Harsh Chaudhary Kalwar
221715
Replacement Task
Set A

Q1a) Why is TCP considered connection-oriented? Justify using its state transition diagram and discuss how each state impacts data flow control.

↳

TCP (Transmission control Protocol) is considered connection-oriented because it establishes a virtual connection between the client and server before the data transfer begins, ensuring reliable, ordered, and error-checked communication.

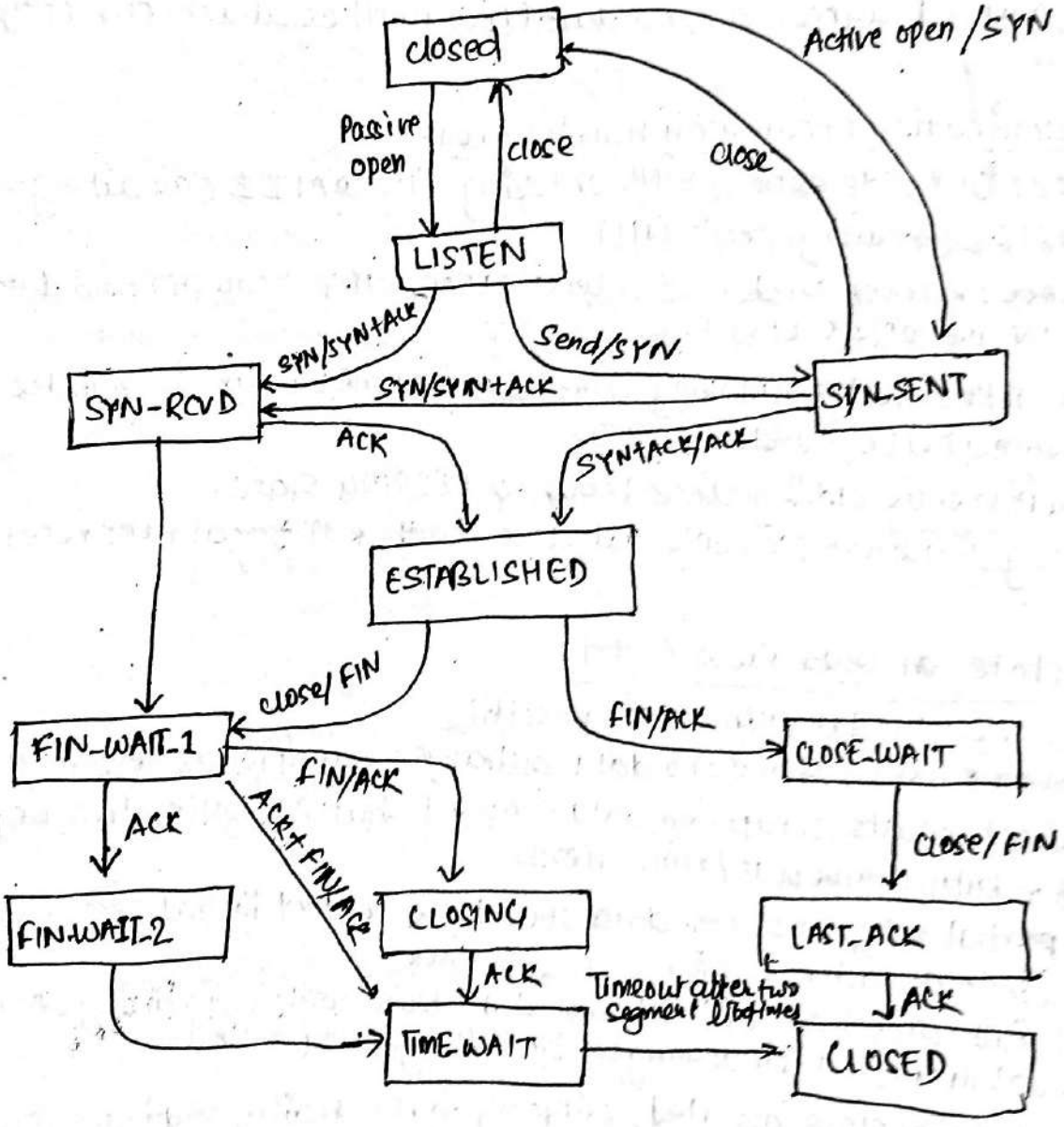

fig: TCP State-Transition Diagram

The TCP State Transition Diagram illustrates the connection-oriented nature by showing distinct phases: establishment, data exchange, and termination. It mandates a handshake to move from CLOSED to ESTABLISHED state, preventing data flow until connection is confirmed.

- Connection Establishment (Three-way Handshake):
  - ↳ Client starts in CLOSED (no connection) and moves to SYN_SENT by sending a SYN segment.
  - ↳ Server, in LISTEN (passive open, awaiting requests), receives ~~SN~~ SYN and transitions to SYN-RCVD by sending SYN+ACK.
  - ↳ Client receives SYN+ACK, sends ACK, and moves to ~~established~~ ESTABLISHED. Server also enters ESTABLISHED upon receiving ACK.
  - ↳ This ensure mutual agreement; no data flow until established, justifying reliability.
- Connection Termination (four-way Handshake):
  - ↳ Active close: One side sends FIN, entering FIN_WAIT_1 (awaiting ACK) then FIN_WAIT_2 (awaiting peer's FIN).
  - ↳ Passive close: Receiver sends ACK, enters CLOSE_WAIT (can still send data), then sends FIN and enters LAST-ACK.
  - ↳ Both sides acknowledge, entering TIME_WAIT (2MSL wait to handle delayed segments) before ~~clo~~ CLOSED.
  - ↳ Rare simultaneous ~~closing~~ close leads to CLOSING state.
  - ↳ This orderly shutdown prevents data loss, with RST for abrupt resets.

Impact of each state on Data flow control:

1. CLOSED: No connection; data flow not possible
2. LISTEN: Server socket passive: no data exchange; but queues incoming SYNs
3. SYN_SENT: Client awaits setup; no data sent yet, but RTT estimation begins for future timeouts / transitions.
4. SYN_RCVD: partial setup; no user data; but flow control initialized via windows advertisements in SYN+ACK
5. ESTABLISHED: full connection: bidirectional data flow with sequencing, ACKs for reliability, and windowing for flow / congestion control.
6. FIN_WAIT_1/2: Active close initiated; outgoing data drains, but incoming data allowed until peer closes

7. CLOSE_WAIT ; Passive close ; incoming data processed, outgoing data sent
8. LAST_ACK/CLOSING : Final ACK/FIN exchange; no new data, focuses on flusing queues and preventing duplicates.
9. TIME_WAIT : No data flow; waits for 2MSL to absorb stray packets, controlling cleanup and preventing erroneus data in new connections.

