

浙江大学实验报告

专业: 计算机科学与技术
姓名: 吴同
学号: 3170104848
邮箱: wutongcs@zju.edu.cn

课程名称: 计算机网络 指导老师: 张泉方 实验时间: 2019 年 12 月
实验名称: 基于 Socket 接口实现自定义协议通信 实验类型: 编程型 同组同学: 无

一、 实验目的

- 学习如何设计网络应用协议
- 掌握 Socket 编程接口编写基本的网络应用软件

二、 实验内容

- 设计一个客户端和服务端之间的应用通信协议，根据自定义的协议规范，使用 Socket 编程接口编写基本的网络应用软件。
- 掌握 C 语言形式的 Socket 编程接口用法，能够正确发送和接收网络数据包。
- 开发一个客户端，实现人机交互界面和与服务器的通信。
- 开发一个服务端，实现并发处理多个客户端的请求。
- 功能要求：运输层协议采用 TCP；客户端采用交互菜单形式，用户可选择连接、断开连接、获取时间、获取名字、活动连接列表、发消息、退出；服务端接收到客户端请求后，根据客户端传过来的指令完成特定任务。
- 本实验涉及到网络数据包发送部分不能使用任何的 Socket 封装类，只能使用最底层的 C 语言形式的 Socket API。

三、 主要仪器设备

- 硬件设备: MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports)
- 操作系统: macOS Catalina 10.15 (19A583)
- 编译环境: clang 11.0.0

四、 操作方法与实验步骤

1. 设计请求、指示、响应数据包的格式
 - 定义两个数据包的边界如何识别
 - 定义数据包的请求、指示、响应类型字段

- 定义数据包的长度字段或者结尾标记
- 定义数据包内数据字段的格式（特别是考虑客户端列表数据如何表达）

2. 客户端编写

2.1 运行初始化, 调用 `socket()`, 向操作系统申请 socket 句柄

2.2 编写一个菜单功能, 列出 7 个选项

2.3 等待用户选择

2.4 根据用户选择, 做出相应的动作 (未连接时, 只能选连接功能和退出功能)

- 选择连接功能: 请用户输入服务器 IP 和端口, 然后调用 `connect()`, 等待返回结果并打印。连接成功后设置连接状态为已连接。然后创建一个接收数据的子线程, 循环调用 `receive()`, 如果收到了一个完整的响应数据包, 就通过线程间通信 (如消息队列) 发送给主线程, 然后继续调用 `receive()`, 直至收到主线程通知退出。
- 选择断开功能: 调用 `close()`, 并设置连接状态为未连接。通知并等待子线程关闭。
- 选择获取时间功能: 组装请求数据包, 类型设置为时间请求, 然后调用 `send()` 将数据发送给服务器, 接着等待接收数据的子线程返回结果, 并根据响应数据包的内容, 打印时间信息。
- 选择获取名字功能: 组装请求数据包, 类型设置为名字请求, 然后调用 `send()` 将数据发送给服务器, 接着等待接收数据的子线程返回结果, 并根据响应数据包的内容, 打印名字信息。
- 选择获取客户端列表功能: 组装请求数据包, 类型设置为列表请求, 然后调用 `send()` 将数据发送给服务器, 接着等待接收数据的子线程返回结果, 并根据响应数据包的内容, 打印客户端列表信息 (编号、IP 地址、端口等)。
- 选择发送消息功能 (选择前需要先获得客户端列表): 请用户输入客户端的列表编号和要发送的内容, 然后组装请求数据包, 类型设置为消息请求, 然后调用 `send()` 将数据发送给服务器, 接着等待接收数据的子线程返回结果, 并根据响应数据包的内容, 打印消息发送结果 (是否成功送达另一个客户端)。
- 选择退出功能: 判断连接状态是否为已连接, 是则先调用断开功能, 然后再退出程序。否则, 直接退出程序。
- 主线程除了在等待用户的输入外, 还在处理子线程的消息队列, 如果有消息到达, 则进行处理, 如果是响应消息, 则打印响应消息的数据内容 (比如时间、名字、客户端列表等); 如果是指示消息, 则打印指示消息的内容 (比如服务器转发的别的客户端的消息内容、发送者编号、IP 地址、端口等)。

3. 服务端编写

- 3.1 运行初始化，调用 `socket()`，向操作系统申请 socket 句柄
- 3.2 调用 `bind()`，绑定监听端口（学号后 4 位），接着调用 `listen()`，设置连接等待队列长度
- 3.3 主线程循环调用 `accept()`，直到返回一个有效的 socket 句柄，在客户端列表中增加一个新客户端的项目，并记录下该客户端句柄和连接状态、端口。然后创建一个子线程后继续调用 `accept()`
 - 调用 `send()`，发送一个 hello 消息给客户端
 - 循环调用 `receive()`，如果收到了一个完整的请求数据包，根据请求类型做相应的动作：
 - 请求类型为获取时间：调用 `time()` 获取本地时间，然后将时间数据组装进响应数据包，调用 `send()` 发给客户端
 - 请求类型为获取名字：将服务器的名字组装进响应数据包，调用 `send()` 发给客户端
 - 请求类型为获取客户端列表：读取客户端列表数据，将编号、IP 地址、端口等数据组装进响应数据包，调用 `send()` 发给客户端
 - 请求类型为发送消息：根据编号读取客户端列表数据，如果编号不存在，将错误代码和出错描述信息组装进响应数据包，调用 `send()` 发回源客户端；如果编号存在并且状态是已连接，则将要转发的消息组装进指示数据包。调用 `send()` 发给接收客户端（使用接收客户端的 socket 句柄），发送成功后组装转发成功的响应数据包，调用 `send()` 发回源客户端。
- 3.4 主线程还负责检测退出指令（如用户按退出键或者收到退出信号），检测到后即通知并等待各子线程退出。最后关闭 Socket，主程序退出

