E-Poll
server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <fcntl.h>

#define PORT 12345
#define MAX_EVENTS 10
#define BUFFER_SIZE 1024

// Set a socket to non-blocking mode
int set_nonblocking(int sock) {
    int flags = fcntl(sock, F_GETFL, 0);
    if (flags == -1) return -1;
    return fcntl(sock, F_SETFL, flags | O_NONBLOCK);
}

int main() {
    int server_fd, client_fd, epoll_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    struct epoll_event event, events[MAX_EVENTS];

    // Create server socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set the socket to non-blocking mode
    if (set_nonblocking(server_fd) == -1) {
        perror("set_nonblocking");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Bind the socket
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_fd, 5) == -1) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

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

    // Register the server socket with epoll
    event.events = EPOLLIN;
    event.data.fd = server_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &event) == -1) {
        perror("epoll_ctl");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

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

    while (1) {
        int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        for (int i = 0; i < nfds; i++) {
            if (events[i].data.fd == server_fd) {
                // Accept new connection
                client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);
                if (client_fd == -1) {
                    perror("accept");
                    continue;
                }
                printf("Accepted connection from client\n");
                set_nonblocking(client_fd);

                // Register client socket with epoll
                event.events = EPOLLIN | EPOLLET; // Edge-triggered
                event.data.fd = client_fd;
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &event) == -1) {
                    perror("epoll_ctl");
                    close(client_fd);
                }
            } else {
                // Handle data from client
                char buffer[BUFFER_SIZE];
                ssize_t count = recv(events[i].data.fd, buffer, sizeof(buffer), 0);
                if (count == -1) {
                    perror("recv");
                    close(events[i].data.fd);
                } else if (count == 0) {
                    // Client has closed connection
                    printf("Client disconnected\n");
                    close(events[i].data.fd);
                } else {
                    // Print received data
                    buffer[count] = '\0'; // Null-terminate the received string
                    printf("Received from client: %s", buffer);
                }
            }
        }
    }

    // Clean up
    close(server_fd);
    close(epoll_fd);
    return 0;
}
```

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define SERVER_IP "127.0.0.1"
#define PORT 12345
#define BUFFER_SIZE 1024

int main() {
    int sock_fd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Create socket
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Connect to server
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

    if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("connect");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    printf("Connected to server. Type messages to send:\n");

    while (1) {
        // Read input from user
        printf("> ");
        fgets(buffer, BUFFER_SIZE, stdin);

        // Send data to server
        send(sock_fd, buffer, strlen(buffer), 0);
    }

    // Clean up
    close(sock_fd);
    return 0;
}
```

Multicasting
server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 12345
#define MULTICAST_ADDR "239.255.255.250"
#define MESSAGE "Hello, Multicast Clients!\n"

int main() {
    int sock_fd;
    struct sockaddr_in multicast_addr;
    // Create a UDP socket
    sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set up the multicast address
    memset(&multicast_addr, 0, sizeof(multicast_addr));
    multicast_addr.sin_family = AF_INET;
    multicast_addr.sin_addr.s_addr = inet_addr(MULTICAST_ADDR);
    multicast_addr.sin_port = htons(PORT);

    // Send messages to the multicast group
    while (1) {
        ssize_t sent_bytes = sendto(sock_fd, MESSAGE, strlen(MESSAGE), 0,
                                    (struct sockaddr *)&multicast_addr, sizeof(multicast_addr));
        if (sent_bytes < 0) {
            perror("sendto");
            close(sock_fd);
            exit(EXIT_FAILURE);
        }
        printf("Sent: %s", MESSAGE);
        sleep(1);  // Send a message every second
    }