Name: Harsh Chaudhary Kalwar Roll no: 221715

1b) UDP lacks reliability guarantees, yet it is widely used in multimedia applications. Explain why and design a basic protocol logic(Pseudo-code or snipper) that adds reliability over UDP.

↳

UDP is connectionless, unreliable, and lacks sequencing, acknowledgement, or retransmissions, leading to potential packet loss, duplication, or out-of-order delivery. However, it is widely used in multimedia applications (e.g., video streaming, online gaming) due to its low overhead and minimal latency.
TCP's reliability mechanisms such as three-way handshakes, ACKs, and retransmissions introduces delay that are unacceptable in real-like scenarios; a lost packet may cause a brief glitch, but retransmission could stall playback entirely.
UDP's simplicity (~~the~~ 8-byte header vs TCP's 20 byte header) enables faster transmission, allowing applications to handle errors at higher layers.
For Instance, in VoIP (e.g., Skype), a minor audio drops are tolerable, prioritizing ~~lowest~~ low jitter over perfect delivery, making UDP ideal for time-sensitive, loss-tolerant media.

To ~~implement~~ add reliability over UDP, a simple stop-and-wait protocol with sequence numbers, ACKs, timeouts, and retransmissions can be implemented. Below is pseudo-code for sender and receiver sides assuming a datagram socket.

Sender Pseudo-Code:

```
Initialize seq_num = 0
initialize timeout = 1 second  // adjustable based on RTT

function send_reliable(data):
    packet = {seq_num : seq_num, payload: data} // Add seq to UDP payload
    sendto(socket, packet, dest_addr)    // UDP send
    start_timer(timeout)
```

```
while True:
  if recvfrom (socket, ack_packet, ~~ser~~ src_addr) and ack_packet.ack_num
      == seq_num:
    seq_num + 1  // Increment for next packet
    break        // ACK received, proceed
  elif timer_expired():
    retransmit (packet)          // Resend on timeout
    reset_timer (timeout * 2)    // Exponential backoff.
```

Receiver Pseudo-Code :

```
initialize expected_seq=0

function receive_reliable():
  while True:
    packet = recvfrom (socket)    // UDP receive
    if packet.seq_num = expected_seq:
    deliver_to_app (packet.payload)  // pass to application)
    sendto (socket, {ack_num: expected_seq}, src_addr) //send ACK
    expected_seq += 1    // update for next
  ~~elif~~
  else:
    sendto (socket, {ack_num: expected_seq - 1}, src addr)
```

The above logic mimics the TCP's basics: ensure ordered-delivery via sequencing, reliability via ACK/ retransmissions, and flow control via timeouts.

Name: Harsh Chaudhary Kalwar | Roll no: 221715

Q2) Why do we need different types of socket address structures like sockaddr_in, ~~sockaddr_out~~ sockaddr_un, sockaddr_in6 ? Illustrate their use cases with brief C code snippets.

↳

Key reasons behind why we need different types of socket address structures like sockaddr_in, sockaddr_un, and sockaddr_in6 include:

1. Protocol-specific Addressing: Each protocol family (AF_INET, AF_UNIX, AF_INET6) has distinct address formats. For example IPV4 uses 32-bit addressed, IPV6 uses 128-bit addresses, and Unix domain uses file paths.
2. Interoperability: The generic ~~socket~~ sockaddr allow socket functions (e.g., bind(), connect()) to work uniformly, but casting to specific structures (sockaddr_in) enables protocol-specific handling.
3. Scalability and Flexibility: Structures like sockaddr_storage accomodate any address type, ensuring future-proofing for new protocols.
4. Alignment and Size: Different structures ensure proper memory alignment and sufficient space for protocol-specific data, avoiding errors in system calls.

Use case for each socket address structure is mentioned below:

1. sockaddr_in (IPV4 Sockets, AF_INET)

↳ used for IPV4-based communication, such as web servers or FTP, requiring a 32-bit IP address and 16-bit port number.

↳ Code

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

void setup_ipv4_server (){
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr;
    bzero (&servaddr, sizeof(serv_addr));
    serv_addr.sin_family = Af_INET;
    serv_addr.sin_addr.s_addr = ~~htonl~~ htonl(INADDR_ANY); //Wildcard IP
```

```
serv_addr.sin_port = htons(8080);    port 8080
bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

}
```

↳ sockaddr_in holds IPv4 address (sin_addr) and port (sin_port). Used for TCP/UDP over IPv4 networks. e.g., HTTP servers.

2. sockaddr_un (Unix Domain Sockets, AF_UNIX)

↳ used for inter-process communication (IPC) on the same host, leveraging file system paths for efficiency (e.g., faster than TCP for local communication).

↳ Code

```
#include <sys/un.h>
#include <sys/socket.h>

void setup_unix_server() {
    int sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    struct sockaddr_un addr;
    bzero(&addr, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, "/tmp/mysocket", sizeof(addr.sun_path)-1);
    unlink("/tmp/mysocket"); Remove old socket file
    bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));

}
```

↳ sockaddr_un uses a pathname (sun_path) for local IPC. Ideal for processes on the same machine. eg database or service communication.

3. sockaddr_in6 (IPV6 Sockets, AF_INET6)

↳ used for IPV6 networks, supporting 128-bits addresses for modern internet-applications, such as cloud services or IoT.

↳ Code

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

void setup_ipv6_client() {
    int sockfd = socket(AF_INET6, SOCK_STREAM, 0);
```

```
struct sockfd sockaddr_in6 serv_addr;
bzero (&serv_addr, sizeof(serv_addr));
serv_addr.sin6_family = AF_INET6;
serv_addr.sin6_port = htons(8080);
inet_pton(AF_INET6, "::1", &serv_addr.sin6_addr); // loopback IPv6
connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
}
```

↳ sockaddr_in6 handle 128-bit IPv6 addresses (sin6_addr) and ports (sin6_port). Used for next gen networks with larger address spaces.

Harsh chaudhary kalwar (221715)

2b) How does incorrect byte ordering affect socket communication between machines of different architecture? Demonstrate how functions like htons() and ntohl() resolve this issue in a TCP program.

↳

Byte ordering refers to how multi-byte data (e.g., integers) is stored in memory. Different architectures use different byte orders. In socket communication, machines with different architectures may misinterpret multi-byte values like IP addresses or port numbers if byte ordering is not standardized. For example:

i) if a little-endian client sends a port number (e.g., 8080) to a big-endian server without conversion, the server may read it as 0x901F (36895), instead of 0x1F90, causing connection failure / incorrect routing.

ii) Similarly, an IP address like 192.168.1.1 (as a 32-bit integer) could be misinterpreted, leading to packets being sent to the wrong destination.

iii) This mismatch disrupt communication, as TCP/IP expects data in network byte order (big-endian) for fields like ports (sin_port) and addresses (sin_addr) in socket structures.

