Client side

This C++ code is a simple client-side implementation of a chat application. It connects to a server over a TCP/IP network and allows the user to send and receive messages. Here's a detailed explanation of how the code works:

Header Inclusions and Macros

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

#define PORT 8080
#define BUFFER_SIZE 1024
```

- `<iostream>` and `<string>`: For standard input-output operations and string handling.
- `<unistd.h>`: For POSIX operating system API functions.
- `<arpa/inet.h>`, `<sys/socket.h>`, `<sys/types.h>`, and `<netinet/in.h>`: For socket programming and internet operations.
- `#define PORT 8080`: Defines the port number where the server is expected to be listening.
- `#define BUFFER_SIZE 1024`: Defines the size of the buffer for message transfers.

Main Function

```
int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
```

- Initializes the socket descriptor `sock` to 0.
- Declares `serv_addr` for the server address structure.
- Initializes a buffer to hold incoming messages.

Socket Creation

```
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    std::cerr << "Socket creation error" << std::endl;
    return -1;
}
```

- Creates a socket using the `socket` function. If it returns a value less than 0, it indicates an error.

Setting Server Address

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(PORT);

if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
    std::cerr << "Invalid address / Address not supported" << std::endl;
    return -1;
}
```

- Sets the server address family to `AF_INET` (IPv4).
- Sets the port to `PORT` (8080), converted to network byte order using `htons`.
- Converts the IP address from text to binary form using `inet_pton`. If it fails, it prints an error message and exits.

Connecting to the Server

```
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    std::cerr << "Connection failed" << std::endl;
    return -1;
}
```

- Attempts to connect to the server. If it fails, it prints an error message and exits.

Chat Loop

```
std::string message;
fd_set readfds;

while (true) {
    FD_ZERO(&readfds);
    FD_SET(sock, &readfds);
    FD_SET(STDIN_FILENO, &readfds);
```

```
    int max_sd = sock > STDIN_FILENO ? sock : STDIN_FILENO;

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

- Declares a `message` string for user input.
- Declares a `fd_set` named `readfds` to manage file descriptors for `select`.
- The `while` loop runs indefinitely to handle user input and server messages.
- `FD_ZERO` clears the `readfds` set.
- `FD_SET` adds the socket and standard input file descriptor to the set.
- `max_sd` is set to the larger of `sock` and `STDIN_FILENO`.
- `select` is used to wait for activity on either the socket or standard input.

Handling User Input

```
    if (FD_ISSET(STDIN_FILENO, &readfds)) {
        std::getline(std::cin, message);
        send(sock, message.c_str(), message.length(), 0);
    }
```

- Checks if there's input from the standard input (user typing).
- If so, reads the input into the `message` string and sends it to the server.

Handling Server Messages

```
    if (FD_ISSET(sock, &readfds)) {
        int valread = read(sock, buffer, BUFFER_SIZE);
        if (valread > 0) {
            buffer[valread] = '\0';
            std::cout << "Server: " << buffer << std::endl;
        }
    }
  }

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

- Checks if there's a message from the server.
- If so, reads it into the `buffer`. If `read` returns a positive value, it means there was data received.
- The buffer is null-terminated and printed as the server's message.

Cleanup

- After exiting the loop (if it ever does), the socket is closed with `close(sock)`.
- The program returns 0, indicating successful execution.

This code sets up a basic TCP client that connects to a server at `127.0.0.1` on port `8080`, allowing for simple text-based communication.

Serverside

This C++ code is a server-side implementation of a chat application. The server listens for incoming client connections, handles multiple clients concurrently, and broadcasts messages from any client to all other clients. Here's a detailed breakdown of the code:

Header Inclusions and Macros

cpp
Copy code

```cpp
#include <iostream>
#include <string>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/select.h>
#include <vector>

#define PORT 8080
#define MAX_CLIENTS 10
#define BUFFER_SIZE 1024
```

- **`<iostream>`** and **`<string>`**: For standard input-output operations and string handling.
- **`<unistd.h>`**: For POSIX operating system API functions.
- **`<arpa/inet.h>`**, **`<sys/socket.h>`**, **`<sys/types.h>`**, **`<netinet/in.h>`**, and **`<sys/select.h>`**: For socket programming and internet operations.
- **`<vector>`**: For managing a dynamic array of client sockets.

- `#define PORT 8080`: Defines the port number where the server will listen.
- `#define MAX_CLIENTS 10`: Defines the maximum number of clients the server can handle simultaneously.
- `#define BUFFER_SIZE 1024`: Defines the size of the buffer for message transfers.

