

浙江大学

本科实验报告

课程名称： 计算机网络基础

实验名称： 基于 Socket 接口实现自定义协议通信

姓 名： 吕镇峰、钱泽诚

学 院： 信息与工程学院

系：

专 业： 电子科学与技术、通信工程

学 号： 3160101256

指导教师： 陆魁军

2018 年 10 月 14 日

浙江大学实验报告

实验名称: 基于 Socket 接口实现自定义协议通信 实验类型: 编程实验

同组学生: 吕镇峰、钱泽诚 实验地点: 计算机网络实验室

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

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
- 程序界面不做要求, 使用命令行或最简单的窗体即可
- 功能要求如下:
 1. 运输层协议采用 TCP
 2. 客户端采用交互菜单形式, 用户可以选择以下功能:
 - a) 连接: 请求连接到指定地址和端口的服务端
 - b) 断开连接: 断开与服务端的连接
 - c) 获取时间: 请求服务端给出当前时间
 - d) 获取名字: 请求服务端给出其机器的名称
 - e) 活动连接列表: 请求服务端给出当前连接的所有客户端信息 (编号、IP 地址、端口等)
 - f) 发消息: 请求服务端把消息转发给对应编号的客户端, 该客户端收到后显示在屏幕上
 - g) 退出: 断开连接并退出客户端程序
 3. 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:
 - a) 向客户端传送服务端所在机器的当前时间
 - b) 向客户端传送服务端所在机器的名称
 - c) 向客户端传送当前连接的所有客户端信息
 - d) 将某客户端发送过来的内容转发给指定编号的其他客户端
 - e) 采用异步多线程编程模式, 正确处理多个客户端同时连接, 同时发送消息的情况
- 根据上述功能要求, 设计一个客户端和服务端之间的应用通信协议
- 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
- 本实验可组成小组, 服务端和客户端可由不同人来完成

三、主要仪器设备

- 联网的 PC 机、Wireshark 软件
- Visual C++、gcc 等 C++ 集成开发环境。

四、操作方法与实验步骤

- 设计请求、指示（服务器主动发给客户端的）、响应数据包的格式，至少要考虑如下问题：
 - a) 定义两个数据包的边界如何识别
 - b) 定义数据包的请求、指示、响应类型字段
 - c) 定义数据包的长度字段或者结尾标记
 - d) 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）
- 小组分工：1 人负责编写服务端，1 人负责编写客户端
- 客户端编写步骤（需要采用多线程模式）
 - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
 - b) 编写一个菜单功能，列出 7 个选项
 - c) 等待用户选择
 - d) 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）
 - 1. 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。
 - 2. 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。
 - 3. 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。
 - 4. 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。
 - 5. 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）。
 - 6. 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）。
 - 7. 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。
 - 8. 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）。
- 服务端编写步骤（需要采用多线程模式）
 - a) 运行初始化，调用 `socket()`，向操作系统申请 `socket` 句柄
 - b) 调用 `bind()`，绑定监听端口（请使用学号的后 4 位作为服务器的监听端口），接着调用 `listen()`，设置连接等待队列长度
 - c) 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()`。该子线程的主要步骤是（刚获得的句柄要传递给子线程，子线程内部要使用该句柄发送和接收数据）：

- ✧ 调用 `send()`，发送一个 `hello` 消息给客户端（可选）
- ✧ 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：
 1. 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
 2. 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
 3. 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
 4. 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。
- d) 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。
- 编程结束后，双方程序运行，检查是否实现功能要求，如果有问题，查找原因，并修改，直至满足功能要求
- 使用多个客户端同时连接服务端，检查并发性
- 使用 Wireshark 抓取每个功能的交互数据包

五、实验数据记录和处理

请将以下内容和本实验报告一起打包成一个压缩文件上传：

- 源代码：客户端和服务端的代码分别在一个目录
- 可执行文件：可运行的 `.exe` 文件或 `Linux` 可执行文件，客户端和服务端各一个
- 描述请求数据包的格式（画图说明），请求类型的定义

请求数据包格式：

请求类型 (1位)：	数据包内容长度 (3位)：	数据包内容：
时间请求0、名字请求1、列表请求2、消息请求3	仅包括该字段之后的“内容”长度（除了消息请求之外其他数据包均省略长度字段和内容字段）	对于消息请求，使用如下格式： 消息长度(3位)编号(2位)消息

