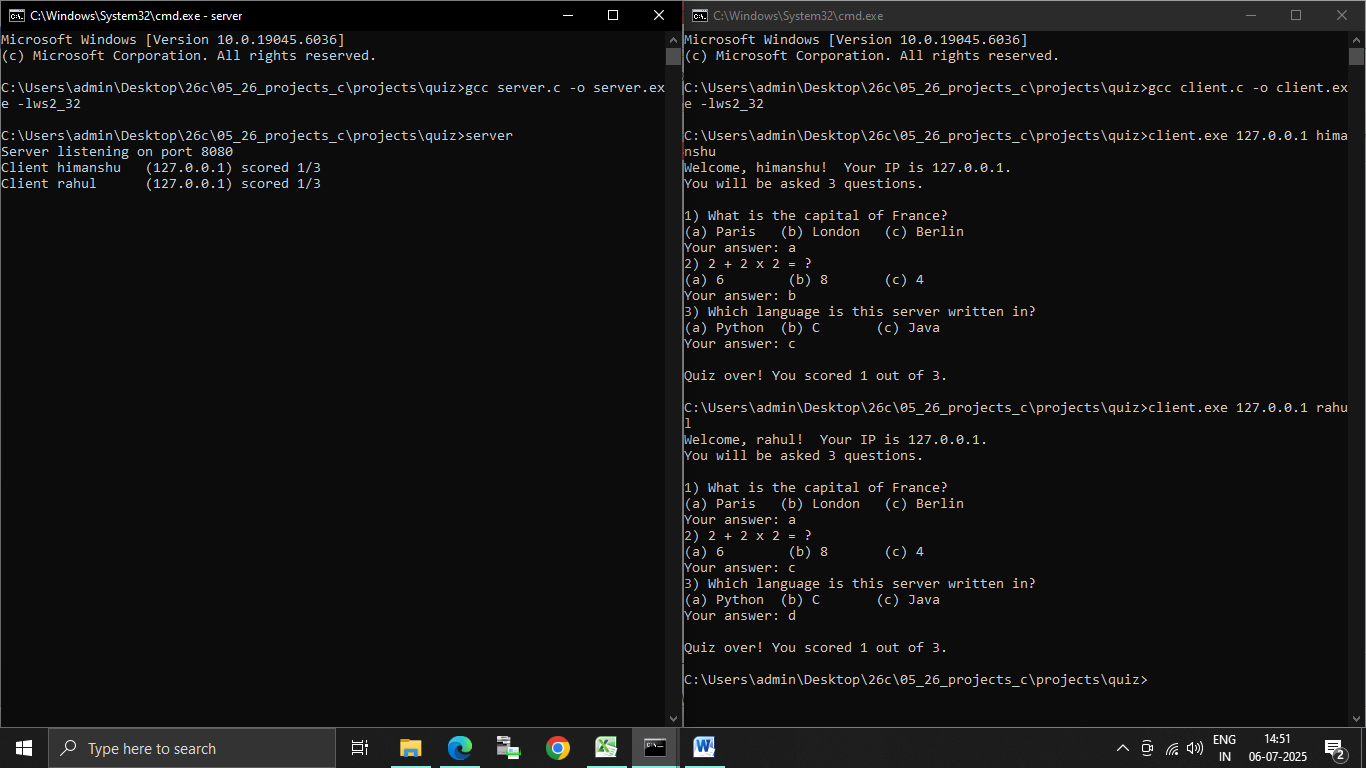
**NETWORK PROGRAMMNG IN C**

**SOCKET PROGRAMMING IN C**

**PROJECT QUIZ SYSTEM IN C (MULLTIPLE CLIENT C))**



**SOURCE CODE:**

**// client.c**

**// Compile with: gcc client.c -o client.exe -lws2\_32**

**#include <winsock2.h>**

**#include <windows.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#pragma comment(lib, "Ws2\_32.lib")**

**#define SERVER\_PORT 8080**

**#define MAXLINE 1024**

**#define QCOUNT 3**

**int main(int argc, char \*argv[]) {**

**if (argc != 3) {**

**printf("Usage: %s <server-ip> <login-id>\n", argv[0]);**

**return 1;**

**}**

**const char \*server\_ip = argv[1];**

**const char \*login = argv[2];**

**WSADATA wsa;**

**SOCKET sock;**

**struct sockaddr\_in servAddr;**

**char buf[MAXLINE];**

**int n;**

**// 1) Initialize Winsock**

**if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {**

**fprintf(stderr, "WSAStartup failed\n");**

**return 1;**

**}**

**// 2) Create socket**

**sock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);**

**if (sock == INVALID\_SOCKET) {**

**perror("socket");**

**WSACleanup();**

**return 1;**

**}**

**// 3) Fill server address**

**servAddr.sin\_family = AF\_INET;**

**servAddr.sin\_addr.s\_addr = inet\_addr(server\_ip);**

**servAddr.sin\_port = htons(SERVER\_PORT);**

**// 4) Connect**

**if (connect(sock,**

**(SOCKADDR\*)&servAddr,**

**sizeof(servAddr)) == SOCKET\_ERROR)**

**{**

**perror("connect");**

**closesocket(sock);**

**WSACleanup();**

**return 1;**

**}**

**// 5) Send login ID (no newline needed)**

**send(sock, login, (int)strlen(login), 0);**

**// 6) Read & print welcome**

**n = recv(sock, buf, MAXLINE-1, 0);**

**if (n <= 0) goto CLEANUP;**

**buf[n] = '\0';**

**printf("%s", buf);**

**// 7) Quiz loop**

**for (int i = 0; i < QCOUNT; ++i) {**

**// Read question**

**n = recv(sock, buf, MAXLINE-1, 0);**

**if (n <= 0) break;**

**buf[n] = '\0';**

**printf("%s", buf);**

**// Prompt and flush**

**printf("Your answer: ");**

**fflush(stdout);**

**// Read user input**

**if (!fgets(buf, sizeof(buf), stdin)) break;**

**send(sock, buf, (int)strlen(buf), 0);**

**}**

**// 8) Read & print result**

**n = recv(sock, buf, MAXLINE-1, 0);**

**if (n > 0) {**

**buf[n] = '\0';**

**printf("%s", buf);**

**}**

**CLEANUP:**

**closesocket(sock);**

**WSACleanup();**

**return 0;**

**}**

**// server.c**

**// Compile with: gcc server.c -o server.exe -lws2\_32**

**// (or MSVC: cl /EHsc server.c ws2\_32.lib)**

**#include <winsock2.h>**

**#include <windows.