TCP, UPD, Socket Programming

Contents

[TCP and UDP Overview 2](#_Toc185531395)

[Network Protocols, UDP, and TCP 4](#_Toc185531396)

[TCP Three-Way Handshake: 6](#_Toc185531397)

[TCP and Data Communication 9](#_Toc185531398)

[Socket Programming 12](#_Toc185531399)

[Client-Server Socket Communication Basics 15](#_Toc185531400)

[Client Code Steps: 15](#_Toc185531401)

[Server Code Steps: 19](#_Toc185531402)

Agenda

* Discuss two important transport layer protocols: TCP and UDP.
* Understand what TCP is and how it works.
* Understand what UDP is and its characteristics.
* Compare TCP and UDP.
* Explore how TCP works internally.
* Introduction to the Socket API used for network communication.
* Understand how applications connect over the network using the Socket API.
* Learn about ephemeral ports and how codes are created.

## TCP and UDP Overview

1. **Transport Layer in Networking:**
   * The transport layer lies just below the application layer in the OSI model.
   * When the application layer needs to send data, it hands it to the transport layer.
   * The transport layer's responsibility is to ensure data is correctly delivered, in the correct order, and without loss.
2. **Analogy of Transport Layer:**
   * Think of the transport layer as a "transporter" in a moving scenario, ensuring data (or "furniture") is delivered safely and in the correct order.
3. **Segmentation of Data:**
   * In the transport layer, the data from the application layer is divided into "segments" in TCP and "datagrams" in UDP.
   * TCP and UDP are the two main transport layer protocols.

**Key Differences Between TCP and UDP**

1. **Reliability:**
   * **TCP** is **reliable**: It ensures data is delivered in the correct order, with no loss or corruption. It guarantees data integrity and delivery.
     + Analogy: TCP is like a guaranteed delivery service (e.g., insured parcel delivery).
   * **UDP** is **best-effort**: It does not guarantee delivery, order, or data integrity. It simply tries to deliver data as best as it can, but with no promises.
     + Analogy: UDP is like a standard courier service with no guarantees.
2. **Connection Type:**
   * **TCP** is **connection-oriented**: It establishes a connection between the sender and receiver before data transfer begins. This ensures reliable delivery and ordered transmission.
     + Analogy: TCP builds a "highway" for data transmission.
   * **UDP** is **connectionless**: It sends data as independent packets without establishing a connection beforehand. Each packet is handled separately.
     + Analogy: UDP is like driving without a predetermined route; each packet goes independently.
3. **Resource Usage:**
   * **TCP** uses more resources than UDP because it handles things like data order, error correction, and data retransmission.
   * **UDP** uses fewer resources as it does not provide the same guarantees and simply sends data as it is.
4. **Header Size:**
   * **TCP** headers are larger because they contain additional information to ensure reliable delivery, error checking, and sequencing.
   * **UDP** headers are smaller because it does not include these guarantees.
5. **Speed:**
   * **TCP** tends to be slower because of the extra overhead associated with reliability and error-checking mechanisms.
   * **UDP** is faster due to its simpler header and lack of reliability features.

**Use Cases for TCP and UDP**

1. **TCP Use Cases** (where reliability is needed):
   * **File transfers**, **emails**, **web browsing** where data integrity and proper order of delivery are essential.
2. **UDP Use Cases** (where speed is more important than reliability):
   * **Video streaming** (e.g., YouTube) where missing a few packets (e.g., pixels) doesn’t significantly impact user experience.
   * **Voice over IP (VoIP)** and **online gaming**, where low latency and fast data transmission are more critical than ensuring every packet arrives in the correct order.

## Network Protocols, UDP, and TCP

**1. Importance of Order vs. Speed in Communication**

* **Speed vs. Order**: In some systems, like voice calls and video streaming, speed is more important than strict order. For instance:
  + In voice calls (e.g., Voice over IP), speed is crucial. A delay of even 1-2 seconds could disrupt the experience, but small variations in the order of data packets may not be noticeable.
  + Similarly, in video streaming, if a few frames are missed, it generally won’t be noticeable to the viewer because human eyes can’t process every frame.
* **Video Streaming**: If a packet for video data is received late (from a previous frame), it is ignored. The system prioritizes faster delivery, not perfect accuracy.