请求类型编码定义：

```
//请求数据包编号
REQ_TIME,      //时间请求, 0
REQ_NAME,      //名字请求, 1
REQ_LIST,      //列表请求, 2
REQ_MSG,       //消息请求, 3
```

- 描述响应数据包的格式（画图说明），响应类型的定义

响应数据包格式：

响应类型（1位）： 时间响应4、名字响应5、列表响应6、 消息响应7、连接响应9	数据包内容长度（3位）： 仅包括该字段之后的“内容”长度	数据包内容
--	---------------------------------	-------

响应类型编码定义：

```
//响应数据包编号
RES_TIME,      //时间响应, 4
RES_NAME,      //名字响应, 5
RES_LIST,      //列表响应, 6
RES_CON       //连接响应, 9
```

- 描述指示数据包的格式（画图说明），指示类型的定义

指示数据包格式：

指示类型（1位）： 仅有消息指示8	数据包内容长度（3位）： 仅包括该字段之后的“内容”长度	数据包内容： 编号_时间_内容
----------------------	---------------------------------	--------------------

指示类型编码定义：

```
//指示数据包编号
COM_MSG,      //消息指示, 8
```

- 客户端初始运行后显示的菜单选项



- 客户端的主线程循环关键代码截图（描述总体，省略细节部分）

菜单循环逻辑：

```
//显示菜单并根据选择进行相应操作
//1. 连接
//2. 断开
//3. 获取时间
//4. 获取名字
//5. 获取客户端列表
//6. 发送消息
//7. 退出
while (1) {
    if (!sClient.isConnected()) {        //根据客户端连接状态判断
        cout << "\n* * * * * * * * * * \n"
            << "* 客户端状态: 未连接... \n"
            << "* 1. 连接 \n"
            << "* 2. 退出 \n"
            << "* * * * * * * * * * \n"
            << "> 输入选项序号以进行选择（按回车键结束）: ";

        while (1) {
            cin >> choice;
            if (choice == 1 || choice == 2) {
                //输入选项符合要求，进行处理
                switch (choice) { ... }
                break;
            }
            else {
                cout << "> 输入有误，请重新输入选择: ";
                cin.clear();                //恢复输入流状态位
                cin.ignore(1024, '\n');    //清空输入缓冲区内容
            }
        }
    }
}

else { //客户端已连接
    mxPrint.lock();                //打印菜单时进行锁定，防止子线程同时输出造成混乱
    cout << "\n* * * * * * * * * * \n"
        << "* 客户端状态: 已连接... \n"
        << "* 1. 连接 \n"
        << "* 2. 断开连接 \n"
        << "* 3. 获取时间 \n"
        << "* 4. 获取名字 \n"
        << "* 5. 获取客户端列表 \n"
        << "* 6. 发送消息 \n"
        << "* 7. 退出 \n"
        << "* * * * * * * * * * \n"
        << "> 输入选项序号以进行选择（按回车键结束）: ";
    mxPrint.unlock();
    while (1) {
        cin >> choice;
        if (choice >= 1 && choice <= 7) {
            //输入选项符合要求，进行处理
            switch (choice) { ... }
            break;
        }
        else {
            cout << "> 输入有误，请重新输入选择: ";
            cin.clear();
            cin.ignore(1024, '\n');
        }
    }
}

return 0;
}
```

响应消息处理函数（阻塞等待）：

```
//消息队列处理函数
void procMsgList(deque<Msg>& msgList, mutex& mx) {
    bool hasResponse=false;
    MsgType type;

    //等待接收到响应消息
    while (!hasResponse) { ... }

    string id;
    string ip;
    string port;
    int index_f = 0;
    int index_l;

    mx.lock(); //开始处理时进行锁定，防止指示消息处理线程对消息队列进行添加/删除导致的迭代器失效
    auto iter = msgList.begin();
    while (iter != msgList.end()) { //遍历消息列表并根据消息类型做出相应处理
        switch (iter->msgType) { ... }
    }
    mx.unlock();
}
```