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <ctype.h>**

**#pragma comment(lib, "Ws2\_32.lib")**

**#define SERVER\_PORT 8080**

**#define MAXLINE 1024**

**#define QCOUNT 3**

**// Quiz questions and correct answers**

**const char \*questions[QCOUNT] = {**

**"1) What is the capital of France?\n(a) Paris (b) London (c) Berlin\n",**

**"2) 2 + 2 x 2 = ?\n(a) 6 (b) 8 (c) 4\n",**

**"3) Which language is this server written in?\n(a) Python (b) C (c) Java\n"**

**};**

**const char answers[QCOUNT] = { 'a', 'a', 'b' };**

**// Per-client data struct**

**typedef struct {**

**SOCKET sock;**

**char ip[16];**

**} CLIENT\_INFO;**

**// Thread function: runs one quiz session**

**DWORD WINAPI clientHandler(LPVOID param) {**

**CLIENT\_INFO \*ci = (CLIENT\_INFO\*)param;**

**SOCKET s = ci->sock;**

**char buf[MAXLINE];**

**int n, score = 0;**

**// 1) Read login ID**

**n = recv(s, buf, MAXLINE-1, 0);**

**if (n <= 0) goto CLEANUP;**

**buf[n] = '\0';**

**char login[64];**

**strncpy(login, buf, sizeof(login));**

**// 2) Send welcome message**

**snprintf(buf, sizeof(buf),**

**"Welcome, %s! Your IP is %s.\n"**

**"You will be asked %d questions.\n\n",**

**login, ci->ip, QCOUNT);**

**send(s, buf, (int)strlen(buf), 0);**

**// 3) Quiz loop**

**for (int i = 0; i < QCOUNT; ++i) {**

**send(s, questions[i], (int)strlen(questions[i]), 0);**

**n = recv(s, buf, MAXLINE-1, 0);**

**if (n <= 0) break;**

**if (tolower(buf[0]) == answers[i]) score++;**

**}**

**// 4) Send result**

**snprintf(buf, sizeof(buf),**

**"\nQuiz over! You scored %d out of %d.\n",**

**score, QCOUNT);**

**send(s, buf, (int)strlen(buf), 0);**

**// 5) Log on server console**

**printf("Client %-10s (%s) scored %d/%d\n",**

**login, ci->ip, score, QCOUNT);**

**CLEANUP:**

**closesocket(s);**

**free(ci);**

**return 0;**

**}**

**int main(void) {**

**WSADATA wsa;**

**SOCKET listenSock;**

**struct sockaddr\_in servAddr, cliAddr;**

**int cliLen = sizeof(cliAddr);**

**// 1) Start Winsock**

**if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {**

**fprintf(stderr, "WSAStartup failed\n");**

**return 1;**

**}**

**// 2) Create listening socket**

**listenSock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);**

**if (listenSock == INVALID\_SOCKET) {**

**perror("socket");**

**WSACleanup();**

**return 1;**

**}**

**// 3) Bind to all interfaces on SERVER\_PORT**

**servAddr.sin\_family = AF\_INET;**

**servAddr.sin\_addr.s\_addr = INADDR\_ANY;**

**servAddr.sin\_port = htons(SERVER\_PORT);**

**if (bind(listenSock, (SOCKADDR\*)&servAddr, sizeof(servAddr)) == SOCKET\_ERROR) {**

**perror("bind");**

**closesocket(listenSock);**

**WSACleanup();**

**return 1;**

**}**

**// 4) Listen**

**if (listen(listenSock, 5) == SOCKET\_ERROR) {**