It leads to errors in establishing connections, data misrouting, or protocol misinterpretation, breaking reliable communication.

P.T.O.

- htons() and ntohl() resolve this issue by:

htons(): Converts 16-bit values (e.g. port numbers) from host to network order.
ntohl(): Converts 32-bit values (e.g. IP addresses) from network to host order.

TCP C Program

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

int main(){
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0){
        perror("socket creation failed");
        return 1;
    }

    struct sockaddr_in serv_addr;
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(8080);
    inet_pton(AF_INET, "192.168.1.1", &serv_addr.sin_addr);

    if(connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr))<0){
        perror("Connection failed");
        return 1;
    }

    struct sockaddr_in peer_addr;
    socklen_t len = sizeof(peer_addr);
    getpeername(sockfd, (struct sockaddr*)&peer_addr, &len);
    char ip_str[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &peer_addr.sin_addr, ip_str, INET_ADDRSTRLEN);
    printf("Connected to Server IP: %s, Port: %u\n", ip_str, ntohs(peer_addr.
                                                        sin_port));
    close(sockfd); return 0;
}
```

↳ Code Explanation:

1. Port Conversion (htons): The client sets serv_addr.sin_port = htons(8080) to convert the port number (16-bit) to network byte order, ensuring the server interprets it correctly regardless of its architecture.

2. IP Address Handling: inet_pton() converts the IP string to a binary 32-bit value in network byte order, avoiding manual conversion for sin_addr.

3. Retrieving Peer Info (ntohs, ntohl): After connecting, getpeername() retrieves the server's address. ntohs(peer_addr.sin_port) converts the port back to host byte order for printing and inet_ntop() handle IP conversion, implicitly using ntohl for the binary-to-string conversion.

4. Impact: These functions ensure the client and server agree on port and IP values, preventing miscommunication. For example, a little-endian client sending 8080 (0x1F90) as ~~0x1f90~~ 0x901F is corrected by htons() to 0x1F90, matching the server's expectation.

This ensures reliable TCP communication across diverse architectures, maintaining protocol integrity.

Harsh Chaudhary Kalwar (221715)

Q3) A Unix server needs to handle SIGNIT without terminating. Explain how signal handling modifies control flow, and demonstrate using sigaction() in a simple socket program.

↳
Signal handling modifies control flow as follows:

i) Normal flow: The process executes sequentially in user mode.

ii) Signal Arrival: The kernal interrupts execution, saves the current context, and switches to kernel mode.

iii) Handler Execution: If a handler is installed, controller jumps to the user-defined handler function, which runs in the interrupted context. The handler can perform actions like logging or cleanup.

iv) Resumption: After the handler returns, the kernel restores the saved context, resuming normal flow from the interruption point. This ensures non-termination if the handler doesn't call exit()

↳ Impact (): It introduces asynchrony, potentially causing race conditions. (e.g. interrupted system calls return EINTR). Handlers should be reentrant and avoid blocking calls.

This allow servers to handle SIGINT without terminating.

Simple Socket Program Demonstration using C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

// SIGINT Handler
void sigint_handler(int signo){
    printf("Received SIGINT (%d). Continuing server operation.\n",
                                                    signo);
}
int main(){

    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGINT, &sa, NULL) == -1){
        perror("sigaction failed");
        exit(1);
    }

int sockfd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in serv_addr;
bzero(&serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
serv_addr.sin_port = htons (8080);
bind(sockfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr));
listen(sockfd, 5);

while(1){
    int connfd = accept(sockfd, NULL, NULL);
    char buf[1024];
    ssize_t n = read(confd, buf, sizeof(buf));
    write(connfd, buf, n);
    close(connfd);
  }
close(sockfd);
return 0;

}
```

Code Explanation:

sigaction() installs sigint_handler, which logs SIGINT but returns control, allowing the server to resume accepting connections. SA_RESTART flag ensures accept() or read() restarts if interrupted, preventing EINTR errors. This keeps the server running despite SIGINT.

3b) Why are daemon processes preferred in network services? Outline the steps to daemonize a server process and show relevant code to detach from terminal and run in background.

↳

Daemon processes are background programs in Unix-like systems that run independently of user sessions, making them ideal for network services. Key reasons behind Daemon processes preferred in network services include:

i) Detachment from Terminal: They detach from controlling terminals, preventing termination when a user logs out or closes the session, ensuring continuous availability for services like HTTP or FTP.

ii) Resource Efficiency: No need for stdin/stdout/stderr, freeing resources; they log via syslog or files, reducing overhead in long-running servers.

iii) Signal and Process Management: Daemons handle signals (e.g. SIGHUP for reload) gracefully, support parentless execution (init/systemd adoption), and avoid zombie processes.

iv) Security and stability: Run with reduced previlges (e.g. non-root after setup), isolated from user interactions, minimizing risks in network-facing services.

v) Scalability: Fork child processes for client handling without cluttering the foreground, enabling persisten listening on the ports.

This ensures reliable, unattended operation for network services, handling multiple connections without user intervention.

Steps to Daemonize a Server Process

1. Fork and Exit Parent: Create a child process; parent exits to detach from shell.
2. Create New Session: Call setsid() in child to become session leader, detaching from terminal.
3. Fork Again (Optional): Fork child again; exit first child to ensure non-session leader
4. Change working directory: Set it to root (/) or safe path to avoid file system issues.
5. Close File Descriptors: Close stdin/stdout/stderr (0,1,2) or redirect to /dev/null.
6. Reset File Creation Mask: umask(0) for full permission control.
7. Handle Signals: Ignore or handle signals like SIGTTOU/SIGCHLD
8. Run in Background: Proceed with server logic (e.g. socket binding /listening)

Relevant Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <signal.h>
#include <netinet/in.h>
```

```
void daemonize() {
    pid_t pid = fork(); // Step 1: Fork and exit parent
    if (pid>0) exit(0); // Parent exits
    if (pid<0) exit(1);

    setsid();     // Step 2: New session.

    pid = fork(); // Step 3: Fork Again
    if (pid>0) exit(0);
    if (pid<0) exit(1);

    ~~umask(0);~~
     umask(0);   // Step 6 : Reset mask
     chdir("/"); // Step 4 : change directory
     close(0); close(1); close(2); // Step 5: close descriptors.
     open("/dev/null", ~~O_RDWR~~ O_RDWR); // Redirect stdin
     dup(0); dup(0);  // Redirect stdout/stderr.

