

华东师范大学软件学院实验报告

实验课程：计算机网络实践	年级：2023 级本科	实验成绩：
实验名称：Lab 7 Socket Programming	姓名：张梓卫	
实验编号：（7）	学号：10235101526	实验日期：2025/01/03
指导老师：刘献忠	组号：	实验时间：2 课时

目录

一 实验目的	1	8 Step Final: 优化细节	14
二 实验内容与实验步骤	2	8.1 通过 flush 优化缓冲区	14
1 实验要求	2	8.2 使用地址复用防止报错	14
2 实验原理	2	8.3 杀死服务端时关闭 Client	14
3 实验步骤	2	8.4 优化客户端连接	15
三 实验环境	3	9 Step More: 实现广播功能	16
四 实验过程与分析	3	五 实验结果展示	17
1 Step 1: 实现 CClient 和 Server 的初始化	3	1 编译指南	20
2 Step 2: 实现服务端和客户端收发消息	5	2 运行指南	20
3 Step 3: 实现多客户端通信	6	六 附录	20
4 Step 4: 实现从文件中读取	8	1 参考资料	20
5 Step 5: 实现双工通信（客户端）	11	2 完整代码	20
6 Step 6: 实现双工通信（服务端）	12	2.1 客户端代码	20
7 Step 7: 动态端口监听	13	2.2 服务器代码	24

Notice

本 PDF 可以点击目录跳转，您可以点击不同位置以获取相关内容。

一 实验目的

该实验是课程《计算机网络实践》第七次实验，全名《Socket Programming》，目标如下：

Itemize — .1

- 熟悉 Socket 编程的基本原理
- 掌握简单的套接字编程
- 掌握通过 Socket 编程实现 C/S 程序的基本方法
- 了解应用层和运输层的作用及相关协议的工作原理和机制

二 实验内容与实验步骤

1 实验要求

要实现多个 Client 客户端与 Server 服务端之间的 Socket 通信，需要满足以下要求：

• Server 端要求：

Itemize 二 .1

- 能够将 Client 的输入信息进行标准输出打印；
- 支持 5 个以上的 Client 同时发送消息并打印；
- 端口号绑定错误时有报错。

• Client 端要求：

Itemize 二 .2

- 能从标准输入或文件中接受消息；
- 标准输入以 两个回车 作为结束标志；
- 连接至错误的地址或端口时会报错。

• 系统整体要求：

Itemize 二 .3

- 系统容错性好，无闪退；
- 支持在 localhost 以及不同机器上运行；
- 支持长文本 (> 20KB)，有缓存区管理功能。

• Bonus 加分项：

Itemize 二 .4

- 实现 Client 和 Server 之间的双工通信
- 支持双向消息传输。
-

2 实验原理

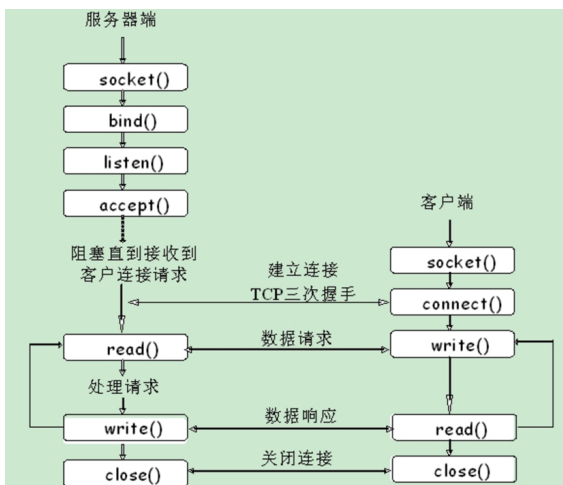


图 1: Socket 通信模型

名称	含义
socket	创建一个通信的管道
bind	把一个地址三元组绑定到 socket 上
listen	准备接受某个 socket 的数据
accept	等待连接到达
connect	主动建立连接
send	发送数据
receive	接受数据
close	关闭连接

表 1: Socket 通信相关操作

3 实验步骤

客户端步骤

- 1、创建套接字
- 2、向服务器发送连接请求 (connect)
- 3、通信 (send/rcv)
- 4、关闭套接字

服务端步骤

- 1、创建用于监听的套接字 (socket)
- 2、将套接字绑定到本地地址和端口上 (bind)
- 3、将套接字设为监听模式 (listen)
- 4、等待客户请求 (accept)，此处要不断的调用 accept

- 5、通信 (send/receive)，完成后返回 4
- 6、关闭套接字 (closesocket)

三 实验环境

使用 Wireshark v4.2.5, Windows 11 Pro, Wget Tools, VSCode, JetBrains Clion, VMWare Workstation 17 Pro - Ubuntu 64 位进行实验。实验报告使用 \LaTeX 进行撰写，使用 Vim 编辑器进行文本编辑。

四 实验过程与分析

Step 0: 运行示例代码

经过以下一系列操作：

```
1 cd
2 vim simplex-talk-server.c
3 vim simplex-talk.c
4 gcc -o server-talk simplex-talk-server.c
5 gcc -o talk simplex-talk.c
6 ./server-talk localhost
7 ./talk localhost
```

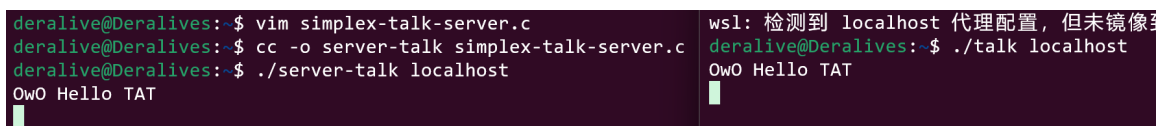


图 2: Example Run

我们成功运行了示例，并且看到了 Client 和 Server 端的基本交互。
接下来我们自己动手开始从零实现 Socket 通信。

1 Step 1: 实现 Client 和 Server 的初始化

头文件的导入

注意，在 Linux 平台之下，`int socket(int af, int type, int protocol);`
但在 Windows 平台下，是没有 `arpa/inet.h` 的，取而代之的是 `winsocket2.h`。
我们需要用到的头文件有：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <strings.h>
5 #include <unistd.h>
6 #include <arpa/inet.h>
```

Include Header

Socket 的创建

`socket()` 函数可以创建要给本地套接字，如果创建失败，报错。

```
1 // af = AF_INET (IPv4), AF_INET6 (IPv6)
2 // type = SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
3 // protocol = 0 (default), IPPROTO_TCP (TCP), IPPROTO_UDP (UDP)
4 int client_socket;
5 client_socket = socket(AF_INET, SOCK_STREAM, 0);
6 if (client_socket < 0) {
```

```
7     perror("Socket creation failed");
8     exit(EXIT_FAILURE);
9 }
```

Create Socket

服务器端配置

我们按照示例文件照猫画虎，即可分析出配置服务器的办法，使用 `struct sockaddr_in` 来定义服务器。

```
1 #define SERVER_PORT 12345
2 // 配置服务器地址
3 struct sockaddr_in server_addr;
4 bzero((char *)&server_addr, sizeof(server_addr)); // 初始化用来清空结构体，保证安全一致
5 server_addr.sin_family = AF_INET;
6 server_addr.sin_addr.s_addr = INADDR_ANY; // 表示本机的所有网卡 IP
7 server_addr.sin_port = htons(SERVER_PORT); // htons 将端口号转换为网络字节序
```

Server Configuration

创建并绑定 Socket

