Networking with c++

Steps to create tcp server :

On windows :

1. Initialize winsock library

2. create the socket

3. get ip and port

4. bind the ip/port with the socket

5. listen on the socket

6. accept

7. recv and send

8. close the socket

9. cleanup the winsock

On linux :

1. Create the socket

2. Get IP and Port

3. Bind the IP/Port with the socket

4. Listen on the socket

5. Accept a connection

6. Receive and send data

7. Close the socket

8. No explicit cleanup

Steps to create a simple tcp client :

Steps to create a simple TCP client:
1.Create the socket
Use socket() to create a socket.

2.Get server IP and port
Specify the server's IP address and port that you want to connect to.

3.Connect to the server
Use connect() to establish a connection to the server using the socket and the server's IP and port.

4.Send data to the server
Use send() to transmit data to the server.

5.Receive data from the server
Use recv() to receive a response from the server.

6.Close the socket
Use close() to close the connection and free up the resources.

7.No explicit cleanup required
Socket resources are released after closing the socket.

Run the process in the background without terminal on linux :
Command :

nohup ./your_server_program &> server.log &

You can check the logs using the command :
tail -f server.log

Running the Server as a System Service :

1. Create a service file: Create a new file at /etc/systemd/system/myserver.service:
sudo nano /etc/systemd/system/myserver.service
2. Add the following content:

```
[Unit]
Description=My TCP Server
After=network.target

[Service]
ExecStart=/path/to/your_server_program
WorkingDirectory=/path/to/your/server
```

```
Restart=always

[Install]
WantedBy=multi-user.target
```

3. Enable and start the service:
sudo systemctl enable myserver.service
sudo systemctl start myserver.service

4. Check status:
sudo systemctl status myserver.service

Command to check % cpu and memory usage of each process on linux :
Command : top

Command to check which process is running on any particular port :
Command : sudo lsof -i :<port_number>

Command to force kill a process using pid :
Command : kill -9 <pid>

code to create a simple tcp server :

on windows :
Server code (windows) : thread per client model

```
#include <iostream>
#include <WinSock2.h>
#include <WS2tcpip.h>
#include <tchar.h>
#include <thread>
#include <vector>
```

```
using namespace std;

#pragma comment(lib,"ws2_32.lib")

/*
	//initialize winsock library

	//create the socket

	//get ip and port

	//bind the ip/port with the socket

	//listen on the socket

	//accept

	//recv and send

	//close the socket

	// cleanup the winsock


*/

bool Initialize() {
	WSADATA data;
	return WSAStartup(MAKEWORD(2,2), &data) == 0;
}


void InteractWithClient(SOCKET clientSocket, vector<SOCKET>& clients) {
	//send/recv client

	cout << "client connected" << endl;
	char buffer[4096];

	while (1) {

		int bytesrecvd = recv(clientSocket, buffer, sizeof(buffer),
0);

		if (bytesrecvd <= 0) {
			cout << "client disconnected" << endl;
```

```
                  break;
            }

            string message(buffer, bytesrecvd);
            cout << "message from client : " << message << endl;

            for (SOCKET client : clients) {
                  if (client != clientSocket) {
                        send(client, message.c_str(), message.length(),
0);
                  }
            }

      }

      auto it = find(clients.begin(), clients.end(), clientSocket);
      if (it != clients.end()) {
            clients.erase(it);
      }

      closesocket(clientSocket);
}

int main() {
      if (!Initialize()) {
            cout << "winsock initialization failed " << endl;
            return 1;
      }
      cout << "server program" << endl;

      SOCKET listenSocket = socket(AF_INET, SOCK_STREAM, 0);

      if (listenSocket == INVALID_SOCKET) {
            cout << "socket creation failed" << endl;
            return 1;
      }

      //create address structure
      int port = 12335;
      sockaddr_in serveraddr;
      serveraddr.sin_family = AF_INET;
      serveraddr.sin_port = htons(port);

      //converet the ipaddress (0.0.0.0) put it inside the sin_family in
biary format
      if (InetPton(AF_INET, _T("0.0.0.0"), &serveraddr.sin_addr) != 1) {
```

```
            cout << "setting address structure failed" << endl;
            closesocket(listenSocket);
            WSACleanup();
            return 1;
      }

      //bind
      if (bind(listenSocket, reinterpret_cast<sockaddr*>(&serveraddr),
sizeof(serveraddr)) == SOCKET_ERROR) {
            cout << "bind failed" << endl;
            closesocket(listenSocket);
            WSACleanup();
            return 1;
      }

      //listen
      if (listen(listenSocket, SOMAXCONN) == SOCKET_ERROR) {
            cout << "listen failed" << endl;
            closesocket(listenSocket);
            WSACleanup();
            return 1;
      }

      cout << "server has started listening on port : " << port << endl;
      vector<SOCKET> clients;

      while (1) {
            //accept
            SOCKET clientSocket = accept(listenSocket, nullptr,
nullptr);
            if (clientSocket == INVALID_SOCKET) {
                  cout << "invalid client socket " << endl;
            }


            clients.push_back(clientSocket);
            thread t1(InteractWithClient, clientSocket,
std::ref(clients));
            t1.detach();

      }

      closesocket(listenSocket);
```

```
      WSACleanup();
      return 0;
}
```

TCP Client Code (Windows) :

```
#include <stdio.h>          // Standard input and output functions
#include <stdlib.h>         // Standard library functions
#include <string.h>         // String manipulation functions
#include <winsock2.h>       // Windows Sockets API

#pragma comment(lib, "ws2_32.lib") // Link with ws2_32.lib

#define SERVER_IP "127.0.0.1" // Server IP address
#define PORT 8080              // Port number to connect to

int main() {
    WSADATA wsaData;       // Structure for WinSock initialization
    SOCKET sock;           // Socket descriptor
    struct sockaddr_in server_addr; // Server address structure
    char *message = "Hello, Server!"; // Message to send
    char buffer[1024] = {0}; // Buffer to store response

    // Initialize Winsock
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        printf("Failed to initialize Winsock. Error Code: %d\n",
WSAGetLastError());
        return EXIT_FAILURE; // Exit if initialization fails
    }

    // Create a socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == INVALID_SOCKET) {
        printf("Could not create socket. Error Code: %d\n",
WSAGetLastError());
        WSACleanup(); // Cleanup Winsock
        return EXIT_FAILURE; // Exit if socket creation fails
    }

    // Setup the server address structure
    server_addr.sin_family = AF_INET; // IPv4
```

```
    server_addr.sin_port = htons(PORT); // Convert port to network byte
order

    // Convert IP address from text to binary form
    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
        printf("Invalid address or Address not supported\n");
        closesocket(sock); // Close socket
        WSACleanup(); // Cleanup Winsock
        return EXIT_FAILURE; // Exit if address conversion fails
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        printf("Connection failed. Error Code: %d\n",
WSAGetLastError());
        closesocket(sock); // Close socket
        WSACleanup(); // Cleanup Winsock
        return EXIT_FAILURE; // Exit if connection fails
    }

    // Send a message to the server
    send(sock, message, strlen(message), 0);

    // Receive a response from the server
    int bytes_received = recv(sock, buffer, sizeof(buffer) - 1, 0);
    if (bytes_received > 0) {
        buffer[bytes_received] = '\0'; // Null-terminate the received
data
        printf("Message from server: %s\n", buffer);
    } else {
        printf("Receive failed. Error Code: %d\n", WSAGetLastError());
    }

    // Cleanup
    closesocket(sock); // Close socket
    WSACleanup(); // Cleanup Winsock

    return EXIT_SUCCESS; // Exit the program
}
```

tcp server code (windows) :

```
#include <stdio.h>          // Standard input and output functions
```

```c
#include <stdlib.h>         // Standard library functions
#include <string.h>         // String manipulation functions
#include <winsock2.h>       // Windows Sockets API

#pragma comment(lib, "ws2_32.lib") // Link with ws2_32.lib

#define PORT 8080           // Port number to listen on

int main() {
    WSADATA wsaData;        // Structure for WinSock initialization
    SOCKET server_socket, client_socket; // Socket descriptors
    struct sockaddr_in server_addr, client_addr; // Server and client
address structures
    int addr_len = sizeof(client_addr); // Length of client address
    char buffer[1024] = {0}; // Buffer to store incoming messages

    // Initialize Winsock
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        printf("Failed to initialize Winsock. Error Code: %d\n",
WSAGetLastError());
        return EXIT_FAILURE; // Exit if initialization fails
    }

    // Create a socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == INVALID_SOCKET) {
        printf("Could not create socket. Error Code: %d\n",
WSAGetLastError());
        WSACleanup(); // Cleanup Winsock
        return EXIT_FAILURE; // Exit if socket creation fails
    }

    // Setup the server address structure
    server_addr.sin_family = AF_INET; // IPv4
    server_addr.sin_addr.s_addr = INADDR_ANY; // Accept connections from
any address
    server_addr.sin_port = htons(PORT); // Convert port to network byte
order

    // Bind the socket
    if (bind(server_socket, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        printf("Bind failed. Error Code: %d\n", WSAGetLastError());
        closesocket(server_socket); // Close socket
        WSACleanup(); // Cleanup Winsock
        return EXIT_FAILURE; // Exit if binding fails
```

```
    }

    // Listen for incoming connections
    if (listen(server_socket, 3) < 0) {
        printf("Listen failed. Error Code: %d\n", WSAGetLastError());
        closesocket(server_socket); // Close socket
        WSACleanup(); // Cleanup Winsock
        return EXIT_FAILURE; // Exit if listen fails
    }

    printf("Waiting for connections...\n");

    // Accept a connection from a client
    client_socket = accept(server_socket, (struct sockaddr
*)&client_addr, &addr_len);
    if (client_socket == INVALID_SOCKET) {
        printf("Accept failed. Error Code: %d\n", WSAGetLastError());
        closesocket(server_socket); // Close server socket
        WSACleanup(); // Cleanup Winsock
        return EXIT_FAILURE; // Exit if accept fails
    }

    printf("Connection accepted.\n");

    // Receive a message from the client
    int bytes_received = recv(client_socket, buffer, sizeof(buffer) - 1,
0);
    if (bytes_received > 0) {
        buffer[bytes_received] = '\0'; // Null-terminate the received
data
        printf("Message from client: %s\n", buffer);
    } else {
        printf("Receive failed. Error Code: %d\n", WSAGetLastError());
    }

    // Send a response to the client
    const char *response = "Hello from server!";
    send(client_socket, response, strlen(response), 0);

    // Cleanup
    closesocket(client_socket); // Close client socket
    closesocket(server_socket); // Close server socket
    WSACleanup(); // Cleanup Winsock

    return EXIT_SUCCESS; // Exit the program
}
```

on linux(simple tcp server) :

You can also use int backlog = SOMAXCONN; // Maximum number of pending connections in listen() api

Need to use this header file :
#include <sys/socket.h> // Socket functions

```
#include <stdio.h>        // For printf and perror
#include <stdlib.h>       // For exit()
#include <string.h>       // For memset
#include <unistd.h>       // For close()
#include <arpa/inet.h>    // For socket functions and structures
(socket, bind, listen, accept, etc.)

#define PORT 8080         // Port number for the server
#define BACKLOG 5         // How many connections can be pending for
accept()

int main() {
    int server_fd, new_socket;      // File descriptors for the server
and new client connection
    struct sockaddr_in address;     // Structure to hold server's
address information
    int addrlen = sizeof(address);  // Size of the address structure
    char buffer[1024] = {0};        // Buffer to store received data
    const char *hello = "Hello from server";  // Message to send to the
client

    // 1. Create a TCP socket using the IPv4 protocol (AF_INET) and
stream type (SOCK_STREAM)
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. Configure the server address
    // Set address family to IPv4 (AF_INET)
    address.sin_family = AF_INET;

    // Bind the socket to all available network interfaces (INADDR_ANY)
and specify the port number
    address.sin_addr.s_addr = INADDR_ANY;
```

```c
    address.sin_port = htons(PORT); // Convert port to network byte order (big-endian)

    // 3. Bind the socket to the IP address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 4. Listen for incoming connections
    // The second argument defines the backlog (number of pending connections allowed)
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    // 5. Accept a connection from a client
    // This call blocks until a client connects
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0) {
        perror("accept failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Connection accepted.\n");

    // 6. Read data sent by the client (this example reads up to 1024 bytes)
    int valread = read(new_socket, buffer, 1024);
    printf("Received: %s\n", buffer);

    // 7. Send a message back to the client
    send(new_socket, hello, strlen(hello), 0);
    printf("Hello message sent to client.\n");

    // 8. Close the client socket when done
    close(new_socket);

    // 9. Close the server socket as well
```

```
    close(server_fd);

    return 0;
}
```