      // Ignore Signals
      signal(SIGCHILD, SIG_IGN);

}
int main(){
    daemonize();    // Daemonize before server setup

// Simple TCP Server logic
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr = { .sin_family = AF_INET, .sin_port =
                htons(8080), ~~.sin_port~~ .sin_addr.s_addr = INADDR_ANY };
    bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(sockfd, 5);

    while(1) {
      int connfd = accept(sockfd, NULL, NULL);
      close(connfd);
    }
    return 0;
}       // This code detach the server, allowing it to run in the backgro-
                                    und without terminal.
```

Harsh Chaudhary Kalwar (Roll: 221715)

4a) Suppose you are building a chat server with multiple clients, Why is select() preferred over fork() in high load environments? write a code sketch to illustrate select()-based socket multiplexing.

↳

In a chat server with multiple clients, fork() creates a child process per client connection, duplicating the parent process's resources. This works for low loads, but it struggles in high load environments (having thousands of clients) due to :

i) Resource Overhead : Each forked process consumes significant amount of memory and CPU for context-switch.

ii) Scalability limits : Process creation/destruction is expensive. Systems have per-user process limits, causing failures under heavy concurrent connections.

iii) Inefficiency for idle clients: chat servers often have many idle connection; forking wastes resources on ~~to~~ inactive clients.

select() enable I/O multiplexing in a single process, monitoring multiple sockets for readiness and it's preferred in high load environment due to reasons mentioned below;

i) Efficiency : Single process handles all clients, reducing memory/CPU usage and context switches.

ii) Scalability: manages hundreds/thousands of ~~users~~ connections without per-client process

iii) Portability and Simplicity : Avoids zombie processes or signal handling issues in forked models, though limited by FD_SETSIZE (e.g., 1024 descriptors; mitigated by poll/epoll)

P.T.O.

Code sketch for select() based multiplexing

```
#include <sys/select.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

#define MAX_CLIENTS 100
#define port 8080

int main(){
    int server_fd, client_fds[MAX_CLIENTS], max_fd, activity, i,
                                                valread;

    fd_set readfds;
    struct sockaddr_in address = {.sin_family = AF_INET, .sinport =
                                htons(8080), .sin_addr.s_addr = INADDR_ANY};

    for(i=0; i<MAX_CLIENTS; i++) client_fds[i]=0;

    server_fd= socket(AF_INET, SOCK_STREAM, 0);
    bind(server_fd, (struct sockaddr*)&address, sizeof(address));
    listen(server_fd, 5);

    while (1){
        FD_ZERO(&readfds);
        FD_SET (server_fd, &readfds);
        max_fd= server_fd;

    for(i=0; i<MAX_CLIENTS; i++){
        if (client_fds[i]>0){
            FD_SET (client_fds[i], &readfds);
            if (client_fds[i] > max_fd = client_fds[i];
        }
    }
```

```
activity = select(max_fd+1, &readfds, NULL, NULL, NULL);

if (FD_ISSET(server_fd, &readfds)){
    int new_socket = accept(server_fd, NULL, NULL);

    for(i=0; i<MAX_CLIENTS; i++){
        if (client_fds[i]==0){
            client_fds[i] = new_socket;
            break;
        }
    }
}

for(i=0; i<MAX_CLIENTS; i++){
    int sd = client_fds[i];
    if (sd>0 && FD_ISSET(sd, &readfds)){
        char buffer[1024] = {0};
        valread = read(sd, buffer, sizeof(buffer));
        if(valread==0){
            close(sd)
            client_fds[i] = 0;
        }else{
            for(int j=0; j<MAX_CLIENTS; j++){
                if (client_fds[j]>0 && client_fds[j]!=sd){
                    write(client_fds[j], buffer, valread);
                }
            }
        }
    }
}
return 0;
}
```

// This sketch uses select() to monitor the server socket and clients in one loop, handling connections and messages efficiently without forking.

Harsh Chaudhary Kalwar; 221715

4b) Blocking vs Non-blocking I/O- explain their conceptual differences with socket system call examples. When would non-blocking I/O be detrimental in network applications?

L)

Blocking I/O	Non-blocking I/O
→ a socket system halts the process until the process completes or an error occurs.	→ It allows the system call to return immediately even if the operation isn't completed.
→ The process waits for data to arrive, a connection to establish or data to be sent, blocking further execution.	→ If no data is available or the operation can not proceed, the call returns an error, letting the process continue other task.
→ The call returns only when the requested action is completed or fails.	→ The process must handle partial results or retry operations, enabling concurrent handling of multiple sockets in a single thread.
→ Suitable for simple applications with few connections, where waiting is acceptable.	→ Ideal for high-load servers managing many client efficiently without seperate threads/processes.

Blocking I/O socket system call example

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

int main(){
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr = {.sin_family = AF_INET, .sin_port =
                htons(8080), .sin_addr.s_addr = INADDR_ANY};
```

```
int connfd = accept(sockfd, NULL, NULL); // Blocks until client connect
char buffer[1024];
int n = read(connfd, buffer, sizeof(buffer)); //Blocks until data is arrived
write(connfd, buffer, n); //Blocks until data is sent.
close(connfd);
close(sockfd);
return 0;
}
```

accept() and read() block until a client connects or data is received, respectively. The process waits simplifying the code but stalling for inactive clients.

Non-blocking I/O example

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <errno.h>

int main(){
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    fcntl(sockfd, F_SETFL, O_NONBLOCK); //set socket to non-blocking
    struct sockaddr_in serv_addr = {.sin_family = AF_INET, .sin_port =
                    htons(8080), .sin_addr.s_addr = INADDR_ANY };
    bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(sockfd, 5);

    int connfd = accept(sockfd, NULL, NULL);
    if (connfd == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
        printf("No client yet, continue.....\n");

    }
    char buffer[1024];
    int n = read(connfd, buffer, sizeof(buffer));
    if (n == -1 && errno == EAGAIN){
        printf("No data yet, retry later");

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

fcntl() sets the socket to non-blocking, accept() and read() return immediately with EAGAIN if no client/data is available, allowing the process to perform other tasks.

Non-blocking I/O, while efficient for scalability, can be detrimental in certain network applications:

1. Increased Complexity
   ↳ non-blocking codes require polling loops or event-driven frameworks, complicating design.

2. CPU overhead in polling
   ↳ constantly polling the sockets can consume CPU, especially if few sockets are active.

3. Latency in Small-scale Apps
   ↳ It may introduce overhead from checking readiness, which can outweigh benefits in application with few connections where blocking delays are tolerable.