```
1 // 创建Socket，同 Client 端一样的创建方法
2 int server_socket;
3 if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
4     perror("Socket creation failed");
5     exit(EXIT_FAILURE);
6 }
```

绑定 Socket，使用 `bind()` 函数告诉操作系统应该监听哪个网络接口和端口

```
1 if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
2     perror("Bind failed");
3     close(server_socket); // 如果绑定失败，就把套接字关闭
4     exit(EXIT_FAILURE);
5 }
```

监听连接，让套接字处于被动监听状态。直到客户端发起请求才会被唤醒。

```
1 if (listen(server_socket, MAX_PENDING) < 0) {
2     perror("Listen failed");
3     close(server_socket); // 这边都是一样的，如果出现错误就把服务器关了
4     exit(EXIT_FAILURE);
5 }
6
7 printf("Server is listening on port %d\n", SERVER_PORT); // 成功，输出提示信息
```

接受客户端

使用 `while` 死循环来卡死，因为 `accept()` 函数是阻塞的，它会从已经在监听的套接字队列中取出一个客户端连接请求，并返回一个新的套接字文件描述符，用于与该客户端进行通信。

但是我们又需要可以有多个客户端连接进来。

```
1 // 等待客户端连接
2 while (1) {
3     client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_addr_len);
4     if (client_socket < 0) {
5         perror("Accept failed");
6         continue; // 若接受新的客户端失败，则进入下一次循环
7     }
8     printf("New client connected.\n");
9 }
```

Accept Client

注意，写到这里时，可以发现如果有第二个客户端接入，`client_socket` 就会被覆盖，所以在当前的写法之下，是只能接受一个客户端的。我们不妨先从这里开始接着写收发逻辑，然后测试一下看看能不能运行。

回到客户端中连接服务端

配置客户端连接到服务端时，也要像服务端那样配置地址，然后调用 `connect()` 函数。注意，`inet_pton()` 函数是用于将十进制的点分地址转换为二进制的服务器地址的。

```
1 // 配置服务器地址
2 struct sockaddr_in server_addr;
3 bzero((char *)&server_addr, sizeof(server_addr));
4 server_addr.sin_family = AF_INET;
5 server_addr.sin_port = htons(SERVER_PORT);
6
7 // 以下都是简单的逻辑了，这里使用的 argv[1] 实际上从命令行获取的服务器地址
8 // 调用示例：./client 127.0.0.1
9 if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0) {
10     perror("Invalid server address");
11     close(client_socket);
12     exit(EXIT_FAILURE);
13 }
14
15 // 连接服务器
16 if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
17     perror("Connection failed");
18     close(client_socket);
19     exit(EXIT_FAILURE);
20 }
21 printf("Connected to server. Enter your message (end with double ENTER):\n");
```

Connect to Server

我们在函数内添加了较多的 `perror()` 语句，用于输出错误信息。并且提供了 `printf()` 输出提示信息。

2 Step 2: 实现服务端和客户端收发消息

从标准流读取输入

从标准输入读取消息，直到两个回车为结束标志，我们实现的思路是使用一个 `buffer` 存储总的字符数组，然后使用 `fgets()` 来逐行读取，并追加至 `buffer` 数组中。

如果连续碰到了两个 `Enter`，则退出。`fgets()` 是安全的，它能够限制读取的最大字节数，防止缓冲区溢出。

```
1 void read_input(char *buffer) {
2     char temp[MAX_LINE];
3     int idx = 0;
4     while (fgets(temp, sizeof(temp), stdin)) { // 从标准输入流中读取
5         // 如果是连续两个回车，移除最后一个回车，保证字符串正确
6         if (strcmp(temp, "\n") == 0 && idx > 0 && buffer[idx - 1] == '\n') {
7             buffer[idx - 1] = '\0'; // 移除最后一个回车
8             return; // 此时可以 return，代表结束标志
9         }
10
11         // 如果一行里输入得太长了，报错并退出
12         if (idx + strlen(temp) >= MAX_LINE) {
13             fprintf(stderr, "Input too large.\n");
14             exit(EXIT_FAILURE);
15         }
16
17         strcpy(buffer + idx, temp); // 追加本行内容到 buffer 数组中
```

```
18     idx += strlen(temp);
19 }
20 }
```

Read Input

发送至服务端

发送消息到服务端，使用 `send()` 函数，将 `buffer` 中的内容发送给服务端。

```
1 while (1) {
2     memset(buffer, 0, sizeof(buffer));
3     printf("> "); // 给用户处于输入状态的提示
4     read_input(buffer); // 调用刚刚写好的函数来从标准输入流中读取
5     send(client_socket, buffer, strlen(buffer), 0);
6 }
```

Send Message

当然，服务端也应该对客户端的消息进行接收，并进行处理。这里我们使用 `recv` 函数来进行处理：

```
1 char buffer[MAX_LINE];
2 while (1) {
3     memset(buffer, 0, sizeof(buffer)); // 清空 buffer
4     ssize_t bytes_received = recv(client_socket, buffer, sizeof(buffer) - 1, 0);
5     if (bytes_received <= 0) {
6         printf("Client disconnected.\n");
7         close(client_socket);
8         break; // 跳出循环，等待下一个客户端
9     }
10    printf("Received: %s\n", buffer); // 打印出接收到的客户端的消息
11 }
```

Receive Message

这一层代码应该写入到一个循环中，在通过建立了 Socket 连接后，循环不断的接收客户端的消息，并进行处理。



```
deralive@Deralives:~$ ./server_init
Server is listening on port 12345
New client connected.
Received: Message From Client_Test

deralive@Deralives:~$ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Message From Client_Test
>
```

图 3: 第一次通信成功

目前为止，我们已经实现了单工通信，满足了以下要求：

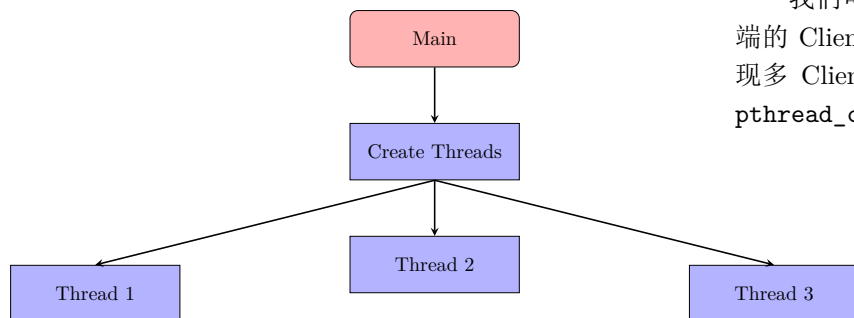
Question

- Server: 能够将 Client 的输入信息进行标准输出打印；
- Client: 能从标准输入中接收消息；
- Client: 标准输入以 两个回车作为结束标志；
- Client: 连接至错误的地址或端口时会报错。
- System: 系统容错性好，无闪退。
- System: 支持在 `localhost` 上运行。

3 Step 3: 实现多客户端通信

线程创建

创建多个 Socket



我们可以考虑这样一种结构,将每一个发送请求接入服务端的 Client 都放置入一个线程当中,通过多线程的方式来实现在多 Client 通信。这里需要放在之前写的 while 循环当中, `pthread_create()` 可以实现这个想法。

它的函数原型如下,其中需要提供一个函数指针,代表这个新建的线程的函数入口。

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
```