Simple tcp client(linux) :

```
#include <iostream>
#include <cstring>      // For memset, memcpy
#include <sys/types.h>  // For data types like `sockaddr_in`
#include <sys/socket.h> // For socket functions
#include <arpa/inet.h>  // For inet_pton
#include <unistd.h>     // For close()

#define SERVER_PORT 8080
#define SERVER_IP "127.0.0.1"
#define BUFFER_SIZE 100

int main() {
    // Create a socket
    int client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket < 0) {
        std::cerr << "Failed to create socket" << std::endl;
        return -1;
    }

    // Define the server address
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address)); // Clear the structure
    server_address.sin_family = AF_INET;                // IPv4
    server_address.sin_port = htons(SERVER_PORT);       // Set port number to
8080

    // Convert IP address from text to binary form
    if (inet_pton(AF_INET, SERVER_IP, &server_address.sin_addr) <= 0) {
        std::cerr << "Invalid address/Address not supported" << std::endl;
        close(client_socket);
        return -1;
```

```
    }

    // Connect to the server
    if (connect(client_socket, (struct sockaddr*)&server_address,
sizeof(server_address)) < 0) {
        std::cerr << "Connection failed" << std::endl;
        close(client_socket);
        return -1;
    }

    // Send 100 bytes of data to the server
    char send_buffer[BUFFER_SIZE];
    memset(send_buffer, 'A', BUFFER_SIZE); // Fill buffer with 'A'

    ssize_t bytes_sent = send(client_socket, send_buffer, BUFFER_SIZE, 0);
    if (bytes_sent < 0) {
        std::cerr << "Failed to send data" << std::endl;
        close(client_socket);
        return -1;
    }

    std::cout << "Sent " << bytes_sent << " bytes to server" << std::endl;

    // Receive echo from the server
    char recv_buffer[BUFFER_SIZE];
    ssize_t bytes_received = recv(client_socket, recv_buffer, BUFFER_SIZE, 0);
    if (bytes_received < 0) {
        std::cerr << "Failed to receive data from server" << std::endl;
        close(client_socket);
        return -1;
    }

    std::cout << "Received " << bytes_received << " bytes from server: "
              << std::string(recv_buffer, bytes_received) << std::endl;

    // Close the socket
    close(client_socket);
    std::cout << "Connection closed" << std::endl;

    return 0;
}
```

When a client sends a large volume of data . Server taking the data and writing to file :

```
#include <iostream>
#include <cstring>      // For memset, memcpy
#include <sys/types.h>  // For data types like `sockaddr_in`
#include <sys/socket.h> // For socket functions
#include <arpa/inet.h>  // For inet_pton
#include <unistd.h>     // For close()
#include <fstream>      // For file I/O

#define BUFFER_SIZE 4096  // Size of the buffer to read each time

void receiveLargeData(int client_socket) {
    char buffer[BUFFER_SIZE];
    ssize_t bytes_received;
    size_t total_bytes_received = 0;

    // Open a file to write the data
    std::ofstream output_file("received_data.txt", std::ios::out |
std::ios::binary);
    if (!output_file.is_open()) {
        std::cerr << "Failed to open file for writing." << std::endl;
        close(client_socket);
        return;
    }

    // Loop to keep receiving data until all is received or the
connection is closed
    while (true) {
        // Receive data into buffer
        bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0);

        if (bytes_received > 0) {
            // Data received successfully
            total_bytes_received += bytes_received;
            std::cout << "Received " << bytes_received << " bytes,
Total: "
                      << total_bytes_received << " bytes" << std::endl;

            // Write the received data to the file
            output_file.write(buffer, bytes_received);
```

```
        } else if (bytes_received == 0) {
            // Connection closed by the client
            std::cout << "Connection closed by client. Total bytes
received: "
                      << total_bytes_received << " bytes" << std::endl;
            break;
        } else {
            // Error occurred during recv()
            std::cerr << "Error in recv()!" << std::endl;
            break;
        }
    }

    // Close the file after receiving all data
    output_file.close();

    // Close the socket after receiving data
    close(client_socket);
}

int main() {
    // Example for handling the server side
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        std::cerr << "Failed to create socket" << std::endl;
        return -1;
    }

    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(8080);

    // Bind the socket to the port
    if (bind(server_socket, (struct sockaddr*)&server_address,
sizeof(server_address)) < 0) {
        std::cerr << "Bind failed!" << std::endl;
        close(server_socket);
        return -1;
    }

    // Start listening for connections
    if (listen(server_socket, 10) < 0) {
        std::cerr << "Listen failed!" << std::endl;
        close(server_socket);
```

```
        return -1;
    }

    std::cout << "Server listening on port 8080..." << std::endl;

    while (true) {
        // Accept a connection from a client
        int client_socket = accept(server_socket, nullptr, nullptr);
        if (client_socket < 0) {
            std::cerr << "Accept failed!" << std::endl;
            continue;
        }

        std::cout << "Client connected!" << std::endl;

        // Receive large data from the client and write it to a file
        receiveLargeData(client_socket);
    }

    // Close the server socket
    close(server_socket);
    return 0;
}
```

socket api socket(AF_INET, SOCK_STREAM, 0) :

AF_INET: The socket will use IPv4 addresses.
SOCK_STREAM: The socket will use the TCP protocol, ensuring a reliable, ordered connection.
0: Use the default protocol (which will be TCP for IPv4 and stream-type sockets).

Common Protocol Options (for use with the socket() function):
1. IPPROTO_TCP
2. IPPROTO_UDP
3. IPPROTO_ICMP
4. IPPROTO_IPV6
5. IPPROTO_RAW
6. IPPROTO_SCTP
7. IPPROTO_IGMP

TCP: Use IPPROTO_TCP or 0 (default for SOCK_STREAM).
UDP: Use IPPROTO_UDP or 0 (default for SOCK_DGRAM).
ICMP: Use IPPROTO_ICMP for raw ping-like operations.

SCTP: Use IPPROTO_SCTP for applications requiring advanced message-oriented features.
RAW sockets: Use IPPROTO_RAW for custom protocol-level implementations.

```
// Wrapper function to get error message from errno
std::string get_error_message(int error_number) {
    return std::string(std::strerror(error_number)); // Convert the
C-style string to std::string
}
```

check the max value of backlog connections in listen() api :
command :
cat /proc/sys/net/core/somaxconn

You can change the backlog limit temporarily using:
command :
echo <new_value> | sudo tee /proc/sys/net/core/somaxconn

make the change permanent using command :
net.core.somaxconn = <new_value>

What happens when we do send() in tcp protocol :

Details of send() api :
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
Parameters:
sockfd:
Type: int
Description: This is the file descriptor of the socket. It is obtained when you create a socket using socket(), and it uniquely identifies the connection over which data is being sent.

buf:
Type: const void *
Description: A pointer to the buffer containing the data to be sent. This is the data that will be transmitted over the network.

len:
Type: size_t
Description: The length of the data in bytes that you want to send from the buffer buf.

flags:
Type: int

Description: Flags to control the behavior of the send operation. Common values include:
0: No special options.
MSG_DONTWAIT: Non-blocking send (does not wait if the send buffer is full).
MSG_OOB: Sends out-of-band data (only available for protocols that support it, like TCP).
MSG_NOSIGNAL: Prevents sending of SIGPIPE if the other end has closed the connection.

Return Value:
On Success:
Returns the number of bytes successfully sent. This may be less than the number of bytes you requested to send (len), indicating a partial send.

On Failure:
Returns -1, and the global variable errno is set to indicate the error.
Possible errno Values:

The send() function in a TCP socket is used to transmit data from a process to a connected socket.

1. User-Space to Kernel-Space Transition :

When the send() function is called, the data to be sent resides in user space.
System Call: The send() function initiates a system call, which causes the process to transition from user space to kernel space. The system call handler in the kernel takes over.

2. Copying Data to Kernel Space

Once in kernel space, the data from the user-space buffer is copied into a kernel-space buffer. This kernel buffer is a part of the socket's send buffer.
Socket Send Buffer: Every socket has its own send buffer, which temporarily holds the data before it is transmitted over the network. The size of this buffer is usually defined by the operating system (e.g., /proc/sys/net/ipv4/tcp_wmem in Linux) but can be adjusted via the SO_SNDBUF socket option.
At this point:

The user space application is freed from the data; it no longer needs to retain it in memory.
The send() call may return, depending on how much data the kernel can fit into the send buffer (more on this in the Blocking vs Non-Blocking section).

3. TCP Segmentation and Packetization

Segmentation: If the data being sent exceeds the Maximum Segment Size (MSS) of the TCP connection (usually around 1,500 bytes for Ethernet), the kernel will break it into smaller chunks. Each chunk will fit into a TCP segment.
TCP Header Generation: Each segment is wrapped with a TCP header, which includes important information like sequence numbers, acknowledgment numbers, flags, etc., for ensuring reliable delivery.

4. Data Transfer to IP Layer

Once the data is segmented and wrapped in TCP headers, it is passed down to the IP (Internet Protocol) layer.

The IP layer adds an IP header containing information like the source and destination IP addresses, fragmentation details, etc.

5. Routing and Delivery to Network Interface

The packet, now encapsulated with TCP and IP headers, is sent to the Network Interface Layer (the Data Link Layer in the OSI model). This is the layer responsible for physically transmitting the packet over the network.

If the destination is on the local network, the data is sent directly to the destination host. If the destination is remote, the packet is passed to a router for further forwarding.

Packet Transmission: At this stage, the network interface card (NIC) handles the actual transmission of the data onto the physical medium (Ethernet, Wi-Fi, etc.).

6. Transmission Control Protocol (TCP) Mechanisms

Sliding Window Protocol: TCP uses a sliding window protocol to control how much data can be sent at once. The window size is dynamic and based on both the receiver's buffer size and network conditions (congestion control). The TCP stack will not send more data than what the receiver can handle.

Acknowledgments (ACKs): After sending a segment, the sender waits for an acknowledgment (ACK) from the receiver. If the ACK is received, the corresponding data is removed from the socket's send buffer.

Retransmission on Failure: If an ACK is not received within a certain time (the retransmission timeout), TCP will retransmit the unacknowledged segments to ensure reliable delivery. This is part of TCP's reliability mechanisms.

7. Congestion Control and Flow Control

Congestion Control: TCP uses various algorithms like Reno, Cubic, or BBR to dynamically adjust the rate of data transmission depending on the current state of the network. This ensures that the sender does not overwhelm the network with too many packets, which could cause packet loss.

Flow Control: This mechanism ensures that the sender does not overwhelm the receiver. The receiver advertises a window size in each TCP acknowledgment, indicating how much more data it can accept. The sender must respect this window.

8. Network Delays and Transmission Time

The data may encounter delays due to various factors such as:

Network congestion

Routing delays

Transmission distance

Packet loss (leading to retransmissions)

During this time, the data is being transmitted over the physical network, hopping between routers and switches until it reaches the destination.

9. Reception by the Receiver

When the data arrives at the receiver, it follows a similar process in reverse:

The data is received at the network interface.

The TCP layer reassembles the segments in the correct order using the sequence numbers. Any missing segments are detected and requested for retransmission.
Finally, the data is passed up to the application layer (to recv() or read() on the receiver side).

10. Blocking vs Non-Blocking send()
In blocking mode (the default behavior):
If the kernel's send buffer is full, send() will block and wait until there is space available. Once some data is acknowledged by the receiver and removed from the send buffer, new data can be written.
In non-blocking mode (if the socket is set with O_NONBLOCK):
If the send buffer is full, send() will return -1 with the error EAGAIN or EWOULDBLOCK, meaning it cannot accept more data at the moment. The application will have to try sending again later.

11. Return Value of send()
The send() function returns the number of bytes successfully sent. This value might be smaller than the size of the buffer you tried to send, depending on the availability of space in the kernel's send buffer.
If send() fails due to an error, it returns -1 and sets errno to the appropriate error code.

recv() :
recv() function is used to receive data from a connected socket in a TCP connection. This API is the counterpart of send() and is essential for reading data transmitted from a remote socket.

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
Parameters:
sockfd:
Type: int
Description: The file descriptor of the connected socket, which is returned by the socket() or accept() functions. This represents the connection from which data is being received.

buf:
Type: void *
Description: A pointer to a buffer where the received data will be stored. This buffer must be allocated by the application and should be large enough to store the expected incoming data.

len:
Type: size_t
Description: The maximum number of bytes to read from the socket. This tells the system the size of the buf and limits how much data can be written to it in a single recv() call.

flags:
Type: int
Description: Flags that modify the behavior of the recv() function. Common flags include:

0: No special behavior (normal blocking receive).
MSG_DONTWAIT: Non-blocking receive. If no data is available, the function returns immediately with -1 and sets errno to EAGAIN or EWOULDBLOCK.
MSG_PEEK: Peek at the incoming data without removing it from the queue. This allows you to inspect the data without actually consuming it.
MSG_WAITALL: Wait until the entire buffer is full before returning, unless the connection is closed or an error occurs. Useful if you want to receive a fixed number of bytes.

Return Value:
On Success:
Returns the number of bytes received. This number can be less than len, especially if fewer bytes are available or if the message was smaller than the buffer size.
If recv() returns 0, it indicates that the remote socket has performed an orderly shutdown (i.e., the connection was closed).

On Failure:
Returns -1, and errno is set to indicate the specific error.
Common errno Values:
EAGAIN or EWOULDBLOCK: The socket is non-blocking, and no data is available to read.
EBADF: The socket file descriptor is not valid.
ECONNRESET: The connection was forcibly closed by the peer.
ENOTCONN: The socket is not connected.
EFAULT: The buffer points to invalid memory.
EINTR: A signal interrupted the recv() call before any data was received.

Now what happens when do recv() behind the scenes :

1. Data Transmission on the Network
Before recv() is called, some data must have been transmitted by the peer (client or server) on the other end of the connection. This data is broken into smaller units called TCP segments and transmitted over the network. Each TCP segment contains:

Data from the application layer (such as a file, a message, or HTTP request).
A TCP header with important control flags (like sequence numbers, window size, etc.).