Broadcast Function

cpp
Copy code

```
void broadcastMessage(int senderSocket, const std::string& message,
const std::vector<int>& clientSockets) {
    for (int clientSocket : clientSockets) {
        if (clientSocket != senderSocket) {
            send(clientSocket, message.c_str(), message.length(), 0);
        }
    }
}
```

- `broadcastMessage`: Sends a message to all connected clients except the sender.
  - `senderSocket`: The socket descriptor of the client that sent the message.
  - `message`: The message to be broadcast.
  - `clientSockets`: A vector containing the socket descriptors of all connected clients.

Main Function

cpp
Copy code

```
int main() {
    int serverSocket, newSocket, maxClients = MAX_CLIENTS;
    int clientSockets[MAX_CLIENTS] = {0};
    int maxSocket;
    struct sockaddr_in address;
    fd_set readfds;
```

- `serverSocket`: The server's listening socket.
- `newSocket`: Socket descriptor for new connections.
- `clientSockets`: Array to hold the socket descriptors of connected clients.
- `maxSocket`: The highest socket descriptor value.
- `address`: Structure to hold the server address.

- **`readfds`**: File descriptor set used by `select`.

Creating Server Socket

cpp
Copy code

```
   if ((serverSocket = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket failed");
        exit(EXIT_FAILURE);
    }

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

- Creates a socket using the `socket` function.
- Configures the server address to accept any incoming IP address (`INADDR_ANY`) on port `PORT`.

Binding and Listening

cpp
Copy code

```
   if (bind(serverSocket, (struct sockaddr *)&address,
sizeof(address)) < 0) {
        perror("Bind failed");
        close(serverSocket);
        exit(EXIT_FAILURE);
    }

    if (listen(serverSocket, 3) < 0) {
        perror("Listen failed");
        close(serverSocket);
        exit(EXIT_FAILURE);
    }

    std::cout << "Listening on port " << PORT << std::endl;
```

- Binds the socket to the specified address and port.
- Starts listening for incoming connections with a backlog of 3.

Main Loop

cpp
Copy code

```
   while (true) {
        FD_ZERO(&readfds);
        FD_SET(serverSocket, &readfds);
        maxSocket = serverSocket;

        for (int i = 0; i < maxClients; i++) {
            int socket = clientSockets[i];
            if (socket > 0) {
                FD_SET(socket, &readfds);
            }
            if (socket > maxSocket) {
                maxSocket = socket;
            }
        }

        int activity = select(maxSocket + 1, &readfds, NULL, NULL,
NULL);

        if ((activity < 0) && (errno != EINTR)) {
            std::cerr << "Select error" << std::endl;
        }

        if (FD_ISSET(serverSocket, &readfds)) {
            if ((newSocket = accept(serverSocket, (struct sockaddr
*)&address, (socklen_t*)&address)) < 0) {
                perror("Accept failed");
                exit(EXIT_FAILURE);
            }

            std::cout << "New connection, socket fd is " << newSocket
<< std::endl;

            for (int i = 0; i < maxClients; i++) {
                if (clientSockets[i] == 0) {
                    clientSockets[i] = newSocket;
```

```
                    std::cout << "Adding to list of sockets as " << i
<< std::endl;
                    break;
                }
            }
        }

        for (int i = 0; i < maxClients; i++) {
            int socket = clientSockets[i];
            if (FD_ISSET(socket, &readfds)) {
                char buffer[BUFFER_SIZE];
                int valread = read(socket, buffer, BUFFER_SIZE);
                if (valread == 0) {
                    close(socket);
                    clientSockets[i] = 0;
                } else {
                    buffer[valread] = '\0';
                    std::string message = std::string(buffer);
                    std::cout << "Message from client: " << message <<
std::endl;
                    broadcastMessage(socket, message,
std::vector<int>(clientSockets, clientSockets + maxClients));
                }
            }
        }
    }

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

- The main loop handles new connections and incoming messages:
    - **`FD_ZERO(&readfds)`**: Clears the `readfds` set.
    - **`FD_SET(serverSocket, &readfds)`**: Adds the server socket to the `readfds` set.
    - **`maxSocket = serverSocket`**: Initializes `maxSocket` with the server socket descriptor.
    - **For loop**: Adds client sockets to the `readfds` set and updates `maxSocket`.

    - `select(maxSocket + 1, &readfds, NULL, NULL, NULL)`: Waits for activity on any socket.
    - **New connection**: If activity is on the server socket, accepts the new connection and adds it to the `clientSockets` array.
    - **Existing connection**: If activity is on a client socket, reads the message, checks if the client disconnected (if `valread` is 0), and broadcasts the message to all other clients.

Cleanup

- After the loop, the server socket is closed with `close(serverSocket)`.
- The program returns 0, indicating successful execution.

This code sets up a basic TCP server that listens for incoming connections on port `8080`, handles up to 10 clients concurrently, and broadcasts messages from any client to all other connected clients.

4o