查阅资料发现, `int pthread_detach(pthread_t thread);` 也有重要作用,具体如代码注释所述。

```

1 while (1) {
2     // 接受客户端连接
3     client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_addr_len);
4     if (client_socket < 0) {
5         perror("Accept failed");
6         continue;
7     }
8     printf("New client connected.\n");
9
10    // 创建线程处理客户端
11    pthread_t thread_id;
12    int *client_socket_ptr = malloc(sizeof(int));
13    *client_socket_ptr = client_socket;
14
15    if (pthread_create(&thread_id, NULL, client_handler, client_socket_ptr) != 0) {
16        perror("Thread creation failed");
17        free(client_socket_ptr);
18        close(client_socket);
19    }
20
21    pthread_detach(thread_id);
22    // 这个函数是线程分离用的。将线程标记为分离状态。
23    // 线程执行结束后系统会自动回收资源,无需通过 pthread_join() 等待线程结束。
24    // 分离线程适用于短期任务,主线程不关心线程的返回值或执行状态。
25 }
  
```

Thread Creation

这里我们定义了 `client_handler()` 函数作为线程的入口,它负责处理每个客户端的连接,可以把处理信息接受的逻辑放在这里:

另外要注意,我们要把各个 `client_socket` 分开,所以采用了一个巧妙的设计:

解答 四.1

- 每个线程接收一个不同的指针 `client_socket_ptr`, 指向各自动态分配的内存区域。
- `*(int *)client_socket_ptr` 提取该线程独立的套接字值。
- 调用 `free(client_socket_ptr)` 释放动态分配的内存, 避免内存泄漏。
- 内存被释放后, 每个线程内部仍然持有 `client_socket` 的副本 (值被解引用并存储在局部变量 `client_socket` 中), 互不影响。

```

1 // POSIX 线程库的线程函数必须接受 void* 类型参数，因此需要强制转换。
2 void *client_handler(void *client_socket_ptr) {
3     int client_socket = *(int *)client_socket_ptr; // 动态分配内存，避免 client_socket 被覆盖
4     free(client_socket_ptr);
5
6     char buffer[MAX_LINE];
7     ssize_t bytes_received;
8
9     while (1) {
10         memset(buffer, 0, sizeof(buffer)); // 接受消息之前，将缓存区清空，防止信息混乱
11         bytes_received = recv(client_socket, buffer, sizeof(buffer) - 1, 0);
12         // 确保保留最后一个字节给字符串的终止符 '\0'，最后的参数 0 为标志位
13         if (bytes_received <= 0) {
14             printf("Client disconnected.\n");
15             break;
16         }
17         printf("Received from client: %s\n", buffer);
18     }
19     close(client_socket);
20 }

```

Thread Handler

Question

至此，我们已经实现支持 5 个以上的 Client 同时发送消息并逐一打印！

```

ws1: 检测到 localhost 代理配置，但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
deralive@Deralives: ~
deralive@Deralives: $ ./server_init
Server is listening on port 12345
New client connected.
New client connected.
New client connected.
New client connected.
New client connected.
New client connected.
Received from client: Message From Client 1
Received from client: Message From Client 2
Received from client: Message From Client 3
Received from client: Message From Client 4
Received from client: Message From Client 5
Received from client: Message From Client 6
deralive@Deralives: ~
deralive@Deralives: $ ./client_init 127.0.0.1
Connection failed: Connection ref used
deralive@Deralives: $ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Message From Client 1
deralive@Deralives: ~
deralive@Deralives: $ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Message From Client 2
deralive@Deralives: ~
deralive@Deralives: $ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Message From Client 3
deralive@Deralives: ~
deralive@Deralives: $ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Message From Client 4
deralive@Deralives: ~
deralive@Deralives: $ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Message From Client 5
deralive@Deralives: ~
deralive@Deralives: $ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Message From Client 6

```

图 4: 多客户端通信

4 Step 4: 实现从文件中读取

在我的设计里，Client 进入了连接 Server 的状态后，是一直处于 ">" 读取用户输入流的。

所以，只需要在输入流中检测是否出现 **FILE:** 字段，并且读取后面跟随的文件名，再使用函数 `fopen()` 打开文件，并

将文件内容发送给 Server 即可。

上面实现的逻辑很简单，可是应该怎样从文件中读取非常大的文本内容呢？

优化思路

我们可以考虑使用 `fread()` 函数来逐块读取文件内容，并逐块发送给 Server。

Notice

在函数 `bytes_read = fread(file_buffer, 1, sizeof(file_buffer), file);` 中：

- 第一个参数 `file_buffer` 是用于存储读取数据的缓冲区。
- 第二个参数 `1` 是单个数据项的大小（字节数），这里是逐字节读取。
- 第三个参数 `sizeof(file_buffer)` 是每次读取的最大数据项数，等于缓冲区大小（1024 字节）。
- `fread` 的返回值是实际读取到的字节数。

```

1 char file_buffer[1024]; 定义分块大小
2 size_t bytes_read;
3 while ((bytes_read = fread(file_buffer, 1, sizeof(file_buffer), file)) > 0) {
4     ssize_t bytes_sent = send(client_socket, file_buffer, bytes_read, 0); // 逐块发送文件内容
5     // 循环读取文件内容，直到文件结束（fread 返回值小于缓冲区大小或为 0）。
6     if (bytes_sent < 0) {
7         perror("Failed to send file");
8         break;
9     }
10 }
```

Read from File

完整代码

根据上述的分析，我们可以将这部分判断是否需要从文件中读取的代码放在主循环中。

```

1 while (1) {
2     memset(buffer, 0, sizeof(buffer));
3     printf("> ");
4     fflush(stdout); // 刷新缓冲区，确保提示符显示
5     read_input(buffer); // 读取输入
6
7     // 如果是从文件中读取文本发送
8     if (strncmp(buffer, "FILE:", 5) == 0) {
9         char filename[MAX_LINE];
10        sscanf(buffer + 5, "%s", filename); // 提取文件名
11
12        FILE *file = fopen(filename, "r");
13        if (!file) {
14            perror("Failed to open file");
15            continue;
16        }
17
18        // 从文件中读取文本并发送
19        char file_buffer[1024];
20        size_t bytes_read;
21        while ((bytes_read = fread(file_buffer, 1, sizeof(file_buffer), file)) > 0) {
22            ssize_t bytes_sent = send(client_socket, file_buffer, bytes_read, 0); // 逐块发送文件内容
23            if (bytes_sent < 0) {
24                perror("Failed to send file");
25                break;
26            }
27        }
28    }
```


Question

至此，我们已经实现 Client 能从标准输入或文件中接受消息。

5 Step 5: 实现双工通信（客户端）

考虑到实际上，在服务端同时接收多个客户端的信息时，实际上采用的是多个线程来处理的。

那么我们可以在每一个创建的 Client 中，新建一个线程来循环检测并接收服务端发来的信息。这样就能够实现双工通信了。

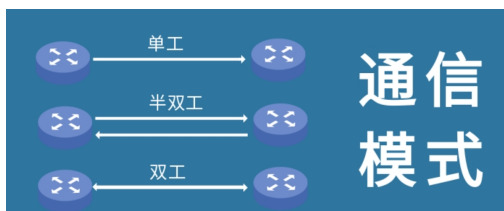


图 7: 双工通信

因为在这边当 $n = 0$ 时，代表了服务端关闭，所以我们可以顺着实现一个功能：在服务端关闭时，发送一条消息给客户端，提示服务端已经关闭。

```

1  typedef struct {
2      int sockfd;
3  } thread_args_t;
4
5  // 接收线程函数：持续从服务器接收消息并打印
6  void *recv_thread_func(void *arg) {
7      thread_args_t *targs = (thread_args_t *)arg;
8      int sockfd = targs->sockfd; // 获取套接字描述符
9      char recv_buffer[1024];
10     ssize_t n;
11
12     while (1) {
13         memset(recv_buffer, 0, sizeof(recv_buffer));
14         n = recv(sockfd, recv_buffer, sizeof(recv_buffer) - 1, 0); // 0 表示阻塞接收模式
15         // 返回值为 0，表示服务器关闭了连接
16         if (n <= 0) {
17             if (n < 0) {
18                 fprintf(stderr, "Error receiving message: %s\n", strerror(errno));
19             } else {
20                 fprintf(stderr, "Server closed connection.\n");
21             }
22             break;
23         }
24         printf("\n[Message from server]: %s\n> ", recv_buffer);
25         fflush(stdout);
26     }
27     return NULL;
28 }

