Socket Communication

# Basic Mechanism of Client-Server Setup

The basic mechanisms of client-server setup are:

* A client app send a request to a server app.
* The server app returns a reply.
* Some of the basic data communications between client and server are:

File transfer - sends name and gets a file.

Web page - sends url and gets a page.

Echo - sends a message and gets it back.

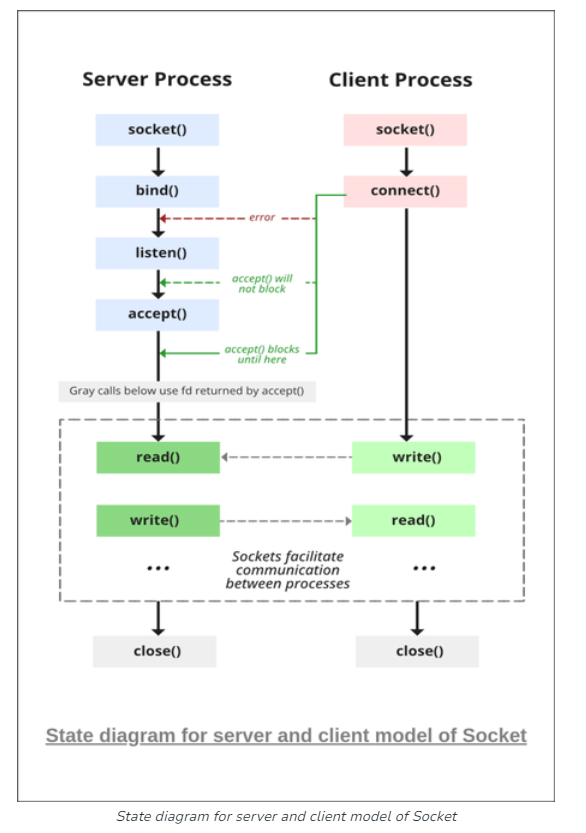
# Client Socket Setup

* create a **socket**.
* **bind\*** - this is probably be unnecessary because you're the client, not the server.
* **connect** to a server.
* **send/recv** - repeat until we have or receive data
* **shutdown** to end read/write.
* **close** to releases data.

# Server Socket Setup

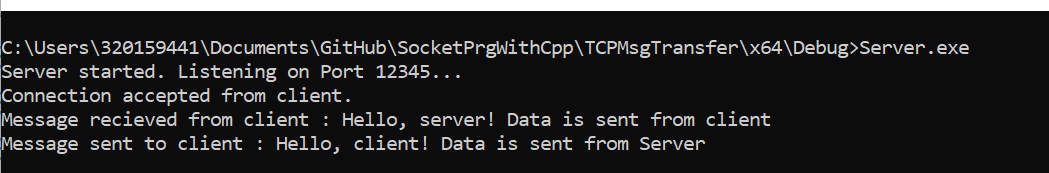
* create a **socket** - Get the file descriptor!
* **bind** to an address -What port am I on?
* **listen** on a port, and wait for a connection to be established.
* **accept** the connection from a client.
* **send/recv** - the same way we read and write for a file.
* **shutdown** to end read/write.
* **close** to releases data.

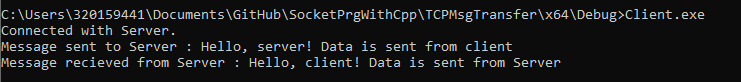
# State Diagram of Client Server Socket Setup



# Output

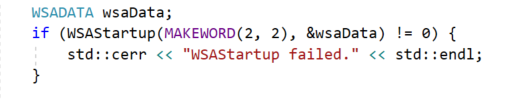
Execute Server.exe and then Client.exe





# **InDetail Explanation of Socket Programming**

# Before Socket Creation



The purpose of this code snippet is to initialize the Winsock library with version 2.2 and to handle any potential initialization errors. If the initialization fails, an error message is printed to the console.

1. **WSADATA** wsaData**;**

WSADATA is a structure provided by the Windows Socket API (Winsock). It contains information about the Windows Sockets implementation.

By declaring an instance of this structure, the program is preparing to initialize the Winsock library.