五、实验数据记录和处理

1. 描述请求数据包的格式，请求类型的定义

请求数据包的类型值为 0、1、2、3、4、5，分别对应断开连接，获取时间，获取服务器主机名，获取客户端列表，发送消息，和一个预留的值。如果是 4 号类型的请求包，则紧跟 ip 地址，端口号，发送的文本，并使用特殊符号进行分隔。

type	ip_addr	^	port	\$
content				

图 1: 请求数据包的格式

2. 描述响应数据包的格式，响应类型的定义

响应数据包的类型值为 11、12、13、14，分别对应获取时间、获取服务器主机名、获取客户端列表、发送消息的响应。11、12 和 14 号类型的响应包的类型值后紧跟获取的文本内容，13 号类型响应包的类型值后则为若干组 ip 地址和端口号，相互之间用特殊符号分隔。

type			
content			
ip_addr	^	port	\$
ip_addr	^	port	\$
ip_addr	^	port	\$

图 2: 响应数据包的格式

3. 描述指示数据包的格式, 指示类型的定义

指示数据包的类型值为 20, 类型值之后是指示的文本内容, 以 0 为结尾或以定义的数据包最大长度为结尾。

type	content

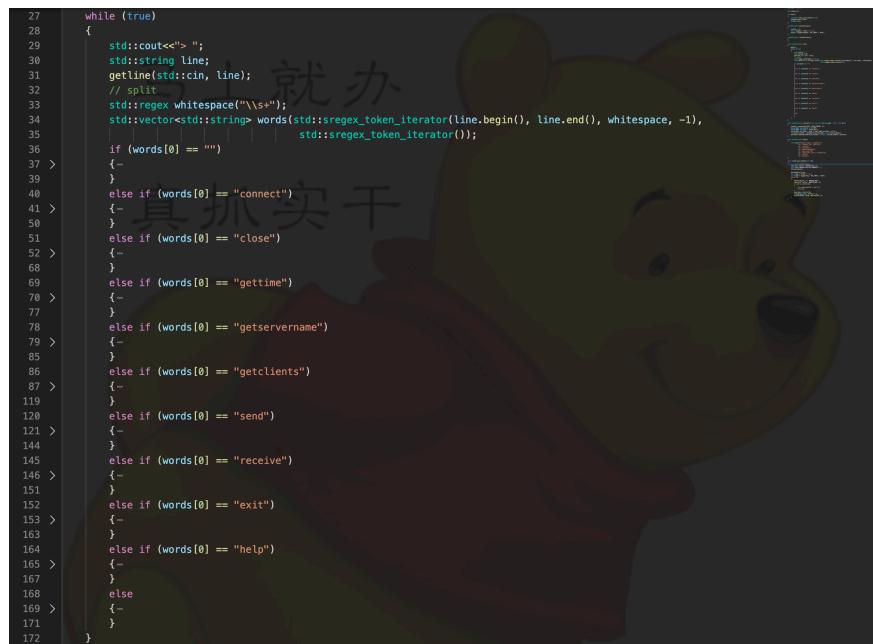
图 3: 指示数据包的格式

4. 客户端初始运行后显示的菜单选项

```
[→ src ./client
Please input a command:
- connect [IP] [port]
- close
- gettime
- getservername
- getclients
- send [IP] [port] [content]
- exit
- help
> ]
```

图 4: 客户端初始运行后显示的菜单选项

5. 客户端的主线程循环关键代码



```
27     while (true)
28     {
29         std::cout<<< "";
30         std::string line;
31         getline(std::cin, line);
32         // split
33         std::regex whitespace("\\s+");
34         std::vector<std::string> words(std::sregex_token_iterator(line.begin(), line.end(), whitespace, -1),
35                                         std::sregex_token_iterator());
36         if (words[0] == "")
37         {-
38     }
39     else if (words[0] == "connect")
40     {-
41     }
42     else if (words[0] == "close")
43     {-
44     }
45     else if (words[0] == "gettime")
46     {-
47     }
48     else if (words[0] == "getservername")
49     {-
50     }
51     else if (words[0] == "getclients")
52     {-
53     }
54     else if (words[0] == "send")
55     {-
56     }
57     else if (words[0] == "receive")
58     {-
59     }
60     else if (words[0] == "exit")
61     {-
62     }
63     else if (words[0] == "help")
64     {-
65     }
66     else
67     {-
68     }
69 }
70 >
71
72
73
74
75
76
77
78
79 >
80
81
82
83
84
85
86
87 >
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121 >
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146 >
147
148
149
150
151
152
153 >
154
155
156
157
158
159
160
161
162
163
164
165 >
166
167
168
169 >
170
171
172 }
```