```

Client Duplex

以上是该新建的线程从服务端不断接收消息并打印的函数，接下来我们只需要在 main 函数中，在连接到服务器之后创建该线程即可。

由于线程函数 `recv_thread_func` 的参数是 `void *`，只能传递单个指针，使用结构体封装参数能够传递多个值。

```

1  pthread_t recv_tid;
2  thread_args_t targs;
3  targs.sockfd = client_socket; // 传递套接字描述符
4  if (pthread_create(&recv_tid, NULL, recv_thread_func, &targs) != 0) {

```

```

5     fprintf(stderr, "Create receive thread failed: %s\n", strerror(errno));
6     close(client_socket);
7     exit(EXIT_FAILURE);
8 }
9 pthread_detach(recv_tid);

```

Client Duplex

6 Step 6: 实现双工通信（服务端）

我初步的设定是，在获得每一个 Client 发送的消息之后，添加消息回显（如时间戳等），这是很经典的双工通信的实现样式。这部分实现逻辑应该放置在之前写的 `client_handler()` 函数中，因为是回显给对应发送过来的客户端的，而非广播。

```

1  #include<time.h>
2
3  // 获取当前时间
4  time_t now = time(NULL);
5  struct tm *t = localtime(&now);
6  char time_str[64];
7  strftime(time_str, sizeof(time_str)-1, "%Y-%m-%d %H:%M:%S", t);
8
9  // 构造回复消息
10 char reply[MAX_LINE];
11 snprintf(reply, sizeof(reply), "Server received your message: \"%s\" at %s\n", buffer, time_str);
12
13 // 发送回复消息给客户端
14 ssize_t bytes_sent = send(client_socket, reply, strlen(reply), 0);
15 if (bytes_sent < 0) {
16     perror("Send response message failed");
17     break;
18 }

```

Server Duplex

实现结果图

可以看到，创建了多个客户端，并且客户端在发送消息至服务端时，服务端会回显消息。服务器使用 Ctrl + C 中断关闭时，所有已连接到服务端的客户端都会显示“Server closed connection”信息。

```

wsl: 检测到 localhost 代理配置，但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
deralive@Deralives: ~$ ./server_init
Server is listening on port 12345
New client connected.
New client connected.
Received from client: Hello!
Received from client: World!
^C
deralive@Deralives: ~$

wsl: 检测到 localhost 代理配置，但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
deralive@Deralives: ~$ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> Hello!

[Message from server]: Server received your message: "Hello!" at 2024-12-20 13:27:03
> Server closed connection.

wsl: 检测到 localhost 代理配置，但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
deralive@Deralives: ~$ ./client_init 127.0.0.1
Connected to server. Enter your message (end with double ENTER):
> World!

[Message from server]: Server received your message: "World!" at 2024-12-20 13:27:07
> Server closed connection.

```

图 8: 双工通信

7 Step 7: 动态端口监听

之前我们是把端口写死的（12345），而实验要求中需要将端口作为一个参数来传递。

```

1 // 服务端的修改
2 #define DEFAULT_SERVER_PORT 12345 // 我们先设置一个默认的端口，当命令行没有传入时就用这个
3 int main(int argc, char *argv[]) {
4     int server_port = DEFAULT_SERVER_PORT;
5     // 解析命令行参数
6     if (argc == 2) {
7         server_port = atoi(argv[1]);
8         if (server_port <= 0 || server_port > 65535) {
9             fprintf(stderr, "Invalid port number: %s\n", argv[1]);
10            exit(EXIT_FAILURE);
11        }
12    } else if (argc > 2) {
13        fprintf(stderr, "Usage: %s [port]\n", argv[0]);
14        exit(EXIT_FAILURE);
15    }
16 }

```

```

1 // 客户端的修改
2 #define DEFAULT_SERVER_PORT 12345
3 int main(int argc, char *argv[]) {
4     int server_port = DEFAULT_SERVER_PORT;
5     char *server_ip;
6     if (argc == 3) {
7         server_ip = argv[1];
8         server_port = atoi(argv[2]);
9         if (server_port <= 0 || server_port > 65535) {
10            fprintf(stderr, "Invalid port number: %s\n", argv[2]);
11            exit(EXIT_FAILURE);
12        }
13    } else {
14        fprintf(stderr, "Usage: %s <server_ip> <server_port>\n", argv[0]);
15        exit(EXIT_FAILURE);
16    }
17 }

```

注意到，当右下角的客户端试图连接到 26420 端口时，会提示此时 Connection Refuse。

```

wsl: 检测到 localhost 代理配置，但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
deralive@Deralives: ~$ ./server_init 26421
Server is listening on port 26421
New client connected.
New client connected.
Received from client: 666 at client 1
Received from client: 777 at client 2
^C
deralive@Deralives: ~$

wsl: 检测到 localhost 代理配置，但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
deralive@Deralives: ~$ ./client_init 127.0.0.1 26421
Connected to server 127.0.0.1:26421.
. Enter your message (end with double ENTER):
> 666 at client 1

>
[Message from server]: Server received your message: "666 at client 1" at 2024-12-20 13:43:55

> Server closed connection.

deralive@Deralives: ~$ ./client_init 127.0.0.1 26420
Connection failed: Connection refused
deralive@Deralives: ~$ ./client_init 127.0.0.1 26421
Connected to server 127.0.0.1:26421.
. Enter your message (end with double ENTER):
> 777 at client 2

>
[Message from server]: Server received your message: "777 at client 2" at 2024-12-20 13:44:00

> Server closed connection.

```

图 9: 动态端口

Question

目前，我们已经实现了：

- 实现了多客户端与服务端之间的双工通信
- 实现了动态端口监听
- 连接到错误端口时会输出提示信息

8 Step Final: 优化细节

8.1 通过 fflush 优化缓冲区

立即清除缓冲区，可以使得提示符马上显示：

```
1 printf("> ");
2 fflush(stdout); // 清空缓冲区，立即显示提示符
```

fflush

8.2 使用地址复用防止报错

SO_REUSEADDR 是套接字的一个选项，用于允许重新绑定地址，即使之前的连接还未完全关闭。

在 TCP 协议中，当关闭一个套接字后，连接可能会进入 TIME_WAIT 状态（通常持续 1-4 分钟）。在 TIME_WAIT 状态下，该端口的地址仍被占用，导致重新启动服务器时，绑定同一端口会报错。

