Socket编程

文件结构

```
chat/
├── CMakeLists.txt
├── README.md
├── make.sh 编译脚本
├── report.pdf 实验报告
├── client.cpp  client源代码
└── server.cpp  server源代码
```

实验环境

Linux / Mac OS

实验要求

选择**高级聊天程序**进行实验：

一个服务器，多个客户端，服务器负责消息中转，客户端之间可以互相聊天。（广播/单播）

实验设计

- 需要先运行`server`程序，再运行`client`程序。
- 每个`client`接入时，需要给出自己的名称，当连接到`server`时，就向服务器发送消息`#new client: clien1`（假设客户端名称为`client`）。如果新客户端加入时使用的名称与在线客户端名称重复，则server向新客户端发送消息，要求其退出后重新以新名字连接`server`，此外不进行任何操作。
- `server`收到这个声明自己名称的消息后，将其`socket`和名称记录在`unordered_map<std::string, int> clnt_socks`中，并向所有客户端广播这条消息，通知所有客户端有新的客户端加入。
- 之后每个客户端发送消息时，如果是广播，则直接输入消息发送即可（假设输入为`message contents`），服务器收到后会转发给所有目前在线的客户端，消息格式为：`[client1] message contents`。
- 如果需要私聊某个客户端，则需要在消息前输入`@客户端名称`，在客户端名称后输入空格，再输入消息内容，如`@cient2 message contents`。服务器收到后，解析消息，根据接收方名称再`clnt_socks`中查找到其对应的`socket`，只将消息转发给发送方和接收方，消息格式为：`[client1] @client2 message contents`。如果在线客户端中无对应名称的客户端，则server向发送方发送消息，告知无此客户端。
- 当客户端退出时，直接输入`quit`或`Quit`，则退出程序。server检测到有客户端退出程序时，向其他所有客户端广播消息`#client client1 leaves the chat room`，告知有客户端离开。客户端退出后，服务器从`clnt_socks`中删除其相应的键值对，并将`clnt_cnt`减一。

- 需要等所有客户端都退出后，再终止`server`程序。

下面对`server`和`client`的实现作详细说明：

server.cpp

int main(int argc,const char argv,const char envp)函数

首先创建套接字：

AF_INET表示使用 IPv4，SOCK_STREAM表示面向连接的数据传输方式，IPPROTO_TCP表示使用 TCP 协议。

```
    serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (serv_sock == -1){
        error_handling("socket() failed!");
    }
```

将套接字和指定的 IP、端口绑定：

首先设置 IPv4，设置 IP 地址为`INADDR_ANY`（指示任意地址），设置端口为默认端口`5208`。

```
    memset(&serv_addr,0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(SERVER_PORT);

    //   绑定
    if (bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) ==
-1){
        error_handling("bind() failed!");
    }
    printf("the server is running on port %d\n", SERVER_PORT);
```

接下来侦听连接请求，循环监听客户端，永远不停止。当没有客户端连接时，`accept()`会阻塞程序执行，直到有客户端连接进来。当有客户端连接时，客户端数量`clnt_cnt`加一，为了保证`clnt_cnt`的正确，需要在修改前后使用`mutex`加锁、解锁。然后为此客户端生成一个线程，调用`handle_clnt`函数。

```
    while(1){   // 循环监听客户端，永远不停止
        clnt_addr_size = sizeof(clnt_addr);
        // 当没有客户端连接时， accept() 会阻塞程序执行，直到有客户端连接进来
        clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr,
&clnt_addr_size);
        if (clnt_sock == -1){
            error_handling("accept() failed!");
        }

        // 增加客户端数量
```

```
        mtx.lock();
        clnt_cnt++;
        mtx.unlock();

        // 生成线程
        std::thread th(handle_clnt, clnt_sock);
        th.detach();

        output("Connected client IP: %s \n", inet_ntoa(clnt_addr.sin_addr));
    }
```

void handle_clnt(int clnt_sock)函数

对每个client进行处理。

设置一个变量`int flag`默认为0，用来标示是否需要向客户端发送消息。

当接收到消息时，首先检查是否为第一次连接服务器时声明自己的名称，即检查接收到的消息最前面是否为`#new client:`。

```
while(recv(clnt_sock, msg, sizeof(msg),0) != 0){
        // 检查是否为第一次进入聊天室时的广播
        if (std::strlen(msg) > std::strlen(tell_name)) {
            // 判断msg最前面是否为 #new client:
            char pre_name[13];
            std::strncpy(pre_name, msg, 12);
            pre_name[12] = '\0';
            if (std::strcmp(pre_name, tell_name) == 0) {
```