4. Error-Prone Partial operations
   ↳ Handling partial reads/writes requires careful buffering and state management.

5. Resource Starvation
   ↳ In poorly-designed non-blocking systems, busy clients may monopolize the event-loop, starving others, unlike blocking I/O where each connection gets dedicated attention.

Example:

In a low-latency, single-connection file transfer app, non-blocking I/O could complicate the code with retries and buffers, while blocking I/O ensures straightforward, reliable data transfer without polling overhead.

Haroh chaudhary Kalwar (221715)

5a) Your server quickly needs to reuse a port after crashing. How does setsockopt() with SO_REUSEADDR help? Write a short TCP socket server snippet with this option.

↳ When a TCP socket server crashes, its socket may remain in the TIME_WAIT state (typically 2x maximum segment lifetime), tying up the port. This prevents immediate reuse, causing "Address Already in Use" errors (EADDRINUSE) on restart. The SO_REUSEADDR socket option, set via setsockopt(), allows the server to bind to the same port even if it's in TIME_WAIT.

Benifits:

i) Quick Recovery
ii) Avoiding Port Conflicts
iii) Use case: Essential for servers that restart frequently or operate in environments with frequent failures, ensuring clients can reconnect without delay.

TCP socket server snippet with SO_REUSEADDR

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

int main(){
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0){
        perror("Socket connection failed");
        return 1;

    }

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

```
struct sockaddr_in serv_addr = { .sin_family = AF_INET, .sin_port = htons(8080),
                                 .sin_addr.s_addr = INADDR_ANY };
if (bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
    perror("Bind failed");
    return 1;
}

~~listen(sockfd, 5)~~
listen(sockfd, 5);
printf("Server listening on port 8080");

while(1){
    int connfd = accept(sockfd, NULL, NULL);
    if (connfd >= 0) {
        char buffer[1024] = {0};
        read(connfd, buffer, sizeof(buffer));
        write(connfd, buffer, strlen(buffer)); // Echo back
        close(connfd);
    }
}
close(sockfd);
return 0;
}
```

Snippet Explanation

↳ setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
enables port reuse before binding opt = 1 activates the option.

↳ If the server crashes and restarts, it can bind to port 8080 immediately.
even if old connections are in TIME_WAIT.

Harsh Chaudhary Kalwar (221715)

5b) Discuss how broadcasting and multicasting differ in socket-level implementation. Demonstrate how ~~each~~ to send a multicast UDP packet from a sender program in C.

↳

~~Broadcar~~

Aspect	Broadcasting	multicasting
Scope and Targeting	Sends data to all hosts on a local subnet (e.g., using addresses like 225.255.255.255 or subnet-specific like 192.168.1.255). It's limited to the local network and floods all devices, regardless of interest.	Sends data to a specific group of hosts that have explicitly joined the multicast group. It uses class D IP addresses and can span routers if TTL is set appropriately, allowing selective delivery across networks.
Socket creation and Options	Uses a UDP socket. Require setsockopt() with SO_BROADCAST to enable broadcasting. No group management; simply sendto() to the broadcast address.	Also used UDP sockets. Sender doesn't join groups but uses setsockopt() for options like IP_MULTICAST_TTL (to control hop count) and optionally IP_MULTICAST_LOOP. Receivers must join via IP_ADD_MEMBERSHIP. Multicast relies on network infrastructure for efficient delivery
Efficiency and Overhead	Inefficient in large networks as it duplicates packets to all hosts, causing unnecessary traffic and processing on uninterested devices. No receiver opt-in; it's all-or-nothing per subnet.	More efficient; routers replicate packets only to subscribed paths, reducing bandwidth. Supports dynamic group membership, ideal for applications like video streaming
Implementation challenges	Simple but prone to loops/storms if not firewalled; limited to LAN.	Complex due to router configuration needs; sender code is straightforward, but receivers handle joins/drops

Demonstrating: sending a multicast UDP Packet in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(){
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0){
        perror("Socket creation failed");
        return 1;
    }

    int ttl=1;
    if (setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl))<0)
        {
        perror("setsockopt failed");
        close(sockfd);
        return 1;
    }

    struct sockaddr_in mcast_addr;
    memset(&mcast_addr, 0, sizeof(mcast_addr));
    mcast_addr.sin_family = AF_INET;
    mcast_addr.sin_port = htons(12345);
    inet_pton(AF_INET, "239.0.0.1", &mcast_addr.sin_addr);

    const char *msg= "Hello, multicasting group!";
    if (sendto(sockfd, msg, strlen(msg), 0, (struct sockaddr *)&mcast_addr)<0){
        &mcastaddr, sizeof(mcastaddr))<0)
        perror("sendto failed");
        close(sockfd);
        return 1;
    } printf("multicast packet sent successfully.\n");
    close(sockfd); return 0;
}
```

Harsh Chaudhary Kalwar. (22f715)

6a) Winsock Programming requires initialization and cleanup. Explain the lifecycle of a Winsock application and implement a basic TCP echo server in winsock with proper structure.

↳

The lifecycle of a Winsock Application ensures proper resource management, version compatibility, and error handling. Key phases:

1. Initialization (WSAStartup): Call WSAStartup() to load the winsock DLL and specify the required version. This returns a WSADATA structure with version info. Failure aborts the app. This step allocates resources and sets up the environment.
2. Socket Creation and Setup: Creates sockets using socket(). For servers: bind to an address/port with bind(), set listening queue with listen(). For clients: connect(). This phase configures the socket for communication.
3. Connection Handling and Data Transfer: Servers use accept() to get client connection, then send()/recv() for data exchange. Clients use send()/recv() post-connection. This is the core operational phase, often in loops for multiple clients.
4. Error Handling and Shutdown: Check return values for errors (use (WSAGetLastError()). Gracefully close connections with shutdown() and closesocket() to release socket descriptors.
5. Cleanup (WSACleanup): Call WSACleanup() to unload the DLL and free resources. Must match each WSAStartup() call. Omitting this can leak resources or cause issues in subsequent runs.

This lifecycle prevents DLL conflicts, ensures thread-safety in multi-threaded apps, and handles Window-specific quirks like overlapped I/O.

Basic TCP echo Server Implementation in Winsock

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iostream>
#include <cstring>

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

```
int main(){
    WSADATA wsaData;
    int result = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (result != 0){
        std:: cerr << "WSA startup failed: " << result << std::endl;
        return 1;
    }