2. Data Arrives at the Local Machine
Once the TCP segments reach the destination (your machine), they traverse through several layers of the network stack before becoming accessible to the recv() call. Here's what happens step-by-step:

a. Link Layer (Network Interface Card - NIC)
The data arrives at your machine's Network Interface Card (NIC) as frames. These frames are processed, removing the Ethernet or Wi-Fi headers, leaving the IP packet.
The NIC may offload certain tasks, such as checksum verification, to improve efficiency.
b. Internet Protocol (IP) Layer

The IP layer receives the IP packets. It removes the IP header, leaving only the TCP segment.
If the data is fragmented (because the TCP segment was too large to fit into a single packet), the IP layer reassembles the fragments into a complete TCP segment.
c. TCP Layer (Transport Layer)
The TCP segment is processed by the TCP layer. This layer ensures reliability through:
Sequence Numbers: TCP ensures that segments are received in the correct order by using sequence numbers. If packets are out of order, TCP will reorder them.
Acknowledgments (ACKs): The receiver acknowledges each segment it receives to inform the sender.
Flow Control: The sender adjusts the amount of data it sends based on the receiver's ability to process it, using a sliding window mechanism.
Retransmissions: If any TCP segment is missing (as detected through sequence numbers), the receiver will ask the sender to retransmit that segment.
The TCP layer strips off the TCP headers, checks for integrity using checksums, and places the resulting data in the receive buffer of the socket.

3. Kernel's Socket Layer
Once the TCP layer processes the data, it is placed in the receive buffer associated with the socket in the kernel. This buffer is a fixed-size queue, and if it fills up, the sender will be instructed (via TCP flow control mechanisms) to stop sending further data until there is space available.

a. Socket Receive Buffer
Each socket in the kernel has its own receive buffer where incoming data is stored. This buffer is limited in size (e.g., 64KB or more depending on the system settings).
Data is stored here until it is read by the application

4. User-Space Application Calls recv()
Now, the user-space application makes a call to recv() to retrieve the data. Here's what happens next:

a. Transition from User Space to Kernel Space (System Call)
When the recv() function is called in user space, a system call is made to switch from user mode to kernel mode. This is done so the application can interact with the kernel's networking stack.
b. Kernel Processes the recv() Call
The kernel checks the socket's receive buffer to see if any data is available.
If data is available, the kernel copies the data from the receive buffer to the user-provided buffer (buf argument in recv()).
c. Blocking vs Non-blocking Behavior
If no data is available:
In Blocking Mode (default): The recv() call blocks, meaning it waits until data is available. The process will be put into a sleep state by the kernel, freeing the CPU for other tasks until data arrives.
In Non-blocking Mode: The recv() call returns immediately with an error (EAGAIN or EWOULDBLOCK) if no data is available.

5. Partial Reads
Even if data is available, the recv() call may only return a portion of the data (i.e., a partial read). This can happen due to:

The size of the data in the socket's receive buffer being smaller than the size of the user-provided buffer.
The TCP segments being smaller than the requested size. In such cases, recv() will return the number of bytes it managed to read, and the application will have to loop and call recv() again to get the remaining data.

6. Return from Kernel to User Space
Once data has been copied into the user-provided buffer, the kernel returns control to the user space:

The return value of recv() indicates the number of bytes read from the socket.
If the connection has been closed by the peer (graceful shutdown), recv() will return 0.
If an error occurs (e.g., the connection was reset), recv() returns -1 and sets errno.

Kernel Space Buffer Management:
The receive buffer is a circular queue that holds incoming data. When data is added, the head pointer moves forward. When data is removed (via recv()), the tail pointer moves forward. If the buffer is full, TCP flow control mechanisms (such as adjusting the window size) will signal the sender to stop sending more data until the application reads from the buffer.
Data Copying (Zero Copy Optimization):
Typically, data is copied from the kernel's buffer to the user's buffer. However, in high-performance systems, optimizations like zero-copy mechanisms might be used. This eliminates the need to copy data between kernel and user space, improving efficiency.

Now creating the forever running tcp server :

```
#include <stdio.h>        // For printf and perror
#include <stdlib.h>       // For exit()
#include <string.h>       // For memset
#include <unistd.h>       // For close()
#include <arpa/inet.h>    // For socket functions and structures
(socket, bind, listen, accept, etc.)

#define PORT 8080         // Port number for the server
#define BACKLOG 5         // How many connections can be pending for
accept()

int main() {
```

```c
    int server_fd, new_socket;      // File descriptors for the server
and new client connection
    struct sockaddr_in address;     // Structure to hold server's
address information
    int addrlen = sizeof(address);  // Size of the address structure
    char buffer[1024] = {0};        // Buffer to store received data
    const char *hello = "Hello from server";  // Message to send to the
client

    // 1. Create a TCP socket using the IPv4 protocol (AF_INET) and
stream type (SOCK_STREAM)
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. Configure the server address
    address.sin_family = AF_INET;  // Use IPv4
    address.sin_addr.s_addr = INADDR_ANY; // Bind to all network
interfaces
    address.sin_port = htons(PORT); // Convert port to network byte
order (big-endian)

    // 3. Bind the socket to the IP address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 4. Listen for incoming connections
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    // 5. Loop to accept and handle client connections
    while (1) {  // Infinite loop to keep the server running
        printf("Waiting for a new connection...\n");

        // Accept a new connection (blocking call)
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
```

```
(socklen_t *)&addrlen)) < 0) {
            perror("accept failed");
            close(server_fd);
            exit(EXIT_FAILURE);
        }

        printf("Connection accepted from client.\n");

        // Read data sent by the client
        int valread = read(new_socket, buffer, 1024);
        printf("Received: %s\n", buffer);

        // Send a message back to the client
        send(new_socket, hello, strlen(hello), 0);
        printf("Hello message sent to client.\n");

        // Close the client socket when done
        close(new_socket);
        printf("Connection closed.\n");
    }

    // 6. Close the server socket when done (never reached in this case)
    close(server_fd);
    return 0;
}
```

Now handle each client in the separate thread ：

```
#include <iostream>
#include <thread>
#include <unistd.h>
#include <netinet/in.h>
#include <cstring>

#define PORT 8080
#define BACKLOG 5

// Function to handle the client connection in a separate thread
void handle_client(int client_socket) {
    char buffer[1024] = {0};
    const char *hello = "Hello from server";

    // Read data from the client
```

```
    int valread = read(client_socket, buffer, 1024);
    std::cout << "Received: " << buffer << std::endl;

    // Send a message back to the client
    send(client_socket, hello, strlen(hello), 0);
    std::cout << "Hello message sent to client." << std::endl;

    // Close the client socket
    close(client_socket);
    std::cout << "Connection closed." << std::endl;
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // 1. Create a TCP socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. Configure the server address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // 3. Bind the socket to the IP address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 4. Listen for incoming connections
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    std::cout << "Server listening on port " << PORT << "...\n";

    // 5. Loop to accept and handle client connections
```

```
    while (true) {
        std::cout << "Waiting for a new connection..." << std::endl;

        // Accept a new connection
        new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)&addrlen);
        if (new_socket < 0) {
            // Log error and continue accepting new connections without
exiting
            std::cerr << "accept failed: " << strerror(errno) <<
std::endl;
            continue;  // Go back to waiting for a new connection
        }

        std::cout << "Connection accepted from client.\n";

        // 6. Create a new thread for each client
        std::thread client_thread(handle_client, new_socket);

        // Detach the thread to automatically clean up when it's done
        client_thread.detach();
    }

    // 7. Close the server socket (never reached in this case)
    close(server_fd);
    return 0;
}
```

Now each client thread also runs forever until client closes the connection :

```
#include <iostream>
#include <thread>
#include <unistd.h>
#include <netinet/in.h>
#include <cstring>

#define PORT 8080
#define BACKLOG 5

// Function to handle the client connection in a separate thread and
```

```cpp
keep running until client closes the connection
void handle_client(int client_socket) {
    char buffer[1024] = {0};
    const char *hello = "Hello from server";

    while (true) {
        // Clear the buffer
        memset(buffer, 0, sizeof(buffer));

        // Read data from the client
        int valread = read(client_socket, buffer, 1024);

        // If valread == 0, the client has closed the connection
        if (valread <= 0) {
            std::cout << "Client disconnected.\n";
            break;
        }

        std::cout << "Received: " << buffer << std::endl;

        // Send a message back to the client
        send(client_socket, hello, strlen(hello), 0);
        std::cout << "Hello message sent to client.\n";
    }

    // Close the client socket after the client disconnects
    close(client_socket);
    std::cout << "Connection closed by server thread.\n";
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // 1. Create a TCP socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. Configure the server address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
```

```
    // 3. Bind the socket to the IP address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 4. Listen for incoming connections
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    std::cout << "Server listening on port " << PORT << "...\n";

    // 5. Loop to accept and handle client connections
    while (true) {
        std::cout << "Waiting for a new connection..." << std::endl;

        // Accept a new connection
        new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)&addrlen);
        if (new_socket < 0) {
            std::cerr << "accept failed: " << strerror(errno) <<
std::endl;
            continue;  // Go back to waiting for a new connection
        }

        std::cout << "Connection accepted from client.\n";

        // 6. Create a new thread for each client and let it run forever
until the client closes the connection
        std::thread client_thread(handle_client, new_socket);

        // Detach the thread to automatically clean up when it's done
        client_thread.detach();
    }

    // 7. Close the server socket (never reached in this case)
    close(server_fd);
    return 0;
}
```

Now creating a new thread for every client can lead to excessive resource usage, especially under heavy load. To address this, we can use a thread pool to limit the number of threads created and reuse them for handling multiple clients.
We will be using open source cpp thread pool library - **ctpl(C++ Thread Pool Library).**
link - https://github.com/vit-vit/CTPL (only download the **ctpl_stl.h** file)

code using the thread pool :

```
#include <iostream>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
#include "ctpl_stl.h"  // Include the thread pool library

#define PORT 8080
#define BACKLOG 5
#define THREAD_POOL_SIZE 4  // Define how many threads the pool should
have

// Function to handle the client connection
void handle_client(int client_socket) {
    char buffer[1024] = {0};
    const char *hello = "Hello from server";

    while (true) {
        // Clear the buffer
        memset(buffer, 0, sizeof(buffer));

        // Read data from the client
        int valread = read(client_socket, buffer, 1024);

        // If valread <= 0, the client has closed the connection or an
error occurred
        if (valread <= 0) {
            std::cout << "Client disconnected or error occurred.\n";
            break;
        }

        std::cout << "Received: " << buffer << std::endl;

        // Send a message back to the client
        send(client_socket, hello, strlen(hello), 0);
        std::cout << "Hello message sent to client.\n";
```

```
    }

    // Close the client socket after the client disconnects
    close(client_socket);
    std::cout << "Connection closed by server thread.\n";
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // Create a thread pool with a fixed number of threads
    ctpl::thread_pool pool(THREAD_POOL_SIZE);

    // 1. Create a TCP socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. Configure the server address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // 3. Bind the socket to the IP address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 4. Listen for incoming connections
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    std::cout << "Server listening on port " << PORT << "...\n";

    // 5. Loop to accept and handle client connections
    while (true) {
        std::cout << "Waiting for a new connection..." << std::endl;
```

```
        // Accept a new connection
        new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)&addrlen);
        if (new_socket < 0) {
            std::cerr << "accept failed: " << strerror(errno) <<
std::endl;
            continue;  // Go back to waiting for a new connection
        }

        std::cout << "Connection accepted from client.\n";

        // 6. Submit the client handling task to the thread pool
        pool.push(handle_client, new_socket);
    }

    // 7. Close the server socket (never reached in this case)
    close(server_fd);
    return 0;
}
```

Still there are a lot of improvements that are required in the above implementations.

Cons of the Above TCP Server:

Limited Scalability (Thread Pool Size):

Even though the server uses a thread pool to manage a fixed number of threads, if too many clients connect simultaneously, the server will reach its thread pool limit. Once the thread pool is fully utilized, new clients will be queued, potentially leading to delays in handling requests.
If the number of clients exceeds the server's ability to process them within a reasonable time, clients may face connection timeouts or slow responses.

Blocking I/O:
The server is using blocking I/O. When a thread is reading from a socket (using read()), it blocks the thread until data is available. If a client is slow or not sending any data, the thread will remain blocked, making inefficient use of the thread pool resources.
This can lead to a situation where threads are stuck waiting on clients, preventing them from serving other clients, even if the server has additional pending connections.

No Load Balancing:
The server doesn't have any mechanism for distributing incoming connections across multiple machines or services. This can lead to a bottleneck if the server is running on a single machine with limited CPU, memory, or network capacity.

No Graceful Shutdown:
The server doesn't provide a way to shut down gracefully. If the server needs to be stopped, it will just exit abruptly, possibly causing incomplete requests or data loss for active connections.

Error Handling and Client State:
Error handling is minimal. While the server logs errors, it doesn't offer any advanced error recovery strategies.
There’s no state management for clients. For example, if a client needs to maintain a persistent connection for long-running sessions, there’s no mechanism for tracking client state or reestablishing connections.

Lack of Security:
The server does not implement any security mechanisms like TLS/SSL to encrypt communication between the server and the clients. Without encryption, data exchanged over the network could be intercepted by malicious actors.

Ways in which this server can be improved . We will mostly look into the timeout and non-blocking i/o for event-driven architecture

Use Non-blocking I/O and Asynchronous Networking:
The biggest improvement would be switching from blocking I/O to non-blocking I/O or an asynchronous I/O model. This prevents threads from blocking on read() and write() calls. epoll (Linux), kqueue (BSD/MacOS), iocp(windows) or select() could be used for handling multiple connections in an event-driven way without requiring a dedicated thread for each client.

An asynchronous I/O library like Boost.Asio (which provides portable support for asynchronous socket programming) would allow the server to handle many more clients with fewer threads by using an event-driven, non-blocking approach.

Example libraries:
Boost.Asio (for cross-platform async I/O).
libuv (used in Node.js for async I/O).

Implement Connection Timeouts:
To avoid having threads blocked indefinitely due to unresponsive clients, you can implement connection timeouts. If a client doesn't send data or respond within a certain timeframe, the server can close the connection.