图 5: 客户端的主线程循环关键代码截图

6. 客户端的接收数据子线程循环关键代码

```
198 void connection_handle(int cfd)
199 {
200     char buffer[BUFFER_MAX];
201     recv(cfd, buffer, BUFFER_MAX, 0);
202     std::cout<<GREEN<<buffer<<NORMAL<<"> ";
203     fflush(stdout);
204
205     messageStruct msg;
206     key_t key = ftok("/", 'a');
207     int msqid = msgget(key, IPC_CREAT | 0666);
208     while (1)
209     {
210         memset(buffer, 0, BUFFER_MAX);
211         recv(cfd, buffer, BUFFER_MAX, 0);
212         if (20 == buffer[0])
213         {
214             std::cout<<buffer + 1<<'\n';
215             continue;
216         }
217         msg.type = buffer[0];
218         strcpy(msg.text, buffer + 1);
219         msgsnd(msqid, &msg, BUFFER_MAX, 0);
220     }
221 }
```

图 6: 客户端的接收数据子线程循环关键代码截图

7. 服务器初始运行后显示的界面

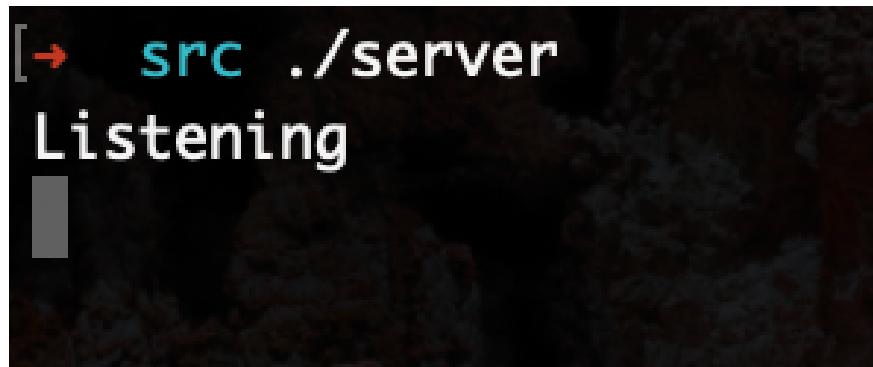


图 7: 服务器初始运行后显示的界面

8. 服务器的主线程循环关键代码

```

30 void socketServer::run()
31 {
32     std::cout<<"Listening\n";
33     while (true)
34     {
35         sockaddr_in client;
36         unsigned int clientAddrLength = sizeof(client);
37         int connection_fd = accept(sockfd, (sockaddr*)&client, (socklen_t*)&clientAddrLength);
38         clientList.push_back(std::pair<int, ip_port>(connection_fd, ip_port(inet_ntoa(client.sin_addr), ntohs(client.sin_port))));
39         std::cout<<inet_ntoa(client.sin_addr)<<":"<<ntohs(client.sin_port)<<" connected.\n";
40         pthread_t connection_thread;
41         pthread_create(&connection_thread, nullptr, thread_handle, &connection_fd);
42     }
43 }
```

图 8: 服务器的主线程循环关键代码截图

9. 服务器的客户端处理子线程循环关键代码

```

53 void connection_handle(int cfd)
54 {
55     char hello[] = "hello\n";
56     send(cfd, hello, strlen(hello), 0);
57
58     char buffer_recv[BUFFER_MAX] = {0};
59     char buffer_send[BUFFER_MAX] = {0};
60     while (true)
61     {
62         recv(cfd, buffer_recv, BUFFER_MAX, 0);
63         memset(buffer_send, 0, BUFFER_MAX);
64         mt.lock();
65         switch (buffer_recv[0])
66         {
67             >>> case 0: ...
68             >>> case 1: ...
69             >>> case 2: ...
70             >>> case 3: ...
71             >>> case 4: ...
72             >>> case 5:
73                 {
74                     break;
75                 }
76         }
77         memset(buffer_recv, 0, BUFFER_MAX);
78         mt.unlock();
79     }
80 }
```

图 9: 服务器的客户端处理子线程循环关键代码截图

10. 客户端选择连接功能时, 客户端和服务端显示内容

```
> connect 127.0.0.1 4848
> hello
>
```

图 10: 客户端显示内容

127.0.0.1:51189 connected.

图 11: 服务端显示内容

No.	Time	Source	Description	Protocol	Len	Info
3	0.000088	127.0.0.1	127.0.0.1 ->	TCP	-	31732 -> 4848 [SYN] Seq=0 Win=65355 Len=64 Tsvl=488314697 Tsecr=488314697 SACK_PERM=1
2	0.000085	127.0.0.1	127.0.0.1 ->	TCP	-	4848 -> 31732 [SYN, ACK] Seq=0 Win=65355 Len=64 Tsvl=488314697 Tsecr=488314697 SAck=1
3	0.000097	127.0.0.1	127.0.0.1 ->	TCP	-	51732 -> 4848 [ACK] Seq=1 Ack=1 Win=488256 Len=64 Tsvl=488314697 Tsecr=488314697
4	0.000106	127.0.0.1	127.0.0.1 ->	TCP	-	[TCP Window Update] 4848 -> 51732 [ACK] Seq=1 Ack=1 Win=488256 Len=0 Tsvl=488314697 Tsecr=488314697
5	0.000269	127.0.0.1	127.0.0.1 ->	TCP	-	4848 -> 51732 [PSH, ACK] Seq=1 Ack=1 Win=488256 Len=6 Tsvl=488314697 Tsecr=488314697
6	0.000279	127.0.0.1	127.0.0.1 ->	TCP	-	51732 -> 4848 [ACK] Seq=1 Ack=7 Win=488256 Len=0 Tsvl=488314697 Tsecr=488314697