**perror("listen");**

**closesocket(listenSock);**

**WSACleanup();**

**return 1;**

**}**

**printf("Server listening on port %d\n", SERVER\_PORT);**

**// 5) Accept loop**

**while (1) {**

**SOCKET clientSock = accept(**

**listenSock,**

**(SOCKADDR\*)&cliAddr,**

**&cliLen**

**);**

**if (clientSock == INVALID\_SOCKET) {**

**perror("accept");**

**continue;**

**}**

**// Prepare client info for thread**

**CLIENT\_INFO \*ci = malloc(sizeof(CLIENT\_INFO));**

**ci->sock = clientSock;**

**strncpy(ci->ip,**

**inet\_ntoa(cliAddr.sin\_addr),**

**sizeof(ci->ip));**

**// Spawn a thread to handle this client**

**HANDLE h = CreateThread(**

**NULL, 0,**

**clientHandler,**

**ci, 0, NULL**

**);**

**if (!h) {**

**perror("CreateThread");**

**closesocket(clientSock);**

**free(ci);**

**} else {**

**CloseHandle(h);**

**}**

**}**

**// Cleanup (never reached)**

**closesocket(listenSock);**

**WSACleanup();**

**return 0;**

**}**

**Below is a breakdown of every major concept and API used in our C “quiz” client/server example,**

along with why and how you’d use each in a real networked C program on Windows. Wherever relevant, I’ve pulled out minimal code snippets to illustrate the concept in isolation.

**1. Winsock Initialization & Cleanup**

Before you can call any socket functions on Windows, you must initialize the Winsock library; when you’re done, you must clean it up.

WSADATA wsa;

if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {

fprintf(stderr, "WSAStartup failed\n");

exit(1);

}

// … your socket code …

WSACleanup();

WSAStartup loads the Winsock DLL and negotiates a version (here 2.2).

WSACleanup releases resources when you’re done.

Without this pair your calls to socket(), bind(), connect(), etc. will fail.

**2. The socket() Call**

SOCKET sock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (sock == INVALID\_SOCKET) { /\* error \*/ }

AF\_INET – IPv4 address family.

SOCK\_STREAM – a reliable, two-way, connection-based byte stream (i.e. TCP).

IPPROTO\_TCP – explicitly select the TCP protocol.

This returns a SOCKET handle (an unsigned integer on Windows) you’ll use for all further operations.

**3. Address Structures: struct sockaddr\_in**

struct sockaddr\_in serv = {0};

serv.sin\_family = AF\_INET;

serv.sin\_port = htons(8080);

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

sin\_family must match the family you passed to socket() (AF\_INET).

sin\_port is a 16-bit port number in network byte order; htons() (“host to network short”) swaps endianness if needed.

sin\_addr holds the IPv4 address; inet\_addr() parses a dotted-quad string into the required 32-bit value.