Graceful Shutdown:
Implement a way to gracefully stop the server. For example, handling a termination signal (**SIGTERM**) to close open sockets and notify clients before shutting down the server.

Load Balancing:
To improve scalability, consider implementing load balancing. Using a load balancer (e.g., HAProxy, NGINX, or Kubernetes) in front of multiple server instances will distribute client connections across multiple machines or containers, allowing you to scale horizontally.

Security Improvements (TLS/SSL):
Add support for TLS/SSL to encrypt communication. This is important if the server is deployed in production or handles sensitive information.
You can use OpenSSL or GnuTLS libraries to implement secure sockets.

Monitoring and Logging:
Implement more comprehensive logging and monitoring to track server performance, connection statistics, and potential issues (such as connection drops, memory leaks, etc.). This will help in diagnosing and improving server performance.
Use tools like Prometheus for real-time monitoring, combined with Grafana for dashboards.

Connection Pooling:
If the clients frequently connect and disconnect, consider implementing connection pooling to reduce the overhead of constantly establishing new connections.

Handling timeout for send() and recv() api code :

```
#include <iostream>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
#include <sys/socket.h>
#include <sys/types.h>
#include <fcntl.h>   // For fcntl()
#include <signal.h>  // For signal handling
#include "ctpl_stl.h"  // Include the thread pool library

#define PORT 8080
#define BACKLOG 5
#define THREAD_POOL_SIZE 4  // Define how many threads the pool should
have
#define TIMEOUT_SECONDS 5   // Timeout duration in seconds

// Function to handle the client connection
void handle_client(int client_socket) {
    char buffer[1024] = {0};
    const char *hello = "Hello from server";
```

```
    // Set receive and send timeout options
    struct timeval timeout;
    timeout.tv_sec = TIMEOUT_SECONDS;  // Set timeout duration
    timeout.tv_usec = 0;               // Microseconds

    // Set receive timeout
    if (setsockopt(client_socket, SOL_SOCKET, SO_RCVTIMEO, (const
char*)&timeout, sizeof(timeout)) < 0) {
        perror("setsockopt(SO_RCVTIMEO) failed");
        close(client_socket);
        return;
    }

    // Set send timeout
    if (setsockopt(client_socket, SOL_SOCKET, SO_SNDTIMEO, (const
char*)&timeout, sizeof(timeout)) < 0) {
        perror("setsockopt(SO_SNDTIMEO) failed");
        close(client_socket);
        return;
    }

    while (true) {
        // Clear the buffer
        memset(buffer, 0, sizeof(buffer));

        // Read data from the client
        int valread = read(client_socket, buffer, sizeof(buffer));

        // If valread <= 0, the client has closed the connection or an
error occurred
        if (valread <= 0) {
            std::cout << "Client disconnected or read timeout
occurred.\n";
            break; // Exit the loop, client connection will be closed
below
        }

        std::cout << "Received: " << buffer << std::endl;

        // Send a message back to the client
        int sent_bytes = send(client_socket, hello, strlen(hello), 0);
        if (sent_bytes < 0) {
            std::cerr << "Error sending message to client.\n";
            break; // Exit the loop, client connection will be closed
below
```

```
        }
        std::cout << "Hello message sent to client.\n";
    }

    // Close the client socket after the client disconnects
    close(client_socket);
    std::cout << "Connection closed by server thread.\n";
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // Create a thread pool with a fixed number of threads
    ctpl::thread_pool pool(THREAD_POOL_SIZE);

    // 1. Create a TCP socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. Configure the server address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // 3. Bind the socket to the IP address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 4. Listen for incoming connections
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    std::cout << "Server listening on port " << PORT << "...\n";

    // 5. Loop to accept and handle client connections
```

```
    while (true) {
        std::cout << "Waiting for a new connection..." << std::endl;

        // Accept a new connection
        new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t *)&addrlen);
        if (new_socket < 0) {
            std::cerr << "accept failed: " << strerror(errno) <<
std::endl;
            continue;  // Go back to waiting for a new connection
        }

        std::cout << "Connection accepted from client.\n";

        // 6. Submit the client handling task to the thread pool
        pool.push(handle_client, new_socket);
    }

    // 7. Close the server socket (never reached in this case)
    close(server_fd);
    return 0;
}
```

Useful api's for socket on linux :

setsocketopt() :
The signature of the setsockopt function in C/C++ :
int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);

Parameters:
int socket:
The file descriptor (or handle) for the socket on which you are setting the options.

int level:
Specifies the protocol level at which the option resides. Common values are:
SOL_SOCKET: Options at the socket level.
IPPROTO_TCP: TCP-specific options.
IPPROTO_IP: IP-specific options.
IPPROTO_IPV6: IPv6-specific options.

int option_name:
The specific socket option to set. For example:
SO_RCVTIMEO: Set receive timeout.
SO_SNDTIMEO: Set send timeout.

SO_REUSEADDR: Allow reuse of local addresses.
TCP_NODELAY: Disable Nagle's algorithm for TCP.
const void *option_value:
A pointer to the option value that will be set. It could be an integer, a structure, or any other data type depending on the option.

socklen_t option_len:
The size, in bytes, of the option value passed in option_value. For example, if option_value is an integer, you would pass sizeof(int).

Return Value:
On success: returns 0.
On failure: returns -1, and errno is set appropriately.

The setsockopt function is used in socket programming to configure various options on a socket. This can control the behavior of the socket and tailor its characteristics according to the requirements of your application. Here are some common scenarios and examples of when to use setsockopt:

Common Uses of setsockopt

Setting Socket Timeout:
You can set timeouts for reading and writing operations on a socket. This is useful in situations where you want to avoid indefinitely blocking on recv or send calls.
Example:

```
struct timeval timeout;
timeout.tv_sec = 5;  // Timeout of 5 seconds
timeout.tv_usec = 0;

// Set receive timeout
if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout,
sizeof(timeout)) < 0) {
    perror("setsockopt(SO_RCVTIMEO) failed");
}

// Set send timeout
if (setsockopt(sockfd, SOL_SOCKET, SO_SNDTIMEO, (const char*)&timeout,
sizeof(timeout)) < 0) {
    perror("setsockopt(SO_SNDTIMEO) failed");
}
```

Enabling or Disabling Address Reuse:

You can set the SO_REUSEADDR option to allow multiple sockets to bind to the same address. This is particularly useful when you want to quickly restart a server that listens on the same port.
Example:

```
int opt = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0)
{
    perror("setsockopt(SO_REUSEADDR) failed");
}
```

Setting the Maximum Socket Buffer Size:
You can use setsockopt to specify the size of the send and receive buffers. This can help optimize network performance based on the expected data traffic.
Example:

```
int buffer_size = 1024 * 1024;  // 1 MB buffer size
if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &buffer_size,
sizeof(buffer_size)) < 0) {
    perror("setsockopt(SO_SNDBUF) failed");
}

if (setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &buffer_size,
sizeof(buffer_size)) < 0) {
    perror("setsockopt(SO_RCVBUF) failed");
}
```

Controlling TCP Keepalive:
The SO_KEEPALIVE option can be used to enable TCP keepalive messages. This is useful for detecting dead connections and keeping idle connections alive.
Example:

```
int opt = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_KEEPALIVE, &opt, sizeof(opt)) < 0)
{
    perror("setsockopt(SO_KEEPALIVE) failed");
}
```

Controlling TCP No Delay:

The TCP_NODELAY option can be set to disable **Nagle's algorithm**, which is used to reduce the number of small packets sent over the network. This is beneficial for applications that require low latency.
Example:

```
int opt = 1;
if (setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)) < 0)
{
    perror("setsockopt(TCP_NODELAY) failed");
}
```

Summary
The setsockopt function is a powerful tool for configuring socket behavior. It can be used in various scenarios to optimize the performance and functionality of your network applications.

Here's a brief summary of when to use it:
Timeouts: To avoid blocking indefinitely on recv or send.
Address Reuse: To allow quick server restarts.
Buffer Sizes: To optimize network performance based on expected traffic.
Keepalive: To maintain connections and detect dead peers.
Nagle's Algorithm: To control packet transmission behavior for low-latency applications.
Considerations
When setting socket options, make sure to choose appropriate values based on your application's needs and the expected network conditions.
Always check the return value of setsockopt to ensure that the option was set successfully, and handle any errors appropriately.

fcntl :

Signature :
int fcntl(int fd, int cmd, ... /* arg */);

Parameters:
int fd:
The file descriptor on which you want to perform an operation. It can be a file descriptor, a socket descriptor, or any other descriptor.

int cmd:
The command that specifies the operation to perform. Some common commands are:
F_GETFL: Get the file status flags.
F_SETFL: Set the file status flags.
F_GETFD: Get the file descriptor flags.
F_SETFD: Set the file descriptor flags.
F_SETLK: Set a file lock (non-blocking).
F_SETLKW: Set a file lock (blocking).
F_GETOWN: Get the process or process group that receives SIGIO or SIGURG signals.
F_SETOWN: Set the process or process group that receives SIGIO or SIGURG signals.

... /* arg */:

A variable argument that can be used based on the command (cmd). For example, it can be a flag value if you are setting file descriptor flags (F_SETFL), or it may be omitted if you're only getting a value (F_GETFL).

Return Value:
On success: returns the value requested (e.g., the current flags) or 0 (if setting a flag).
On failure: returns -1, and errno is set appropriately.

ioctl :

Signature of ioctl :

int ioctl(int fd, unsigned long request, ...);

Parameters:
int fd:
This is the file descriptor on which to perform the control operation. It could be a socket, file descriptor, or any other device descriptor.

unsigned long request:
This is the control request or command that specifies the operation to be performed. Each command can require a different type of argument and perform a specific action.
Some common ioctl requests are:
FIONBIO: Set or clear the non-blocking mode on a socket.
SIOCGIFADDR: Get the address of an interface (used in network programming).
TIOCGWINSZ: Get terminal window size (used for terminals).
...:

A variable argument list, depending on the command. This can be a pointer to a structure or an integer value, depending on the request command. The exact argument needed depends on the specific request being issued.

Return Value:
On success: 0.
On failure: -1, and errno is set appropriately.

Example :

1. Set a Socket to Non-blocking Mode :

```c
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int set_socket_non_blocking(int sockfd) {
    int flags = 1;

    // FIONBIO is the request for setting non-blocking mode
    if (ioctl(sockfd, FIONBIO, &flags) == -1) {
        perror("ioctl(FIONBIO) failed");
        return -1;
    }

    printf("Socket is set to non-blocking mode.\n");
    return 0;
}
```

2. Getting and Setting Network Interface Information :

```c
#include <sys/ioctl.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void get_ip_address(const char *interface) {
    int fd;
    struct ifreq ifr;

    fd = socket(AF_INET, SOCK_DGRAM, 0);

    // Specify the interface name
    strncpy(ifr.ifr_name, interface, IFNAMSIZ-1);

    // Issue the ioctl call to get the IP address
    if (ioctl(fd, SIOCGIFADDR, &ifr) == -1) {
        perror("ioctl(SIOCGIFADDR) failed");
        close(fd);
        return;
    }
```

```
    // Extract the IP address from the ifr structure
    struct sockaddr_in *ipaddr = (struct sockaddr_in *)&ifr.ifr_addr;
    printf("IP Address: %s\n", inet_ntoa(ipaddr->sin_addr));

    close(fd);
}

int main() {
    get_ip_address("eth0");  // Replace "eth0" with your network
interface
    return 0;
}
```

ioctl :
Signature of ioctl :

int ioctl(int fd, unsigned long request, ...);

Parameters:
int fd:
This is the file descriptor on which to perform the control operation. It could be a socket, file descriptor, or any other device descriptor.

unsigned long request:
This is the control request or command that specifies the operation to be performed. Each command can require a different type of argument and perform a specific action.
Some common ioctl requests are:
FIONBIO: Set or clear the non-blocking mode on a socket.
SIOCGIFADDR: Get the address of an interface (used in network programming).
TIOCGWINSZ: Get terminal window size (used for terminals).
...:

A variable argument list, depending on the command. This can be a pointer to a structure or an integer value, depending on the request command. The exact argument needed depends on the specific request being issued.

Return Value:
On success: 0.
On failure: -1, and errno is set appropriately.

Example :

1. Set a Socket to Non-blocking Mode :

```c
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int set_socket_non_blocking(int sockfd) {
    int flags = 1;

    // FIONBIO is the request for setting non-blocking mode
    if (ioctl(sockfd, FIONBIO, &flags) == -1) {
        perror("ioctl(FIONBIO) failed");
        return -1;
    }

    printf("Socket is set to non-blocking mode.\n");
    return 0;
}
```

2. Getting and Setting Network Interface Information :

```c
#include <sys/ioctl.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void get_ip_address(const char *interface) {
    int fd;
    struct ifreq ifr;

    fd = socket(AF_INET, SOCK_DGRAM, 0);

    // Specify the interface name
    strncpy(ifr.ifr_name, interface, IFNAMSIZ-1);

    // Issue the ioctl call to get the IP address
    if (ioctl(fd, SIOCGIFADDR, &ifr) == -1) {
        perror("ioctl(SIOCGIFADDR) failed");
        close(fd);
        return;
    }
```

```
    // Extract the IP address from the ifr structure
    struct sockaddr_in *ipaddr = (struct sockaddr_in *)&ifr.ifr_addr;
    printf("IP Address: %s\n", inet_ntoa(ipaddr->sin_addr));

    close(fd);
}

int main() {
    get_ip_address("eth0");  // Replace "eth0" with your network
interface
    return 0;
}
```

To check your Linux kernel version :
Command : uname -r
Output : 5.15.0-47-generic

Or

Command : cat /proc/version
Output : Linux version 5.15.0-47-generic (buildd@lcy02-amd64-063) (gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #51~20.04.1-Ubuntu SMP Tue Aug 23 17:31:12 UTC 2022

TCP Keep-Alive Connections in a TCP Server :

Scenarios Where TCP Keep-Alive is Useful:
Long-lived idle connections: When a connection might be idle for a long time (e.g., a server maintaining connections to clients for real-time messaging), keep-alive ensures that stale connections are closed, freeing resources.
Network failures: If the client unexpectedly disconnects (due to network failure, or device going offline), without keep-alive, the server might not know and could keep the connection open indefinitely.
Resource Management: Helps the server clean up resources associated with dead connections, ensuring that connections aren't held open forever, which could lead to resource exhaustion.
High-availability systems: Systems that need to detect failed connections quickly (e.g., distributed systems or high-availability architectures) benefit from keep-alive to quickly detect and handle failures.