图 12: Wireshark 抓包内容

11. 客户端选择获取时间功能时, 客户端和服务端显示内容

> gettime
Server time: Tue Dec 31 16:04:17 2019

图 13: 客户端显示内容

4 gettime: 1577779609

图 14: 服务端显示内容

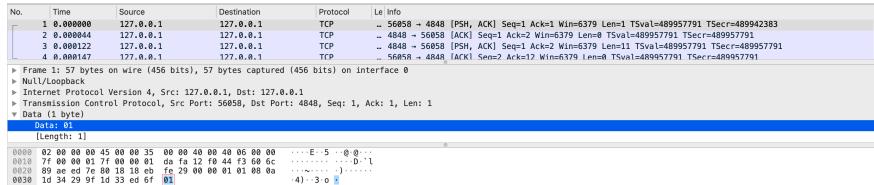


图 15: Wireshark 抓包内容 (请求包)

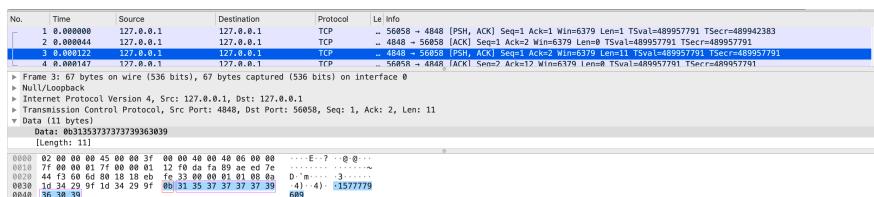


图 16: Wireshark 抓包内容 (响应包)

12. 客户端选择获取名字功能时，客户端和服务端显示内容

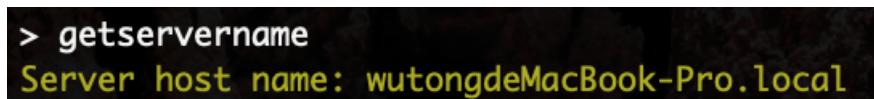


图 17: 客户端显示内容



图 18: 服务端显示内容

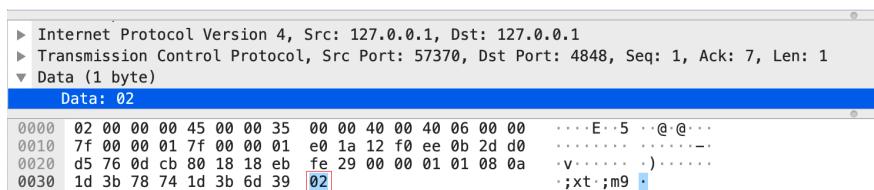


图 19: Wireshark 抓包内容 (请求包)

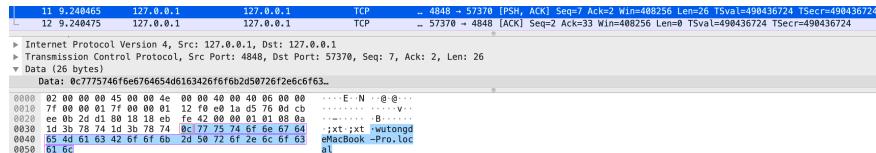


图 20: Wireshark 抓包内容 (响应包)

```
std::cout<<cfd<<" getname"<<'\n';
buffer_send[0] = 12;
gethostname(buffer_send + strlen(buffer_send), sizeof(buffer_send) - sizeof(char));
send(cfd, buffer_send, strlen(buffer_send), 0);
break;
```

图 21: 服务器的处理代码

13. 客户端选择获取客户端列表功能时，客户端和服务端显示内容

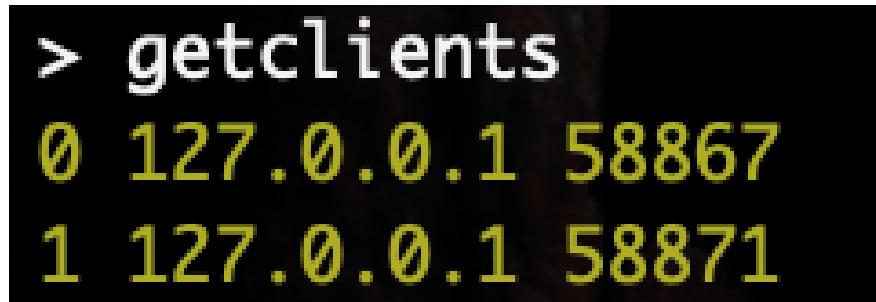


图 22: 客户端显示内容



图 23: 服务端显示内容

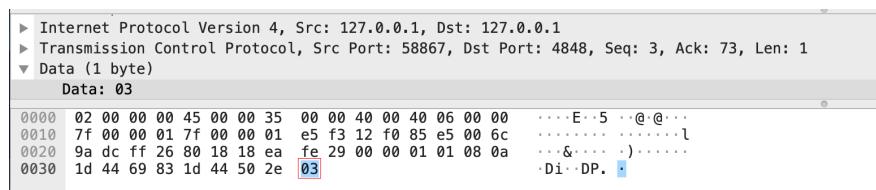


图 24: Wireshark 抓包内容 (请求包)