**4. Binding & Listening (Server Only)**

bind(listenSock, (SOCKADDR\*)&serv, sizeof(serv));

listen(listenSock, 5);

bind() associates your socket with a local IP & port so the OS knows to deliver incoming connections there.

listen() marks the socket as a listening socket and sets the maximum “backlog” (pending connections) to 5.

After listen() your socket is ready to accept new connections.

**5. Accepting Connections**

struct sockaddr\_in cli;

int cliLen = sizeof(cli);

SOCKET clientSock = accept(listenSock, (SOCKADDR\*)&cli, &cliLen);

accept() blocks until a client connects.

It returns a new SOCKET handle (clientSock) dedicated to that client; the original listenSock remains free to accept more.

You’ll almost always accept in a loop, so you can serve many clients:

while ((clientSock = accept(...)) != INVALID\_SOCKET) {

// hand off clientSock to handler

}

**6. Connecting (Client Only)**

if (connect(sock, (SOCKADDR\*)&serv, sizeof(serv)) == SOCKET\_ERROR) {

perror("connect");

exit(1);

}

connect() initiates a three-way TCP handshake with the remote server’s IP & port.

On success, you can immediately send() and recv() on sock.

**7. Sending and Receiving Data**

// Send a buffer

send(sock, buffer, (int)strlen(buffer), 0);

// Receive into buffer

int n = recv(sock, buffer, MAXLEN-1, 0);

if (n > 0) {

buffer[n] = '\0'; // null-terminate for printing

}

send() and recv() are your primary data I/O calls.

They work exactly like read()/write() on sockets, but take a flags parameter (we passed 0).

Always check return values:

send() returns the number of bytes actually sent (or SOCKET\_ERROR).

recv() returns 0 at orderly shutdown, negative on error.

**8. Byte-Order Utilities**

htons() / htonl(): Host-to-network for 16-bit / 32-bit integers.

ntohs() / ntohl(): Network-to-host.

Network byte order is big-endian; Intel and many other CPUs are little-endian, so these macros swap bytes on the fly where needed.

**9. Concurrency: Per-Client Threads**

After accept() returns a new socket, we spin up a thread:

typedef struct { SOCKET sock; char ip[16]; } CLIENT\_INFO;

DWORD WINAPI clientHandler(LPVOID param) {

CLIENT\_INFO \*ci = param;

// … handle quiz on ci->sock …

closesocket(ci->sock);

free(ci);

return 0;

}

// In main accept loop:

CLIENT\_INFO \*ci = malloc(sizeof(\*ci));

ci->sock = clientSock;

strcpy(ci->ip, inet\_ntoa(cliAddr.sin\_addr));

HANDLE h = CreateThread(NULL, 0, clientHandler, ci, 0, NULL);

if (h) CloseHandle(h);

else { /\* handle error \*/ }

Each client gets its own thread, so multiple users can take the quiz simultaneously.

We pass a small struct {sock, ip} so the thread knows which socket and remote IP it’s serving.

Don’t forget to free() that struct and closesocket() when the thread finishes.

**10. IP-Address Conversion**

inet\_addr("x.x.x.x") → 32-bit value, used for client → server connections.

inet\_ntoa(addr\_struct) → dotted-quad string, used by the server to log the client’s IP.

These are the simplest, POSIX-inspired APIs available on Windows.

**11. Console I/O and Prompting**

In our client we did:

printf("Your answer: ");

fflush(stdout);

fgets(buf, sizeof(buf), stdin);

send(sock, buf, strlen(buf), 0);

fflush(stdout) forces the prompt to appear before we block in fgets().

fgets() reads a line (including the trailing \n), so when sending back we’re including whatever the user typed plus a newline—exactly what the server expects to read.

**12. Cleanup and Error Handling**

Always check every Winsock call for error returns (INVALID\_SOCKET, SOCKET\_ERROR, or <0).

On error, you can call WSAGetLastError() to get the detailed Windows socket error code.

Always closesocket() every socket you open, and call WSACleanup() exactly once per successful WSAStartup().