**2. UDP (User Datagram Protocol) and TCP (Transmission Control Protocol)**

* **UDP Characteristics**:
  + **Faster but Less Reliable**: UDP is used where speed is more important than accuracy, like in real-time applications (gaming, voice calls).
  + **No Guarantees**: UDP doesn’t guarantee packet delivery or order, and it doesn’t have mechanisms to ensure data integrity.
  + **Usage**: Often used in gaming (e.g., PUBG) and voice communications, where speed takes precedence over order.
* **TCP Characteristics**:
  + **Reliable but Slower**: TCP ensures data is delivered in order, guarantees accurate delivery, and checks for data integrity.
  + **Protocols that use TCP**: HTTP, web pages, file transfers.
  + **Header Details**:
    - **Sequence Numbers and Acknowledgment**: TCP assigns a sequence number to each segment to ensure proper ordering and uses acknowledgments to confirm successful delivery.
    - **Checksum**: Ensures data integrity and detects errors.
  + **Applications**: Used in scenarios where the reliability and accuracy of data delivery are more important than speed (e.g., HTTP).

**3. Combining TCP and UDP**

* It’s possible to use both TCP and UDP for different parts of a communication, depending on the need for speed or reliability.
  + Example: Video streaming might use UDP for real-time video data, but HTTP-based requests (e.g., recommendations) would use TCP for accuracy.

**4. Network Analysis Tools (Wireshark)**

* **Wireshark**: A tool for inspecting network traffic and analyzing protocols. It shows which protocol (TCP or UDP) is used in different requests.
* **Packet Details**: You can see source and destination ports, headers, and sequence numbers.

**5. TCP and UDP Headers**

* **TCP Header**: Includes information like source/destination port, sequence numbers, acknowledgment numbers, checksum, and more. TCP’s header is larger and more complex because it includes more features for reliability and order.
* **UDP Header**: Much simpler and smaller. It includes just the source/destination port, length, and checksum.
* **TCP vs. UDP Header Size**: TCP’s larger header makes it slower but more reliable. UDP’s small header makes it faster.

**6. Checksum and Data Integrity**

* **TCP** has a more stringent checksum for ensuring data integrity, whereas **UDP**’s checksum is simpler and may not catch as many errors.
* **Packet Loss**: In UDP, packet loss might occur without any recovery mechanisms, while TCP ensures data is resent if lost.

**7. Application Port Numbers**

* Both TCP and UDP use port numbers to identify the source and destination applications. These port numbers help route the data to the correct application on the receiving machine.

**8. Internet Protocols for Voice and Video Calls**

* **Voice over IP (VoIP)**: Typically uses UDP for faster transmission, as speed is more important than perfect order.
* **Video Streaming**: Can use both TCP (e.g., HTTP video streaming) or UDP (e.g., real-time streaming protocols), depending on the application’s need for speed versus accuracy.

**Conclusion:**

* The decision between using UDP or TCP comes down to the trade-off between speed and reliability.
* For applications like gaming and voice calls, where speed is essential, UDP is preferred. For applications like HTTP, where accuracy and reliability are critical, TCP is used.

## TCP Three-Way Handshake:

**Introduction to TCP and its Guarantees:**

* **TCP (Transmission Control Protocol)** ensures reliable data transfer between a client and a server by providing several guarantees:
  + Data correctness
  + Data delivery
  + Retries for undelivered data
* These guarantees are implemented using the **Three-Way Handshake** process before data transfer begins.

**Entities Involved in TCP Communication:**

* Communication always occurs between **two entities**:
  + Client and Server (or in peer-to-peer communication, it’s peer-to-peer communication, but still between two entities).
  + In broadcasting, multiple pairwise communications still occur.

**What is the Three-Way Handshake?**

* Before starting data transfer, TCP establishes a connection using a **three-way handshake**:
  1. **Client sends a SYN (synchronize) request** to the server.
  2. **Server acknowledges (SYN-ACK)** the client’s synchronization request and sends its own.
  3. **Client sends an ACK (acknowledgement)** to the server confirming synchronization and readiness to proceed.
* This handshake ensures that both entities are synchronized and ready for data transfer.