如果是第一次声明名称的消息，则将其声明的名字复制出来，如果在`clnt_socks`中无相同名称的客户端，则成功连接，将其名称和socket记录在`clnt_socks`中。

```
                char name[20];
                std::strcpy(name, msg+12);
                if(clnt_socks.find(name) == clnt_socks.end()){
                    output("the name of socket %d: %s\n", clnt_sock, name);
                    clnt_socks[name] = clnt_sock;
                }
```

如果在`clnt_socks`中有相同名称的客户端，即新客户端使用了与在线客户端重复的名称时，server向其发送一个错误消息，此外不向在线客户端转发消息，因此`flag=1`。向新客户端发送错误提示消息，要求其退出后重新以另一个名字连接。并将`clng_cnt`减一，即错误增加了此客户端，实际此客户端并没有加入聊天。

```
                else {
                    // 客户端名字重复
                    std::string error_msg = std::string(name) + " exists
already. Please quit and enter with another name!";
                    send(clnt_sock, error_msg.c_str(), error_msg.length()+1,
0);
                    mtx.lock();
                    clnt_cnt--;
                    mtx.unlock();
                    flag = 1;
                }
            }
        }
```

如果`flag=0`，即需要转发此消息，则调用`send_msg`函数发送消息。

```
        if(flag == 0)
            send_msg(std::string(msg));
```

当客户端连接关闭后，从`clnt_socks`中删除此客户端相应键值对，客户端数量减一，向所有在线客户端发送消息，告知有客户端离开。

```
    if(flag == 0){
        // 客户端关闭连接，从clnt_socks中删除此客户端
        std::string leave_msg;
        std::string name;
        mtx.lock();
        for (auto it = clnt_socks.begin(); it != clnt_socks.end(); ++it ){
            if(it->second == clnt_sock){
                name = it->first;
                clnt_socks.erase(it->first);
            }
        }
        clnt_cnt--;
        mtx.unlock();
        leave_msg = "client " + name + " leaves the chat room";
        send_msg(leave_msg);
        output("client %s leaves the chat room\n", name.c_str());
        close(clnt_sock);
    }
    else {
        close(clnt_sock);
    }
```

void send_msg(const std::string &msg)函数

处理消息发送。

首先使用互斥量加锁。

```
    mtx.lock();
```

server收到的消息为`[client1] @client2 message`或`[client1] message`。

首先判断是否为私聊消息，私聊消息在第一个空格后是`@`：

```
    std::string pre = "@";
    int first_space = msg.find_first_of(" ");
    if (msg.compare(first_space+1, 1, pre) == 0){
```

如果是私聊消息，则处理消息，得到发送方和接收方的名称。

```
        // space为recv_clnt和消息间的空格
        int space = msg.find_first_of(" ", first_space+1);
        std::string receive_name = msg.substr(first_space+2, space-
first_space-2);
        std::string send_name = msg.substr(1, first_space-2);
```

从`clnt_socks`中查找接收方对应的socket，如果不存在，则向发送方发送错误提示，告知此客户端不存在。

```
        if(clnt_socks.find(receive_name) == clnt_socks.end()) {
            // 如果私聊的用户不存在
            std::string error_msg = "[error] there is no client named " +
receive_name;
            send(clnt_socks[send_name], error_msg.c_str(),
error_msg.length()+1, 0);
        }
```

如果存在，则向发送方和接收方发送此消息，即只有私聊的两人可以看到消息。

```
        else {
            send(clnt_socks[receive_name], msg.c_str(), msg.length()+1, 0);
            send(clnt_socks[send_name], msg.c_str(), msg.length()+1, 0);
        }
    }
```