我们可以在创建 Socket 之后进行设置地址复用，解决这个问题。

```
1 // 创建Socket
2 if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
3     perror("Socket creation failed");
4     exit(EXIT_FAILURE);
5 }
6
7 // 复用地址避免重启服务器时的"Address already in use"
8 int opt = 1;
9 setsockopt(server_addr, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

Address Reuse

8.3 杀死服务端时关闭 Client

在服务端使用 Ctrl + C 关闭时，只是关闭了 socket 连接，然而 Client 仍然还处于阻塞状态，这是由于不断读取消息的 while 循环导致的。

所以本来写的 **while(1)** 我想更换为 **while(running)**，通过设置一个标志位来保证客户端程序的退出。

所有的 while 循环中都加入一个 running 的判断条件，就可以避免阻塞。

```
1 volatile sig_atomic_t running = 1; // 全局标志变量，用于控制主线程的退出
2
3 // 客户端修改位置：
4 if (n <= 0) {
5     if (n < 0) {
6         fprintf(stderr, "Error receiving message: %s\n", strerror(errno));
7     } else {
8         fprintf(stderr, "Server closed connection.\n");
9     }
10    running = 0; // 设置全局标志变量为 0，通知主线程退出
11    close(sockfd); // 关闭套接字
12    break;
13 }
```

```

14
15 while (running) {
16     if {
17         // 判断是使用文字输入还是文件输入，并发送消息
18     }
19     close(client_socket);
20     return 0;
21 }

```

Kill Client

8.4 优化客户端连接

connect 函数会阻塞，直到成功连接到目标地址或发生错误（如超时或拒绝连接）。如果输入的是一个无效的 IP 地址（如 asas），或者是一个无法路由的地址，connect 会一直等待。

所以我们可以添加一个连接超时的设置，这样使得客户端在连接到不对的 IP 地址时，过了指定的时间就会自动退出。

因为有时候我们也有使用域名进行连接的需求，而非简单的 IP 地址，这时候可以使用 netdb.h 库中的 getaddrinfo 来实现这个需求。

```

1  # include <netdb.h>
2
3  // 使用 getaddrinfo 解析域名或 IP 地址
4  struct addrinfo hints, *res;
5  memset(&hints, 0, sizeof(hints));
6  hints.ai_family = AF_INET; // IPv4
7  hints.ai_socktype = SOCK_STREAM; // TCP
8
9  int ret = getaddrinfo(server_ip, NULL, &hints, &res);
10 if (ret != 0) {
11     fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(ret));
12     close(client_socket);
13     exit(EXIT_FAILURE);
14 }
15
16 struct sockaddr_in *addr_in = (struct sockaddr_in *)res->ai_addr;
17 server_addr.sin_family = AF_INET;
18 server_addr.sin_addr = addr_in->sin_addr;
19 server_addr.sin_port = htons(server_port);
20
21 freeaddrinfo(res);
22
23 struct timeval timeout;
24 timeout.tv_sec = 3; // 超时时间为 3 秒
25 timeout.tv_usec = 0;
26 setsockopt(client_socket, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof(timeout));
27 // 这里只设置发送的超时时间，不要设置接收的 SO_RCVTIMEO 超时，否则连接上了也会断开
28
29 // 尝试连接
30 if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
31     ...
32 }

```

Connect


```
32 pthread_mutex_lock(&client_list_mutex);
33 for (int i = 0; i < MAX_CLIENTS; i++) {
34     if (client_sockets[i] == client_socket) {
35         client_sockets[i] = 0;
36         break;
37     }
38 }
39 pthread_mutex_unlock(&client_list_mutex);
```

Server Broadcast

五 实验结果展示

Notice

- 连接至错误的 IP 地址 / 端口号时会提示出错信息
- 能在标准输出打印客户端发送的消息
- 标准输入消息以两次回车为结束标志

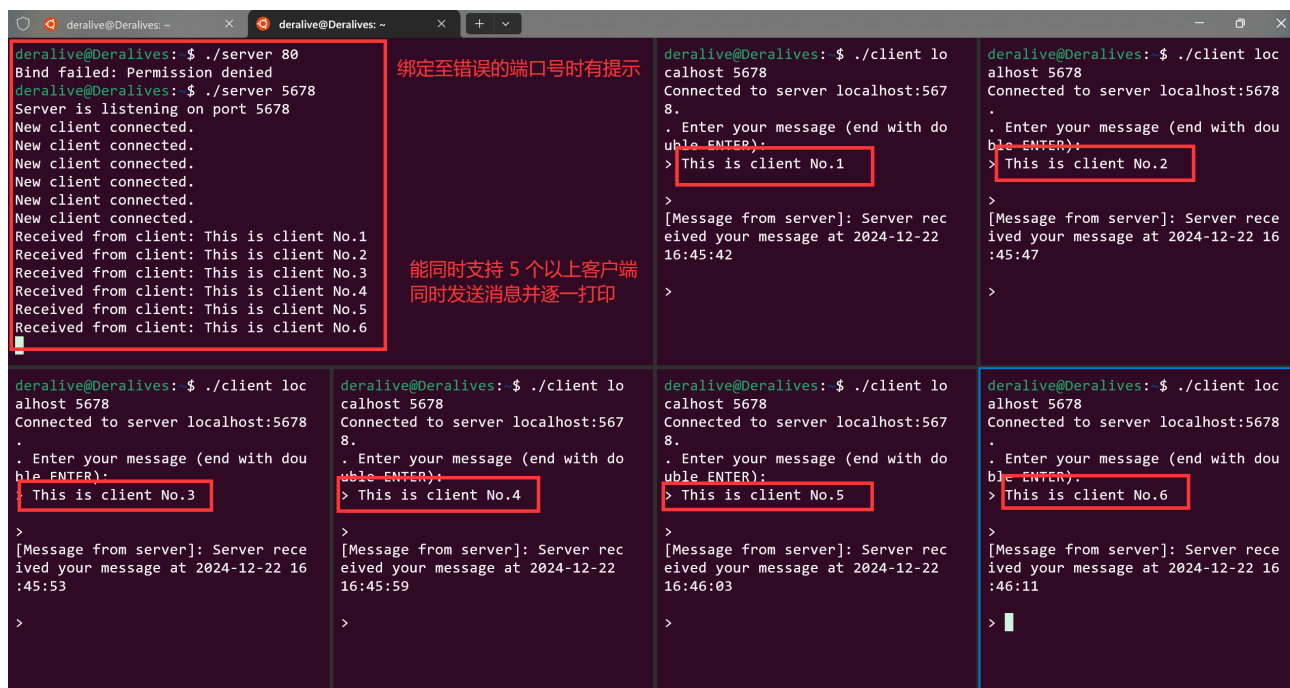
```
deralive@Deralives: ~$ ./server www.baidu.com
Invalid port number: www.baidu.com
deralive@Deralives: ~$ ./server www.baidu.com 999
Usage: ./server [port]
deralive@Deralives: ~$ ./server 5678
Server is listening on port 5678
New client connected.
Received from client: Hello, I'm Client 1, This is the first message.
Received from client: Hello I'm Client 2, This is the second message.
能够在标准输出打印客户端发送的消息

deralive@Deralives: ~$ ./client www.baidu.com
Usage: ./client <server_ip> <server_port>
deralive@Deralives: ~$ ./client www.baidu.com 5678
Connection failed: Operation now in progress
deralive@Deralives: ~$ ./client localhost 5678
Connected to server localhost:5678.
. Enter your message (end with double ENTER):
> Hello, I'm Client 1, This is the first message.
标准输入消息以两次回车为结束标志
[Message from server]: Server received your message at 2024-12-22 16:37:43
>