A diagram of a computer network

Description automatically generated

**Analogy of Three-Way Handshake:**

* Imagine three friends (A, B, and C). A and B want to make fun of C.
  + A signals B to get synchronized.
  + B acknowledges and agrees to participate.
  + A acknowledges B’s participation and confirms synchronization.
  + This is analogous to the **three-way handshake** in TCP:
    - **SYN**: A signals B (Client sends SYN).
    - **SYN-ACK**: B acknowledges and agrees (Server sends SYN-ACK).
    - **ACK**: A acknowledges B’s participation (Client sends ACK).

**Why Three Messages Are Needed (Why Not Just Two?):**

* **Single message** (client sends SYN) is not sufficient due to potential **data loss** in networks:
  + If the **SYN** message from the client is lost, the server never knows the client’s intention to connect. The client won’t be aware of this loss, and no further communication can occur.
  + If there is **no acknowledgment** (ACK) for the initial SYN-ACK message, the client and server cannot confirm synchronization.
* With **three messages**:
  + If the SYN message is lost, the client will resend it, ensuring the server eventually receives it.
  + If the SYN-ACK message is lost, the server will resend it, ensuring the client eventually receives it.
  + This ensures that both entities are synchronized and confirms data will flow.

**TCP Retries and Acknowledgments:**

* Data loss in TCP (during any message exchange) is handled by retransmission. For example, if:
  + **Client’s SYN message is lost**, the client will eventually retry after a timeout.
  + **Server’s SYN-ACK is lost**, the server will resend it after a timeout.
  + **ACK messages are lost**: If the client or server does not receive the ACK, it will resend the message to ensure synchronization.

**Conclusion of Three-Way Handshake:**

* The **three-way handshake** ensures reliable communication:
  + **SYN** initiates the connection.
  + **SYN-ACK** confirms synchronization.
  + **ACK** finalizes the connection and ensures both entities are ready for data transfer.
* Without the three-way handshake, there would be no certainty that both sides are synchronized, leading to potential communication failure due to packet loss.

**Summary:**

The Three-Way Handshake is a crucial step in TCP to ensure that both the client and server are synchronized and ready for data exchange. It helps prevent issues like data loss and ensures that both sides agree to the connection before any data transfer begins.

## TCP and Data Communication

**Key Concepts in TCP Communication**

1. **Sequence Numbers:**
   * Both client and server maintain independent sequence numbers for their messages.
   * Sequence numbers ensure ordered data delivery and allow missing segments to be identified.
   * They can start from any number (e.g., 1521 or 241).
2. **Three-Way Handshake (Connection Establishment):**
   * **Step 1:** Client sends a synchronization (SYN) request with its sequence number (e.g., 1521).
   * **Step 2:** Server acknowledges (ACK) the client’s sequence and sends its own sequence number (e.g., 241).
   * **Step 3:** Client acknowledges the server’s sequence number.
   * This handshake ensures both parties are synchronized before data transfer begins.
3. **Data Transfer with Acknowledgments:**
   * After the handshake, the client sends data packets in sequence (e.g., 1522, 1523).
   * The server acknowledges each packet (e.g., ACK 1523) and indicates readiness for the next one.
   * Similarly, the server sends its data with its sequence numbers, acknowledged by the client.
4. **Reordering and Missing Segments:**
   * TCP can rearrange out-of-order packets using sequence numbers before delivering them to the application.
   * If a segment is missing, the receiver requests retransmission (e.g., "Resend 1522").
5. **Connection Termination:**
   * **FIN Flag:** Signals the end of communication.
   * If no acknowledgment or data is received for some time, the connection is considered lost and terminated.
6. **Headers and Flags in TCP:**
   * **SYN Flag:** Indicates synchronization requests during the handshake.
   * **ACK Flag:** Acknowledges received segments.
   * **FIN Flag:** Signals connection termination.
   * TCP headers contain this information along with sequence and acknowledgment numbers.