    SOCKET listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenSocket == INVALID_SOCKET){
        std::cerr << "Socket Creation failed: " << WSAGetLastError() <<
                                                    std::endl;
        WSACleanup();
        return 1;
    }

    sockaddr_in serverAddr;
    std::memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(8080);

    if (bind(listenSocket, (sockaddr*)&serverAddr, sizeof(serverAddr))
            == SOCKET_ERROR){
        std::cerr << "Bind failed: " << WSAGetLastError() << std::endl;
        closesocket(listenSocket);
        WSACleanup();
        return 1;
    }
    if (listen(listenSocket, SOMAXCONN) == SOCKET_ERROR){
        std::cerr << "Listen failed: " << WSAGetLastError() << std::endl;
        closesocket(listenSocket);
        WSACleanup();
        return 1;
    }
    Std:: Cout << "Server listening on port 8080..." << std::endl;
```

```
SOCKET clientsocket = accept(listenSocket, NULL, NULL);
if (clientSocket == INALID_SOCKET){
  std::cerr<<"Accept failed : "<< WSAGetLastError() << std::endl;
  closesocket(listenSocket);
  WSACleanup();
  return 1;
 }

char butter[1024];
while(true){
   int bytesReceived = recv(clientsocket, butter, sizeof(butter), 0);
   if (bytesReceived>0){
      send(clientSocket, butter, bytesReceived, 0);
    }
   elseif
   else if (bytesReceived ==0){
      std::cout<<" client disconnected."<<std::endl;
      break;
    }
    else {
     std::cerr<<" Recv failed :"<< WSAGetLastError()<<std::endl;
     break;
    }
  }
 closesocket(clientSocket);
 closesocket
closesocket(listenSocket);
WSACleanup();
 return 0;

}
```

Harsh chaudhary Kalwar (221715)

6b) How does error handling differ between unix and ~~winsocket~~ winsock socket APIs? Show code fragments of error handling using WSAGetLastError() and contrast with perror() or errno.

↳ Error handling in unix and winsock socket APIs differs due to their underlying systems, error reporting mechanisms, and API design.

Aspect	unix	winsock
Error Reporting Mechanism	Socket functions return -1 on failure and set the global errno variable to a specific error code. The perror() function or strerror (errno) translates errno to human-readable messages. This is consistent across unix system calls, leveraging a kernal-level error framework.	Socket functions return SOCKET_ERROR(-1) or INVALID_SOCKET. Errors are retrieved via WSAGetLastError(), which returns a Windows-specific error code. Unlike errno, WSAGetLastError() is specific to winsock and not shared with other system calls. Use formatMessage() for ~~defined~~ detailed error strings
Error code Namespace	Error codes are defined in <<errno.h>>. They are simple integers, shared across system calls, and standardized in POSIX.	Error codes start at 10000 to avoid overlap with windows system errors. They are windows-specific and not POSIX-compatible.
Thread Safety	errno is thread-specific in modern systems, ensuring safety in multi-threaded apps.	WSAGetLastError() is also thread-safe, returning the last error for the calling thread's winsock operation.
Initialization Dependency	No explicit initialization; errno is always available as part of the C library.	Requires WSAStartup() to initialize the Winsock DLL. Errors before initialization may not set WSAGetLastError() correctly.
Portability and Usage	perror() or strerror() is portable across POSIX systems, simpler for quick debugging.	WSAGetLastError() requires window-specific handling, and formatMessage() is verbose ☹ for detailed messages, reducing portability.

Code Fragments for Error Handling (Unix)

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1){
        perror("socket creation bailed");
        fprintf(stderr, "errno: %d (%s)\n", errno, strerror(errno));
        exit(1);
    }

    struct sockaddr_in addr = {.sin_family = AF_INET, .sin_port = htons(2020)
                               , .sin_addr.s_addr = INADDR_ANY};
    if (bind(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1){
        perror("Bind bailed");
        exit(1);
    }
    close(sockfd);
    return 0;
}
```

Explanation: perror() uses errno to print a descriptive error messages to stderr. ~~stde~~ strerror(errno) can be used for custom formatting. Errors like EADDRINUSE (98) are checked after each system call.