deralive@Deralives: ~$ ./client 180.160.66.66 5678
Connection failed: Operation now in progress
deralive@Deralives: ~$ ./client 127.0.0.1 5678
Connected to server 127.0.0.1:5678.
. Enter your message (end with double ENTER):
> Hello I'm Client 2, This is the second message.
[Message from server]: Server received your message at 2024-12-22 16:38:03
>
```

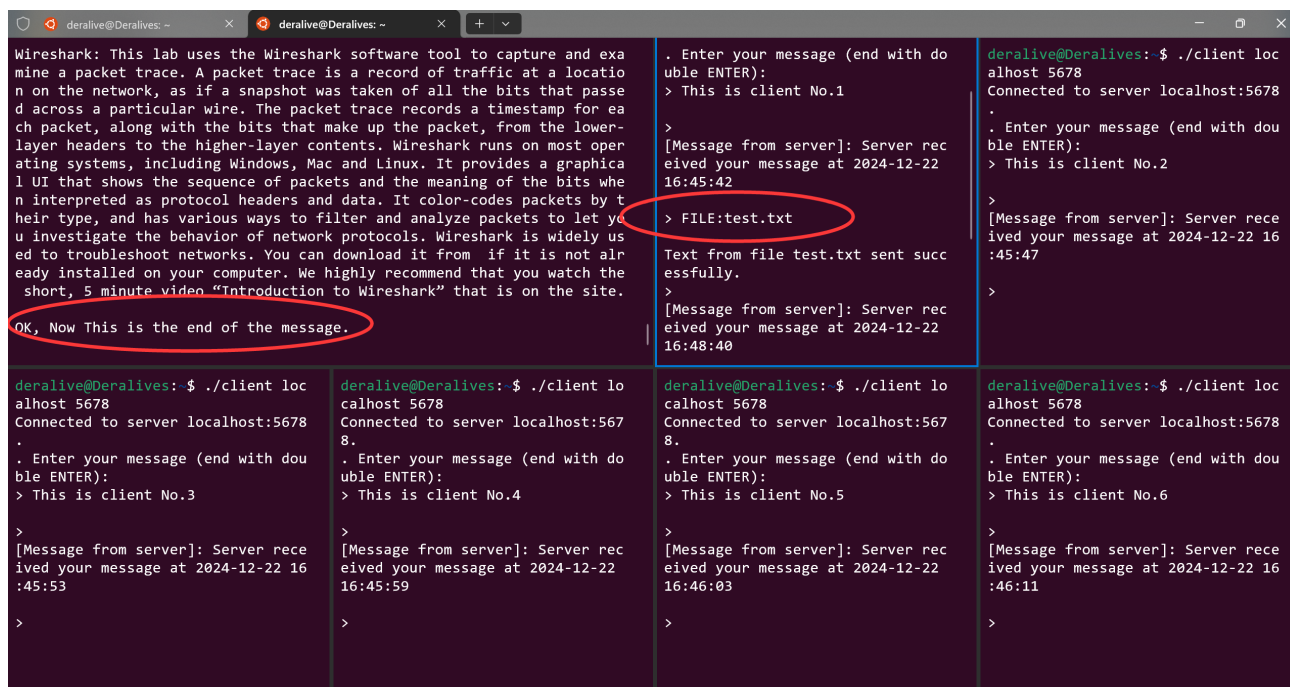
Notice

- 绑定至错误的端口号时会提示出错信息
- 能从标准输入或文件中接收消息
- 支持 5 个以上客户端同时发送消息并逐一打印
- 支持长文本消息，有缓冲区管理

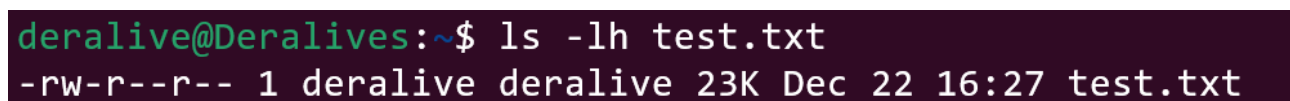


Notice

- 能从标准输入或文件中接收消息
- 支持长文本消息，有缓冲区管理



其中，使用 `ls` 命令可以查看文件的大小：



Notice

- 容错性好，无闪退
- 支持双工通信

服务端不仅可以接收到客户端的消息，还能及时反馈一条消息告诉客户端现在收到消息的时间戳。

同时，我还添加了服务端向客户端广播的功能，如果一个客户端想要发送广播给其他客户端，使用 **BROADCAST:** 命令即可。

```

ating systems, including Windows, Mac and Linux. It provides a graphical UI that shows the sequence of packets and the meaning of the bits when interpreted as protocol headers and data. It color-codes packets by their type, and has various ways to filter and analyze packets to let you investigate the behavior of network protocols. Wireshark is widely used to troubleshoot networks. You can download it from if it is not already installed on your computer. We highly recommend that you watch the short, 5 minute video "Introduction to Wireshark" that is on the site.

OK, Now This is the end of the message.

Received from client: BROADCAST: Hello, I'm Client No.2
^C
deralive@Deralives:~$

Connected to server localhost:5678
. Enter your message (end with double ENTER):
> This is client No.3

[Message from server]: Server received your message at 2024-12-22 16:45:53
>

[Message from server]: Hello, I'm Client No.2
> Server closed connection.
Input anything and <Enter> to exit.

Connected to server localhost:5678
. Enter your message (end with double ENTER):
> This is client No.4

[Message from server]: Server received your message at 2024-12-22 16:45:59
>

[Message from server]: Hello, I'm Client No.2
> Server closed connection.
Input anything and <Enter> to exit.

Connected to server localhost:5678
. Enter your message (end with double ENTER):
> This is client No.5

[Message from server]: Server received your message at 2024-12-22 16:46:03
>

[Message from server]: Hello, I'm Client No.2
> Server closed connection.
Input anything and <Enter> to exit.

Connected to server localhost:5678
. Enter your message (end with double ENTER):
> This is client No.6

[Message from server]: Server received your message at 2024-12-22 16:46:11
>

[Message from server]: Hello, I'm Client No.2
> Server closed connection.
Input anything and <Enter> to exit.
  
```

Notice

- 支持在 localhost 和不同机器上运行

我使用我的另一台电脑安装了适用于 WSL 的 Windows 子系统，在这里编译使用了我编写的代码，成功运行。

```

deralive-side@DeralivesLaptop:~$ gcc -o client client.c
/usr/bin/ld: /tmp/ccGuC09N.o: in function 'main':
client.c:(.text+0x5d5): undefined reference to 'pthread_create'
/usr/bin/ld: client.c:(.text+0x62b): undefined reference to 'pthread_detach'
collect2: error: ld returned 1 exit status
deralive-side@DeralivesLaptop:~$ gcc -o client client.c -pthread
/usr/bin/ld: cannot find -pthread
collect2: error: ld returned 1 exit status
deralive-side@DeralivesLaptop:~$ gcc -o client client.c -lpthread
deralive-side@DeralivesLaptop:~$ vim server.c
deralive-side@DeralivesLaptop:~$ gcc -o server server.c -lpthread
deralive-side@DeralivesLaptop:~$
  
```

```

deralive-side@DeralivesLaptop:~$ ./server 8877
Server is listening on port 8877
New client connected.
Received from client: Hello, I'm Deralive-Side PC

deralive-side@DeralivesLaptop:~$ ./client localhost 8877
Connected to server localhost:8877.
. Enter your message (end with double ENTER):
> Hello, I'm Deralive-Side PC