**TCP vs. UDP**

1. **TCP (Transmission Control Protocol):**
   * Reliable, connection-oriented.
   * Ensures order and error-free data delivery.
   * Uses sequence numbers and acknowledgments.
2. **UDP (User Datagram Protocol):**
   * Connectionless, faster.
   * No guarantees for order or delivery.

**Additional Points**

* **Segmentation:** Large files (e.g., 1GB) are split into smaller packets, each with an order number.
* **Connection Continuity:**
  + Data and headers are sent together after the handshake.
  + Retransmissions occur if acknowledgments are not received.

**Practical Uses**

* TCP is used for applications requiring reliable and ordered data delivery, e.g., file transfers, web browsing.
* UDP is preferred for speed-critical tasks like video streaming or online gaming.

## Socket Programming

**1. What is Socket Programming?**

* **Socket**: An object/interface provided by the operating system, used by applications to send and receive data over a network.
* **Analogy**: Similar to how you use a file object to interact with the filesystem, you use a socket object to interact with the network.
* **Socket API**: Operating systems provide socket APIs (usually in C/C++). High-level programming languages (e.g., Python, Java) provide wrappers around these APIs.
  + Application developers use the language’s wrapper API.
  + The wrapper internally calls the operating system’s socket API.

**2. Python and Sockets**

* Python’s socket API wraps the operating system’s socket API.
* It simplifies writing client-server applications.

**3. Client-Server Model Using Sockets**

**Client Side:**

1. **Create Socket**: The client creates a socket object.
2. **Connect**: The client connects to the server using the server’s port.
   * The server’s port is predefined or shared with the client.
3. **Send Data**: The client sends requests to the server.
4. **Receive Response**: The client receives responses from the server.
5. **End Connection**: The client disconnects once the communication is complete.

**Server Side:**

1. **Create Socket**: The server creates a socket object.
2. **Bind**: The server binds to a specific port.
   * This step informs the operating system to route incoming data on this port to the server.
3. **Start Listening**: The server begins listening for incoming connections.
4. **Accept Connection**: The server accepts a connection request from a client.
5. **Receive Data**: The server receives data from the client.
6. **Send Data**: The server sends responses back to the client.
7. **End Connection**: The server disconnects after completing the communication.

**4. Key Differences Between Client and Server Operations**

|  |  |
| --- | --- |
| **Client** | **Server** |
| Creates a socket object. | Creates a socket object. |
| Connects to the server. | Binds to a port and starts listening. |
| Sends requests to the server. | Accepts connections from clients. |
| Receives responses from the server. | Receives and processes client data. |
| Disconnects after communication. | Disconnects after communication. |

**5. Ephemeral Ports**

* **Client**: Uses ephemeral ports (temporary, automatically assigned).
* **Server**: Binds to a well-known port (e.g., port 80 for HTTP).

**6. Protocols and Socket Programming**

* Sockets are protocol-agnostic. They support communication over:
  + **TCP**: Reliable, connection-oriented.
  + **UDP**: Unreliable, connectionless.
* The choice of protocol determines behaviour (e.g., three-way handshake for TCP).

**7. Lifecycle of a Socket**

**Client Perspective:**

1. Create a socket object.
2. Connect to the server (via port).
3. Send data.
4. Receive data.
5. Disconnect.

**Server Perspective:**

1. Create a socket object.
2. Bind to a port.
3. Start listening.
4. Accept connections.
5. Receive data.
6. Send data.
7. Disconnect.

**8. Three-Way Handshake**

* **When it Happens**: During the connection step for TCP.
* **Who Performs It**: The operating system handles the handshake when the application requests a connection using the socket API.
* **Layer**: The application code resides at the application layer; the handshake occurs at the transport layer (handled by the OS).

**9. Summary**

* Socket programming enables network communication between applications.
* Python’s socket API simplifies interaction with the operating system’s socket API.
* The client initiates connections, while the server listens and processes requests.
* TCP/UDP protocols define how data is transmitted.
* The lifecycle of a socket involves creating, connecting, data exchange, and disconnection.

## Client-Server Socket Communication Basics

1. **Client and Server Files**:
   * The client initiates a connection to the server.
   * The server listens for incoming client connections and processes them.
2. **IP Address and Port**:
   * The server binds to an **IP address** and a **port** (e.g., 127.0.0.1:8838).
   * The client connects to the server's IP and port.