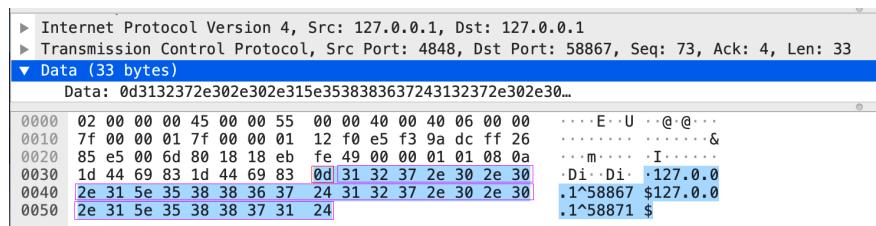


图 25: Wireshark 抓包内容 (响应包)

```
std::cout<<cfd<<" getclients"<<'\n';
buffer_send[0] = 13;
for (auto& it: clientList)
{
    sprintf(buffer_send + strlen(buffer_send), "%s", it.second.first.c_str());
    sprintf(buffer_send + strlen(buffer_send), "%^");
    sprintf(buffer_send + strlen(buffer_send), "%d", it.second.second);
    sprintf(buffer_send + strlen(buffer_send), "$");
}
send(cfd, buffer_send, strlen(buffer_send), 0);
break;
```

图 26: 服务器的处理代码

14. 客户端选择发送消息功能时，客户端和服务端显示内容

```
> send 127.0.0.1 60456 good morning
Success.
```

图 27: 发送方客户端显示内容

```
> good morning
```

图 28: 接收方客户端显示内容

```
4 send message to 127.0.0.1:60456
```

图 29: 服务端显示内容

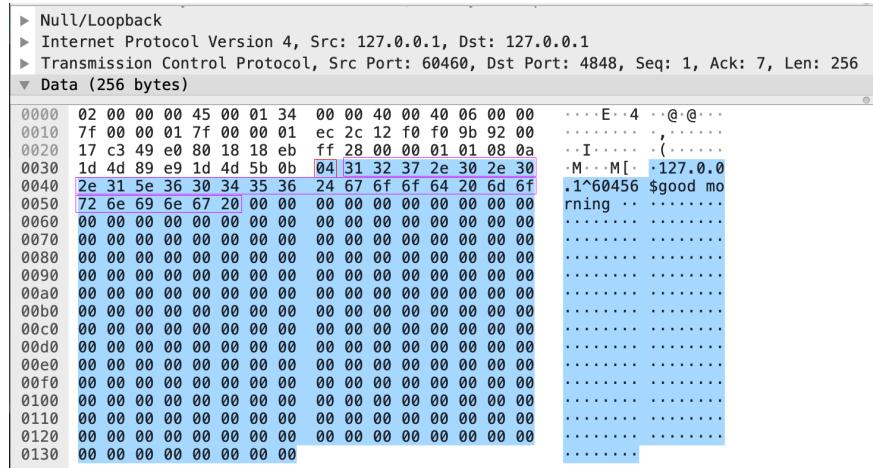


图 30: Wireshark 抓包内容 (请求包)

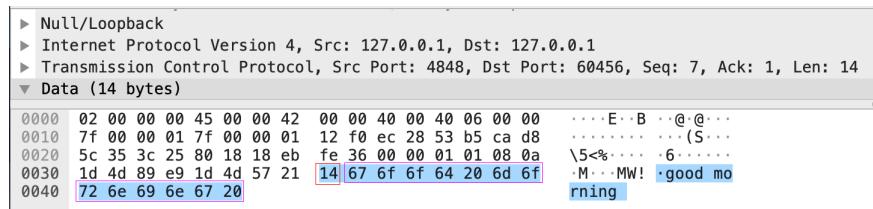


图 31: Wireshark 抓包内容 (指示包)

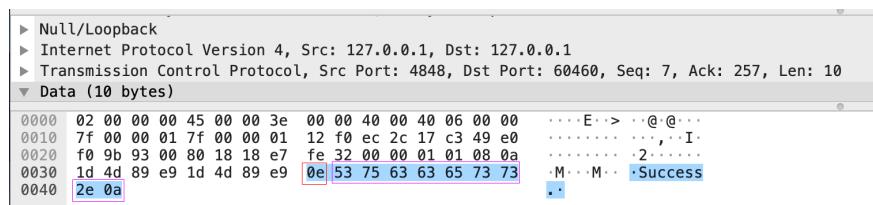


图 32: Wireshark 抓包内容 (响应包)

```

    std::string msg(buffer_recv + 1);
    size_t pos0 = msg.find("^\n"), pos1 = msg.find("$");
    std::string ip_addr = msg.substr(0, pos0);
    int port = atoi(msg.substr((pos0 + 1), pos1 - pos0 - 1).c_str());
    std::string content = msg.substr(pos1 + 1);
    int sock = -1;
    for (auto it = clientList.begin(); it != clientList.end(); ++it)
    {
        if (it->second.first == ip_addr && it->second.second == port)
        {
            sock = it->first;
            break;
        }
    }
    std::cout<<<cfld<<" send message to "<<ip_addr<<"."<<port<<'\n';
    buffer_send[0] = 14;
    if (-1 == sock)
    {
        sprintf(buffer_send + 1, "Fail.\n");
    }
    else
    {
        send_msg(sock, content);
        sprintf(buffer_send + 1, "Success.\n");
    }
    send(cfd, buffer_send, strlen(buffer_send), 0);
    break;
}