How TCP Keep-Alive Works:
The TCP stack sends keep-alive probes at intervals when there is no data exchange, checking if the other side is still responsive.
If the remote side fails to respond to a keep-alive probe (after several retries), the connection is considered dead, and the server closes the connection.
By default, the keep-alive timeout and interval settings can be quite large (e.g., 2 hours for the first probe), but they can be adjusted as needed.

example code ：

```
#include <sys/types.h>  // For data types used in sockets
#include <sys/socket.h> // For socket functions
#include <netinet/in.h> // For sockaddr_in and IP protocols
#include <arpa/inet.h>  // For inet_pton and address conversions
#include <unistd.h>     // For close()
#include <string.h>     // For memset
#include <iostream>     // For input-output stream
#include <netinet/tcp.h> // For TCP keep-alive options

#define PORT 8080
#define BACKLOG 10

void enableKeepAlive(int sockfd) {
    int optval = 1;
    socklen_t optlen = sizeof(optval);

    // Enable keep-alive option on the socket
    if (setsockopt(sockfd, SOL_SOCKET, SO_KEEPALIVE, &optval, optlen) <
0) {
        std::cerr << "Failed to enable SO_KEEPALIVE on the socket." <<
std::endl;
        return;
    }

    // Set the idle time before the first keep-alive probe is sent (in
seconds)
    optval = 10;  // 10 seconds
    if (setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPIDLE, &optval, optlen) <
0) {
        std::cerr << "Failed to set TCP_KEEPIDLE." << std::endl;
        return;
    }

    // Set the interval between subsequent keep-alive probes (in
seconds)
    optval = 5;  // 5 seconds
```

```
    if (setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPINTVL, &optval, optlen)
< 0) {
        std::cerr << "Failed to set TCP_KEEPINTVL." << std::endl;
        return;
    }

    // Set the maximum number of failed probes before the connection is
considered dead
    optval = 3;  // 3 probes
    if (setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPCNT, &optval, optlen) <
0) {
        std::cerr << "Failed to set TCP_KEEPCNT." << std::endl;
        return;
    }

    std::cout << "Keep-alive enabled on the socket with custom
settings." << std::endl;
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);

    // Create socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        std::cerr << "Socket failed" << std::endl;
        exit(EXIT_FAILURE);
    }

    // Forcefully attach socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt))) {
        std::cerr << "setsockopt failed" << std::endl;
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind the socket to the network address and port
    if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) <
0) {
```

```
        std::cerr << "Bind failed" << std::endl;
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_fd, BACKLOG) < 0) {
        std::cerr << "Listen failed" << std::endl;
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    std::cout << "Waiting for a connection..." << std::endl;

    // Accept an incoming connection
    if ((new_socket = accept(server_fd, (struct sockaddr*)&address,
(socklen_t*)&addrlen)) < 0) {
        std::cerr << "Accept failed" << std::endl;
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    std::cout << "Connection accepted." << std::endl;

    // Enable TCP keep-alive on the accepted socket
    enableKeepAlive(new_socket);

    // Simulate some server logic (e.g., reading/writing data)
    const char* hello = "Hello from server";
    send(new_socket, hello, strlen(hello), 0);
    std::cout << "Hello message sent" << std::endl;

    // Close sockets
    close(new_socket);
    close(server_fd);

    return 0;
}
```

Now coming back to the tcp server we were creating ：

When using a thread pool to create a TCP server where each client connection is handled by a thread from the pool, there is a key issue: Blocking I/O operations like send() and recv(). If the server is performing blocking I/O, a thread might get stuck waiting on slow

clients. This can lead to poor utilization of the thread pool because once a thread is blocked, it can't handle any other client until the operation completes.

In a high-throughput server with many clients, this could significantly limit scalability. To resolve this, we can adopt non-blocking sockets along with an I/O multiplexing technique such as epoll.

Key Concepts to Fix the Problem:

Non-blocking Sockets: By making the sockets non-blocking, send() and recv() return immediately if the operation cannot be performed, instead of blocking the thread. This allows the server to manage multiple connections without getting stuck on one slow connection.

epoll for I/O Multiplexing:
epoll is a Linux-specific I/O multiplexing mechanism that allows efficient monitoring of multiple file descriptors (like sockets).
With epoll, we can register multiple client sockets and wait for events (like data availability or readiness to write) without blocking. It avoids the overhead of iterating over many file descriptors, as it only focuses on sockets with pending events.

Thread Pool: The thread pool will handle CPU-intensive tasks or short I/O operations without getting blocked. When data is ready to be processed (detected via epoll), a thread from the pool can be assigned to handle it.

```
#include <iostream>
#include <thread>
#include <vector>
#include <unistd.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <cstring>
#include <functional>
#include <queue>
#include <mutex>
#include <condition_variable>

const int MAX_EVENTS = 10;
const int PORT = 8080;
const int BACKLOG = 128;

// Thread pool to manage worker threads
```

```cpp
class ThreadPool {
public:
    ThreadPool(size_t numThreads);
    ~ThreadPool();
    void enqueue(std::function<void()> task);

private:
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex queueMutex;
    std::condition_variable condition;
    bool stop;
};

ThreadPool::ThreadPool(size_t numThreads) : stop(false) {
    for (size_t i = 0; i < numThreads; ++i) {
        workers.emplace_back([this] {
            while (true) {
                std::function<void()> task;

                {
                    std::unique_lock<std::mutex> lock(this->queueMutex);
                    this->condition.wait(lock, [this] {
                        return this->stop || !this->tasks.empty();
                    });

                    if (this->stop && this->tasks.empty()) return;

                    task = std::move(this->tasks.front());
                    this->tasks.pop();
                }

                task();
            }
        });
    }
}

ThreadPool::~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(queueMutex);
        stop = true;
    }
    condition.notify_all();
    for (std::thread &worker : workers) worker.join();
}
```

```
void ThreadPool::enqueue(std::function<void()> task) {
    {
        std::unique_lock<std::mutex> lock(queueMutex);
        if (stop) throw std::runtime_error("enqueue on stopped
ThreadPool");
        tasks.push(std::move(task));
    }
    condition.notify_one();
}

// Utility to set a socket to non-blocking mode
void setNonBlocking(int sockfd) {
    int flags = fcntl(sockfd, F_GETFL, 0);
    fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
}

// Function to handle a single client connection
void handleClient(int client_fd) {
    char buffer[1024];
    while (true) {
        ssize_t bytes_read = recv(client_fd, buffer, sizeof(buffer), 0);
        if (bytes_read == -1) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                // No data to read, try again later
                break;
            } else {
                std::cerr << "recv error: " << strerror(errno) <<
std::endl;
                close(client_fd);
                return;
            }
        } else if (bytes_read == 0) {
            std::cout << "Client disconnected" << std::endl;
            close(client_fd);
            return;
        } else {
            // Echo back the message
            send(client_fd, buffer, bytes_read, 0);
        }
    }
}

int main() {
    // Create thread pool with 4 threads
    ThreadPool pool(4);
```

```
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        std::cerr << "Failed to create socket" << std::endl;
        return -1;
    }

    sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (sockaddr*)&server_addr, sizeof(server_addr)) ==
-1) {
        std::cerr << "Bind failed" << std::endl;
        close(server_fd);
        return -1;
    }

    if (listen(server_fd, BACKLOG) == -1) {
        std::cerr << "Listen failed" << std::endl;
        close(server_fd);
        return -1;
    }

    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        std::cerr << "Epoll create failed" << std::endl;
        close(server_fd);
        return -1;
    }

    epoll_event event{}, events[MAX_EVENTS];
    event.data.fd = server_fd;
    event.events = EPOLLIN;

    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &event) == -1) {
        std::cerr << "Epoll ctl failed" << std::endl;
        close(server_fd);
        close(epoll_fd);
        return -1;
    }

    while (true) {
        int n_fds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        for (int i = 0; i < n_fds; i++) {
```

```
            if (events[i].data.fd == server_fd) {
                // Accept new client connection
                int client_fd = accept(server_fd, nullptr, nullptr);
                if (client_fd == -1) {
                    std::cerr << "Accept failed" << std::endl;
                    continue;
                }
                setNonBlocking(client_fd);

                // Add new client socket to epoll for monitoring
                event.data.fd = client_fd;
                event.events = EPOLLIN | EPOLLET;  // Edge-triggered
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd,
&event) == -1) {
                    std::cerr << "Epoll ctl add client failed" <<
std::endl;
                    close(client_fd);
                }
            } else {
                // Data available on existing client socket
                int client_fd = events[i].data.fd;
                pool.enqueue([client_fd]() {
                    handleClient(client_fd);
                });
            }
        }
    }

    close(server_fd);
    close(epoll_fd);
    return 0;
}
```

In the above tcp server implementation with thread pool and epoll. In that let's use the ctpl thread pool library instead of writing our own . Then we want that client and server can communicate indefinitely until client send "q" in the message in which case we need to close the client socket .
Also we can have a dedicated thread to keep checking the epoll_wait and once an event arrives, we can check the event and push the task according to whether it is "read or write" to the thread pool. **Code for this ->**

```
#include <iostream>
#include <thread>
```

```cpp
#include <vector>
#include <unistd.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <cstring>
#include "ctpl_stl.h" // Include CTPL thread pool library

const int MAX_EVENTS = 10;
const int PORT = 8080;
const int BACKLOG = 128;
const int TIMEOUT = 5000; // 5 seconds

// Utility function to set a socket to non-blocking mode
void setNonBlocking(int sockfd) {
    int flags = fcntl(sockfd, F_GETFL, 0);
    fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
}

// Function to create a listening socket
int createSocket() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        std::cerr << "Failed to create socket" << std::endl;
        return -1;
    }

    int opt = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    sockaddr_in addr{};
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY; // Listen on all interfaces
    addr.sin_port = htons(PORT);

    if (bind(sockfd, (sockaddr*)&addr, sizeof(addr)) == -1) {
        std::cerr << "Bind failed" << std::endl;
        close(sockfd);
        return -1;
    }

    if (listen(sockfd, BACKLOG) == -1) {
        std::cerr << "Listen failed" << std::endl;
        close(sockfd);
```

```
        return -1;
    }

    setNonBlocking(sockfd);
    return sockfd;
}

// Function to handle client communication
void handleClient(int client_fd) {
    char buffer[1024];
    while (true) {
        memset(buffer, 0, sizeof(buffer));
        ssize_t bytes_received = recv(client_fd, buffer, sizeof(buffer)
- 1, 0);
        if (bytes_received > 0) {
            std::string message(buffer);
            std::cout << "Received from client: " << message <<
std::endl;

            // Check for termination message
            if (message.find("q") != std::string::npos) {
                std::cout << "Client requested to close the connection."
<< std::endl;
                break;
            }

            // Echo the message back to the client
            send(client_fd, buffer, bytes_received, 0);
        } else if (bytes_received == 0) {
            // Client closed the connection
            std::cout << "Client disconnected." << std::endl;
            break;
        } else if (errno != EWOULDBLOCK && errno != EAGAIN) {
            std::cerr << "Recv error occurred!" << std::endl;
            break;
        }
    }
    close(client_fd); // Close the client connection
    std::cout << "Connection closed with client." << std::endl;
}

int main() {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        std::cerr << "Epoll create failed" << std::endl;
        return -1;
```

```
    }

    // Create listening socket
    int listen_fd = createSocket();
    if (listen_fd == -1) {
        return -1;
    }

    // Setup CTPL Thread Pool with 4 threads
    ctpl::thread_pool pool(4);

    // Register listening socket with epoll
    epoll_event event{}, events[MAX_EVENTS];
    event.events = EPOLLIN;
    event.data.fd = listen_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &event) == -1) {
        std::cerr << "Failed to add listen socket to epoll" <<
std::endl;
        close(listen_fd);
        return -1;
    }

    // Dedicated thread to monitor epoll events
    std::thread epoll_thread([&]() {
        while (true) {
            int n_fds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
            for (int i = 0; i < n_fds; i++) {
                if (events[i].data.fd == listen_fd) {
                    // New client connection
                    int client_fd = accept(listen_fd, nullptr, nullptr);
                    if (client_fd != -1) {
                        std::cout << "New client connected" <<
std::endl;
                        setNonBlocking(client_fd);

                        // Register the new client socket for EPOLLIN
                        epoll_event client_event{};
                        client_event.events = EPOLLIN | EPOLLET; //
Edge-triggered
                        client_event.data.fd = client_fd;
                        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD,
client_fd, &client_event) == -1) {
                            std::cerr << "Failed to add client socket to
epoll" << std::endl;
                            close(client_fd);
                        }
```

```
                    } else {
                        std::cerr << "Accept failed" << std::endl;
                    }
                } else {
                    // Existing client has sent data, handle in a
separate thread from the pool
                    int client_fd = events[i].data.fd;
                    if (events[i].events & EPOLLIN) {
                        pool.push([client_fd](int id) {
                            handleClient(client_fd);
                        });
                    }
                }
            }
        }
    });

    epoll_thread.join();

    close(listen_fd);
    close(epoll_fd);
    return 0;
}
```

epoll api in detail :
The epoll API is a scalable I/O multiplexing mechanism available in the Linux kernel (introduced in version 2.5.44). It allows applications to monitor multiple file descriptors to see if I/O is possible on any of them. It is more efficient than older methods like poll() or select() when working with a large number of file descriptors.