## Client Code Steps:

1. Create a Socket Object:

|  |
| --- |
| auto clientSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);  if (clientSocket == INVALID\_SOCKET) {  std::cerr << "Socket creation failed!" << std::endl;  WSACleanup();  return 1;  } |

* AF\_INET specifies that the socket will use the IPv4 protocol. AF\_INET6 for IPv6
* SOCK\_STREAM specifies that the socket type is streaming (TCP).
* IPPROTO\_TCP specifies the TCP protocol.

1. Setup Server Address:

|  |
| --- |
| sockaddr\_in serverAddr; serverAddr.sin\_family = AF\_INET;  serverAddr.sin\_port = htons(PORT);  serverAddr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP); // Use localhost IP |

* The sockaddr\_in structure is used in C and C++ network programming to specify an endpoint address for IPv4. It contains information about the address family, port number, and IP address.
* The sockaddr\_in6 structure for IPV6.

1. Connect to the Server:

|  |
| --- |
| if (connect(clientSocket, (struct sockaddr\*)&serverAddr,  sizeof(serverAddr)) == SOCKET\_ERROR)  {  std::cerr << "Connect failed!" << std::endl;  closesocket(clientSocket);  WSACleanup();  return 1;  } |

* You are using the clientSocket to connect to the server. Here's a breakdown of what's happening:
  + clientSocket: This is the socket that you created earlier using the socket function.
  + (struct sockaddr\*)&serverAddr: This casts the serverAddr structure to the generic struct sockaddr\* type. The serverAddr structure contains the server's IP address and port number.
  + sizeof(serverAddr): This specifies the size of the serverAddr structure.

1. Send Data:

|  |
| --- |
| std::string message = "Hello, Server!";  send(clientSocket, message.c\_str(), message.length(), 0); |

* In C++, the send function used in socket programming requires a pointer to the data to be sent and the size of the data. When you use message.c\_str(), you are providing a pointer to the null-terminated character array (C-string) representing the string, and message.length() gives the length of that string.
* When you want to send data other than characters, such as integers, floating-point numbers, or custom structures, you need to convert the data into a byte array (or a contiguous block of memory) that can be transmitted over the socket. This process is often referred to as **serialization** or **marshalling**.

1. Receive Data:

|  |
| --- |
| int recvSize;  if ((recvSize = recv(clientSocket, buffer,  sizeof(buffer), 0)) > 0) {  buffer[recvSize] = '\0';  std::cout << "Response from server: " << buffer << "\n";  }  else {  std::cerr << "Recv failed!" << std::endl;  } |

* The same socket is used to **receive** data from the server, and the data will indeed be received as a stream of bytes. Therefore, you need an array (or a buffer) to store the incoming data, as it arrives in chunks, which is why the code uses a buffer (like char buffer[BUFFER\_SIZE]) to hold this data.

1. Close Connection:

|  |
| --- |
| closesocket(clientSocket); |

**When We Call** closesocket(clientSocket);**:**

1. **Releases Resources**:

* It releases the resources allocated by the operating system for the socket, such as memory and buffers.
* It ensures that the operating system knows that the socket is no longer in use and that any associated resources can be freed.

1. **Finishes Communication**:

* If you call closesocket() on a connected socket (like clientSocket in your case), the socket is gracefully closed. The TCP connection will be terminated in a controlled way, allowing both ends of the connection (client and server) to cleanly shut down their communication.
* For TCP sockets, a **four-way handshake** occurs when the socket is closed, where the connection is properly closed from both the client and the server. This helps ensure that both sides know the connection is being closed and no further data is expected.

1. **Marks the Socket as Inactive**:

* After calling closesocket(), the socket is no longer usable for communication (i.e., you can't send or receive data using it).
* If you attempt to use the socket again after calling closesocket(), you’ll get an error.

1. **Network Cleanup**:

* For a client socket, calling closesocket() will allow the operating system to clean up any lingering network connections or buffers that may still be allocated to the socket.

