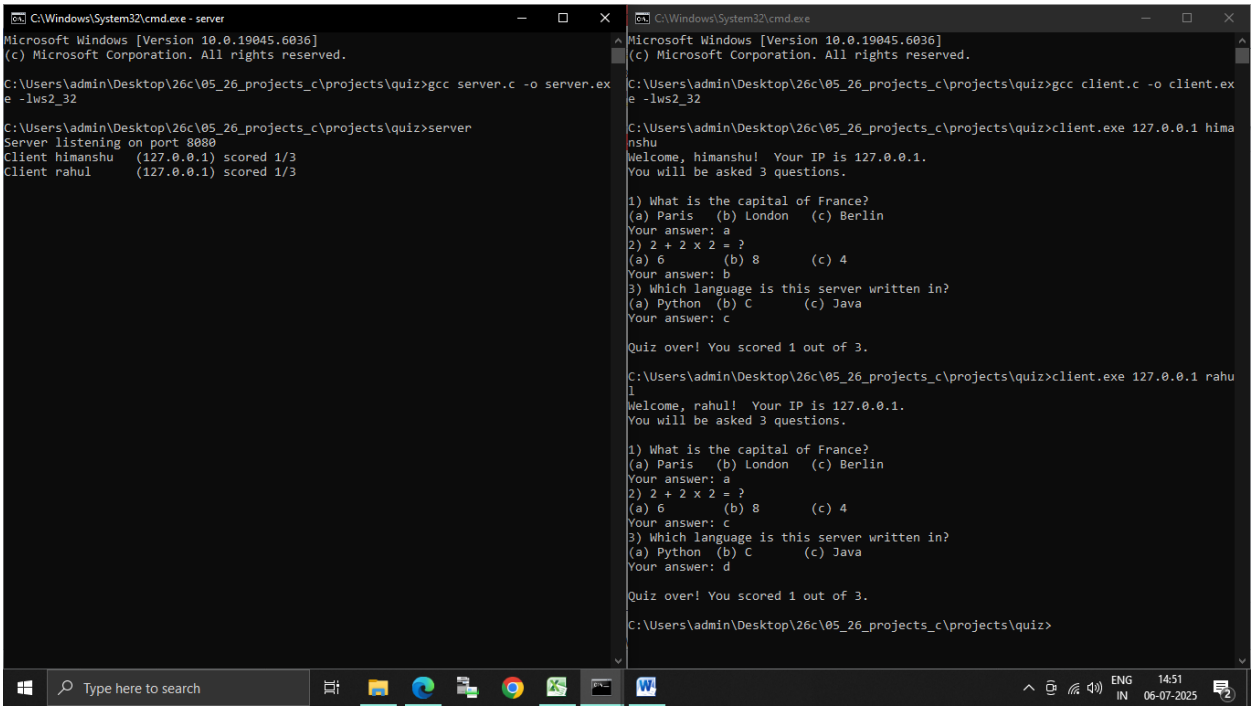
NETWORK PROGRAMMNG IN C

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

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.