```

图 33: 服务器的处理代码

15. 拔掉客户端的网线, 然后退出客户端程序

由于本实验无法模拟断开网线的情况, 所以用关闭终端代替。但是, 关闭终端时, 操作系统会进行相关处理, 与真实的物理掉线有很大的不同。理论上分析, 客户端意外断网时, 服务器无法获知, TCP 连接不会被释放。

```

[~] ~ netstat -an | grep 4848
tcp4      0      0 127.0.0.1.4848          127.0.0.1.49475      CLOSE_WAIT
tcp4      0      0 127.0.0.1.49475          127.0.0.1.4848      FIN_WAIT_2
tcp4      0      0 *.4848                  *.*                   LISTEN

```

图 34: 客户端的连接状态为 FIN_WAIT_2

No.	Time	Source	Destination	Protocol	Le Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	_ 49475 -> 4848 [FIN, ACK] Seq=1 Ack=1 Win=6379 Len=0 TStamp=457592538 TSecr=457510933
2	0.000035	127.0.0.1	127.0.0.1	TCP	_ 4848 -> 49475 [ACK] Seq=1 Ack=2 Win=6379 Len=0 TStamp=457592538 TSecr=457592538

图 35: TCP 连接释放的消息

```
[+ ~ netstat -an | grep 4848
tcp4      0      0 127.0.0.1.4848          127.0.0.1.50594      ESTABLISHED
tcp4      0      0 127.0.0.1.50594          127.0.0.1.4848      ESTABLISHED
tcp4      0      0 127.0.0.1.4848          127.0.0.1.49475      CLOSE_WAIT
tcp4      0      0 *.4848                  *.*                  LISTEN
```

图 36: 较长时间后服务端的连接不再存在

16. 再次连上客户端的网线, 重新运行客户端程序

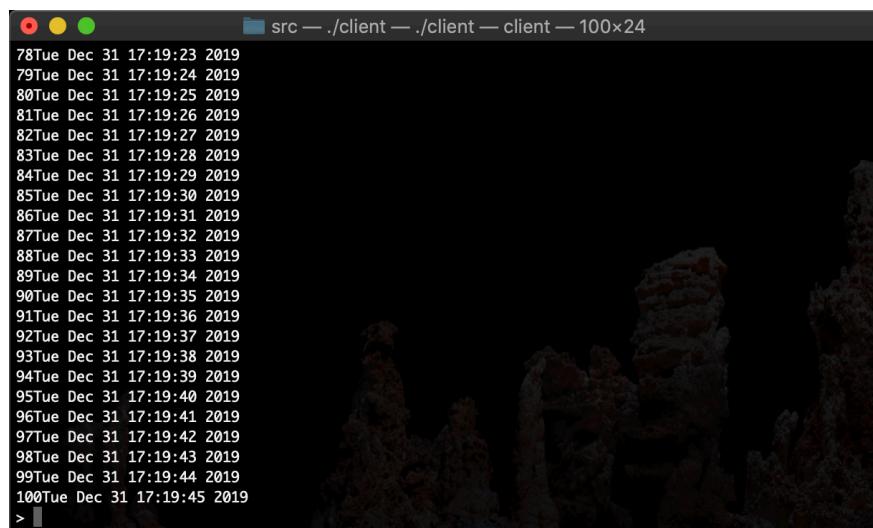
```
[+ ~ netstat -an | grep 4848
tcp4      0      0 127.0.0.1.4848          127.0.0.1.50594      ESTABLISHED
tcp4      0      0 127.0.0.1.50594          127.0.0.1.4848      ESTABLISHED
tcp4      0      0 127.0.0.1.4848          127.0.0.1.49475      CLOSE_WAIT
tcp4      0      0 127.0.0.1.49475          127.0.0.1.4848      FIN_WAIT_2
tcp4      0      0 *.4848                  *.*                  LISTEN
```

图 37: 之前异常退出的连接还在

```
> send 127.0.0.1 4848 test
Fail.
```

图 38: 发送消息失败

17. 修改获取时间功能, 改为用户选择 1 次, 程序内自动发送 100 次请求



The terminal window shows the command `src ./client` running. The output consists of 100 lines of text, each containing a timestamp in the format "xxTue Dec 31 17:19:yy 2019", where xx and yy are digits ranging from 78 to 99. The background of the terminal window is dark, and the text is white.

```
78Tue Dec 31 17:19:23 2019
79Tue Dec 31 17:19:24 2019
80Tue Dec 31 17:19:25 2019
81Tue Dec 31 17:19:26 2019
82Tue Dec 31 17:19:27 2019
83Tue Dec 31 17:19:28 2019
84Tue Dec 31 17:19:29 2019
85Tue Dec 31 17:19:30 2019
86Tue Dec 31 17:19:31 2019
87Tue Dec 31 17:19:32 2019
88Tue Dec 31 17:19:33 2019
89Tue Dec 31 17:19:34 2019
90Tue Dec 31 17:19:35 2019
91Tue Dec 31 17:19:36 2019
92Tue Dec 31 17:19:37 2019
93Tue Dec 31 17:19:38 2019
94Tue Dec 31 17:19:39 2019
95Tue Dec 31 17:19:40 2019
96Tue Dec 31 17:19:41 2019
97Tue Dec 31 17:19:42 2019
98Tue Dec 31 17:19:43 2019
99Tue Dec 31 17:19:44 2019
100Tue Dec 31 17:19:45 2019
```