1. WSAStartup**(MAKEWORD**(2, 2), &wsaData**):**

WSAStartup is a function from the Winsock library that initializes the Winsock library.

The MAKEWORD(2, 2) macro is used to request version 2.2 of Winsock. This version is widely supported and is commonly used in many applications.

The second argument is a pointer to the WSADATA structure declared earlier. The function will fill this structure with information about the Winsock implementation after successful initialization.

1. if (WSAStartup(**MAKEWORD**(2, 2), &wsaData) != 0) { ... }:

This line checks the return value of WSAStartup.

If WSAStartup fails (returns a value other than 0), the code inside the curly braces will be executed.

In the provided code, an error message is printed to the standard error stream (std::cerr) if WSAStartup fails.

# Socket Creation

int socketCreation = **socket**(domain,type, protocol)

int socketCreation = **socket**(**AF\_INET**, **SOCK\_STREAM**, 0)

* include Library : #include <winsock2.h>

## Socket Domain

In socket programming, the term "socket domain" refers to the communication domain or address family that determines the type of addresses used by the socket. Different socket domains are suitable for different types of communication. Here are some common socket domains:

**1. AF\_INET (IPv4):**

* + **Description**: Address Family IPv4 is the most widely used socket domain. It supports communication over the Internet using IPv4 addresses (e.g., "192.168.1.1").
  + **Use Cases**: General-purpose networking applications, web servers, and many internet-based services.

**2. AF\_INET6 (IPv6): [Preferred]**

* ***Description***: Address Family IPv6 supports communication over the Internet using IPv6 addresses, which provide a larger address space than IPv4.
* ***Use Cases***: Modern networking applications that need to support IPv6, as the Internet transitions from IPv4 to IPv6.

**3. AF\_UNIX (Unix Domain):**

* ***Description***: Address Family Unix Domain provides communication between processes on the same machine using file system paths as addresses. It is also known as Local Domain or Inter-Process Communication (IPC).
* ***Use Cases***: Inter-process communication on the same machine, communication between a server and its local clients.

**4. AF\_ISO (ISO protocols):**

* ***Description***: Address Family ISO supports communication using ISO (International Organization for Standardization) protocols, such as the OSI (Open Systems Interconnection) model.
* ***Use Cases***: Less common and mainly used in specialized applications requiring ISO protocol support.

**5. AF\_NETLINK (Linux Netlink):**

* ***Description***: Address Family Netlink is a communication protocol between the Linux kernel and user-space processes, providing a mechanism for information exchange and control.
* ***Use Cases***: Kernel-user space communication in Linux, such as networking configuration.

**6. AF\_PACKET (Packet Interface):**

* ***Description***: Address Family Packet provides direct access to network packets at the link layer, allowing low-level packet manipulation.
* ***Use Cases***: Network sniffing, packet capturing, and low-level networking tasks.

**7. AF\_ROUTE (Routing Sockets):**

* ***Description***: Address Family Route is used for communication with the routing table in the kernel.
* ***Use Cases***: Networking applications that need to interact with the system's routing information.

**8. AF\_X25 (X.25 WAN):**

* ***Description***: Address Family X.25 provides communication over X.25 wide-area networks.
* ***Use Cases***: Legacy systems and applications that still use X.25 for wide-area networking.

## Socket Communication Types

1. **SOCK\_STREAM (TCP): [Preferred]**

* ***Use Case:*** Reliable, connection-oriented communication.
* ***Description***: TCP (Transmission Control Protocol) provides a reliable, stream-oriented connection between two hosts. It ensures data integrity and order of delivery. It is suitable for applications that require a reliable and error-checked stream of data, such as file transfer, web browsing, and email.

1. **SOCK\_DGRAM (UDP):**

* ***Use Case***: Unreliable, connectionless communication.
* ***Description***: UDP (User Datagram Protocol) provides a connectionless, low-overhead communication mechanism. It is faster than TCP but does not guarantee data integrity or order. UDP is suitable for real-time applications, such as streaming media, online gaming, and DNS.

1. **SOCK\_RAW:**

* **Use Case**: Low-level packet manipulation.
* **Description**: SOCK\_RAW allows applications to access raw network packets at the transport layer. This type of socket is used for low-level network protocol implementation and packet crafting. It is more commonly used in networking infrastructure or security applications.