如果不是私聊消息，则向所有在线客户端转发此消息。

```
    else {
        // 广播
        for (auto it = clnt_socks.begin(); it != clnt_socks.end(); it++) {
            send(it->second, msg.c_str(), msg.length()+1, 0);
        }
    }
```

对互斥量解锁。

```
    mtx.unlock();
```

辅助函数

这里我们使用到了`output`函数进行`server`的输出，`error_handling`函数进行错误处理。实现简单，在此不作赘述。

client.cpp

int main(int argc,const char argv,const char envp)函数

首先检查调用client时的参数是否合法，调用格式为`./client client_name`。

```
    if (argc!=2){
        error_output("Usage : %s <Name> \n",argv[0]);
        exit(1);
    }
```

为了方便消息发送，我们将客户端名称两边加上方括号。

```
    name="["+std::string(argv[1])+"]";
```

创建socket：

```
    sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == -1){
        error_handling("socket() failed!");
    }
```

将套接字和指定的 IP、端口绑定，这里IP默认`127.0.0.1`，端口默认`5208`。

```
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(IP);
    serv_addr.sin_port = htons(SERVER_PORT);
```

连接服务器，并向服务器发送自己的名字，消息格式为`#new client: client_name`。

```
    if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1){
        error_handling("connect() failed!");
    }
    // 向服务器发送自己的名字
    std::string my_name = "#new client:" + std::string(argv[1]);
    send(sock, my_name.c_str(), my_name.length() + 1, 0);
```

生成发送消息、接受消息的线程。

```
    std::thread snd(send_msg, sock);
    std::thread rcv(recv_msg, sock);

    snd.join();
    rcv.join();
```

void send_msg(int sock)函数

处理消息的发送。如果输入的是`quit`或者`Quit`，则客户端退出，关闭`socket`并退出程序。否则生成发送的消息，将其发送给服务器。

```
    while(1){
        getline(std::cin, msg);
        if (msg == "Quit"|| msg == "quit"){
            close(sock);
            exit(0);
        }
        // 生成消息格式（[name] massage）
        std::string name_msg = name + " " + msg;
        send(sock, name_msg.c_str(), name_msg.length() + 1, 0);
    }
```

void recv_msg(int sock)函数

处理消息的接受。当收到消息时，打印出来。

```
    char name_msg[BUF_SIZE + name.length() + 1];
    while (1){
        int str_len = recv(sock, name_msg, BUF_SIZE+name.length() + 1, 0);
        if (str_len == -1){
            exit(-1);
        }
        std::cout<<std::string(name_msg)<<std::endl;
    }
```

辅助函数

这里我们使用了`error_output`函数处理错误，实现简单，在此不作赘述。

编译运行

编译

```
$ g++ server.cpp -o server -pthread -std=c++11
$ g++ client.cpp -o client -pthread -std=c++11
```

或者直接运行编译脚本

```
$ ./make.sh
```

编译结束后得到`server`和`client`可执行文件。

运行

需要先运行server：

```
$ ./server
```

再运行client，需要给出此客户端的名称：

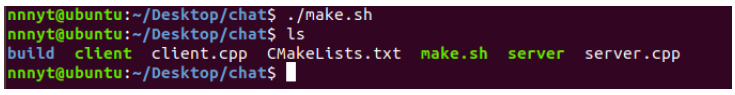
```
$ ./client client1
```

类似的，可以增加多个客户端，需要不同的名称，如：

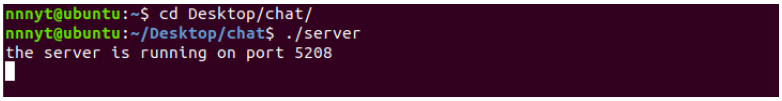
```
$ ./client client2
$ ./client client3
```