13. Putting It All Together

**Server**

WSAStartup() → socket() → bind() → listen()

Loop: accept() → spawn CreateThread() → back to accept

In each thread: recv(login), send(welcome), quiz loop (send(question), recv(answer)), send(result), log, then closesocket().

**Client**

WSAStartup() → socket() → connect() → send(login)

recv(welcome) → loop quiz (recv(question), prompt & fgets(), send(answer)) → recv(result) → closesocket().

Why These Concepts Matter

Low-level control: C sockets give you raw access to TCP/IP, so you can build protocols (like our quiz handshake) on top.

Portability: Winsock APIs mirror POSIX sockets closely, so once you understand them in C you can move between Windows and Linux.

Concurrency: Threads (or fork() on Unix) let you service multiple clients without blocking the whole server.

Byte-order & structs: Understanding htons(), inet\_addr(), and sockaddr\_in is crucial for any networked C program.

With these building blocks in hand, you’re ready to tackle anything from chat servers to HTTP clients in pure C!

Below is a structured deep-dive into

**Network programming in C**,

covering the core APIs, concepts, and a minimal TCP “echo” server and client. Wherever you see a function or struct you don’t recognize, scroll back up to its explanation.

**1. Sockets & the Client–Server Model**

At its heart, network programming in C is all about sockets—endpoints for communication between two machines (or two processes on the same machine). You typically adopt a client–server pattern:

Server: creates a listening socket, waits for clients, then serves each one.

Client: creates a socket, connects to the server’s address/port, then sends and receives data.

**2. TCP vs UDP**

TCP (SOCK\_STREAM): reliable, connection-oriented, byte stream. Guarantees in-order delivery, retransmits lost packets.

UDP (SOCK\_DGRAM): unreliable, connectionless, message-oriented. Lower overhead, no guarantees.

All examples below use TCP.

**3. Key Data Types & Byte Order**

// IPv4 address + port

struct sockaddr\_in {

short sin\_family; // AF\_INET

unsigned short sin\_port; // port in network byte order

struct in\_addr sin\_addr; // 32-bit IPv4 address

};

// Convert host <→ network byte order:

uint16\_t htons(uint16\_t hostshort); // host to network (short)

uint32\_t htonl(uint32\_t hostlong); // host to network (long)

uint16\_t ntohs(uint16\_t netshort); // network to host

uint32\_t ntohl(uint32\_t netlong);

sin\_port must be in network (big-endian) order—use htons().

sin\_addr.s\_addr is also big-endian; you can set it via inet\_addr("1.2.3.4") or inet\_pton().

**4. The Socket API: Function Primer**

| Function | Role |
| --- | --- |
| socket(family, type, protocol) | Create a new socket descriptor |
| bind(fd, struct sockaddr\*, len) | Assign a local address & port to a socket |
| listen(fd, backlog) | Mark socket as passive, ready to accept() |
| accept(fd, addr, &len) | Accept an incoming connection |
| connect(fd, addr, len) | Initiate connection to server |
| send(fd, buf, len, flags) | Send data over a connected socket |
| recv(fd, buf, len, flags) | Receive data from a connected socket |
| close(fd) (POSIX) closesocket(fd) (Win) | Tear down the socket |

**5. Minimal TCP Server (POSIX)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h> // close()

#include <arpa/inet.h> // sockaddr\_in, htons(), inet\_addr()

#include <sys/socket.h> // socket API

#define PORT 8080

#define BACKLOG 5

int main() {

int listen\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in addr = {

.sin\_family = AF\_INET,

.sin\_port = htons(PORT),

.sin\_addr.s\_addr = INADDR\_ANY

};

bind(listen\_fd, (struct sockaddr\*)&addr, sizeof(addr));

listen(listen\_fd, BACKLOG);

printf("Listening on port %d …\n", PORT);

while (1) {

int client\_fd = accept(listen\_fd, NULL, NULL);

if (client\_fd < 0) continue;

// Echo phase:

char buf[512];

ssize\_t n;

while ((n = recv(client\_fd, buf, sizeof(buf), 0)) > 0) {

send(client\_fd, buf, n, 0);

}

close(client\_fd);

}

close(listen\_fd);

return 0;

}

Walk-through:

