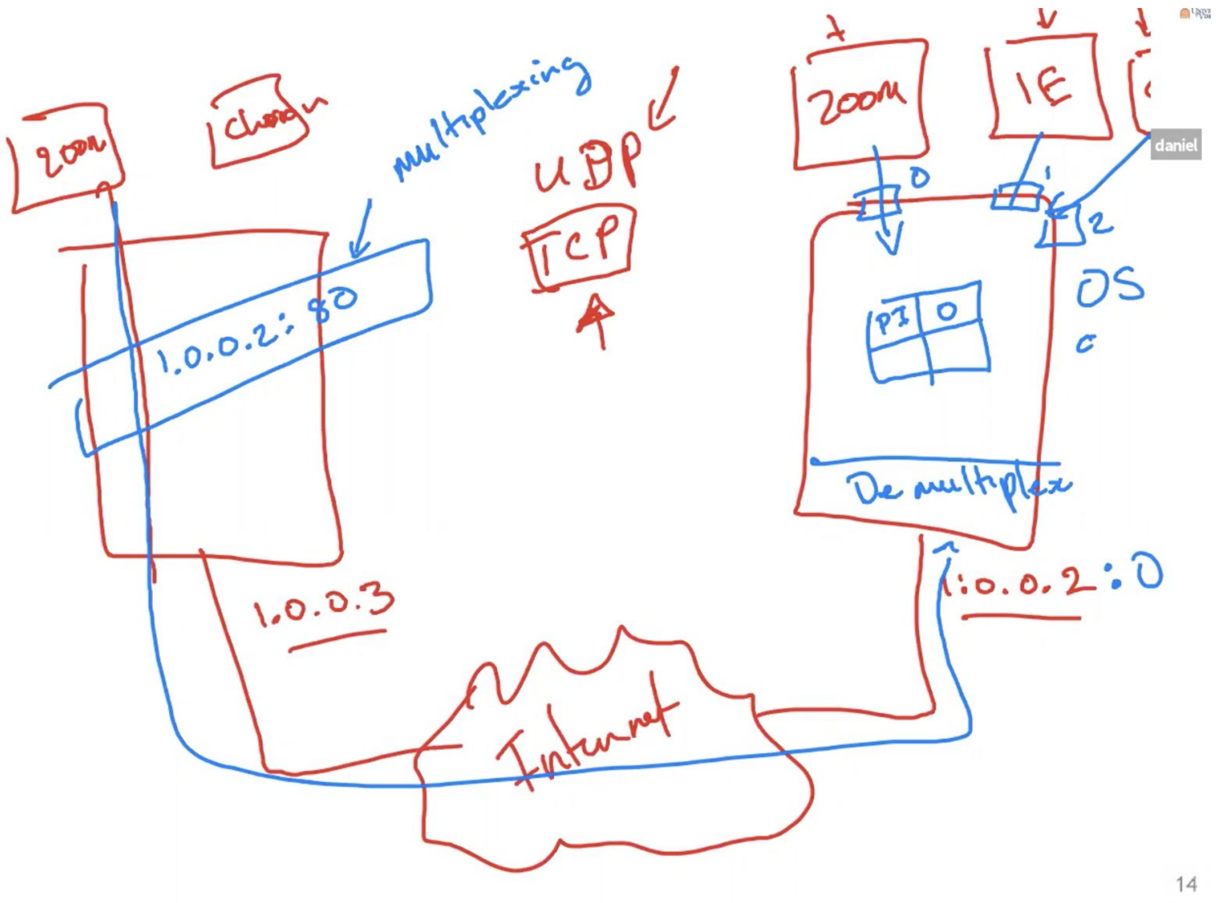
<u>Coding Style Question</u>

Write an application program that uses sockets to create a half duplex text messaging client?