1. **SOCK\_SEQPACKET**:

* **Use Case**: Reliable, sequenced packet communication.
* **Description**: SOCK\_SEQPACKET is a reliable, connection-oriented socket type that provides a sequenced, reliable, and unduplicated flow of data. It is less commonly used than SOCK\_STREAM and is suitable for specific applications that require strict ordering of data.

1. **SOCK\_RDM (Reliably Delivered Message):**

* ***Use Case***: Reliable message-oriented communication.
* ***Description***: SOCK\_RDM is a reliable message-oriented socket type. It ensures that messages are delivered reliably but does not guarantee their order. It is less commonly used than SOCK\_STREAM and SOCK\_SEQPACKET.

1. **SOCK\_NONBLOCK:**

* ***Use Case***: Non-blocking communication.
* ***Description***: SOCK\_NONBLOCK is not a separate socket type but rather a flag that can be combined with other socket types. It enables non-blocking mode, allowing applications to perform other tasks while waiting for data or connections.

## **Socket Protocol**

In socket programming, the protocol parameter is used to specify the communication protocol to be used with the socket. The protocol parameter is commonly used in the socket() function call. Different values of the protocol parameter indicate the specific communication protocol or protocol family that the socket will use. Here are some common values for the protocol parameter:

1. **0 (Automatically Select):**

* If the protocol parameter is set to 0, the system will automatically choose the appropriate protocol based on the specified address family (AF\_INET, AF\_INET6, etc.) and the socket type (SOCK\_STREAM, SOCK\_DGRAM, etc.). This is the most common choice for general-purpose sockets.

1. **IPPROTO\_TCP (Transmission Control Protocol - TCP):**

* If you specifically want to use the TCP protocol, you can set the protocol parameter to IPPROTO\_TCP. This is often used with SOCK\_STREAM sockets for reliable, connection-oriented communication.

1. **IPPROTO\_UDP (User Datagram Protocol - UDP):**

* If you specifically want to use the UDP protocol, you can set the protocol parameter to IPPROTO\_UDP. This is used with SOCK\_DGRAM sockets for connectionless, fast communication.

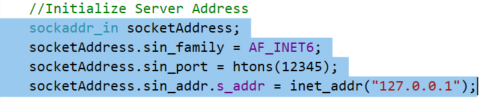
1. **IPPROTO\_ICMP (Internet Control Message Protocol - ICMP):**

* If you specifically want to use ICMP, set the protocol parameter to IPPROTO\_ICMP. ICMP is commonly used for network diagnostics and error reporting.

1. **IPPROTO\_RAW (Raw IP Packets):**

* If you want to work with raw IP packets, set the protocol parameter to IPPROTO\_RAW. This allows you to send and receive raw network packets.

## **Socket Address Initialization**



They are used to set up the server's address information, specifically when working with IPv4.

* sockaddr\_in serverAddress;

The **sockaddr\_in** structure is used to represent an IPv4 socket address. It contains fields that store information about the address, such as the address family, port number, and IP address.

* serverAddress.sin\_family = AF\_INET;

This line sets the **sin\_family** field of the **serverAddress** structure. The **sin\_family** field specifies the address family, and in this case, it's set to **AF\_INET**, indicating the use of IPv4.

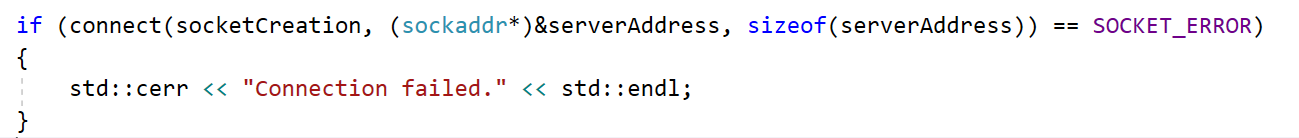
* serverAddress.sin\_port = htons(12345);

This line sets the **sin\_port** field of the **serverAddress** structure. The **sin\_port** field represents the port number on which the server will listen for incoming connections. The **htons()** function is used to convert the port number from host byte order to network byte order. Port numbers are typically specified in network byte order in networking functions.