**socket()→TCP socket.**

**bind()→attach to all local interfaces on port 8080.**

**listen()→start listening queue.**

**accept()→block until a client connects, yielding client\_fd.**

**recv()/send()→echo loop until client closes.**

**close() the client socket, then loop to accept the next.**

**6. Minimal TCP Client (POSIX)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#define PORT 8080

int main(int argc, char \*argv[]) {

if (argc != 2) { fprintf(stderr, "Usage: %s <server-ip>\n", argv[0]); return 1; }

int sock = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in serv = {

.sin\_family = AF\_INET,

.sin\_port = htons(PORT)

};

inet\_pton(AF\_INET, argv[1], &serv.sin\_addr);

connect(sock, (struct sockaddr\*)&serv, sizeof(serv));

char \*msg = "Hello, server!\n";

send(sock, msg, strlen(msg), 0);

char buf[512];

ssize\_t n = recv(sock, buf, sizeof(buf)-1, 0);

buf[n] = '\0';

printf("Echo: %s", buf);

close(sock);

return 0;

}

Walk-through:

**socket()→TCP socket.**

**inet\_pton()→parse the server’s IP.**

**connect()→perform TCP handshake.**

**send()→send greeting.**

**recv()→read echo.**

**close()→clean up.**

**7. Porting to Windows (Winsock2)**

On Windows you must:

WSAStartup() before any socket calls.

Link Ws2\_32.lib and #include <winsock2.h>.

Use closesocket() instead of close().

Call WSACleanup() at the end.

Everything else is nearly identical.

**8. DNS & getaddrinfo()**

For robust, IPv4+IPv6–aware code, replace inet\_pton()+struct sockaddr\_in with:

struct addrinfo hints = {0}, \*res;

hints.ai\_family = AF\_UNSPEC; // v4 or v6

hints.ai\_socktype = SOCK\_STREAM;

getaddrinfo(hostname, "http", &hints, &res);

// loop res linked list, try socket() + connect() on each until success

And free with freeaddrinfo(res).

**9. Concurrency & Multiplexing**

To serve multiple clients you can:

fork() a new process (Unix).

CreateThread() or \_beginthread() (Windows).

select() or poll() on a single thread: monitor multiple sockets for readability/writability.

epoll (Linux) or kqueue (BSD/macOS).

**Example using select():**

fd\_set read\_fds;

FD\_ZERO(&read\_fds);

FD\_SET(listen\_fd, &read\_fds);

select(max\_fd+1, &read\_fds, NULL, NULL, NULL);

if (FD\_ISSET(listen\_fd, &read\_fds)) {

// accept new client

}

**10. Best Practices & Next Steps**

Always check return values for errors.

Use non-blocking or timeouts on recv() if you need to avoid hanging indefinitely.

For large data, loop on send() until all bytes are transmitted.

Consider TLS (OpenSSL) on top of your sockets for encryption.

Explore higher-level libraries, e.g. libcurl, ZeroMQ, or gRPC for richer protocols.

With these fundamentals—socket creation, binding/listening, connecting, sending/receiving, and concurrency—you can build everything from simple chat servers to full HTTP clients and beyond, all in plain C. Enjoy coding your own networked applications!

Below are several advanced topics and patterns you’ll encounter as you dive deeper into network programming in C—complete with explanations and minimal code snippets.

**1. UDP Sockets (Datagram)**

Unlike TCP’s stream, UDP is message-oriented and connectionless.

// UDP echo server (POSIX)

int sock = socket(AF\_INET, SOCK\_DGRAM, 0);

struct sockaddr\_in serv = { .sin\_family = AF\_INET,

.sin\_addr.s\_addr = INADDR\_ANY,

.sin\_port = htons(8081) };

bind(sock, (struct sockaddr\*)&serv, sizeof(serv));

char buf[512];

struct sockaddr\_in cli;

socklen\_t cliLen = sizeof(cli);

while (1) {

ssize\_t n = recvfrom(sock, buf, sizeof(buf), 0,

(struct sockaddr\*)&cli, &cliLen);

if (n > 0) {

sendto(sock, buf, n, 0,

(struct sockaddr\*)&cli, cliLen);

}

}

recvfrom/sendto let you specify the peer address per-packet.

Good for simple request-response, streaming telemetry, DNS, real-time voice/video where occasional loss is acceptable.