Three way handshake

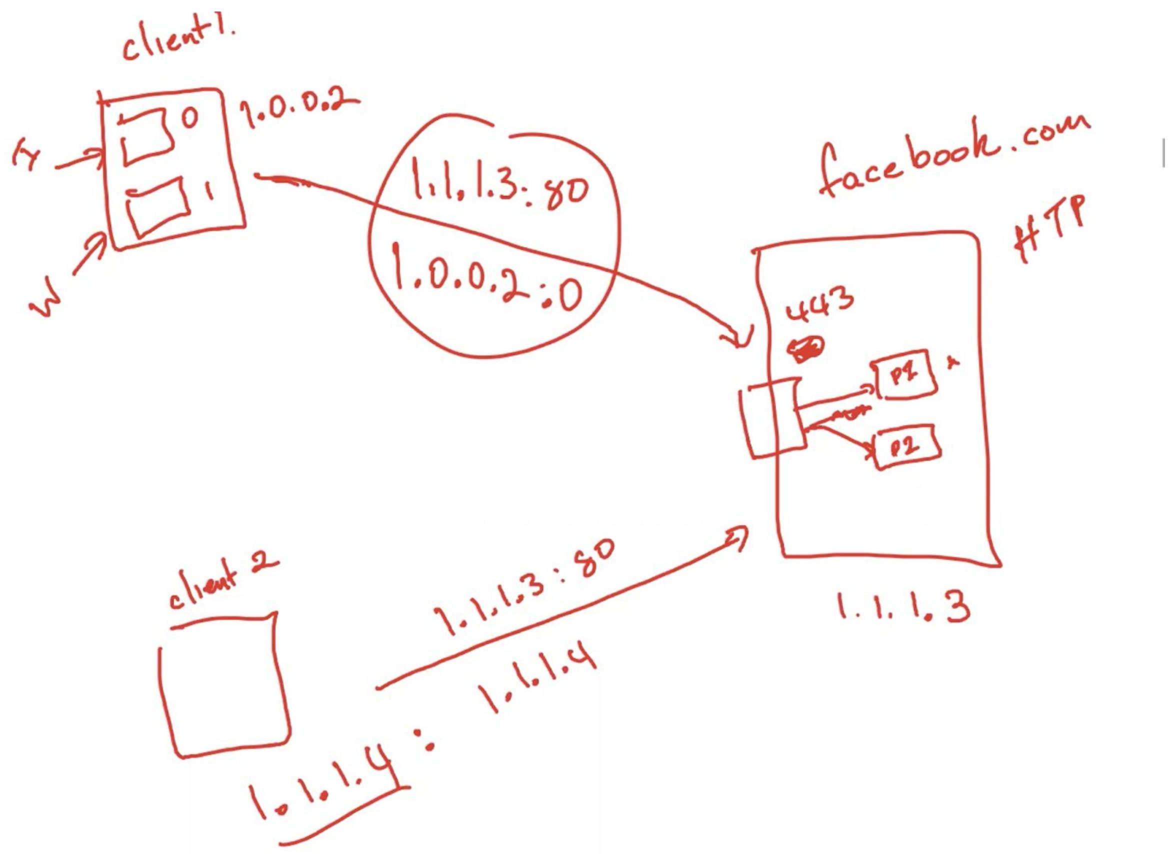


Multiplexing and Demultiplexing



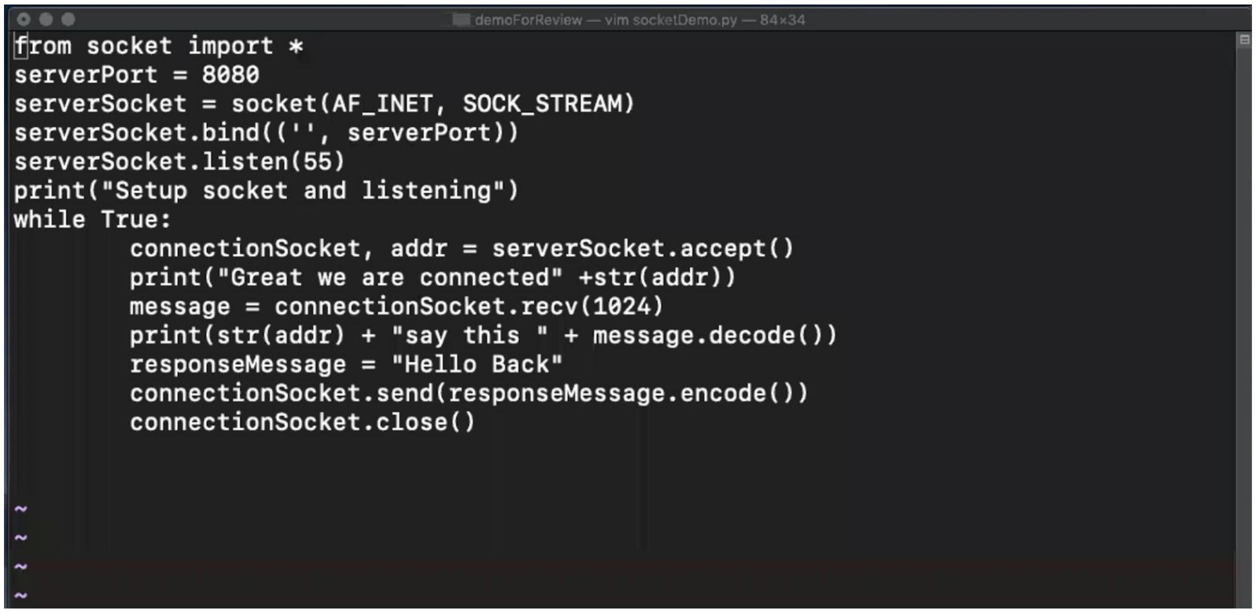
- You start out with hosts(1.0.0.3 and 1.0.0.2) and internet infrastructure. Hosts might be running multiple things(Ex. Zoom, IE, Chrome), each a process
- 1.0.0.3 sends zoom data to destination 1.0.0.2, **demultiplexing** involves figuring out which process the data is destined for on the 1.0.0.2 host. OS is involved with figuring that out. Each process creates a port and OS creates data structure for mapping the process ID and the port assignment.
- Data transmission includes extra port information(**multiplexing**)

- Addressing strategy for UDP and TCP. Example above is for UDP, TCP includes extra level, example below



- Extra level of complexing for TCP comes from multiple mappings between process
- Starting with a server(facebook.com- 1.1.1.3) and your clients(1.0.0.2 and 1.1.1.4)
- Facebook process is running on port 443(more secure than port 80)
- Sending data from 1.1.1.4 to 1.1.1.3: 443 and from 1.0.0.2 to 1.1.1.3: 443
- There is a separate process for each individual client, using the sender's info to help with mapping

Live Server TCP Socket Code Example


demoForReview — vim socketDemo.py — 84×34

```
from socket import *
serverPort = 8080
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(55)
print("Setup socket and listening")
while True:
        connectionSocket, addr = serverSocket.accept()
        print("Great we are connected" +str(addr))
        message = connectionSocket.recv(1024)
        print(str(addr) + "say this " + message.decode())
        responseMessage = "Hello Back"
        connectionSocket.send(responseMessage.encode())
        connectionSocket.close()



~
~
~
~
```

Lines

1. Imports socket library
2. Sets port to 8080, higher number because lower ports are taken already
3. Creating a socket of type of AF_INET(ipv6) of SOCK_STREAM(TCP)
4. Binding current process to socket, leaving empty string defaults to machines IP. passing in serverPort/8080
5. Sets number of connections to possibly listen for
6. Print statement for user reassurance
7. Constant loop
8. Accepts connection, Performs three way handshake. Returns a socket and address of type tuple(ip address, port)
9. Print statement for user reassurance
10. Receive message - window size 1024 bytes, returns content
11. Printing out message from sender
12. Response string
13. Sending response string back to sender, needs to be encoded(encoding needed on mac and some windows machines)
14. Closing connections, sending fin messages