[Message from server]: Server received your message at 2024-12-22 17:51:10
>
  
```

能在标准输出打印客户端发送的消息（20 分）	支持 5 个以上客户端同时发送消息并逐一打印（20 分）
✓	✓
能从标准输入或文件接收消息（20 分）	标准输入消息以两次回车作为结束标志（5 分）
✓	✓
绑定至错误的端口号时能提示出错信息（5 分）	连接至错误的 IP 地址/端口号时能提示出错信息（5 分）
✓	✓
支持在 localhost 及在两台不同机器上运行（10 分）	支持长文本消息（不少于 20KB），有缓冲区管理（10 分）
✓	✓
容错性好，无闪退（5 分）	支持双工通信（5 分）
✓	✓

表 2: 评分表

1 编译指南

由于我们已经实现了全双工通信，此时用到了多线程处理，因此需要使用 pthread 库。按照中国科学技术大学的 Linux Socket 编程指南，我们需要在最后添加 -pthread 编译选项：
依次使用以下命令：

```
1 $ gcc -o client client.c -pthread
2 $ gcc -o server server.c -pthread
```

Notice

后来我发现，在我的主电脑上，其实不加也可以... 应该是因为某些 Linux 正式发行版中已经预置了常用了链接库。但是在我的副电脑上编译时，如果不加入这个参数就会报链接错误。

2 运行指南

```
1 > ./server [port]
```

```
1 > ./client <server_ip> <server_port> or > ./client localhost <server_port>
2 > FILE:<file_name> # 需要加后缀
3 > BROADCAST:<message> # 发送广播消息
```

六 附录

1 参考资料

- 中科大 Linux Socket 编程指南: http://staff.ustc.edu.cn/~mengning/np/linux_socket/new_page_4.htm
- Linux Manual Page: <https://man7.org/linux/man-pages/man7/pthreads.7.html>

2 完整代码

2.1 客户端代码 (client.c)

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <strings.h>
5  #include <unistd.h>
6  #include <arpa/inet.h>
7  #include <errno.h>
8  #include <signal.h>
9  #include <pthread.h>
10 #include <netdb.h> // for getaddrinfo
11
12 #define DEFAULT_SERVER_PORT 12345
13 #define MAX_LINE 20480
14
15 typedef struct {
16     int sockfd;
17 } thread_args_t;
18
19 volatile sig_atomic_t running = 1; // 全局标志变量，用于控制主线程的退出
20
21 // 接收线程函数：持续从服务器接收消息并打印
22 void *recv_thread_func(void *arg) {
23     thread_args_t *targs = (thread_args_t *)arg;
24     int sockfd = targs->sockfd; // 获取套接字描述符
25     char recv_buffer[1024];
26     ssize_t n;
27
28     while (running) {
29         memset(recv_buffer, 0, sizeof(recv_buffer));
30         n = recv(sockfd, recv_buffer, sizeof(recv_buffer) - 1, 0); // 0 表示阻塞接收模式
31         // 返回值为 0，表示服务器关闭了连接
32         if (n <= 0) {
33             if (n < 0) {
34                 fprintf(stderr, "Error receiving message: %s\n", strerror(errno));
35             } else {
36                 fprintf(stderr, "Server closed connection.\n Input anything and <Enter> to exit.");
37             }
38             running = 0;
39             close(sockfd);
40             exit(EXIT_SUCCESS);
41         }
42         printf("\n[Message from server]: %s\n> ", recv_buffer);
43         fflush(stdout);
44     }
45     return NULL;
46 }
47
48 // 从标准输入读取消息，直到两个回车为结束标志
49 void read_input(char *buffer) {
50     char temp[MAX_LINE];
51     int idx = 0;
52     while (fgets(temp, sizeof(temp), stdin) && (running)) { // 从标准输入流中读取
53         // 如果是连续两个回车，移除最后一个回车，保证字符串正确
54         if (strcmp(temp, "\n") == 0 && idx > 0 && buffer[idx - 1] == '\n') {
55             buffer[idx - 1] = '\0'; // 移除最后一个回车
56             return; // 此时可以 return，代表结束标志
57         }
58
59         // 如果一行里输入得太长了，报错并退出
60         if (idx + strlen(temp) >= MAX_LINE) {
61             fprintf(stderr, "Input too large.\n");
62             exit(EXIT_FAILURE);
63         }

```

```

64     strcpy(buffer + idx, temp); // 追加本行内容到 buffer 数组中
65     idx += strlen(temp);
66 }
67 }
68
69 int main(int argc, char *argv[]) {
70     int client_socket;
71     struct sockaddr_in server_addr;
72     char buffer[MAX_LINE];
73     int server_port = DEFAULT_SERVER_PORT;
74     char *server_ip;
75
76     if (argc == 3) {
77         server_ip = argv[1];
78         server_port = atoi(argv[2]);
79         if (server_port <= 0 || server_port > 65535) {
80             fprintf(stderr, "Invalid server port: %s\n", argv[2]);
81             exit(EXIT_FAILURE);
82         }
83     }
84     else {
85         fprintf(stderr, "Usage: %s <server_ip> <server_port>\n", argv[0]);
86         exit(EXIT_FAILURE);
87     }
88
89     // 创建Socket
90     if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
91         perror("Socket creation failed");
92         exit(EXIT_FAILURE);
93     }
94
95     // 使用 getaddrinfo 解析域名或 IP 地址
96     struct addrinfo hints, *res;
97     memset(&hints, 0, sizeof(hints));
98     hints.ai_family = AF_INET; // IPv4
99     hints.ai_socktype = SOCK_STREAM; // TCP
100
101     int ret = getaddrinfo(server_ip, NULL, &hints, &res);
102     if (ret != 0)
103     {
104         fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(ret));
105         close(client_socket);
106         exit(EXIT_FAILURE);
107     }
108
109     struct sockaddr_in *addr_in = (struct sockaddr_in *)res->ai_addr;
110     server_addr.sin_family = AF_INET;
111     server_addr.sin_addr = addr_in->sin_addr;
112     server_addr.sin_port = htons(server_port);
113
114     freeaddrinfo(res);
115
116     // 设置连接超时
117     struct timeval timeout;
118     timeout.tv_sec = 3; // 超时时间为 5 秒
119     timeout.tv_usec = 0;
120     setsockopt(client_socket, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof(timeout));
121     // 这里只设置发送的超时时间，不要设置接收的 SO_RCVTIMEO 超时，否则连接上了也会断开
122
123     // 尝试连接
124     if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
125         perror("Connection failed");
126         close(client_socket);
127         exit(EXIT_FAILURE);

```

```

128     }
129
130     printf("Connected to server %s:%d.\n. Enter your message (end with double ENTER):\n", server_ip,
server_port);
131
132     // 创建接收线程
133     pthread_t rcv_tid;
134     thread_args_t targs;
135     targs.sockfd = client_socket;
136     if (pthread_create(&rcv_tid, NULL, rcv_thread_func, &targs) != 0)
137     {
138         fprintf(stderr, "Creating receive thread failed: %s\n", strerror(errno));
139         close(client_socket);
140         exit(EXIT_FAILURE);
141     }
142     pthread_detach(rcv_tid);
143
144     // 主循环：读取输入并发送消息
145     while (running) {
146         memset(buffer, 0, sizeof(buffer));
147         printf("> ");
148         fflush(stdout); // 刷新缓冲区，确保提示符显示
149         read_input(buffer); // 读取输入
150
151         // 如果是从文件中读取文本发送
152         if (strncmp(buffer, "FILE:", 5) == 0) {
153             char filename[MAX_LINE];
154             sscanf(buffer + 5, "%s", filename); // 提取文件名
155
156             FILE *file = fopen(filename, "r");
157             if (!file) {
158                 perror("Failed to open file");
159                 continue;
160             }
161
162             // 从文件中读取文本并发送
163             char file_buffer[1024];
164             size_t bytes_read;
165             while (((bytes_read = fread(file_buffer, 1, sizeof(file_buffer), file)) > 0) && (running)) {
166                 ssize_t bytes_sent = send(client_socket, file_buffer, bytes_read, 0); // 逐块发送文件内容
167                 if (bytes_sent < 0) {
168                     perror("Failed to send file");
169                     break;
170                 }
171             }
172             fclose(file);
173             printf("Text from file %s sent successfully.\n", filename);
174         } else {
175             // 否则，发送普通消息
176             size_t total_sent = 0;
177             size_t message_length = strlen(buffer);
178             while ((total_sent < message_length) && (running)) {
179                 ssize_t bytes_sent = send(client_socket, buffer + total_sent, message_length - total_sent,
0);
180                 if (bytes_sent < 0) {
181                     perror("Failed to send message");
182                     break;
183                 }
184                 total_sent += bytes_sent;
185             }
186         }
187     }
188
189     close(client_socket);