**2. Multicast**

To receive a multicast group:

struct ip\_mreq mreq;

mreq.imr\_multiaddr.s\_addr = inet\_addr("239.255.0.1");

mreq.imr\_interface.s\_addr = htonl(INADDR\_ANY);

setsockopt(sock, IPPROTO\_IP, IP\_ADD\_MEMBERSHIP,

&mreq, sizeof(mreq));

IP\_ADD\_MEMBERSHIP subscribes you to a multicast group.

Use sendto to transmit; all group members on the LAN will get the packet.

**3. Socket Options (setsockopt / getsockopt)**

int reuse = 1;

setsockopt(listenSock, SOL\_SOCKET, SO\_REUSEADDR,

&reuse, sizeof(reuse));

Common options:

SO\_REUSEADDR: bind to a port in TIME\_WAIT.

SO\_KEEPALIVE: detect dead peers.

SO\_LINGER: control shutdown behavior.

SO\_RCVTIMEO / SO\_SNDTIMEO: timeouts on blocking I/O.

**4. Non-Blocking I/O & select() Multiplexing**

Turn a socket non-blocking (POSIX):

int flags = fcntl(sock, F\_GETFL, 0);

fcntl(sock, F\_SETFL, flags | O\_NONBLOCK);

Then use select() to wait for readability/writability:

fd\_set readfds;

FD\_ZERO(&readfds);

FD\_SET(sock, &readfds);

struct timeval tv = {5,0}; // 5s timeout

int ready = select(sock+1, &readfds, NULL, NULL, &tv);

if (ready > 0 && FD\_ISSET(sock, &readfds)) {

recv(sock,...);

}

select() scales to ~1024 fds on many systems.

Returns when any watched socket is ready, or timeout expires.

**5. poll() and epoll (Linux-only)**

On Linux, for large numbers of connections, use poll() or epoll:

struct pollfd fds[100];

fds[0].fd = listenSock; fds[0].events = POLLIN;

int count = poll(fds, nfds, 5000);

if (count > 0 && (fds[0].revents & POLLIN)) {

accept(...);

}

epoll is similar but more efficient for thousands of fds.

**6. Raw Sockets & Packet Crafting**

int rsock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_ICMP);

Allows you to read/write full IP packets.

Used for ping (ICMP), traceroute, or custom protocol prototypes.

Requires elevated privileges (root / Administrator).

**7. IPv6 Support**

struct sockaddr\_in6 serv6 = {

.sin6\_family = AF\_INET6,

.sin6\_port = htons(8080),

.sin6\_addr = in6addr\_any

};

socket(AF\_INET6, SOCK\_STREAM, 0);

Use getaddrinfo() with AF\_UNSPEC to write dual-stack code that works on both IPv4 and IPv6.

**8. UNIX Domain Sockets (POSIX Only)**

int sock = socket(AF\_UNIX, SOCK\_STREAM, 0);

struct sockaddr\_un addr = {

.sun\_family = AF\_UNIX,

.sun\_path = "/tmp/mysock"

};

bind(sock, (struct sockaddr\*)&addr, sizeof(addr));

listen(sock, 5);

Fast inter-process communication on the same host.

Supports SOCK\_STREAM and SOCK\_DGRAM.

**9. Asynchronous I/O**

On Windows you can use Overlapped I/O (with WSARecv, WSASend, and OVERLAPPED structs).

On Linux you can explore aio\_read(), io\_uring, or event libraries like libuv.

**10. Security & TLS**

To secure your socket channel, layer OpenSSL or mbedTLS:

**SSL\_library\_init();**

**SSL\_CTX \*ctx = SSL\_CTX\_new(TLS\_client\_method());**

**SSL \*ssl = SSL\_new(ctx);**

**SSL\_set\_fd(ssl, sock);**

**SSL\_connect(ssl);**

**SSL\_write(ssl, buf, len);**

**SSL\_read(ssl, buf, sizeof(buf));**

Encrypts all data.

Requires cert management but is essential for production services.

**11. High-Level Frameworks**

Once you understand the low-level APIs, you can step up to libraries that abstract away the details:

libevent / libev: event loops with built-in select/poll/epoll support.

ZeroMQ: socket-style messaging patterns (REQ/REP, PUB/SUB).

gRPC: modern RPC over HTTP/2 with code generation.