WinSock Code

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

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

↳

```
int main(){
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2,2), &wsaData)!=0) {
      fprintf(stderr, "WSA startup failed : %d \n", WSAGetLastError());
      return 1;
    }

    SOCKET sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sockfd==INVALID_SOCKET){
      fprintf(stderr, "socket creation failed : %d \n", WSAGetLastError());
      WSACleanup();
      return 1;
    }

    struct sockaddr_in addr={.sin_family=AF_INET, .sin_port =
                htons(8080), .sin_addr.s_addr =INADDR_ANY};
    if (bind(sockfd, (struct sockaddr*)&addr, sizeof(addr)) ==
                                    SOCKET_ERROR){
      fprintf (stderr, "Bind Failed : %d \n", WSAGetLastError());
      closesocket (sockfd);
      WSACleanup();
      return 1;
    }

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

Explanation: WSAGetLastError() retrieves the winsock-specific error code after a failure. No direct equivalent to perror() exists, so fprintf() is used. For detailed messages, FormatMessage() can be added.

7a) A developer uses select() with winsock but experiences poor performance. What alternatives exist in Winsock for asynchronous I/O ? Explain WSAEventSelect() with an example.

↳

Alternatives to select() in winsock for Asynchronous I/O are listed below:

i) WSAEventSelect(): Associate sockets with windows event objects, allowing the application to wait on multiple events.

ii) WSAAsyncSelect(): Ties socket events to windows messages, posting notifications to a window's message queue.

iii) Overlapped I/O (completion routines or events): Uses WSASend(), WSARecv(), etc. with overlapped structures.

iv) I/O Completion Ports (IOCP): Windows most scalable model, associating sockets with completion port.

v) Registered I/O (RIO): A newer, high-performance API for low-latency I/O with completion queues. Less common due to complexity and platform requirements.

WSAEventSelect()

It associates a socket with a windows event object and specifies network events to monitor. When an event occurs, the event objects is signaled, and the application can use WSAWaitForMultipleEvents() or WSAWaitForMultipleObjects() to detect it. Key points are below mentioned;

i) Event-Driven: Non-blocking; the application waits on events rather than polling descriptors.

ii) Scalability: more efficient than select() as it avoids linear scans, supporting more sockets.

iii) Work flow: create an event with WSACreateEvent(), associates it with a socket via WSAEventSelect(), waits for events, and checks specific events with WSAEnumNetworkEvents().

iv) Use Case: Suitable for servers handling moderate concurrent connection with better performance than select().

Example:

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iostream>

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

int main(){

    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2,2),&wsaData) != 0){
        std:: cerr <<"WSAStartup failed:" << WSAGetLastError() << std:: endl;
        return 1;
    }

    SOCKET listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenSocket == Invalid Socket) {
        std:: cerr << "Socket Creation failed:" << WSAGetLastError() << std:: endl;
        WSACleanup();
        return 1;
    }

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

    if (bind(listenSocket, (sockaddr *) &serverAddr, sizeof(serverAddr)) ==
                                        SOCKET_ERROR) {
        std:: cerr << "Bind failed :" << WSAGetLastError() << std:: endl;
        closesocket(listenSocket);
        WSACleanup()
        return 1;
    }
    if( listen(listenSocket, SOMAXCONN) == SOCKET_ERROR){
        std:: cerr << "Listen failed :" << WSAGetLastError() << std:: endl;
        closesocket(listenSocket);
        WSACleanup();
        return 1;
    }
```

```
WSAEvent
WSAEVENT listenEvent = WSACreateEvent();
if (listenEvent == WSA_INVALID_EVENT){
  std::cerr << "Event Creation failed: " << WSAGetLastError() << std::endl;
  closesocket(listenSocket);
  WSACleanup();
  return 1;
}

if (WSAEventSelect(listenSocket, listenEvent, FD_ACCEPT) == SOCKET_ERROR){
  std::cerr << "WSAEventSelect failed " << WSAGetLastError() << std::endl;
  WSACloseEvent(listenEvent);
  closeSocket(listenSocket);
  WSACleanup()
  return 1;
}

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

WSAEVENT events[2] = {listenEvent};
SOCKET clientSocket = INVALID_SOCKET;
WSAEvent clientEvent = WSA_INVALID_EVENT;

while(true){
  DWORD index = WSAWaitForMultipleEvents(clientSocket == INVALID_
      SOCKET ? 1:2, events, FALSE, WSA_INFINITE, FALSE);
  if (index == WSA_WAIT_FAILED){
    std::cerr << "wait failed" << WSAGetLastError() << std::endl;
    break;
  }
  WSAResetEvent(events[index]);

  WSANETWORKEVENTS netEvents;
  SOCKET targetSocket = (index == 0) ? listenSocket : clientSocket;
  if (WSAEnumNetworkEvents(targetSocket, events[index], &netEvents)
      == SOCKET_ERROR){
    std::cerr << "Enum Events failed: " << WSAGetLastError() << std::endl;
    break;
  }
```

```
if (index == 0 && (netEvents.lNetworkEvents & FD_ACCEPT)) {
clientSocket = accept (listenSocket, NULL, NULL);
if (clientSocket == INVALID_SOCKET) {
    std::cerr << "Accept failed : " << WSAGetLastError() << std::endl;
    continue;
}
clientEvent = WSACreateEvent();
WSAEventSelect (clientSocket, clientEvent, FD_READ | FD_CLOSE);
events[i] = clientEvent;
std::cout << "Client Connected: " << std::endl;
}

if (index == 1 && (netEvents.lNetworkEvents & FD_READ)) {
    char buffer[1024];
    int bytes = recv (clientSocket, buffer, sizeof(buffer), 0);
    if (bytes > 0) {
        send (clientSocket, buffer, bytes, 0);
    }
}

if (index == 1 && (netEvents.lNetworkEvents & FD_CLOSE)) {
    std::cout << "Client Disconnected." << std::endl;
    closeSocket (clientSocket);
    WSACloseEvent (clientEvent);
    clientSocket = INVALID_SOCKET;
    clientEvent = WSA_INVALID_EVENT;
}

if (clientSocket != INVALID_SOCKET) closeSocket (clientSocket);
if (clientEvent != WSA_INVALID_EVENT) WSACloseEvent (clientEvent);
closeSocket (listenSocket);
WSACloseEvent (listenEvent);
WSACleanup();
return 0;
}
```

7b) Why is multithreading important in network applications? Show how to implement a thread based TCP server in winsock that handles each client in a seperate thread.

↳

Multithreading is critical in network applications for handling multiple clients efficiently, improving performance, and ensuring responsiveness. Key reasons behind are mentioned below:

i) Concurrency: Multithreading allows a server to handle multiple client connections simultaneously without blocking.

ii) Scalability: Utilize single threaded models, threads leverage CPU cores, improving throughput for high-load scenarios like chat servers or web servers with many concurrent users.

iii) Responsiveness: Dedicates threads per client prevents slow clients from blocking others, ensuring timely response in real-time applications.

iv) Simplified logic: Each thread can use blocking I/O, simplifying code compared to non-blocking or event-driven models, as each thread handles on client's lifecycle independently.

v) Resource Utilization: Threads share the process's memory, making them lighter than processes, though still heavier than event-based models like IOCP.

Implementation of thread-based TCP server in WinSock

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iostream>
#include <thread>

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

→ P.T.O. →

```
void handleClient(SOCKET clientSocket){
    char buffer[1024];
    while(true){
        int bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);
        if(bytesReceived <= 0){
            std::cout << "Client disconnected or error:" << WSAGetLastError()
                                                    << std::endl;
            closeSocket(clientSocket);
            break;
        }
        if(send(clientSocket, buffer, bytesReceived, 0) == SOCKET_ERROR){
            std::cerr << "send failed: " << WSAGetLastError() << std::endl;
            closeSocket(clientSocket);
            break;
        }
    }
}
int main(){
    WSADATA wsaData;
    if(WSAStartup(MAKEWORD(2,2), &wsaData) != 0){
        std::cerr << "WSAStartup failed: " << WSAGetLastError() <<
                    std::endl;
        return 1;
    }
    SOCKET listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if(listenSocket == INVALID_SOCKET){
        std::cerr << "Socket Creation failed:" << WSAGetLastError() << std::endl;
        WSACleanup();
        return 1;
    }
```

```
sockaddr_in serverAddr = {. sin_family = AF_INET, . sin_port =
        htons(8080), .sin_addr.s_addr = INADDR_ANY};
if (bind (listenSocket, (sockaddr *) &serverAddr, sizeof (serverAddr))
                == SOCKET_ERROR) {

    std:: Cerr << "Bind failed : "<<WSAGetLastError ()<<std:: endl;
    closeSocket(listenSocket);
    WSACleanup();
    return 1;
    }

    if (listen(listenSocket, SOMAXCONN) == SOCKET_ERROR){
      std::cerr<<" listenFailed: "<< WSAGetLastError()<<std:: endl;
      closeSocket (listenSocket);
       WSACleanup();
       return 1;
     }

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

     while (true){
        SOCKET clientSocket = accept (listenSocket, NULL, NULL);
        if (clientSocket = INVALID_SOCKET) {
          std:: cerr << "Accept failed : " <<WSAGetLastError()<<std:: end);
          continue;
        }
        std:: cout<<"New Client connected : " << std:: endl;
        std:: thread clientThread (handleClient, clientSocket);
        clientThread. detach();

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

Harsh chaudhary Kalwar (221715)

8a) TLS/SSL provides security over sockets. Explain the role of certificates and handshake in securing communication. Outline the steps to integrate openssl into a Unix TCP server.

L>

Role of Certificates and Handshake in Securing Communication are listed below;

i) Certificates:

L> These are digital documents issued by Certificate Authorities containing a public key, identity info, and a CA signature. Their role is to authenticate endpoints: the server presents its certificate during handshake; the client verifies it against trusted CAs to confirm the server's identity. Certificates enable asymmetric encryption for key exchange and symmetric encryption for data. Self-signed certificates can be used for testing but lack trust in production.

ii) Hand shake

L> Negotiates encryption algorithm & keys before data transfer.

steps:

a) Client Hello – client sends supported cipher suites, TLS version

b) Server Hello – server selects cipher suite and sends its certificate

c) Certificate Verification – client checks server's certificate with CA

d) Key exchange – session key generated (RSA etc.)

e) Finished messages – both sides confirm secure channel is established.

Integrating OpenSSL in a Unix TCP server :

1. Install & Include OpenSSL:

L>
```
#include <openssl/ssl.h>
#include <openssl/err.h>
```

2. Initialize Openssl library:

```
SSL_library_init();
SSL_load_error_strings();
OpenSSL_add_all_algorithms();
```

3. Create SSL Context

```
SSL_CTX *ctx = SSL_CTX_new(TLS_server_method());
```

4. Load certificate & Private key.

```
SSL_CTX_use_certificate_file(ctx, "server.crt", SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "server.key", SSL_FILETYPE_PEM);
```

5. Create TCP Socket & Accept Connection (normal socket(), bind(), listen(), accept()).

6. Wrap Socket with SSL:

```
SSL *ssl = SSL_new(ctx);
SSL_set_fd(ssl, client_fd);
SSL_accept(ssl);
```

7. Send / Receive Data

```
SSL_write(ssl, buffer, len);
SSL_read(ssl, buffer, sizeof(buffer));
```

8. Shutdown & Clean up

```
SSL_shutdown(ssl);
SSL_free(ssl);
SSL_CTX_free(ctx);
EVP_cleanup();
```

8b) Compare WebSocket and gRPC with respect to real-time bidirectional communication. Design a simple use case where gRPC is preferable over traditional REST or socket programming.

↳

Feature	WebSocket	gRPC
Protocol	works over TCP (usually upgraded from HTTP)	works over HTTP/2
Data Format	Text (UTF-8) or Binary (Frames)	Binary (ProtocolBuffers - Protobuf)
Communication	Full-duplex persistent connection	Supports bidirectional streaming (client & server)
Performance	Low latency, but text data can be larger	very fast (compact protobuf + multiplexing)
Interoperability	Easy for browsers (JS support)	multi-language API ~~subs~~ stubs, better for backend systems
Use case	Live chats, gaming, notifications	High-performance microservices, IoT control, data streaming
Type safety	No built-in schema enforcement	Strongly type via protobuf schema

Use case where gRPC is preferrable:

Let's take a scenario of a real-time sensor monitoring system in an industrial plant where:

↳ hundreds of IoT devices send temperature, pressure, and vibration data continuously to a central processing service.

↳ processing service streams alerts back to devices when readings exceed thresholds.

In this case gRPC is more preferrable than WebSocket due to:

1. Binary & Compact: Protobuf reduces bandwidth usage
2. strong Typing: Schema ensures devices and servers agree on data format
3. Streaming RPC: Server & client can send continuous data in parallel without re-opening connections.
4. Multiplexing: Multiple streams over single HTTP/2 connection.
5. Multi-language support 🌍: Devices → C++/Python, Server → Go/Java.

9) Software-Defined Networking (SDN) centralizes control but introduces new challenges. Explain how OpenFlow changes traditional network flow behavior and why it matters in programmable networks.

↳

a Software-Defined Networking (SDN) seperates the control plane (decision-making) from the data plane (packet forwarding). Control is centralized in an SDN controller, enabling programmable and dynamic network management.

In Traditional networks:

↳ Each switch/router makes independent forwarding decisions based on its own routing tables.

↳ Control logic is embedded in each device (vendor-specific firmware)

- With OpenFlow:

1. Flow table in Switches – Contains match fields, actions, and stats.
2. Controller Updates Flows – Switches query the controller when no matching rule exists.
3. Centralized Updates – controllers can push new flow rules dynamically to all switches.
4. Programmable behavior – Flows can be altered for QoS, load balancing, firewalling, or routing changes without touching hardware.

In programmable network it matters due to following reasons:

1. Centralized Intelligence: Global network view for optimal routing & policies.
2. Rapid Changes: Modify traffic flows in seconds.
3. Vendor Neutrality: Works across different hardware via OpenFlow API.
4. Advanced use cases: Dynamic security rules, traffic engineering, real-time failover.
5. Innovation: Researchers and operators can test new protocols without replacing physical devices.

Harsh Chaudhary Kalwar (22175)

10) You're asked to troubleshoot network performance on server. Choose any five of these tools and briefly explain their use: ping, netstat, iperf, telnet, ifconfig, traceroute, nmap.

↳

Network troubleshooting tools & their uses are explained below:

1. Ping
   - ↳ It tests basic connectivity between two hosts
   - ↳ Sends ICMP Echo Request packets and waits for echo reply.
   - ↳ Key Output: Round Trip Time (RTT), packet loss percentage
   - ↳ Detects connectivity problems, high latency, or packet drops between server and remote host

2. netstat
   - ↳ Displays active TCP/UDP connections, listening ports, routing tables and interface statistics.
   - ↳ Key output: local/remote IPs, ports, protocol type, connection status (ESTABLISHED, LISTENING, etc.)
   - ↳ Use in troubleshooting:
     - find which services are running on which ports
     - Detect suspicious or unauthorized connections

3. iperf
   - ↳ measures network throughput and bandwidth between two hosts.
   - ↳ Client-server model; sends TCP/UDP data streams and measures transfer rate.
   - ↳ Use in troubleshooting:
     - Identify bandwidth bottlenecks
     - Test maximum achievable speed between two points

4. traceroute
   - ↳ Displays the route and intermediates hops taken by packets to reach a destination
   - ↳ Sends packets with increasing TTL (Time-To-Live) and records each hop's response time.

→

↳ Use in troubleshooting:
Pinpoint where delays and or packet loss occur
Detect misconfigured routers or long routing paths.

8. nmap
↳ Scans network hosts and discovers open ports and services
↳ ~~list of~~ key output: list of detected hosts, open ports, service versions, OS details
↳ Use in troubleshooting:
security auditing and vulnerability detection
Inventory of services running on a server.