1. **epoll_create1()**
int epoll_create1(int flags);

Parameters:
flags: If 0, it behaves like the older epoll_create(). Otherwise, use EPOLL_CLOEXEC to set the FD_CLOEXEC (close-on-exec) flag on the new epoll file descriptor.
Return Value:
Returns a file descriptor for the new epoll instance.
On failure, it returns -1 and sets errno accordingly.

2. **epoll_ctl()**
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

Parameters:
epfd: The epoll instance file descriptor returned by epoll_create1().
op: The operation to be performed. Possible values:
EPOLL_CTL_ADD: Add a new file descriptor to be monitored.
EPOLL_CTL_MOD: Modify the monitoring of an existing file descriptor.
EPOLL_CTL_DEL: Remove a file descriptor from monitoring.
fd: The target file descriptor to operate on.
event: A pointer to struct epoll_event which describes the desired events to be monitored.
Struct epoll_event:

```
struct epoll_event {
    uint32_t events;    // Epoll events
    epoll_data_t data;  // User data
};

typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

events: Specifies the events to monitor, such as:
EPOLLIN: The file descriptor is ready for reading.
EPOLLOUT: The file descriptor is ready for writing.
EPOLLERR: An error has occurred on the file descriptor.
EPOLLET: Enables edge-triggered behavior (the default is level-triggered).
EPOLLONESHOT: The event will be triggered only once, after which the file descriptor is disabled.
data: Used to store user-defined information like the file descriptor

Return Value:
Returns 0 on success.
Returns -1 on failure and sets errno.

3. epoll_wait()

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

Parameters:
epfd: The epoll instance file descriptor returned by epoll_create1().
events: A pointer to an array of struct epoll_event. This array is used to return information about file descriptors that have events ready.
maxevents: The number of events that can be returned (i.e., the size of the events array).
timeout: Specifies the maximum time (in milliseconds) that epoll_wait will wait for an event. A timeout of:
0 means non-blocking (returns immediately).
-1 means infinite blocking (wait indefinitely).
A positive value is the wait time in milliseconds.

Return Value:
Returns the number of file descriptors that are ready for I/O.
Returns -1 on failure and sets errno.

how to remove fd from epoll :

```
if (epoll_ctl(epfd, EPOLL_CTL_DEL, sockfd, NULL) == -1) {
perror("epoll_ctl: EPOLL_CTL_DEL"); close(epfd); return -1; }
```

Modify fd in epoll :

```
// Modify the file descriptor to monitor for writing (EPOLLOUT) instead
event.events = EPOLLOUT; // Changing from EPOLLIN to EPOLLOUT
 if (epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd, &event) == -1)
 {
 perror("epoll_ctl: EPOLL_CTL_MOD");
 close(epfd); return -1;
 }
```

Now coming back again to the tcp server , we can make it more efficient using the system api call io_uring :

io_uring :
io_uring is a Linux kernel API introduced in Linux 5.1 to provide high-performance asynchronous I/O operations. It aims to improve both the performance and scalability of I/O operations, especially for applications that need to handle large numbers of concurrent I/O requests, like high-performance network servers, databases, and storage systems.

io_uring is built on the idea of submission and completion queues, allowing user-space applications to submit I/O requests to the kernel without the need for system calls for each I/O operation, thus reducing the overhead of context switches between user-space and kernel-space.

How io_uring Works :
Submission Queue (SQ): This is where applications submit their I/O requests. Instead of making a system call for each I/O operation, the application writes the request to the submission queue.

Completion Queue (CQ): After the I/O request is completed by the kernel, the result is placed in the completion queue. The application reads from the completion queue to get the status of completed requests.

This model allows for batching and asynchronous I/O operations, greatly improving performance, particularly when handling many I/O operations in parallel.

Comparison with select and epoll :

1. Blocking Nature of select and epoll
select: An older mechanism used to monitor multiple file descriptors to see if any of them are ready for I/O (read, write, or exceptional conditions). It has a limit on the number of file descriptors and scales poorly.

epoll: A more efficient mechanism introduced to address the scalability issues of select. It allows monitoring a large number of file descriptors and performs better with many concurrent connections.

io_uring: Unlike select and epoll, io_uring is not just about monitoring file descriptors but also about providing an asynchronous interface to I/O operations (e.g., reading, writing, accepting connections). It eliminates the need for repeated system calls, which reduces overhead.

2. System Call Overhead
select and epoll: Both make system calls to monitor file descriptors. Each time you need to poll for events, you make a system call (select(), epoll_wait(), etc.).

io_uring: Reduces system call overhead by allowing multiple I/O requests to be submitted to the kernel at once. With SQPOLL mode (where a kernel thread monitors the submission queue), even the system call for submitting the request can be avoided, making I/O almost entirely asynchronous from the application's point of view.

3. Performance and Scalability
select: Performance degrades significantly with a large number of file descriptors since it has to scan through all descriptors each time.

epoll: More scalable than select, as it uses an event-driven model, so only file descriptors with events are processed. However, it still involves system calls, which add context-switching overhead.

io_uring: Designed for high scalability, it allows batch submissions and completions of I/O requests with minimal system call overhead. This provides lower latency and better performance, especially when handling many concurrent I/O operations.

4. Ease of Use
select and epoll: Both are relatively easy to use and well-established in the Linux ecosystem.

io_uring: Being a newer interface, it is more complex and requires more effort to implement effectively, as it introduces new concepts like submission and completion queues.

5. Kernel Support
select and epoll: These are stable and widely available even in older kernels (available since Linux 2.x).

io_uring: Only available in Linux 5.1 and later. Some features of io_uring (e.g., SQPOLL) require Linux 5.6 or later. Therefore, its usage is limited to more modern systems.

6. Reduced Context Switching: With io_uring, you can submit many I/O requests without requiring a system call for each one, reducing the number of context switches between user-space and kernel-space.

7. Support for Asynchronous I/O: Unlike epoll which is for event-driven I/O readiness notification, io_uring provides a true asynchronous I/O API, making it more suitable for systems that need to perform actual non-blocking I/O operations (like reading or writing to files/sockets).

8. Thread Safety: While io_uring can reduce system call overhead and improve performance, managing io_uring structures in multi-threaded environments requires careful synchronization, which can add some complexity.

10. Kernel Version Dependency: io_uring is only available on newer kernels (Linux 5.1+), and some features are available only in even newer kernel versions (like Linux 5.6+). This limits its use on older systems.

Example :
Using io_uring to submit a file read request :

```
#include <liburing.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
#include <string.h>

int main() {
    struct io_uring ring;
    struct io_uring_cqe *cqe;
    struct io_uring_sqe *sqe;
    int ret, fd;
    char buffer[4096];

    // Initialize io_uring with 8 entries in the submission/completion
queues
    ret = io_uring_queue_init(8, &ring, 0);
    if (ret < 0) {
        std::cerr << "io_uring_queue_init failed: " << strerror(-ret) <<
std::endl;
        return 1;
    }

    // Open a file
    fd = open("testfile.txt", O_RDONLY);
    if (fd < 0) {
```

```
        std::cerr << "Failed to open file: " << strerror(errno) <<
std::endl;
        io_uring_queue_exit(&ring);
        return 1;
    }

    // Get a submission queue entry
    sqe = io_uring_get_sqe(&ring);
    if (!sqe) {
        std::cerr << "Failed to get SQE" << std::endl;
        io_uring_queue_exit(&ring);
        return 1;
    }

    // Prepare a read operation
    io_uring_prep_read(sqe, fd, buffer, sizeof(buffer), 0);

    // Submit the request to the kernel
    ret = io_uring_submit(&ring);
    if (ret < 0) {
        std::cerr << "io_uring_submit failed: " << strerror(-ret) <<
std::endl;
        close(fd);
        io_uring_queue_exit(&ring);
        return 1;
    }

    // Wait for the completion of the request
    ret = io_uring_wait_cqe(&ring, &cqe);
    if (ret < 0) {
        std::cerr << "io_uring_wait_cqe failed: " << strerror(-ret) <<
std::endl;
        close(fd);
        io_uring_queue_exit(&ring);
        return 1;
    }

    // Check if the operation was successful
    if (cqe->res < 0) {
        std::cerr << "Read failed: " << strerror(-cqe->res) <<
std::endl;
    } else {
        std::cout << "Read " << cqe->res << " bytes: " <<
std::string(buffer, cqe->res) << std::endl;
    }
```

```
    // Mark the completion entry as seen
    io_uring_cqe_seen(&ring, cqe);

    // Clean up
    close(fd);
    io_uring_queue_exit(&ring);

    return 0;
}
```

In the above code , we are using **"liburing" which is a wrapper above the io_uring.** This makes it easy to use io_uring by avoiding to write a lot of boilerplate code for setting io_uring.

io_uring_queue_init: Initializes the io_uring instance with a queue depth of 8.
io_uring_get_sqe: Retrieves a submission queue entry (SQE).
io_uring_prep_read: Prepares a read operation to read from a file.
io_uring_submit: Submits the request to the kernel.
io_uring_wait_cqe: Waits for the request to complete.
io_uring_cqe_seen: Marks the completion queue entry as "seen" once processed.

Async Programming :

Async programming and I/O multiplexing are essential techniques for handling multiple I/O-bound tasks efficiently. They address the problem of blocking I/O operations in a program that needs to handle multiple connections or perform several tasks simultaneously.

The Problem: Synchronous I/O
Consider a scenario where we want to handle multiple client connections in a server. A synchronous approach would handle each client one by one, blocking the server's main thread until each operation completes.

Synchronous I/O Example

```
#include <iostream>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

void handleClient(int clientSocket) {
    char buffer[1024] = {0};
```

```
    read(clientSocket, buffer, 1024);  // Blocking read
    std::cout << "Received: " << buffer << std::endl;
    write(clientSocket, "Hello from server", 17);
    close(clientSocket);
}

int main() {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    sockaddr_in address = {AF_INET, htons(8080), INADDR_ANY};
    bind(serverSocket, (sockaddr*)&address, sizeof(address));
    listen(serverSocket, 3);

    while (true) {
        int clientSocket = accept(serverSocket, nullptr, nullptr);  //
Blocking accept
        handleClient(clientSocket);
    }

    close(serverSocket);
    return 0;
}
```

Issues with Synchronous I/O:
Blocking Calls: Both accept() and read() are blocking calls, meaning the program waits until the operation completes, wasting CPU time.

Poor Scalability: If there are multiple clients, the server must handle them one by one, leading to poor performance.

Unresponsive Server: While handling one client, other clients cannot be served, making the server unresponsive.

Solving the Problem with Non-Blocking I/O :

By setting the socket to non-blocking mode, read() or accept() calls will return immediately if no data is available, allowing the server to perform other tasks.

```
#include <iostream>
#include <unistd.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <cstring>
```

```
void setNonBlocking(int socket) {
    int flags = fcntl(socket, F_GETFL, 0);   // Get current flags for
the socket
    fcntl(socket, F_SETFL, flags | O_NONBLOCK);  // Set the socket to
non-blocking
}

void handleClient(int clientSocket) {
    char buffer[1024];
    memset(buffer, 0, sizeof(buffer));

    // Try reading data from the client socket
    int bytesRead = read(clientSocket, buffer, sizeof(buffer));
    if (bytesRead > 0) {
        std::cout << "Received from client: " << buffer << std::endl;
        write(clientSocket, "Hello from server", 17);  // Send response
back to the client
    } else if (bytesRead == -1 && (errno != EWOULDBLOCK && errno !=
EAGAIN)) {
        std::cerr << "Error reading from client socket!" << std::endl;
        close(clientSocket);
    } else {
        // If no data to read, return and keep the socket open for
future reads
    }
}

int main() {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        std::cerr << "Failed to create socket" << std::endl;
        return -1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(8080);

    if (bind(serverSocket, (struct sockaddr*)&serverAddr,
sizeof(serverAddr)) == -1) {
        std::cerr << "Bind failed" << std::endl;
        close(serverSocket);
        return -1;
    }
```

```
    if (listen(serverSocket, 5) == -1) {
        std::cerr << "Listen failed" << std::endl;
        close(serverSocket);
        return -1;
    }

    setNonBlocking(serverSocket);  // Set the server socket to
non-blocking

    while (true) {
        int clientSocket = accept(serverSocket, nullptr, nullptr);
        if (clientSocket == -1) {
            if (errno != EWOULDBLOCK && errno != EAGAIN) {
                std::cerr << "Accept failed!" << std::endl;
            }
            // No client connection to accept; continue looping
        } else {
            setNonBlocking(clientSocket);  // Set the client socket to
non-blocking
            std::cout << "Accepted new client connection" << std::endl;

            // Handle client in non-blocking mode
            handleClient(clientSocket);
        }