图 39: 客户端显示内容

No.	Time	Source	Destination	Protocol	Info
181	333.566316	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [ACK] Seq=1 Ack=7 Win=48256 Len=8 Tsval=4942309796 Tsec=r494299786
182	333.566316	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [PSH, ACK] Seq=1 Ack=7 Win=48256 Len=1 Tsval=494231425 Tsec=r494299786
183	335.218142	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [ACK] Seq=7 Ack=8 Win=48256 Len=1 Tsval=494231425 Tsec=r494211425
184	335.218199	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [PSH, ACK] Seq=7 Ack=8 Win=48256 Len=1 Tsval=494231425 Tsec=r494211425
185	335.218199	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [ACK] Seq=8 Ack=9 Win=48256 Len=1 Tsval=494231425 Tsec=r494211425
186	335.218199	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [PSH, ACK] Seq=8 Ack=9 Win=48256 Len=1 Tsval=494231425 Tsec=r494211425
187	336.216153	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [ACK] Seq=18 Ack=18 Win=48256 Len=8 Tsval=494212428 Tsec=r494212428
188	336.216153	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [PSH, ACK] Seq=18 Ack=18 Win=48256 Len=1 Tsval=494212428 Tsec=r494212428
189	336.216203	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [ACK] Seq=19 Ack=19 Win=48256 Len=1 Tsval=494212428 Tsec=r494212428
190	336.216203	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [PSH, ACK] Seq=19 Ack=19 Win=48256 Len=1 Tsval=494212428 Tsec=r494212428
191	337.223396	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [ACK] Seq=23 Ack=4 Win=48256 Len=8 Tsval=494213428 Tsec=r494213428
192	337.223408	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [PSH, ACK] Seq=23 Ack=4 Win=48256 Len=1 Tsval=494213428 Tsec=r494213428
193	337.223408	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [ACK] Seq=4 Ack=5 Win=48256 Len=8 Tsval=494213428 Tsec=r494213428
194	338.226214	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [PSH, ACK] Seq=4 Ack=5 Win=48256 Len=1 Tsval=494213428 Tsec=r494213428
195	338.226214	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [ACK] Seq=5 Ack=6 Win=48256 Len=1 Tsval=494213428 Tsec=r494213428
196	338.226352	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [PSH, ACK] Seq=5 Ack=6 Win=48256 Len=1 Tsval=494214429 Tsec=r494214429
197	338.226400	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [ACK] Seq=5 Ack=5 Win=48256 Len=1 Tsval=494214429 Tsec=r494214429
198	339.238292	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [PSH, ACK] Seq=5 Ack=5 Win=48256 Len=1 Tsval=494214429 Tsec=r494214429
199	339.238348	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [ACK] Seq=51 Ack=6 Win=48256 Len=8 Tsval=494215429 Tsec=r494215429
200	339.238441	127.0.0.1	127.0.0.1	TCP	- 4848 -> 51218 [PSH, ACK] Seq=51 Ack=6 Win=48256 Len=1 Tsval=494215429 Tsec=r494215429
201	339.238441	127.0.0.1	127.0.0.1	TCP	- 51218 -> 4848 [ACK] Seq=6 Ack=7 Win=48256 Len=8 Tsval=494215429 Tsec=r494215429