* serverAddress.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

This line sets the **sin\_addr.s\_addr** field of the **serverAddress** structure. The **sin\_addr** structure within **sockaddr\_in** contains the IP address. Here, **inet\_addr("127.0.0.1")** converts the string representation of an IPv4 address ("127.0.0.1") to its binary form. In this example, "127.0.0.1" is the loopback address, representing the local machine. You would change this to the actual IP address of your server.

# Socket Connection [ For Client Side ]



The code attempts to establish a connection to a server using a previously created socket. If the connection attempt fails, an error message is printed to the console indicating the failure.

1. if (connect(socketCreation, (sockaddr\*)&serverAddress, sizeof(serverAddress)) == SOCKET\_ERROR) { ... }:

connect function: This is a standard function from the Winsock library used to initiate a connection on a socket. It is used to establish a connection to a specified address (in this case, the server's address).

socketCreation: This appears to be a previously created socket descriptor (or socket handle) that was obtained using the socket function. This descriptor represents a communication endpoint that can be used for network communication.

(sockaddr\*)&serverAddress:

serverAddress is a sockaddr structure (or its derivatives like sockaddr\_in for IPv4) that contains the address information of the server to which the connection is to be established.

The (sockaddr\*)&serverAddress is a type-casting operation. Since connect expects a pointer to a sockaddr structure as its second argument, the address of serverAddress is passed by reference.

sizeof(serverAddress): This provides the size of the serverAddress structure in bytes. It indicates to the connect function the size of the address structure it is working with.

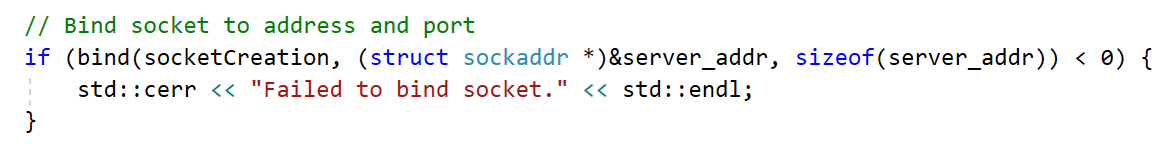
== SOCKET\_ERROR: The connect function returns SOCKET\_ERROR (a constant defined in Winsock) if the connection attempt fails.

The block inside the curly braces { ... }:

This block contains the code that gets executed if the connection attempt fails.

Inside this block, an error message "Connection failed." is printed to the standard error stream (std::cerr).

# Socket Binding [ For Server Side ]



The code attempts to bind the previously created socket to a specific network address and port. If the binding operation fails, an error message is printed to the console indicating the failure.

* if (bind(socketCreation, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) { ... }:

bind function: This is a standard function from the Winsock library used to associate a socket with a specific network address and port number.

socketCreation: This is the socket descriptor (or socket handle) that represents a communication endpoint, which was previously created using the socket function.

(struct sockaddr \*)&server\_addr:

server\_addr is a structure (typically sockaddr\_in for IPv4) that contains the address information (IP address and port number) to which the socket should be bound.

(struct sockaddr \*)&server\_addr is a type-casting operation. Since the bind function expects a pointer to a sockaddr structure (or its derivatives) as its second argument, the address of server\_addr is passed by reference.

sizeof(server\_addr): This provides the size of the server\_addr structure in bytes. It indicates to the bind function the size of the address structure it is working with.

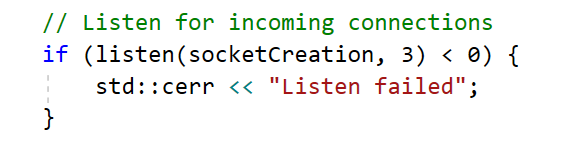
< 0: The bind function returns -1 (or a value less than 0) if the binding operation fails.

The block inside the curly braces { ... }:

This block contains the code that gets executed if the binding operation fails.

Inside this block, an error message "Failed to bind socket." is printed to the standard error stream (std::cerr).

# Socket Listening [ For Server Side ]



The purpose of this code is to set a socket to listen for incoming connections. Let's break it down step by step.

