ECE361 Lab 3 Report

Ian Hu 1006939244

Santiago Ibagon 1006831591

Ethan Sovde 1006747040

Shawn Zhai 1006979389

5 What are TCP and UDP Sockets?

1. In addition to the type of data, what other aspects of UDP protocol that you studied in your classroom makes it suitable for communicating video and audio files? (Hint: maybe it has something to do with the UDP header size.)

UDP is suitable for transmitting video and audio files not only because of the type of data it supports but also due to its efficiency in handling data transmission. Its header size is only 8 bytes, much smaller than TCP's 20 bytes, which reduces the communication overhead and increases speed. Additionally, UDP's connectionless design allows data to be sent without the need for handshake or connection establishment, which is ideal for real-time applications. This simplicity ensures low latency and quick data transfer, making it perfect for scenarios where minor data loss is tolerable, such as streaming video or audio.

2. If UDP is more suitable than TCP for communicating video and audio files, why online video sharing platforms (e.g., YouTube) use HTTP and HTTPs for video and audio delivery? Don't HTTP and HTTPs use TCP?

Although UDP is faster and more efficient for real-time data like video and audio, platforms like YouTube use HTTP and HTTPS, which rely on TCP, because they prioritize reliability and compatibility. TCP ensures that all packets are delivered in order, eliminating the possibility of missing or corrupted data, which is critical for ensuring high-quality video playback. Additionally, TCP handles congestion control and flow management, making it ideal for delivering content over diverse network conditions. While UDP sacrifices reliability for speed, TCP guarantees a smooth and uninterrupted user experience by resending lost packets and maintaining a consistent data flow. Moreover, video platforms use adaptive streaming techniques over TCP to adjust video quality dynamically based on network speed, addressing latency concerns typically associated with TCP. These features make TCP more suited for large-scale content delivery despite UDP's inherent speed advantages.

3. Why do you think we need UDP when we already have a reliable transport layer protocol like TCP?

UDP remains a necessity even with the availability of TCP because it offers a lightweight and faster option for applications prioritizing low latency over guaranteed delivery. For real-time activities such as online gaming, live video streaming, or voice-over-IP, UDP provides the speed and simplicity required for a seamless user experience. Unlike TCP, UDP avoids the overhead of establishing and maintaining a connection, allowing it to handle scenarios where minor data loss does not significantly affect the application’s functionality. Its ability to broadcast or multicast data also makes it indispensable for specific use cases like live events or group communication.

4. Why not having UDP as the only transport layer protocol is a good idea? Can’t applications develop their own transport needs?

Relying solely on UDP would not be practical because TCP offers essential features like congestion control, guaranteed delivery, and data ordering, which are critical for many applications. If UDP were the only transport protocol, developers would need to implement these mechanisms manually for every application, increasing complexity and risk of inconsistencies. Additionally, TCP’s built-in reliability ensures compatibility across diverse networks and devices, reducing the need for custom solutions. Having both TCP and UDP allows applications to choose the protocol that best suits their requirements without reinventing the wheel, thus ensuring efficient and robust communication for a wide range of use cases.

6 TCP Socket in a nutshell

6.1 TCP Client and Server Interaction Model

1. Why the same as the TCP server the TCP client does not bind to an IP address and port number?

In the TCP communication model, the client does not typically bind to a specific IP address and port number because it does not need to reserve resources or act as a persistent listener for incoming connections. The client’s primary role is to initiate the connection to a server, which is achieved by specifying the server's IP address and port during the connection process. The operating system automatically assigns an ephemeral port to the client, which is dynamically allocated from a pool of available ports. This simplifies the client’s implementation and allows multiple clients on the same machine to communicate with the same server simultaneously without port conflicts.

2. What happens if the TCP client also uses the bind function?

If the TCP client explicitly binds to an IP address and port number, it forces the client to use that specific port for outgoing communication. While this might be useful in scenarios where the client must use a fixed port (e.g., for firewall rules or legacy systems), it can lead to complications. For instance, if another process is already using the specified port, the binding will fail, and the client will not be able to establish a connection. Additionally, binding to a fixed port may limit the ability to run multiple instances of the client on the same machine, as port conflicts can occur.

3. How can the server use client-side binding to improve server security?

Servers can use client-side binding information (i.e., the client's IP address and port) to enhance security by implementing access control and filtering mechanisms. For example, the server can maintain a whitelist of allowed client IP addresses and ports, rejecting connection requests from untrusted or unknown sources. Additionally, the server can monitor client-side binding information to detect and mitigate potential attacks, such as IP spoofing or port scanning. By tracking and validating the source IP and port of incoming connections, the server can ensure that only legitimate clients are granted access, thereby improving overall security.

6.2 Simple TCP Client and Server Code in Python 3

1. Since server awaits new clients connections, what happens if two or more clients attempt to connect to the server?

The server.listen() function in the code specifies that the server will wait for incoming connections. While the code does not explicitly set the backlog queue size in listen(), the default behavior allows multiple clients to attempt connections. The server handles each connection sequentially using accept(). If too many clients attempt to connect and the queue size exceeds the default (or specified) limit, additional connection attempts may be refused.

