南開大學
Nankai University

计算机学院

计算机网络实验一

简单对话的实现

姓名：岳一名

学号：2212472

专业：计算机科学与技术

2024 年 10 月 19 日

目录

1 协议的实现

这段代码实现的客户端与服务器之间的通信基于传输控制协议 (TCP)。首先端口号为 8888，使用 IPV4 的地址形式，地址为“127.0.0.1”。TCP 是一种面向连接的协议，提供可靠的字节流传输，确保数据包按顺序到达，并具有错误检测和重传机制。在代码中，客户端通过 socket(AF_INET, SOCK_STREAM, 0) TCP

客户端和服务器之间的消息传递遵循简单的文本协议。客户端通过 send() 函数发送用户输入的消息，并使用 recv() 函数接收服务器的响应。此协议不涉及复杂的消息格式或状态管理，我们信息的内容就是一个字符串，因为 c++ 语言中的字符串使用 utf-8 编码格式，所以我们能够直接输入中英文，进行中英文的对话。

2 具体对话的流程

2.1 创建服务端

```
if (bind(server_fd, (sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
    std::cerr << "Bind failed\n";
    return -1;
}

if (listen(server_fd, MAX_CLIENTS) < 0) {
    std::cerr << "Listen failed\n";
    return -1;
}
```

在上面判断代码之前，省略了一些变量的声明，上面两个函数是建立服务端的重要函数

bind 函数将一个套接字 (server_fd) 与一个特定的地址 (server_addr) 绑定在一起。这是服务器设置的一部分，确保服务器可以在指定的 IP 地址和端口上监听连接请求。该函数的参数包括创建的套接字描述符、指向包含地址信息的结构体的指针 (需转换为 sockaddr 类型) 和地址结构体的大小。成功返回 0，失败返回-1，若绑定失败，通常表示端口被占用、权限问题或网络配置错误。

listen 函数将套接字设置为监听状态，准备接受客户端的连接请求。它允许指定最大客户端连接数 (MAX_CLIENTS)，我们这里设置的是 10，即同时能够处理的未决连接数。此函数的参数包括需要监听的套接字描述符和最大等待连接的客户端数量。成功返回 0，失败返回-1，若监听失败，通常是因为 bind 未成功执行、系统限制或资源不足。若一切正常，listen 使服务器能够接受来自客户端的连接请求。

经过上面的函数，我们的服务端就处在监听的状态，能够监听来自客户端链接的请求。服务端储在就绪状态。

2.2 客户端发起链接

```
if (connect(sockfd, (sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
    std::cerr << "Connection failed\n";
    return -1;
}
```

这段代码使用 connect 函数尝试建立与服务器的连接。它的主要功能是将客户端套接字(sockfd)连接到服务器指定的地址和端口（server_addr)。该函数的参数包括套接字描述符、指向服务器地址的结构体的指针（需转换为 sockaddr 类型）以及地址结构体的大小。

connect 函数的返回值是建立连接的结果。成功时返回 0，失败时返回-1。如果连接失败，程序会输出错误信息并返回-1，表示客户端未能成功连接到服务器。连接失败可能的原因包括服务器未启动、网络不可达、指定的 IP 地址或端口错误等。这些因素会直接影响客户端与服务器之间的通信。上面能够发起连接，并且判断是否链接成功，如果成功，就说明客户端与服务端建立了链接，就相当于客户端进入了“聊天室”，这样我们就能进行客户之间的对话。

信息的发送共享

```
void broadcast_message(const std::string &message, int sender_fd) {
    std::lock_guard<std::mutex> guard(clients_mutex);
    for (int client_fd : clients) {
        if (client_fd != sender_fd) {
            send(client_fd, message.c_str(), message.length(), 0);
        }
    }
}
void handle_client(int client_fd) {
    char buffer[1024];
    std::string welcome_msg = "Welcome to the chat room!\n";
    send(client_fd, welcome_msg.c_str(), welcome_msg.length(), 0);

    while (true) {
        memset(buffer, 0, sizeof(buffer));
        int bytes_received = recv(client_fd, buffer, sizeof(buffer), 0);

        if (bytes_received <= 0) {
            std::lock_guard<std::mutex> guard(clients_mutex);
            clients.erase(std::remove(clients.begin(), clients.end(), client_fd),
                clients.end());
            close(client_fd);
            std::cout << "Client disconnected: " << client_fd << std::endl;
            break;
        }

        std::string message = "Client " + std::to_string(client_fd) + ": " + buffer;
        std::cout << message << std::endl;
        broadcast_message(message, client_fd);
    }
}
```

上面两个函数就是服务端，handle_client() 函数就是处理收发信息的处理，设置一个持续执行的 while，并且 recv 函数就是接收不同客户端的信息。如果接收成功的话，就进行消息的广播，能够将信息传播到其他聊天室内。广播函数会遍历链接服务端的所有客户端的 ID，除了发送消息的客户端以外，其他的都需要进行消息的广播操作。并且在广播的过程中，会使用互斥锁进行线程的锁定，以确保在发送信息的时候不会导致信息数据的混乱。

上面就是发送到服务端的信息如何传播到客户端实现信息的共享。下面来看客户端如何发送到服务端

```
void receive_messages(int sockfd) {
    char buffer[1024];
    while (true) {
        memset(buffer, 0, sizeof(buffer));
        int bytes_received = recv(sockfd, buffer, sizeof(buffer), 0);
        if (bytes_received <= 0) {
            std::cerr << "Connection closed by server.\n";
            close(sockfd);
            exit(0);
        }
        std::cout << buffer << std::endl;
    }
}

void send_messages(int sockfd) {
    std::string message;
    while (true) {
        std::getline(std::cin, message);
        if (message.find("overover") != std::string::npos) {
            std::cout << "Client stop!\n";
            close(sockfd);
            exit(0);
            return;
        }
        send(sockfd, message.c_str(), message.length(), 0);
    }
}
```

上面是客户端的信息发送和接受的操作。接收到信息直接通过 cout 进行输出。在发送信息的时候，我们需要对发送到信息进行特殊判断，我设置了 overover 的退出命令，如果我们输入了 overover，那么就会执行客户端的退出操作，调用 close 的函数，将客户端关闭，但是对于客户端来说，其接收信息和发送信息是采用两个子线程来进行维持的，最后需要使用 exit(0) 来终止主线程，这样就会使得子线程都终结，完成客户端的退出。

3 用户信息的存储

在这次实验中，信息都是通过服务端进行传播，所以信息传播到服务端的时候，我们直接通过 vector 来进行信息的存储，存储后进行广播，本次实验默认服务端是长久在线，如果服务端掉线，那么数据就会丢失。这就是对话的信息存储。

4 总结

本文介绍了一个基于 TCP 协议的简单聊天系统，客户端和服务器之间通过可靠的字节流进行通信。服务端在特定 IP 和端口上监听连接请求，客户端通过连接建立通信。信息通过简单的文本协议

进行传输，支持中英文对话，增强了系统的灵活性。信息广播机制保证了所有连接的客户端能够实时接收消息，同时使用互斥锁确保线程安全。

系统的设计强调了客户端和服务端的分工，通过独立的子线程实现并发处理，有效地支持了信息的发送与接收。然而，当前实现仍存在一些潜在问题，例如信息存储依赖于服务端的稳定性，未能实现持久化存储。因此，未来可以考虑引入数据库来保存聊天记录，以提升系统的可靠性和用户体验。此外，进一步增强消息确认机制、错误处理能力和用户管理功能，将使得该聊天应用更加健壮和用户友好。