**Important Notes:**

* **Graceful Shutdown for TCP**: In the case of a TCP socket (used in your example with SOCK\_STREAM), the connection will be terminated with a **FIN (finish) packet** being sent to the server. The server will also send back a **FIN packet**, confirming the shutdown of the connection. This process ensures that both sides are aware that the connection is being closed, and all pending data is transmitted.
* **Unreliable UDP**: If you were using UDP (SOCK\_DGRAM), calling closesocket() would simply close the socket without the graceful termination of a connection (since UDP is connectionless).
* **Multiple** closesocket() **Calls**: It’s safe to call closesocket() multiple times on the same socket, but only the first call actually performs the close operation.

## Server Code Steps:

1. Create a Socket Object:

|  |
| --- |
| SOCKET serverSocket, clientSocket; serverSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);  if (serverSocket == INVALID\_SOCKET) {  std::cerr << "Socket creation failed!" << std::endl;  WSACleanup();  return 1;  } |

* In socket programming, **two sockets**—serverSocket and clientSocket—are needed because the server and client have different roles in the communication process. Each socket handles a different part of the connection:

**a. Server Socket (**serverSocket**):**

* The **server socket** is used by the server to **listen for incoming connections** and **accept** client connections.
* The server socket is created once when the server starts up, and it is bound to a specific port (e.g., port 8080 in your example).
* It is in **listening** mode, waiting for incoming client connections.
* When a client attempts to connect, the server uses this socket to **accept the connection** and create a new **client socket** to handle communication with that specific client.

**b. Client Socket (**clientSocket**):**

* The **client socket** is created when a connection has been **accepted** by the server and represents the actual communication channel between the client and the server.
* The clientSocket is used for **sending and receiving data** between the server and the client after the connection is established.
* This socket is unique to the connection between the server and the client.
* The client socket is typically created dynamically by the server in response to a client connection (via accept()), and it’s used specifically to interact with that particular client.

**Why Two Sockets?**

* **Separation of Concerns**: The server needs one socket to **listen for incoming connections** and another to **communicate with each connected client**. The server socket is just for listening, while the client socket is used for actual communication.

**Listening vs. Communication**:

* The serverSocket is bound to a port and listens for incoming connections from clients, but it doesn’t communicate directly with the clients.
* When the server receives a connection request, it creates a new **client socket** (clientSocket) that handles communication with that specific client.

**Multiple Clients**:

* If the server wants to handle **multiple clients**, it will continue using the same serverSocket to listen for new connections. Each time a new client connects, the server creates a new clientSocket to handle communication with that client, while the serverSocket keeps listening for additional incoming connections.

1. Bind to Address and Port:

|  |
| --- |
| serverAddr.sin\_family = AF\_INET;  serverAddr.sin\_port = htons(PORT);  serverAddr.sin\_addr.s\_addr = INADDR\_ANY;  // Bind the socket  if (bind(serverSocket, (struct sockaddr\*)&serverAddr,  sizeof(serverAddr)) == SOCKET\_ERROR) {  std::cerr << "Bind failed!" << std::endl;  closesocket(serverSocket);  WSACleanup();  return 1;  } |

**Why Do We Need bind() on the Server Side?**

* **Port Number**: A server needs to listen on a specific port so that clients know where to connect. bind() assigns a port (like 8080) to the socket so that the server knows to listen for incoming connections on that port.
* **IP Address**: You can bind the server to a specific IP address (e.g., 127.0.0.1 for localhost) or all available interfaces (e.g., INADDR\_ANY). By binding the server to an IP, you're telling the system that you want the server to handle incoming connections on that address and port.

**What Happens When You Call bind()?**

* **Binding the Socket**: When you call bind(), the operating system assigns the serverSocket to a specific local IP address and port (defined in the serverAddr structure).
  + For example, serverAddr.sin\_port = htons(8080) binds the socket to port 8080, and serverAddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1") binds it to the local IP address 127.0.0.1.
* **Port and IP Association**: After the call to bind(), the server socket is associated with the specified address (in this case, IP 127.0.0.1 and port 8080).
  + This means that the server can now listen for incoming connections from clients on the specified IP and port.
* **Required Before** listen**()**: The bind() function must be called before listen() on the server side. The listen() function tells the socket to start accepting incoming connections, but the socket must first be bound to an address and port using bind().