    // Clean up
    close(sock_fd);
    return 0;
}
```

client.c

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

#define PORT 12345
#define MULTICAST_ADDR "239.255.255.250"
#define BUFFER_SIZE 1024

int main() {
    int sock_fd;
    struct sockaddr_in local_addr, multicast_addr;
    char buffer[BUFFER_SIZE];
    int n;
    // Create a UDP socket
    sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Allow multiple sockets to use the same PORT number
    int reuse = 1;
    if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&reuse,
sizeof(reuse)) < 0) {
        perror("setsockopt");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    // Bind the socket to any valid IP address and the specified port
    memset(&local_addr, 0, sizeof(local_addr));
    local_addr.sin_family = AF_INET;
    local_addr.sin_addr.s_addr = INADDR_ANY; // Bind to all local interfaces
    local_addr.sin_port = htons(PORT);

    if (bind(sock_fd, (struct sockaddr *)&local_addr, sizeof(local_addr)) < 0) {
        perror("bind");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    // Set up the multicast group address
    memset(&multicast_addr, 0, sizeof(multicast_addr));
    multicast_addr.sin_family = AF_INET;
    multicast_addr.sin_addr.s_addr = inet_addr(MULTICAST_ADDR);
    multicast_addr.sin_port = htons(PORT);

    // Join the multicast group
    struct ip_mreq mreq;
    mreq.imr_multiaddr = multicast_addr.sin_addr;
    mreq.imr_interface.s_addr = htonl(INADDR_ANY);

    if (setsockopt(sock_fd, IPPROTO_IP, IP_ADD_MEMBERSHIP, (void *)&mreq,
sizeof(mreq)) < 0) {
        perror("setsockopt");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    printf("Listening for multicast messages on %s:%d\n", MULTICAST_ADDR, PORT);

    while (1) {
        socklen_t addr_len = sizeof(multicast_addr);
        n = recvfrom(sock_fd, buffer, BUFFER_SIZE - 1, 0, (struct sockaddr
*)&multicast_addr, &addr_len);
        if (n < 0) {
            perror("recvfrom");
            close(sock_fd);
            exit(EXIT_FAILURE);
        }
        buffer[n] = '\0'; // Null-terminate the received message
        printf("Received: %s", buffer);
    }

    // Clean up
    close(sock_fd);
    return 0;
}
```

Multiplexing
server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/select.h>

#define PORT 12345
#define MAX_CLIENTS 10
#define BUFFER_SIZE 1024

int main() {
    int server_fd, client_fd, max_sd, activity, sd;
    int client_sockets[MAX_CLIENTS];
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    char buffer[BUFFER_SIZE];

    // Initialize client sockets
    for (int i = 0; i < MAX_CLIENTS; i++) {
        client_sockets[i] = 0;
    }

    // Create server socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Bind the socket
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_fd, 3) == -1) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

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

    while (1) {
        fd_set readfds;

        // Clear the socket set
        FD_ZERO(&readfds);

        // Add server socket to the set
        FD_SET(server_fd, &readfds);
        max_sd = server_fd;

        // Add child sockets to the set
        for (int i = 0; i < MAX_CLIENTS; i++) {
            sd = client_sockets[i];

            // If valid socket descriptor then add to read list
            if (sd > 0) {
                FD_SET(sd, &readfds);
            }

            // Keep track of the maximum socket descriptor
            if (sd > max_sd) {
                max_sd = sd;
            }
        }

        // Wait for activity on the sockets
```

```
        activity = select(max_sd + 1, &readfds, NULL, NULL, NULL);
        if (activity < 0) {
            perror("select");
            exit(EXIT_FAILURE);
        }

        // If something happened on the master socket, then it's an incoming connection
        if (FD_ISSET(server_fd, &readfds)) {
            client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);
            if (client_fd < 0) {
                perror("accept");
                exit(EXIT_FAILURE);
            }
            printf("Accepted connection from client\n");

            // Add new socket to array of sockets
            for (int i = 0; i < MAX_CLIENTS; i++) {
                if (client_sockets[i] == 0) {
                    client_sockets[i] = client_fd;
                    break;
                }
            }
        }

        // Check for I/O operations on other sockets
        for (int i = 0; i < MAX_CLIENTS; i++) {
            sd = client_sockets[i];

            if (FD_ISSET(sd, &readfds)) {
                int n = recv(sd, buffer, sizeof(buffer), 0);
                if (n <= 0) {
                    // Client disconnected
                    printf("Client disconnected\n");
                    close(sd);
                    client_sockets[i] = 0;
                } else {
                    buffer[n] = '\0'; // Null-terminate the received string
                    printf("Received: %s", buffer);
                }
            }
        }
    }