        // Perform other tasks or continue handling other clients
        usleep(1000);  // Simulate some processing time
    }

    close(serverSocket);
    return 0;
}
```

Explanation

Non-Blocking Mode:
The setNonBlocking() function sets the socket to non-blocking mode using fcntl() with the O_NONBLOCK flag. This is done for both the server socket and any accepted client sockets.

Accepting New Connections:

accept() is called in a loop. If there is no new connection, it immediately returns -1 with errno set to EWOULDBLOCK or EAGAIN, which are non-blocking indicators. The server continues its loop without getting blocked.

Reading from Clients:
The read() call on the client socket is also non-blocking. If no data is available, it immediately returns -1 with errno set to EWOULDBLOCK or EAGAIN, allowing the program to continue without blocking.

Handling Clients Efficiently:
The server can handle multiple clients by repeatedly accepting new connections and reading from existing ones without blocking on any single socket.

Benefits of Non-Blocking I/O

Improved Responsiveness: The server is not blocked waiting for I/O operations to complete, making it more responsive to incoming connections.

Concurrent Handling: Multiple clients can be handled concurrently without multi-threading or forking processes.

CPU Utilization: The server can utilize CPU time more effectively by not idling during blocking calls.

I/O MULTIPLEXING :
select() and poll() are system calls used for monitoring multiple file descriptors (like sockets, pipes, etc.) to see if they have become "ready" for some class of I/O operation (e.g., input/output). Both are widely used in network programming to implement I/O multiplexing, allowing a single-threaded program to handle multiple I/O sources concurrently.

select()
select() is a system call that allows a program to monitor multiple file descriptors, waiting until one or more of them become "ready" for some class of I/O operation.
It uses three separate "sets" of file descriptors—one for reading, one for writing, and one for exceptions. The kernel modifies these sets to indicate which file descriptors are ready.
select() has a limitation: it has a fixed maximum number of file descriptors it can handle (often FD_SETSIZE, which is typically 1024).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
```

```c
#include <fcntl.h>

int main() {
    fd_set readfds;   // Set of file descriptors to monitor for reading
    struct timeval tv; // Timeout structure
    int retval;

    // Set timeout to 5 seconds
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    FD_ZERO(&readfds);  // Clear the set
    FD_SET(STDIN_FILENO, &readfds);  // Add standard input (file
descriptor 0) to the set

    // Wait for an input on stdin, with a 5-second timeout
    retval = select(STDIN_FILENO + 1, &readfds, NULL, NULL, &tv);

    if (retval == -1) {
        perror("select()");  // Error occurred
    } else if (retval) {
        printf("Data is available now.\n");
        // FD_ISSET(STDIN_FILENO, &readfds) will be true
    } else {
        printf("No data within five seconds.\n");
    }

    return 0;
}
```

poll()
poll() is similar to select(), but it scales better with a larger number of file descriptors.
Unlike select(), poll() does not have a limit on the number of file descriptors it can handle.
It uses a structure array (struct pollfd) to specify the file descriptors to monitor and the events of interest (e.g., read, write, exceptions).
poll() can handle large sets of file descriptors and is more efficient when monitoring thousands of file descriptors.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <poll.h>
```

```c
#include <fcntl.h>

int main() {
    struct pollfd fds[1];  // Array of pollfd structures
    int ret;

    // Monitor stdin (file descriptor 0) for input
    fds[0].fd = STDIN_FILENO;  // File descriptor to monitor
    fds[0].events = POLLIN;    // Monitor for input

    // Poll with a timeout of 5 seconds
    ret = poll(fds, 1, 5000); // Timeout in milliseconds (5000 ms = 5 seconds)

    if (ret == -1) {
        perror("poll()");  // Error occurred
    } else if (ret == 0) {
        printf("No data within five seconds.\n");  // Timeout
    } else {
        if (fds[0].revents & POLLIN) {
            printf("Data is available now.\n");  // Input is available
        }
    }

    return 0;
}
```

Key Differences:

Key Differences:

Feature	`select()`	`poll()`
Data Structure	`fd_set`	`struct pollfd[]`
Limitations	Limited by `FD_SETSIZE` (typically 1024)	No inherent limit on file descriptors
Efficiency	Less efficient for large numbers of FDs	More efficient for large numbers of FDs
Usability	Three separate sets for read, write, and exceptions	One array with multiple event types

Both functions have been largely superseded by more modern alternatives like epoll (Linux) and kqueue (BSD/MacOS) or IOCP for windows for high-performance I/O operations in real-world applications.

Epoll :

What is epoll?
epoll is a Linux I/O event notification facility that allows a program to efficiently monitor multiple file descriptors to see if they have I/O available. It is used for handling large numbers of file descriptors in a scalable way and is an alternative to older polling mechanisms like select() and poll().

epoll is especially useful in high-performance server applications (like HTTP or game servers) that need to handle many simultaneous connections. It is an edge-triggered or level-triggered interface, which means it only provides notifications when the state of a monitored file descriptor changes.

Key Components of epoll
epoll_create() / epoll_create1(): Creates an epoll instance.
epoll_ctl(): Used to add, modify, or remove file descriptors from the epoll instance.
epoll_wait(): Waits for events on the file descriptors that have been added to the epoll instance.
Example of epoll in C++
Below is a simple example of how to use epoll to handle multiple file descriptors (e.g., multiple client connections on a server) in C++.

```
#include <iostream>
#include <unistd.h>
#include <sys/epoll.h>
#include <arpa/inet.h>
#include <cstring>
#include <fcntl.h>
```

```c
#define MAX_EVENTS 10
#define PORT 8080

// Set socket to non-blocking mode
void setNonBlocking(int socket) {
    int opts = fcntl(socket, F_GETFL);
    if (opts < 0) {
        perror("fcntl(F_GETFL)");
        exit(EXIT_FAILURE);
    }
    opts = (opts | O_NONBLOCK);
    if (fcntl(socket, F_SETFL, opts) < 0) {
        perror("fcntl(F_SETFL)");
        exit(EXIT_FAILURE);
    }
}

int main() {
    int server_fd, client_fd, epoll_fd;
    struct sockaddr_in address;
    struct epoll_event ev, events[MAX_EVENTS];
    int addrlen = sizeof(address);
    char buffer[1024] = {0};

    // Create server socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind socket to port
    if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) <
0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_fd, 5) < 0) {
        perror("listen failed");
        close(server_fd);
```

```
        exit(EXIT_FAILURE);
    }

    // Create epoll instance
    epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1 failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    ev.events = EPOLLIN;  // Input event
    ev.data.fd = server_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &ev) == -1) {
        perror("epoll_ctl: server_fd");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    while (true) {
        int num_fds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        for (int i = 0; i < num_fds; ++i) {
            if (events[i].data.fd == server_fd) {  // Incoming
connection
                client_fd = accept(server_fd, (struct
sockaddr*)&address, (socklen_t*)&addrlen);
                if (client_fd < 0) {
                    perror("accept failed");
                    continue;
                }
                setNonBlocking(client_fd);  // Set client socket to
non-blocking

                ev.events = EPOLLIN | EPOLLET;  // Input event,
edge-triggered
                ev.data.fd = client_fd;
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &ev)
== -1) {
                    perror("epoll_ctl: client_fd");
                    close(client_fd);
                    continue;
                }
                std::cout << "Accepted new connection: " << client_fd <<
std::endl;
            } else {  // Data from a client
                memset(buffer, 0, sizeof(buffer));
```

```
                ssize_t bytes_read = read(events[i].data.fd, buffer,
sizeof(buffer));
                if (bytes_read <= 0) {
                    // Connection closed or error
                    if (epoll_ctl(epoll_fd, EPOLL_CTL_DEL,
events[i].data.fd, NULL) == -1) {
                        perror("epoll_ctl: remove client_fd");
                    }
                    close(events[i].data.fd);
                    std::cout << "Closed connection: " <<
events[i].data.fd << std::endl;
                } else {
                    // Print received data
                    std::cout << "Received from " << events[i].data.fd
<< ": " << buffer << std::endl;
                    // Echo back to the client
                    send(events[i].data.fd, buffer, bytes_read, 0);
                }
            }
        }
    }

    close(server_fd);
    close(epoll_fd);
    return 0;
}
```

Level-Triggered vs. Edge-Triggered

The terms edge-triggered and level-triggered refer to how notifications are generated when monitored file descriptors (like sockets or pipes) are ready for I/O operations (such as reading or writing).

Level-Triggered (LT) Mode:

Definition: This is the default mode of operation for epoll. In level-triggered mode, epoll repeatedly notifies the user as long as the file descriptor is ready for an I/O operation (like reading or writing).
Behavior: When a file descriptor becomes "ready" (e.g., data is available to read), epoll continues to report it in every call to epoll_wait() until the readiness condition is cleared (e.g.,

the data is read). This means if the application doesn't consume the event, it will keep receiving notifications.
Use Case: Useful when you want to ensure that no events are missed, such as in applications where handling every piece of available data is critical.

Edge-Triggered (ET) Mode:

Definition: Edge-triggered mode, on the other hand, only notifies the user when a "change" occurs in the state of the file descriptor. It only triggers when the state transitions (e.g., from no data to available data).
Behavior: In ET mode, once epoll_wait() reports that a file descriptor is ready, it will not report it again until the readiness condition changes again. For example, if there is new data to read after reading the previous data.
Use Case: This mode is more efficient for scenarios where high performance is needed and you want to avoid handling the same event multiple times. However, it requires careful handling to ensure that all data is read in one go.

Key Differences Summarized

Feature	Level-Triggered (LT)	Edge-Triggered (ET)
Notification	Repeatedly, as long as the condition is true	Only once, when the condition changes (e.g., new data arrives)
Use Case	Simpler to implement but less efficient	More efficient but requires more careful handling
Ideal Scenario	When you need to ensure all data is processed	When you want to minimize system calls and overhead
Data Handling	Can be read partially, and the next call will notify	Must read until `EAGAIN` to consume all data

Level-Triggered Example :

```
// Level-triggered example using epoll
#include <sys/epoll.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```c
#define MAX_EVENTS 5
#define READ_SIZE 10

int main() {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        exit(EXIT_FAILURE);
    }

    int server_fd = /* setup your server socket */;

    struct epoll_event event, events[MAX_EVENTS];
    event.events = EPOLLIN;  // Level-triggered by default
    event.data.fd = server_fd;
    epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &event);

    while (1) {
        int event_count = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        for (int i = 0; i < event_count; i++) {
            if (events[i].data.fd == server_fd) {
                int client_fd = /* accept a new client */;
                event.events = EPOLLIN;
                event.data.fd = client_fd;
                epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &event);
            } else {
                char buffer[READ_SIZE + 1];
                int bytes_read = read(events[i].data.fd, buffer,
READ_SIZE);
                if (bytes_read == -1) {
                    perror("read");
                    close(events[i].data.fd);
                    continue;
                }
                buffer[bytes_read] = '\0';
                printf("Read: %s\n", buffer);
                // Even if we read partially, epoll_wait() will continue
to trigger this file descriptor
                // until all data is read
            }
        }
    }

    close(server_fd);
    close(epoll_fd);
    return 0;
```

```
}
```

Explanation: In level-triggered mode, if there is more data available on a socket than READ_SIZE, epoll_wait() will keep returning events for that file descriptor until all data is read.

Edge-Triggered Example :

```
// Edge-triggered example using epoll
#include <sys/epoll.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define MAX_EVENTS 5
#define READ_SIZE 10

int main() {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        exit(EXIT_FAILURE);
    }

    int server_fd = /* setup your server socket */;
    struct epoll_event event, events[MAX_EVENTS];
    event.events = EPOLLIN | EPOLLET;  // EPOLLET for edge-triggered
mode
    event.data.fd = server_fd;
    epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &event);

    while (1) {
        int event_count = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        for (int i = 0; i < event_count; i++) {
            if (events[i].data.fd == server_fd) {
                int client_fd = /* accept a new client */;
                event.events = EPOLLIN | EPOLLET;  // Edge-triggered for
```

```
clients too
                event.data.fd = client_fd;
                epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &event);
            } else {
                char buffer[READ_SIZE + 1];
                while (1) {  // Keep reading until all data is read
                    int bytes_read = read(events[i].data.fd, buffer,
READ_SIZE);
                    if (bytes_read == -1) {
                        if (errno == EAGAIN || errno == EWOULDBLOCK)
break;  // All data read
                        perror("read");
                        close(events[i].data.fd);
                        break;
                    } else if (bytes_read == 0) {  // Connection closed
                        close(events[i].data.fd);
                        break;
                    }
                    buffer[bytes_read] = '\0';
                    printf("Read: %s\n", buffer);
                }
            }
        }
    }

    close(server_fd);
    close(epoll_fd);
    return 0;
}
```

Explanation: In edge-triggered mode, epoll_wait() will only notify the application once after the file descriptor becomes ready. If all data is not read in one go, no further notifications will be received until new data arrives. Therefore, the code must read until it gets EAGAIN or EWOULDBLOCK to ensure that all data is read.

Pros of Using select() Over epoll() :

Portability:

select() is a part of the POSIX standard, which makes it highly portable across different platforms like Linux, macOS, BSD, and Windows. This is unlike epoll(), which is specific to Linux.

Simplicity and Ease of Use:

select() has a simpler and more straightforward API, making it easier to understand and use for smaller applications or quick prototyping. It's often used in introductory networking examples because of its simplicity.

Well-Documented:
select() has been around for a long time and is well-documented. It is widely understood by developers, and a lot of existing codebases still use it.
Sufficient for Small Applications:

If an application is dealing with a small number of file descriptors (less than 1024), select() is usually sufficient and doesn't introduce significant overhead.

Cons of Using select() Over epoll()

Scalability Issues:
select() has a limit on the maximum number of file descriptors it can monitor, defined by FD_SETSIZE (typically 1024). This limit can be increased, but doing so increases memory usage linearly with the number of file descriptors.

Performance Overhead:

select() requires the kernel to scan all file descriptors passed to it, which means it has O(n) complexity. This leads to inefficiencies as the number of file descriptors increases. For large-scale applications, this can lead to performance degradation.

No Edge-Triggered Notifications:

Unlike epoll(), which supports both level-triggered and edge-triggered notifications, select() only provides level-triggered notifications. This means that you need to call select() repeatedly to check the status of file descriptors, which can lead to increased CPU usage.
Modification of fd_set:

The fd_set passed to select() is modified by the kernel, meaning you need to reset it before every call. This makes the code more verbose and error-prone.

Copying Overhead:

Every time select() is called, the fd_set is copied from user space to kernel space, which adds extra overhead, especially when handling a large number of file descriptors.

Simple code in nodejs server to check the performance against our tcp server :

index.js file -