实验结果

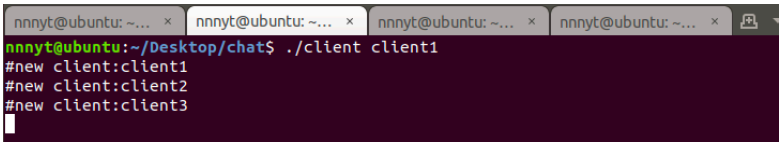
编译：

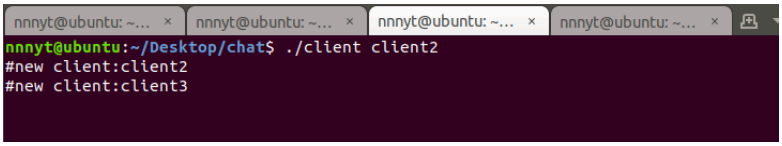


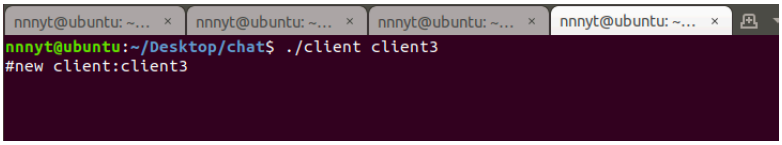
运行`server`：



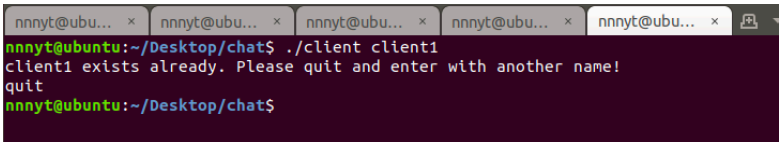
依次运行三次`client`，名称分别为`client1`,`client2`,`client3`，三个客户端分别显示如下：



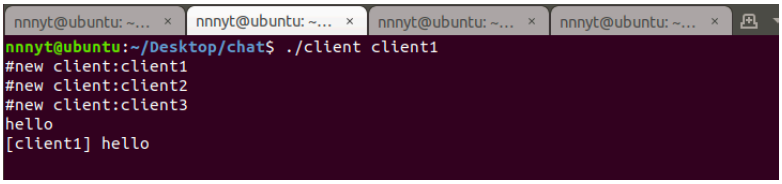


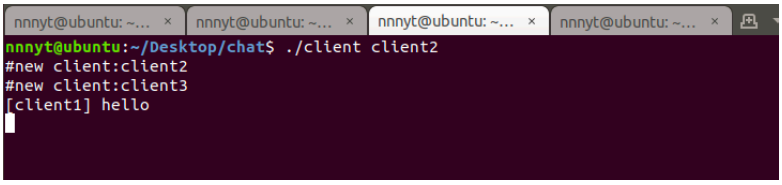


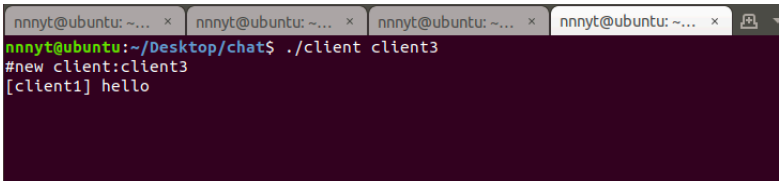
这时我们如果再运行一个`client`，名称为`client1`，则报错，提醒客户端退出后以新名字连接。输入`quit`退出此程序。