    // Clean up
    close(server_fd);
    return 0;
}
```

client.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/select.h>

#define SERVER_IP "127.0.0.1"
#define PORT 12345
#define BUFFER_SIZE 1024

int main() {
    int sock_fd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Create socket
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Connect to server
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

    if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("connect");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    printf("Connected to server. Type messages to send:\n");

    while (1) {
        // Use select to wait for user input or server response
        fd_set readfds;
        FD_ZERO(&readfds);
        FD_SET(sock_fd, &readfds);
        FD_SET(STDIN_FILENO, &readfds);

        // Wait for activity
        int max_fd = sock_fd > STDIN_FILENO ? sock_fd : STDIN_FILENO;
        select(max_fd + 1, &readfds, NULL, NULL, NULL);

        // Check if there is data to read from the socket
        if (FD_ISSET(sock_fd, &readfds)) {
            ssize_t n = recv(sock_fd, buffer, sizeof(buffer), 0);
            if (n > 0) {
                buffer[n] = '\0'; // Null-terminate the received string
                printf("Server: %s", buffer);
            } else {
                // Server closed the connection
                printf("Server disconnected\n");
                close(sock_fd);
                exit(EXIT_SUCCESS);
            }
        }

        // Check if there is input from the user
        if (FD_ISSET(STDIN_FILENO, &readfds)) {
            printf("> ");
            fgets(buffer, BUFFER_SIZE, stdin);
            send(sock_fd, buffer, strlen(buffer), 0);
        }
    }

    // Clean up
    close(sock_fd);
    return 0;
}
```

Multithreading
server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define PORT 12345
#define BUFFER_SIZE 1024

void *handle_client(void *client_socket) {
    int sock = *(int *)client_socket;
    free(client_socket);

    char buffer[BUFFER_SIZE];
    ssize_t n;
    while ((n = recv(sock, buffer, sizeof(buffer) - 1, 0)) > 0) {
        buffer[n] = '\0'; // Null-terminate the received string
        printf("Received: %s", buffer);

        // Optionally, send a response back to the client
        send(sock, "Message received\n", 17, 0);
    }

    if (n == 0) {
        printf("Client disconnected\n");
    } else {
        perror("recv");
    }

    close(sock);
    return NULL;
}

int main() {
    int server_fd, *client_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);

    // Create server socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Bind the socket
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_fd, 5) == -1) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
```

```c
    printf("Server is running on port %d...\n", PORT);

    while (1) {
        client_socket = malloc(sizeof(int));
        *client_socket = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);
        if (*client_socket == -1) {
            perror("accept");
            free(client_socket);
            continue;
        }
        printf("Accepted connection from client\n");

        // Create a new thread to handle the client
        pthread_t tid;
        if (pthread_create(&tid, NULL, handle_client, (void *)client_socket) != 0) {
            perror("pthread_create");
            close(*client_socket);
            free(client_socket);
        }

        pthread_detach(tid); // Detach the thread to reclaim resources when it finishes
    }

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

client.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define SERVER_IP "127.0.0.1"
#define PORT 12345
#define BUFFER_SIZE 1024

int main() {
    int sock_fd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Create socket
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Connect to server
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

    if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("connect");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    printf("Connected to server. Type messages to send:\n");

    while (1) {
        // Read input from user
        printf("> ");
        fgets(buffer, BUFFER_SIZE, stdin);

        // Send data to server
        send(sock_fd, buffer, strlen(buffer), 0);

        // Optionally, receive response from server
        ssize_t n = recv(sock_fd, buffer, sizeof(buffer), 0);
        if (n > 0) {
            buffer[n] = '\0'; // Null-terminate the received string
            printf("Server: %s", buffer);
        }
    }

    // Clean up
    close(sock_fd);
    return 0;
}
```

Openssl
server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

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

#define PORT 4433
#define BUFFER_SIZE 1024

void handle_errors() {
   ERR_print_errors_fp(stderr);
   abort();
}

int main() {
   SSL_CTX *ctx;
   SSL *ssl;
   int server_fd, client_fd;
   struct sockaddr_in server_addr, client_addr;
   socklen_t addr_len = sizeof(client_addr);
   char buffer[BUFFER_SIZE];

   // Initialize OpenSSL
   SSL_library_init();
   OpenSSL_add_all_algorithms();
   SSL_load_error_strings();
   ctx = SSL_CTX_new(TLS_server_method());
   if (!ctx) {
      handle_errors();
   }

   // Load certificates
   if (SSL_CTX_use_certificate_file(ctx, "server.crt", SSL_FILETYPE_PEM) <= 0 ||
      SSL_CTX_use_PrivateKey_file(ctx, "server.key", SSL_FILETYPE_PEM) <= 0) {
      handle_errors();
   }

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

   // Bind the socket
   memset(&server_addr, 0, sizeof(server_addr));
   server_addr.sin_family = AF_INET;
   server_addr.sin_addr.s_addr = INADDR_ANY;
   server_addr.sin_port = htons(PORT);
   if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
      perror("bind");
      close(server_fd);
      exit(EXIT_FAILURE);
```

```
    }

    // Listen for incoming connections
    if (listen(server_fd, 5) < 0) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Secure server is running on port %d...\n", PORT);

    while (1) {
        // Accept incoming connection
        client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);
        if (client_fd < 0) {
            perror("accept");
            continue;
        }

        // Create SSL structure
        ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client_fd);

        // Accept SSL connection
        if (SSL_accept(ssl) <= 0) {
            ERR_print_errors_fp(stderr);
        } else {
            // Communicate with client
            int bytes;
            while ((bytes = SSL_read(ssl, buffer, sizeof(buffer))) > 0) {
                buffer[bytes] = '\0'; // Null-terminate the received string
                printf("Received: %s\n", buffer);
                SSL_write(ssl, "Message received\n", 17);
            }
        }

        // Cleanup
        SSL_free(ssl);
        close(client_fd);
    }

    // Clean up
    SSL_CTX_free(ctx);
    close(server_fd);
    return 0;
}
```