```
const express = require('express');
const app = express();

// Define the route for the root URL
app.get('/', (req, res) => {
    // Get current date and time in the local time zone
    const currentDateTime = new Date().toLocaleString();

    // Send the response with the current date and time in red text
    res.send(`<html><body><h1
style="color:red;">${currentDateTime}</h1></body></html>`);
});

// Start the server on port 3000
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

//to install nodejs on linux :
commands :
sudo apt-get install nodejs
sudo apt install npm

npm init -y
npm install express
node index.js

Full code for tcp server on linux using epoll :

Lin.h

```
#ifndef LINSERVER_H
#define LINSERVER_H

#include "../tcpserver.h"
#include <string>
#include <sys/epoll.h>
#include "ctpl_stl.h"
#include <unordered_map>
```

```
struct ClientState {
    int client_socket;
    std::string recvBuffer;  // Buffer for holding incoming data
    std::string sendBuffer;  // Buffer for holding outgoing data
    size_t bytesSent;        // Tracks how many bytes have been sent
    bool waitingForSend;     // True if waiting to send more data
    bool waitingForRecv;     // True if waiting to receive more data

    ClientState() : client_socket(-1), bytesSent(0), waitingForSend(false),
waitingForRecv(true) {}
    explicit ClientState(int csocket) : client_socket(csocket), bytesSent(0),
waitingForSend(false), waitingForRecv(true){}
};

class LinServer : public TCPServer
{
public:
    LinServer();
    virtual ~LinServer();

    bool initialize(int port, const std::string &ip_address = "0.0.0.0")
override;
    void start() override;
    void handleClient(int client_socket);
    bool setSocketNonBlocking(int socketId);

private:
    int server_fd;                          // File descriptor for the server
socket
    int epoll_fd;                           // File descriptor for epoll
    // Map to store state for each client
    std::unordered_map<int, ClientState> clientStates;
    std::thread epollThread;                // dedicated thread for epoll_wait()
    static ctpl::thread_pool threadPool;  // Static thread pool shared by all
instances
    static int determineThreadPoolSize(); // Helper to determine the thread
pool size
    void handleRecv(int client_socket);
    void handleSend(int client_socket);
    void handleError(int client_socket);
    void epollLoop(); // The dedicated epoll thread
```

```
    void prepareAndSendResponse(int client_socket, const std::string
&response);
    std::string getDateTimeHTMLResponse();
    bool isRequestComplete(int client_socket);
};

#endif // LINSERVER_H
```

Lin.cpp :

```
#include "lin.h"
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <thread>
#include <mutex>
#include <fcntl.h>
#include <ctime>
#include <cstring>
#include <errno.h> // For errno

using namespace std;
// Buffer size for receiving messages
#define BUFFER_SIZE 1024

#define MAX_EVENTS 10

mutex coutMutex; // mutex to sync the output on console . we can comment the
lock code.

LinServer::LinServer() : server_fd(-1), epoll_fd(-1) {}

LinServer::~LinServer()
{
    if (server_fd != -1)
    {
```

```
        close(server_fd);
    }
    if (epollThread.joinable())
    {
        epollThread.join();
    }
}

// Initialize the static thread pool using the hardware concurrency
ctpl::thread_pool LinServer::threadPool(LinServer::determineThreadPoolSize());

// Helper function to determine the number of threads based on hardware
concurrency
int LinServer::determineThreadPoolSize()
{
    int concurrency = std::thread::hardware_concurrency();
    return (concurrency > 0) ? concurrency : 4; // Default to 4 if hardware
concurrency is unavailable
}

bool LinServer::setSocketNonBlocking(int socketId)
{
    int flags = fcntl(socketId, F_GETFL, 0);
    if (flags == -1)
    {
        std::cerr << "Error getting socket flags!" << std::endl;
        return false;
    }

    flags |= O_NONBLOCK;
    if (fcntl(socketId, F_SETFL, flags) == -1)
    {
        std::cerr << "Error setting socket to non-blocking!" << std::endl;
        return false;
    }

    std::cout << "Socket set to non-blocking mode." << std::endl;
    return true;
}

void LinServer::handleClient(int client_socket)
{
```

```
    char buffer[BUFFER_SIZE] = {0};

    // Continue to receive data until the client closes the connection
    while (true)
    {
        // Clear the buffer before receiving new data
        memset(buffer, 0, sizeof(buffer));

        // Receive data from the client
        int valread = read(client_socket, buffer, BUFFER_SIZE);

        // If the read is less than or equal to 0, the client has closed the
connection
        if (valread <= 0)
        {
            std::lock_guard<std::mutex> lock(coutMutex);
            std::cout << "Client disconnected or error occurred. Closing
connection." << std::endl;
            break; // Exit the loop to close the connection
        }

        // Log the received data
        {
            std::lock_guard<std::mutex> lock(coutMutex);
            std::cout << "Received: " << buffer << std::endl;
        }

        // Prepare the HTTP response
        std::string http_response =
            "HTTP/1.1 200 OK\r\n"
            "Content-Type: text/plain\r\n"
            "Content-Length: " +
            std::to_string(valread) + "\r\n"
                                      "\r\n" +
            std::string(buffer, valread); // Echo the received data

        // Send the HTTP response back to the client
        send(client_socket, http_response.c_str(), http_response.length(), 0);

        {
            std::lock_guard<std::mutex> lock(coutMutex);
            std::cout << "Echo response sent!" << std::endl;
```

```
        }
    }

    // Close the connection
    close(client_socket);
    {
        std::lock_guard<std::mutex> lock(coutMutex);
        std::cout << "Client connection closed." << std::endl;
    }
}

bool LinServer::initialize(int port, const std::string &ip_address)
{
    cout << "initialising the server..." << endl;
    struct sockaddr_in address;
    int opt = 1;

    // Create socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        std::cerr << "Socket creation failed!" << std::endl;
        return false;
    }

    // Set socket options to reuse address and port
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt,
sizeof(opt)))
    {
        std::cerr << "setsockopt failed!" << std::endl;
        return false;
    }

    epoll_fd = epoll_create1(0);
    if (epoll_fd == -1)
    {
        std::cerr << "Epoll creation failed!" << std::endl;
        return false;
    }

    // Set up the server address structure
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY; // all available ip addresses
```

```
    address.sin_port = htons(port);       // Set the port number

    // Bind the socket to the IP address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
    {
        std::cerr << "Bind failed!" << std::endl;
        return false;
    }

    // Start listening for connections
    if (listen(server_fd, 3) < 0)
    {
        std::cerr << "Listen failed!" << std::endl;
        return false;
    }

    std::cout << "Server initialized on " << ip_address << ":" << port <<
std::endl;
    return true;
}

void LinServer::start()
{
    epollThread = std::thread(&LinServer::epollLoop, this);
    struct sockaddr_in client_address;
    socklen_t addrlen = sizeof(client_address);
    char buffer[BUFFER_SIZE] = {0};
    int client_socket = -1;

    std::cout << "Waiting for connections..." << std::endl;

    while (true)
    { // Infinite loop to accept connections continuously
        // Accept an incoming connection
        if ((client_socket = accept(server_fd, (struct sockaddr
*)&client_address, &addrlen)) < 0)
        {
            std::cerr << "Accept failed! Error: " << strerror(errno) <<
std::endl;
            continue; // Continue to the next iteration to accept new
connections
        }
```

```
        std::cout << "Connection accepted!" << std::endl;

        // Set client socket to non-blocking
        setSocketNonBlocking(client_socket);

        // Add the new client socket to the epoll set for monitoring
        struct epoll_event ev;
        ev.events = EPOLLIN | EPOLLOUT | EPOLLET;
        ev.data.fd = client_socket;
        epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_socket, &ev);

        std::cout << "New client connected and added to epoll." << std::endl;
    }
}

void LinServer::epollLoop()
{
    struct epoll_event events[MAX_EVENTS];

    while (true)
    {
        int event_count = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        for (int i = 0; i < event_count; i++)
        {
            int fd = events[i].data.fd;

            if (events[i].events & EPOLLIN)
            {
                // Push to thread pool for receiving data
                threadPool.push([this, fd](int thread_id)
                                { this->handleRecv(fd); });
            }
            else if (events[i].events & EPOLLOUT)
            {
                // Push to thread pool for sending data
                threadPool.push([this, fd](int thread_id)
                                { this->handleSend(fd); });
            }
            else if (events[i].events & EPOLLERR)
            {
                // Handle errors
                threadPool.push([this, fd](int thread_id)
```

```
                                { this->handleError(fd); });
            }
        }
    }
}

// Function to get the current date and time in HTML format
std::string LinServer::getDateTimeHTMLResponse()
{
    // The HTML content includes a script to display the time based on the
client's timezone
    std::string htmlResponse =
        "<html><head><title>Echo Server</title></head>"
        "<body><h1>Echo Server</h1>"
        "<p>Below is the current time in your timezone:</p>"
        "<p style='color:red;' id='localTime'></p>"
        "<script>"
        "function getLocalTime() {"
        "  const now = new Date();"
        "  const options = { weekday: 'long', year: 'numeric', month: 'long',
day: 'numeric', hour: 'numeric', minute: 'numeric', second: 'numeric',
timeZoneName: 'short' };"
        "  const localTime = now.toLocaleString(undefined, options);"
        "  document.getElementById('localTime').innerText = localTime;"
        "}"
        "window.onload = getLocalTime;"
        "</script>"
        "</body></html>";

    std::string httpResponse =
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html\r\n"
        "Content-Length: " +
        std::to_string(htmlResponse.size()) + "\r\n"
                                              "\r\n" +
        htmlResponse;

    return httpResponse;
}

void LinServer::handleRecv(int client_socket)
{
```

```
    // Ensure client state is initialized the first time
    if (clientStates.find(client_socket) == clientStates.end())
    {
        clientStates[client_socket] = ClientState(client_socket); //
Initialize client state
    }

    auto &state = clientStates[client_socket]; // Access the client's state
    char buffer[BUFFER_SIZE];
    int bytesRead = recv(client_socket, buffer, BUFFER_SIZE, 0);

    if (bytesRead > 0)
    {
        // std::cout << "Received data from client: " << std::string(buffer,
bytesRead) << std::endl;

        state.recvBuffer.append(buffer, bytesRead);

        // Check if the request is complete (e.g., HTTP would check for
\r\n\r\n or content length)
        if (isRequestComplete(state.client_socket))
        {
            // Once the entire request is received, prepare the response
            // Prepare and send a response (Echo + Date/Time HTML response)
            std::string response = getDateTimeHTMLResponse();
            prepareAndSendResponse(client_socket, response);
        }
        else
        {
            // If the request is incomplete, mark that we're still waiting for
more data
            state.waitingForRecv = true;
        }
    }
    else if (bytesRead == 0)
    {
        std::cout << "Client disconnected." << std::endl;
        close(client_socket);
        clientStates.erase(client_socket); // Clean up state
    }
    else
    {
```

```
        if (errno != EWOULDBLOCK && errno != EAGAIN)
        {
            std::cerr << "Error in recv: " << strerror(errno) << std::endl;
            handleError(client_socket);
        }
        else
        {
            // more data to recv
            return;
        }
    }
}

void LinServer::handleSend(int client_socket)
{
    auto &state = clientStates[client_socket]; // access client state

    if (state.bytesSent < state.sendBuffer.size())
    {
        // Calculate how much more data needs to be sent
        int bytesToSend = state.sendBuffer.size() - state.bytesSent;
        int bytesSent = send(client_socket, state.sendBuffer.c_str() +
state.bytesSent, bytesToSend, 0);

        if (bytesSent > 0)
        {
            state.bytesSent += bytesSent; // Update the number of bytes sent
so far
            if (state.bytesSent == state.sendBuffer.size())
            {
                // If all data has been sent, reset the state
                std::cout << "All data sent to client." << std::endl;
                state.bytesSent = 0;
                state.sendBuffer.clear();
                state.waitingForSend = false;
                state.waitingForRecv = true; // Ready to receive again
            }
        }
        else
        {
            if (errno == EAGAIN || errno == EWOULDBLOCK)
            {
```

```
                //wait for the next opportunity to send
                return;
            }
            else
            {
                // An error occurred, handle it
                std::cerr << "Error in send." << std::endl;
                handleError(client_socket);
            }
        }
    }
}

// Prepare the response data and initiate sending process
void LinServer::prepareAndSendResponse(int client_socket, const std::string
&response)
{
    auto &clientState = clientStates[client_socket];
    clientState.client_socket = client_socket;
    clientState.sendBuffer = response;
    clientState.bytesSent = 0;

    handleSend(client_socket); // Start sending the data
}

void LinServer::handleError(int client_socket)
{
    std::cerr << "Socket error, closing connection." << std::endl;
    close(client_socket);
}

// Check if the request has been fully received
bool LinServer::isRequestComplete(int client_socket) {
    // Example logic: Check if the request ends with "\r\n\r\n"
    return true;//clientStates[client_socket].recvBuffer.find("\r\n\r\n") !=
std::string::npos;
}
```