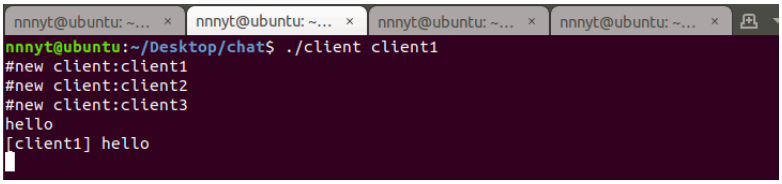
使用`client1`发送一个广播消息，三个客户端分别显示如下：

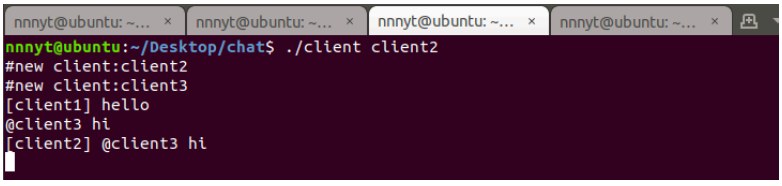


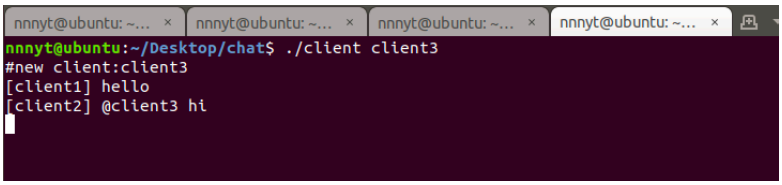




使用`client2`给`client3`发送一个私聊消息，三个客户端分别显示如下：

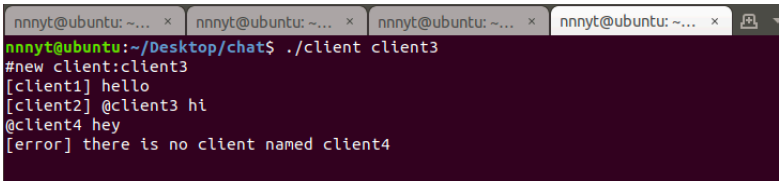




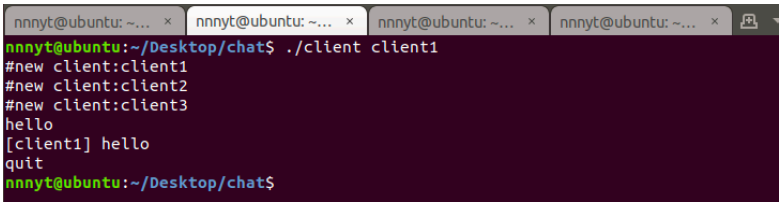


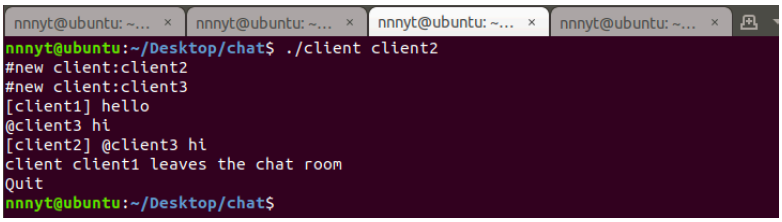
可以看到只有`client2`和`client3`显示了这条私聊消息。

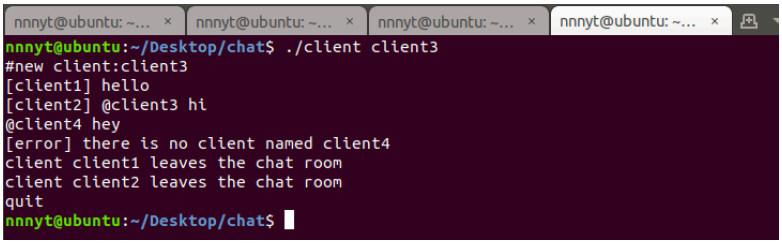
使用`client3`给不存在的`client4`发送一条消息，服务器向`client3`报错如下，其他客户端无任何消息：



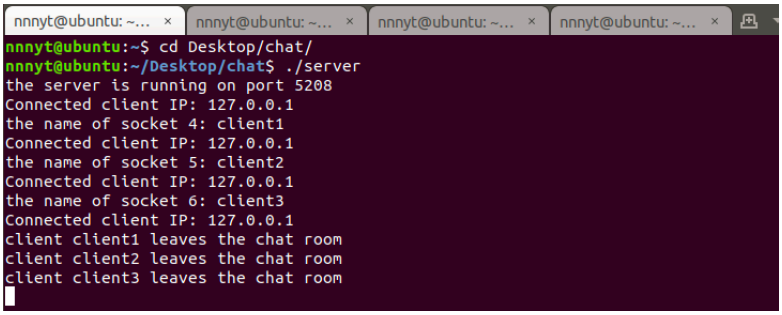
依次退出`client1`、`client2`、`client3`，三个客户端分别显示如下：







这时查看`server`的输出记录：



终止`server`，测试结束。