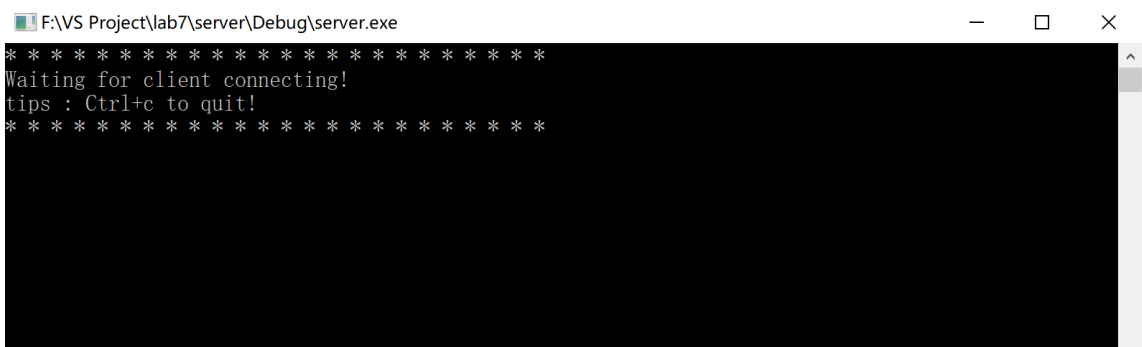
- 客户端的接收数据子线程循环关键代码截图（描述总体，省略细节部分）

```
//消息监听线程函数
void threadListen(SocketClient& sc, deque<Msg>& msgList, mutex& mx, const bool& exitSignal) {
    static enum RecState {
        rec_type, //类型识别阶段
        rec_length, //长度识别阶段
        rec_content //内容识别阶段
    };

    while (sc.isConnected() == false); //开启后进入循环，当客户端连接成功之后进行消息监听

    Msg msgTemp;
    char c;
    char strLen[3];
    int iLen;
    MsgType _type;
    string strContent;
    RecState rs = rec_type;
    SOCKET sock = sc.getSocket();
    int cnt = 0;
    while (1) {
        if (exitSignal) return;
        recv(sock, &c, 1, 0);
        switch (rs) {
            case rec_type: //识别状态处于rec_type，表明未开始接受消息包，则进行消息包类别判断，修改状态为rec_length，开始读取长度字段
                _type= MsgType(c - '0');
                if (_type >= 0 && _type <= 9) { ... }
                else continue;
            case rec_length: //识别状态处于rec_length，读取长度字段中
                strLen[cnt++] = c; //将三位长度字段存放放到strLen数组内
                if (cnt == LENGTH_BITS) { ... }
                break;
            case rec_content: //识别状态处于rec_content，读取内容字段中
                strContent.push_back(c);
                cnt++;
                if (cnt == iLen) { ... }
                break;
        }
    }
}
```

- 服务器初始运行后显示的界面



- 服务器的主线程循环关键代码截图（描述总体，省略细节部分）

```
//阻塞等待客户端连接
while (true)
{
    //如果总的线程退出信号为true, 则跳出循环而不是阻塞在等待连接的地方
    if (exitsignal == true)
        break;
    sServer = accept(sListen, (struct sockaddr *)&saClient, &length);
    if (sServer == INVALID_SOCKET)
    {
        printf("accept() failed! code:%d Please connect again!\n", WSAGetLastError());
        continue; //如果连接失败, 则进入下一轮循环等待
    }
    printf("Accepted Client: %s:%d\n", inet_ntoa(saClient.sin_addr), ntohs(saClient.sin_port)); //如果连接成功, 打印连接成功的客户端信息
    saClientlist[clientcount].sServer = sServer;
    saClientlist[clientcount].clientnumebr = clientcount; //client编号
    saClientlist[clientcount].isalive = true; //标志为存活的client
    saClientlist[clientcount].saClient = saClient; //存放client的地址结构信息
    handle[clientcount] = (HANDLE)_beginthreadex(NULL, 0, ThreadFun, &saClientlist[clientcount], 0, NULL); //线程句柄
    clientcount++; //客户端计数加1
}
```

主线程负责监听连接请求，当收到连接请求时创建新线程，让新线程负责消息收发。主线程采用的是阻塞式模型，当出现新的连接之前，阻塞在 `accept` 函数处。创建子线程前传入结构指针主要用于线程间通信。