* if (listen(socketCreation, 3) < 0) { ... }:

listen function: This is a standard function from the Winsock library used to set the socket into a listening mode, allowing it to accept incoming connection requests.

socketCreation: This is the socket descriptor (or socket handle) that represents a communication endpoint, which was previously created using the socket function and possibly bound to an address using the bind function.

3: The second argument to the listen function specifies the maximum length to which the queue of pending connections may grow. In other words, it indicates how many incoming connection requests can be queued up before the application starts accepting them. Here, it's set to 3, meaning the socket can handle up to 3 pending connections.

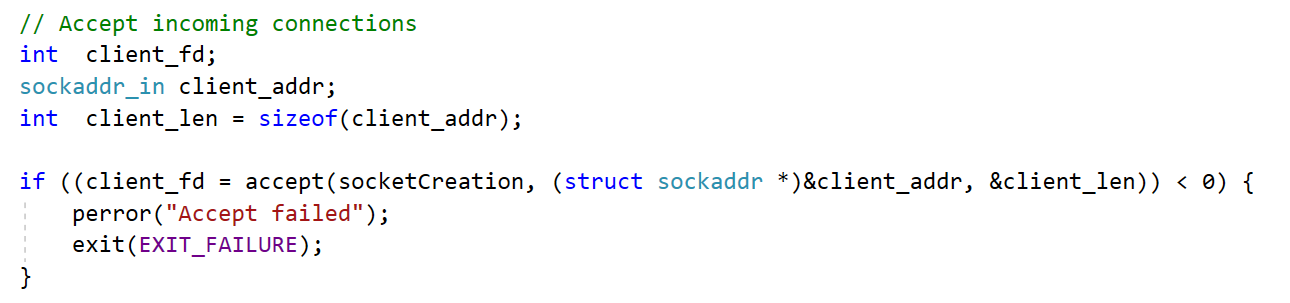
< 0: The listen function returns -1 (or a value less than 0) if the listening operation fails.

The block inside the curly braces { ... }:

This block contains the code that gets executed if the listening operation fails.

Inside this block, an error message "Listen failed" is printed to the standard error stream (std::cerr).

# Socket Accept Connection [ For Server Side ]



The code attempts to accept an incoming connection request on the listening socket (**socketCreation**). If the acceptance fails, an error message is printed, and the program exits with a failure status. If successful, **client\_fd** will contain the socket descriptor for the new connection, and **client\_addr** will contain the address information of the connecting client.

* int client\_fd;:

This declares an integer variable client\_fd which will store the socket descriptor (or socket handle) for the accepted connection. This descriptor represents a communication endpoint for the newly established connection.

* sockaddr\_in client\_addr;:

This declares a sockaddr\_in structure named client\_addr that will hold the address information of the client that is connecting. The sockaddr\_in structure is commonly used for IPv4 addresses in Winsock programming.

* int client\_len = sizeof(client\_addr);:

This initializes an integer variable client\_len with the size of the client\_addr structure. It's used to specify the size of the address structure when accepting a connection.

* if ((client\_fd = accept(socketCreation, (struct sockaddr \*)&client\_addr, &client\_len)) < 0) { ... }:

accept function: This is a standard function from the Winsock library used to accept a connection request on a listening socket.

socketCreation: This is the socket descriptor (or socket handle) that represents the listening socket on which the connection request is received.

(struct sockaddr \*)&client\_addr:

This specifies a pointer to the client\_addr structure where the address information of the connecting client will be stored.

The type-casting is done because the accept function expects a pointer to a sockaddr structure (or its derivatives like sockaddr\_in).

&client\_len:

This specifies the address of the client\_len variable, which contains the size of the client\_addr structure. After the call to accept, this variable will be updated with the actual size of the client's address.

< 0: The accept function returns -1 (or a value less than 0) if the connection acceptance operation fails.