1. Listen for Incoming Connections:

|  |
| --- |
| if (listen(serverSocket, 1) == SOCKET\_ERROR) {  std::cerr << "Listen failed!" << std::endl;  closesocket(serverSocket);  WSACleanup();  return 1;  } |

* The listen() function in socket programming is used to **mark a server socket as ready to accept incoming connection requests** from clients. It prepares the socket to listen for incoming connections and specifies how many connections can be waiting in the queue before the operating system starts rejecting new connection attempts.

**Purpose of** listen()**:**

* **Prepares the server to accept incoming connections**: The listen() function is a prerequisite before accepting client connections.
* **Defines the backlog (queue size)**: It defines the maximum number of connections that can be queued while the server is waiting to process them.

int listen(SOCKET s, int backlog);

* + s: The socket file descriptor (e.g., serverSocket), which has already been created and bound to an address and port (using socket() and bind()).
  + backlog: The **maximum number of connections** that can be queued before the operating system starts rejecting new connections. If the backlog is full, new connection attempts will fail until the queue has space.

1. Accept a Client Connection:

|  |
| --- |
| clientSocket = accept(serverSocket, (struct sockaddr\*)&clientAddr,  &clientAddrSize);  if (clientSocket == INVALID\_SOCKET) {  std::cerr << "Accept failed!" << std::endl;  closesocket(serverSocket);  WSACleanup();  return 1;  } |

* The accept() function in socket programming is used by a **server** to **accept an incoming client connection**. It is called after the server socket has been set to listen for connections using the listen() function. When accept() is called, the server accepts a new incoming connection request from a client and returns a new socket (known as the **client socket**) that is specifically used for communication with that client.

SOCKET accept(SOCKET s, struct sockaddr\* addr, int\* addrlen);

* s: The server socket (e.g., serverSocket) that was previously bound and is listening for incoming connections.
* addr: A pointer to a sockaddr structure that will be populated with the client's address information (such as the IP address and port of the client).
* addrlen: A pointer to an integer that initially contains the size of the sockaddr structure. Upon return, it will contain the actual size of the client's address.

**Purpose of accept():**

* **Accepts a client connection**: When a client requests to connect to the server, accept() establishes the communication link between the server and that specific client.
* **Creates a new socket for communication**: The server uses the original socket (serverSocket) to listen for incoming connections, but once a connection is accepted, accept() creates a **new socket** (clientSocket) dedicated to communication with that client.

1. Receive and Process Data:
2. Send Response:

|  |
| --- |
| int recvSize;  while ((recvSize = recv(clientSocket, buffer,  sizeof(buffer), 0)) > 0) {  buffer[recvSize] = '\0';  std::cout << "Received from client: " << buffer << std::endl;  // Send a response to the client  std::string response = "Hello Client!, We received: ";  response += buffer;  send(clientSocket, response.c\_str(), response.length(), 0);  } |

* Use the clientSocket to send the response.

1. Close Connection:

|  |
| --- |
| closesocket(clientSocket);  closesocket(serverSocket); |

* The closesocket() function in Windows socket programming is used to **close a socket**. When you call closesocket() on a socket, it marks the socket for closure, releasing all associated resources, such as memory, network buffers, and file descriptors.

**Purpose of** closesocket()**:**

* **Release resources**: It releases the system resources associated with the socket.
* **Terminate the connection**: It properly terminates the communication on the socket, whether it's a client or server socket.
* **Prevent further communication**: After calling closesocket(), no more data can be sent or received on that socket.

**What Happens When You Call** closesocket()**?**

* **For Client Socket (**clientSocket**)**:
  + If you call closesocket(clientSocket) after finishing communication with the client (e.g., after sending and receiving data), it terminates the communication between the client and server.
  + It closes the socket that was dedicated to the specific client, freeing up any resources used for that connection.
* **For Server Socket (**serverSocket**)**:
  + If you call closesocket(serverSocket) after finishing the server’s task, it closes the server socket and releases the resources associated with it.
  + In many cases, you'll only close the server socket after stopping the server from accepting new connections. If you want the server to continue accepting connections, you would generally only close serverSocket when the server is shutting down completely.