```

```

190     return 0;
191 }

```

2.2 服务器代码 (server.c)

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <strings.h>
5  #include <unistd.h>
6  #include <pthread.h>
7  #include <arpa/inet.h>
8
9  #define DEFAULT_SERVER_PORT 12345
10 #define MAX_LINE 20480 // 支持长文本消息
11 #define MAX_PENDING 5
12 #define MAX_CLIENTS 10
13
14 // 客户端套接字列表和互斥锁
15 int client_sockets[MAX_CLIENTS];
16 pthread_mutex_t client_list_mutex = PTHREAD_MUTEX_INITIALIZER;
17
18 // 处理广播消息的函数
19 void broadcast_message(const char *message)
20 {
21     pthread_mutex_lock(&client_list_mutex); // 加锁，避免并发修改客户端列表
22
23     for (int i = 0; i < MAX_CLIENTS; i++)
24     {
25         if (client_sockets[i] != 0)
26         {
27             ssize_t bytes_sent = send(client_sockets[i], message, strlen(message), 0);
28             if (bytes_sent < 0)
29             {
30                 perror("Failed to send message to client");
31             }
32         }
33     }
34
35     pthread_mutex_unlock(&client_list_mutex); // 解锁
36 }
37
38 // POSIX 线程库的线程函数必须接受 void* 类型参数，因此需要强制转换。
39 void *client_handler(void *client_socket_ptr) {
40     int client_socket = *(int *)client_socket_ptr; // 动态分配内存，避免 client_socket 被覆盖
41     free(client_socket_ptr);
42
43     char buffer[MAX_LINE];
44     ssize_t bytes_received;
45
46     // 将客户端加入到客户端列表
47     pthread_mutex_lock(&client_list_mutex);
48     for (int i = 0; i < MAX_CLIENTS; i++) {
49         if (client_sockets[i] == 0) {
50             client_sockets[i] = client_socket;
51             break;
52         }
53     }
54     pthread_mutex_unlock(&client_list_mutex);
55
56     while (1) {
57         memset(buffer, 0, sizeof(buffer)); // 接受消息之前，将缓存区清空，防止信息混乱
58         bytes_received = recv(client_socket, buffer, sizeof(buffer) - 1, 0);

```



```

59 // 确保保留最后一个字节给字符串的终止符 '\0', 最后的参数 0 为标志位
60 if (bytes_received <= 0) {
61     printf("Client disconnected.\n");
62
63     // 客户端断开连接时, 从客户端列表中移除
64     pthread_mutex_lock(&client_list_mutex);
65     for (int i = 0; i < MAX_CLIENTS; i++)
66     {
67         if (client_sockets[i] == client_socket)
68         {
69             client_sockets[i] = 0;
70             break;
71         }
72     }
73     pthread_mutex_unlock(&client_list_mutex);
74
75     break;
76 }
77 printf("Received from client: %s\n", buffer);
78
79 // 检查是否是广播命令
80 if (strcmp(buffer, "BROADCAST:", 10) == 0) {
81     char *message = buffer + 10; // 跳过 "broadcast:" 部分
82     broadcast_message(message); // 广播消息
83 } else {
84     // 获取当前时间
85     time_t now = time(NULL);
86     struct tm *t = localtime(&now);
87     char time_str[64];
88     strftime(time_str, sizeof(time_str) - 1, "%Y-%m-%d %H:%M:%S", t);
89
90     // 构造回复消息
91     char reply[MAX_LINE];
92     snprintf(reply, sizeof(reply), "Server received your message at %s\n", time_str);
93
94     // 发送回复消息给客户端
95     ssize_t bytes_sent = send(client_socket, reply, strlen(reply), 0);
96     if (bytes_sent < 0) {
97         perror("Send response message failed");
98         break;
99     }
100 }
101 }
102
103 close(client_socket);
104 return NULL;
105 }
106
107 int main(int argc, char *argv[]) {
108     int server_socket, client_socket;
109     struct sockaddr_in server_addr, client_addr;
110     socklen_t client_addr_len = sizeof(client_addr);
111     int server_port = DEFAULT_SERVER_PORT;
112
113     // 解析命令行参数
114     if (argc == 2) {
115         server_port = atoi(argv[1]);
116         if (server_port <= 0 || server_port > 65535) {
117             fprintf(stderr, "Invalid port number: %s\n", argv[1]);
118             exit(EXIT_FAILURE);
119         }
120     } else if (argc > 2) {
121         fprintf(stderr, "Usage: %s [port]\n", argv[0]);
122         exit(EXIT_FAILURE);

```

```

123     }
124
125     // 创建Socket
126     if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
127         perror("Socket creation failed");
128         exit(EXIT_FAILURE);
129     }
130
131     // 复用地址避免重启服务器时的"Address already in use"
132     int opt = 1;
133     setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
134
135     // 配置服务器地址
136     bzero((char *)&server_addr, sizeof(server_addr));
137     server_addr.sin_family = AF_INET;
138     server_addr.sin_addr.s_addr = INADDR_ANY;
139     server_addr.sin_port = htons(server_port);
140
141     // 绑定Socket
142     if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
143         perror("Bind failed");
144         close(server_socket);
145         exit(EXIT_FAILURE);
146     }
147
148     // 监听连接
149     if (listen(server_socket, MAX_PENDING) < 0) {
150         perror("Listen failed");
151         close(server_socket);
152         exit(EXIT_FAILURE);
153     }
154
155     printf("Server is listening on port %d\n", server_port);
156
157     while (1) {
158         // 接受客户端连接
159         client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_addr_len);
160         if (client_socket < 0) {
161             perror("Accept failed");
162             continue;
163         }
164         printf("New client connected.\n");
165
166         // 创建线程处理客户端
167         pthread_t thread_id;
168         int *client_socket_ptr = malloc(sizeof(int));
169         *client_socket_ptr = client_socket;
170
171         if (pthread_create(&thread_id, NULL, client_handler, client_socket_ptr) != 0) {
172             perror("Thread creation failed");
173             free(client_socket_ptr);
174             close(client_socket);
175         }
176         pthread_detach(thread_id);
177         // 这个函数是线程分离用的。将线程标记为分离状态。
178         // 线程执行结束后系统会自动回收资源，无需通过 pthread_join() 等待线程结束。
179         // 分离线程适用于短期任务，主线程不关心线程的返回值或执行状态。
180     }
181     close(server_socket);
182     return 0;
183 }

```