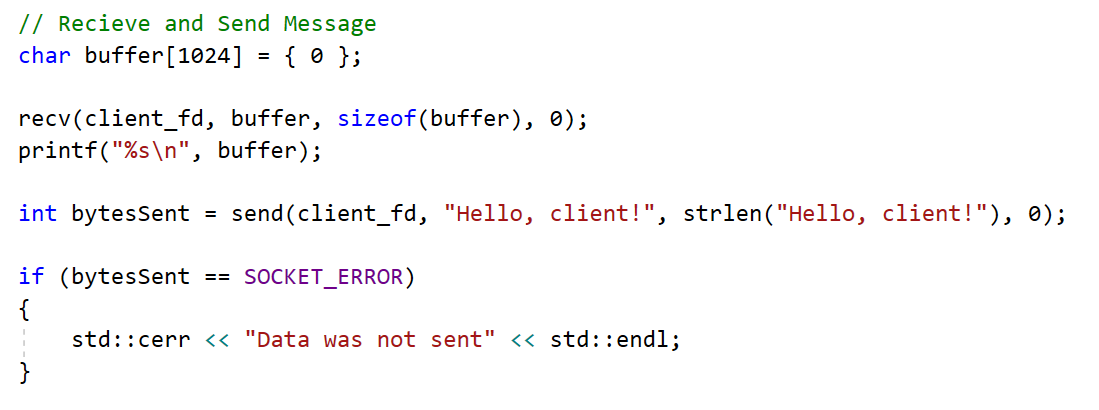
The block inside the curly braces { ... }:

This block contains the code that gets executed if the connection acceptance operation fails.

The perror("Accept failed"); line prints an error message to the standard error stream (stderr) using the perror function, which provides a more detailed error message based on the last encountered error related to sockets.

The exit(EXIT\_FAILURE); line terminates the program with a failure status.

# Receive and Send Message [ For Server Side ]



The code receives data from a connected client socket into a buffer, prints the received data to the console, and then sends a "Hello, client!" message back to the client. If the sending operation fails, an error message is printed.

* char buffer[1024] = { 0 }:

This declares an array of characters named buffer with a size of 1024 bytes and initializes it to zeros. This buffer will be used to store the received data.

* recv(client\_fd, buffer, sizeof(buffer), 0);:

recv function: This is a standard function from the Winsock library used to receive data from a connected socket.

client\_fd: This is the socket descriptor for the connected client socket.

buffer: This is the buffer where the received data will be stored.

sizeof(buffer): This specifies the size of the buffer, indicating the maximum number of bytes that can be received.

0: This is a flag parameter; setting it to 0 means no special flags are used during the operation.

printf("%s\n", buffer);:

This line prints the received data (stored in the buffer) to the console using the printf function.

* int bytesSent = send(client\_fd, "Hello, client!", strlen("Hello, client!"), 0);:

send function: This is a standard function from the Winsock library used to send data on a connected socket.

client\_fd: This is the socket descriptor for the connected client socket.

"Hello, client!": This is the data (a string) to be sent to the client.

strlen("Hello, client!"): This computes the length of the string "Hello, client!" to determine the number of bytes to be sent.

0: This is a flag parameter; setting it to 0 means no special flags are used during the operation.

* if (bytesSent == SOCKET\_ERROR) { ... }:

This checks if the send operation was successful.

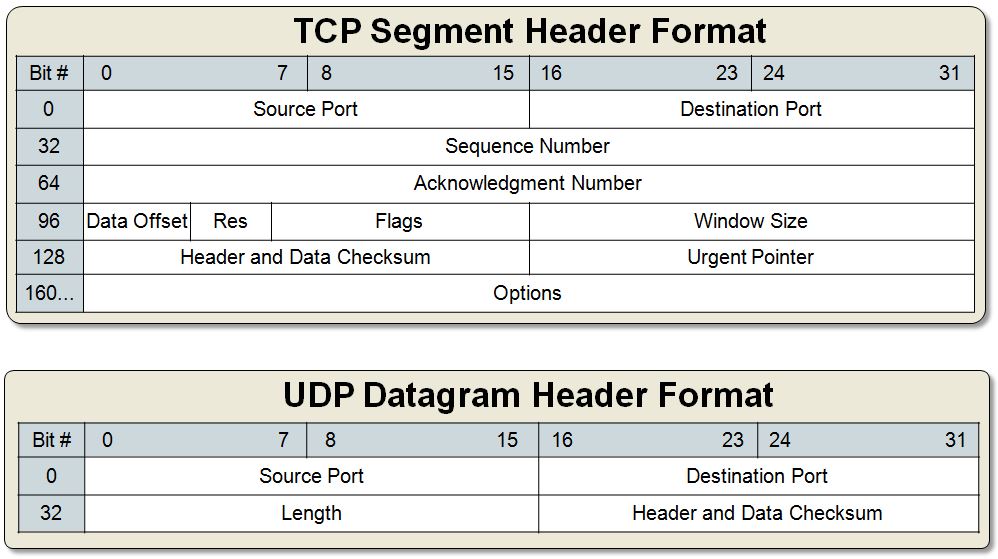
SOCKET\_ERROR: This is a constant defined in Winsock, typically representing an error condition for socket operations.