2. What does happen if a client process close the socket but the server does not?

In the provided server code:

read_buffer = connection.recv(1024)

If the client closes its socket, the server will detect an EOF when trying to receive data from the client. Since the server does not close the socket explicitly (except with server.close()), the resources associated with the connection remain allocated until explicitly released.

3. What does happen if the server uses connection.close() in line 20 of the server code instead of server.close()? Explain if a change to connection.close() in line 20 would be useful for the server side?

Using connection.close() in line 20 instead of server.close() closes only the specific client connection, allowing the server’s main listening socket to remain open and accept new connections. This change is useful as it ensures the server can handle multiple clients sequentially or concurrently, releasing resources for each client after interaction without terminating the server. In contrast, server.close() shuts down the entire server, preventing further connections, which is not practical for a multi-client setup. Thus, connection.close() is the better choice for scalability and continuous operation.

4. Which endpoint does the IP address “127.0.0.1” refer to? Explain how a socket works when the IP address of the server and client computers are the same? Isn’t a socket a form of network IPC?

The IP address 127.0.0.1 refers to the loopback interface, enabling network communication within the same host. When both the server and client use 127.0.0.1, the data flows entirely through the local networking stack, simulating external communication without involving physical network interfaces. This setup allows processes on the same machine to communicate using standard networking protocols.. A socket, whether used locally or across machines, is a form of network IPC, abstracting the complexities of data transmission and providing a consistent interface for processes to exchange information.

7 UDP Socket in a nutshell

1. If the UDP server does not need to manage connections anymore, how does the server know how many clients are connected to the server?

A UDP server does not maintain connections, so it inherently has no built-in mechanism to track the number of clients "connected" to it. Instead, the server identifies clients based on the source IP address and port number included in each received message. If needed, the server can maintain a list of these client identifiers (source IP and port) to approximate the number of clients that have interacted with it. However, this tracking is entirely application-specific and must be implemented manually, as UDP is connectionless and stateless by design.

2. How can a UDP client know whether a message has been delivered to the server?

A UDP client cannot inherently know whether its message has been delivered because UDP does not provide delivery guarantees or acknowledgments. To confirm delivery, the client must rely on application-level mechanisms, such as implementing a response system where the server explicitly sends an acknowledgment message upon receiving data. Without such mechanisms, the client remains unaware of delivery status, as UDP does not provide feedback for lost or undelivered packets.

7.1 Simple UDP Client and Server Code in Python 3

1. What happens to a UDP client process if the UDP server closes its socket?

The UDP client does not receive any notification if the server closes its socket, as UDP is connectionless and does not maintain a persistent link between client and server.

```
read_buffer = client.recvfrom(1024)
```

If the server has closed its socket, the line above will not return any data, and the client will simply continue operating without knowing the server is unavailable. The client can still send messages, but they will not be received or acknowledged.

2. What does socket.SOCK_DGRAM parameter refer to in the UDP client and server codes? How is it different from the socket.SOCK_STREAM in the TCP client and server codes?

socket.SOCK_DGRAM is used to specify that the socket is a datagram socket, which operates using UDP. In contrast, socket.SOCK_STREAM is used for TCP, which provides a connection-oriented, reliable stream of data. The main difference is that SOCK_DGRAM does not establish a connection or guarantee delivery, while SOCK_STREAM requires a connection and ensures reliable, ordered data transfer.

3. Since UDP is connection-less, can a UDP server receives a UDP client message if the server runs after the client has sent its message to the server?

No, the server will not receive messages sent before it starts running. In the provided server code:

```
read_buffer = server.recvfrom(1024)
```

The server must be actively running and bound to the appropriate IP and port to receive messages. If the client sends a message before the server executes this line, the message will be lost because UDP does not queue messages for servers that are not currently active.

4. How does the port number 2024 we use in the UDP client/server codes is different from the same port number in the TCP client/server codes?

The port number 2024 in the UDP client/server codes is entirely separate from the same port number in the TCP client/server codes. Although they share the same numerical value, UDP and TCP operate independently, and the operating system maintains separate port spaces for each protocol. This means that using port 2024 for a UDP socket does not conflict with using port 2024 for a TCP socket. They are treated as distinct by the networking stack.

5. If UDP offers an unreliable and connection-less service, why don't we directly use IP protocol when we need unreliable and connection-less communication? We note that IP protocol also offers an unreliable and connection-less service in the network layer. (Hint: maybe UDP ports that are offered in the transport layer play a role in the answer).

Although IP provides an unreliable and connectionless service similar to UDP, we use UDP instead of directly relying on IP because UDP operates at the transport layer and introduces additional functionality, most notably port numbers. These ports allow multiple applications or processes to communicate over the network simultaneously by differentiating traffic based on port numbers. Without UDP or a similar protocol, there would be no standardized way for the network layer (IP) to distinguish between data meant for different applications on the same device.

Moreover, UDP provides a minimal structure for application data with its header, which includes source and destination port numbers, length, and checksum. The checksum helps verify the integrity of transmitted data, something IP does not guarantee. These features make UDP lightweight but more functional and practical than directly using the IP protocol, which lacks the application-layer demultiplexing provided by ports and the optional error detection facilitated by the checksum.