图 40: Wireshark 抓包内容

18. 多个客户端同时连接服务器，同时发送时间请求

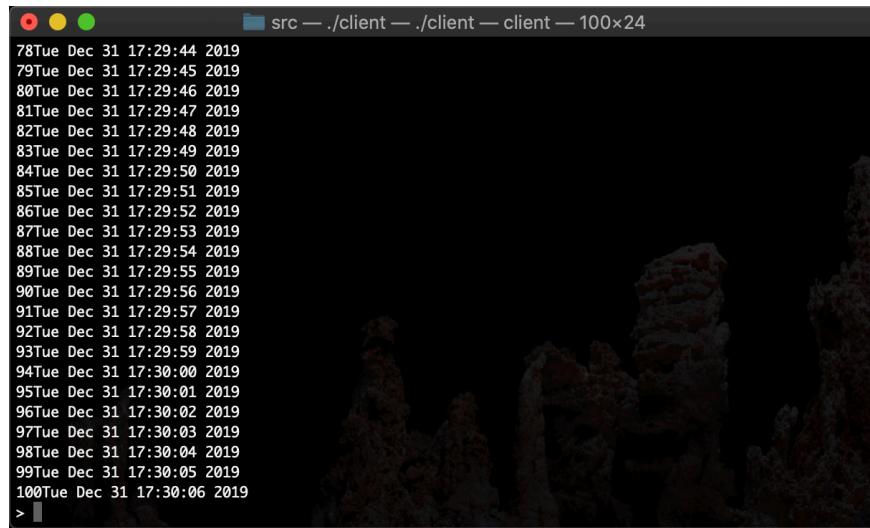


图 41: 客户端 1 显示内容

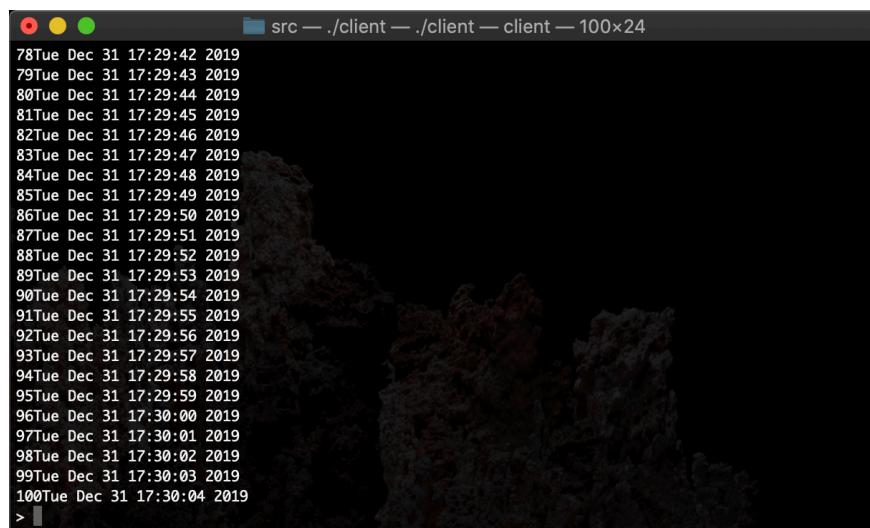
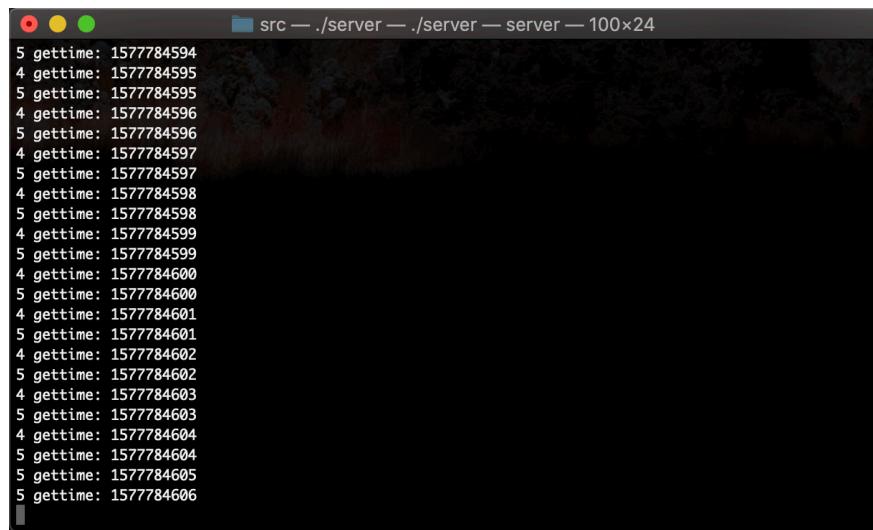


图 42: 客户端 2 显示内容



The screenshot shows a terminal window with the title bar "src — ./server — ./server — server — 100x24". The window contains a series of log entries from a server application. Each entry consists of a timestamp followed by the string "_gettime: 1577784594". The timestamps are repeated in a loop: 1577784594, 1577784595, 1577784595, 1577784596, 1577784596, 1577784597, 1577784597, 1577784598, 1577784598, 1577784599, 1577784599, 1577784600, 1577784600, 1577784601, 1577784601, 1577784602, 1577784602, 1577784603, 1577784603, 1577784604, 1577784604, 1577784605, 1577784605, 1577784606.

图 43: 服务端显示内容

六、 实验结果与分析

1. 客户端是否需要调用 `bind` 操作？它的源端口是如何产生的？每一次调用 `connect` 时客户端的端口是否都保持不变？

客户端不需要 `bind` 操作，调用 `connect` 时 `socket` 为程序自动分配一个未被占用的端口，每一次调用 `connect` 客户端的端口会发生变化。

2. 假设在服务端调用 `listen` 和调用 `accept` 之间设了一个调试断点，暂停在此断点时，此时客户端调用 `connect` 后是否马上能连接成功？

不能连接成功。

3. 连续快速 `send` 多次数据后，通过 Wireshark 抓包看到的发送的 Tcp Segment 次数是否和 `send` 的次数完全一致？

不完全一致。

4. 服务器在同一个端口接收多个客户端的数据，如何能区分数据包是属于哪个客户端的？

通过已经建立的 `socket` 句柄进行区分。

5. 客户端主动断开连接后，当时的 TCP 连接状态是什么？这个状态保持了多久？

客户端由 ESTABLISHED 状态转为 FIN_WAIT1 状态，再转为 FIN_WAIT2 状态。FIN_WAIT1 状态会保持到服务端确认。

6. 客户端断网后异常退出，服务器的 TCP 连接状态有什么变化吗？服务器该如何检测连接是否继续有效？

服务器的 TCP 连接状态不会有变化。可以使用心跳检测的方法。

七、讨论与心得

通过本次编程实验，我了解了如何使用 UNIX 的 Socket 接口进行网络编程，并对网络协议有了更深入的理解。实验过程中，在线程间的同步和通信上遇到过一些困难。我通过调试和查看 Wireshark 的抓包，最终解决了问题，实现了所要求的效果。本实验对数据包协议的定义存在一些可改进之处，如，ip 地址和端口号都可调整为存储为二进制数，而不是文本格式。