If the send operation fails (returns SOCKET\_ERROR), the code inside the curly braces { ... } is executed.

Inside this block, an error message "Data was not sent" is printed to the standard error stream (std::cerr).

# TCP v/s UDP

There are several different types of socket that determine the structure of the transport layer. The most common types are **stream** sockets and **datagram** sockets.



|  |  |
| --- | --- |
| TCP (Streams) | UDP (Datagrams) |
| Connections | Connectionless sockets We don't have to maintain an open connection as we do with stream sockets. We just build a packet, put an IP header on it with destination information, and send it out. No connection needed: datagram sockets also use IP for routing, but they don't use TCP  \*note: can be connect()'d if we really want. |
| SOCK\_STREAM | SOCK\_DGRAM |
| If we output two items into the socket in the order "A, B", they will arrive in the order "A, B" at the opposite end. They will also be error-free. | If we send a datagram, it may arrive. But it may arrive out of order. If it arrives, however, the data within the packet will be error-free. |
|  | Why would we use an unreliable protocol? Speed! We just ignore the dropped packets. |
| Arbitrary length content | Limited message size |
| Flow control matches sender to receiver | Can send regardless of receiver state |
| Congestion control matches sender to network | Can send regardless of network state |
| http, telnet | tftp (trivial file transfer protocol), dhcpcd (a DHCP client), multiplayer games, streaming audio, video conferencing  \*note: They use complementary protocol on top of UDP to get more reliability |

* **Stream Sockets**  
  Stream sockets provide **reliable two-way** communication similar to when we call someone on the phone. One side initiates the connection to the other, and after the connection is established, either side can communicate to the other.  
  In addition, there is immediate confirmation that what we said actually reached its destination.  
  Stream sockets use a **Transmission Control Protocol (TCP)**, which exists on the transport layer of the Open Systems Interconnection (OSI) model. The data is usually transmitted in packets. TCP is designed so that the packets of data will arrive without errors and in sequence.  
  Webservers, mail servers, and their respective client applications all use TCP and stream socket to communicate.
* **Datagram Sockets**  
  Communicating with a datagram socket is more like mailing a letter than making a phone call. The connection is **one-way** only and **unreliable**.  
  If we mail several letters, we can't be sure that they arrive in the same order, or even that they reached their destination at all. Datagram sockets use **User Datagram Protocol (UDP)**. Actually, it's not a real connection, just a basic method for sending data from one point to another.  
  Datagram sockets and UDP are commonly used in networked games and streaming media.  
  Though in this section, we mainly put focus on applications that maintain connections to their clients, using connection-oriented TCP, there are cases where the overhead of establishing and maintaining a socket connection is unnecessary.  
  For example, just to get the data, a process of creating a socket, making a connection, reading a single response, and closing the connection, is just too much. In this case, we use UDP.  
  Services provided by UDP are typically used where a client needs to make a short query of a server and expects a single short response. To access a service from UDP, we need to use the UDP specific system calls, **sendto()** and **recvfrom()** instead of **read()** and **write()** on the socket.

UDP is used by app that doesn't want reliability or bytestreams.

Voice-over-ip (unreliable) such as conference call. (visit [VoIP](http://www.bogotobogo.com/VideoStreaming/VoIP.php))

DNS, RPC (message-oriented)

DHCP (bootstrapping)