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define SERVER_IP "127.0.0.1"
#define PORT 4433
#define BUFFER_SIZE 1024

void handle_errors() {
    ERR_print_errors_fp(stderr);
    abort();
}

int main() {
    SSL_CTX *ctx;
    SSL *ssl;
    int sock_fd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Initialize OpenSSL
    SSL_library_init();
    OpenSSL_add_all_algorithms();
    SSL_load_error_strings();
    ctx = SSL_CTX_new(TLS_client_method());
    if (!ctx) {
        handle_errors();
    }

    // Create socket
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Connect to server
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

    if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("connect");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    // Create SSL structure
    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, sock_fd);

    // Initiate SSL connection
    if (SSL_connect(ssl) <= 0) {
        ERR_print_errors_fp(stderr);
    } else {
        printf("Connected to secure server.\n");

        // Communicate with server
        while (1) {
            printf("> ");
            fgets(buffer, BUFFER_SIZE, stdin);
            SSL_write(ssl, buffer, strlen(buffer));

            int bytes = SSL_read(ssl, buffer, sizeof(buffer));
            if (bytes > 0) {
                buffer[bytes] = '\0'; // Null-terminate the received string
                printf("Server: %s", buffer);
            }
        }
    }

    // Cleanup
    SSL_free(ssl);
    close(sock_fd);
    SSL_CTX_free(ctx);
    return 0;
}
```

P2P
server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

void send_file(FILE *fp, int socket) {
    char buffer[BUFFER_SIZE];
    while (fgets(buffer, BUFFER_SIZE, fp) != NULL) {
        if (send(socket, buffer, strlen(buffer), 0) == -1) {
            perror("send");
            exit(EXIT_FAILURE);
        }
        memset(buffer, 0, BUFFER_SIZE);
    }
}

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    char *filename = "shared_file.txt";

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

    // Set up the server address structure
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind the socket
    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_fd, 5) == -1) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Peer-to-peer server is running on port %d...\n", PORT);

    while (1) {
        // Accept an incoming connection
        client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);
        if (client_fd == -1) {
            perror("accept");
            continue;
        }
        printf("Client connected.\n");

        // Open the file to send
        FILE *fp = fopen(filename, "r");
        if (fp == NULL) {
            perror("File not found");
            close(client_fd);
            continue;
        }

        // Send the file to the client
        send_file(fp, client_fd);
        printf("File sent successfully.\n");

        // Clean up
        fclose(fp);
        close(client_fd);
    }

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

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

void receive_file(int socket) {
    char buffer[BUFFER_SIZE];
    FILE *fp = fopen("received_file.txt", "w");
    if (fp == NULL) {
        perror("File could not be opened");
        exit(EXIT_FAILURE);
    }

    int n;
    while ((n = recv(socket, buffer, BUFFER_SIZE, 0)) > 0) {
        fwrite(buffer, sizeof(char), n, fp);
        memset(buffer, 0, BUFFER_SIZE);
    }
    fclose(fp);
}

int main() {
    int sock_fd;
    struct sockaddr_in server_addr;

    // Create socket
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set up the server address structure
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr); // Change IP if needed

    // Connect to the server
    if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("connect");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    // Receive the file from the server
    receive_file(sock_fd);
    printf("File received successfully.\n");

    // Clean up
    close(sock_fd);
    return 0;
}
```