- 服务器的客户端处理子线程循环关键代码截图（描述总体，省略细节部分）

```
while (true)
{
    //如果收到当前线程退出的信号或收到全部线程退出信号
    if (curClient->prosignal == true || exitsignal == true)
    {
        //如果主线程收到结束信号, 则终止各个子线程, 子线程用return方式终止, 最为安全的方式
        curClient->isalive = false;
        closesocket(curClient->sServer); //关闭连接套接字
        return -1;
    }

    ret = pacRecv(curClient, (char *)&pactype, PAC_TYPE_LEN); //指定消息接受函数去接受两个字符长度的数据包类型字段

    //当pacRecv函数没有检测到远端退出信号的时候再去发送数据包, 不然会多发一次
    if (!(curClient->prosignal == true || exitsignal == true))
        pacSend((struct clientlist *)pM, pactype);
}
```

子线程检查全局量是否指示子线程退出，然后选择退出或者准备收发消息。`pacRecv` 阻塞等待消息。收到消息后再进入 `pacSend` 准备发送响应消息。

- 客户端选择连接功能时，客户端和服务端显示内容截图。

客户端：

```

* * * * *
* 客户端状态：未连接...
* 1. 连接
* 2. 退出
* * * * *
> 输入选项序号以进行选择（按回车键结束）：1
> 请输入目标服务器的ip地址：10.110.6.33

[系统消息] 连接成功!
[服务器响应] Hello

```

服务器：

```

* * * * *
Waiting for client connecting!
tips : Ctrl+c to quit!
* * * * *
Accepted Client: 10.110.6.33:13782
Server say Hello to client: 10.110.6.33:13782
* * * * *

```

Wireshark 抓取的数据包截图：

0000	00 1c 42 41 04 ca 00 1c 42 00 00 18 08 00 45 00	..BA....B.....E.
0010	00 32 c3 d3 00 00 80 06 e8 54 0a b4 42 12 0a d3	.2.....T..B...
0020	37 05 01 8c f6 56 dc 36 95 54 16 d7 b5 5e 50 18	7....V.6 .T...^P.
0030	40 00 1e 49 00 00 39 30 30 35 48 65 6c 6c 6f 00	@..I..90 05Hello.

*画线部分为交互字符串：绿色为类型编码（1位，“9”表示连接响应数据包），红色为内容长度（3位），黄色为内容

- 客户端选择获取时间功能时，客户端和服务端显示内容截图。

客户端：

```

* * * * *
* 客户端状态：已连接...
* 1. 连接
* 2. 断开连接
* 3. 获取时间
* 4. 获取名字
* 5. 获取客户端列表
* 6. 发送消息
* 7. 退出
* * * * *
> 输入选项序号以进行选择（按回车键结束）：3

[服务器响应] 当前服务器时间为（年-月-日-时-分-秒）：2018-10-15-20-47-49

```

服务器：

```

Server time: 2018-10-15-20-47-49
Client Number 0 requestes system time on server!
* * * * *

```

Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的时间数据对应的位置）：

0000	00 1c 42 41 04 ca 00 1c 42 00 00 18 08 00 45 00	..BA.... B.....E.
0010	00 3f c4 bf 00 00 80 06 e7 5b 0a b4 42 12 0a d3	.?..... [..B...
0020	37 05 01 8c f6 56 dc 36 95 5e 16 d7 b5 60 50 18	7....V.6 .^....`P.
0030	40 00 56 3d 00 00 34 30 31 39 32 30 31 38 2d 31	@.V=..40 192018-1
0040	30 2d 31 36 2d 31 38 2d 34 34 2d 31 37	0-16-18- 44-17

*画线部分为交互字符串：红色为类型编码（1位，“4”表示时间响应数据包），绿色为内容长度（3位），黄色为内容

- 客户端选择获取名字功能时，客户端和服务端显示内容截图。

客户端：

```
*****
* 客户端状态：已连接...
* 1.连接
* 2.断开连接
* 3.获取时间
* 4.获取名字
* 5.获取客户端列表
* 6.发送消息
* 7.退出
*****
> 输入选项序号以进行选择（按回车键结束）：4
[服务器响应] 服务器主机名为：ASUS-Kaze
```

服务器：

```
Hostname on server:ASUS-Kaze
Client Number 0 requestes hostname on server!
*****
```

Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的名字数据对应的位置）：

0000	00 1c 42 41 04 ca 00 1c 42 00 00 18 08 00 45 00	..BA.... B.....E.
0010	00 38 c6 f2 00 00 80 06 e5 2f 0a b4 42 12 0a d3	.8..... ./...B...
0020	37 05 01 8c f6 56 dc 36 95 75 16 d7 b5 61 50 18	7....V.6 .u....aP.
0030	40 00 f2 c0 00 00 35 30 31 32 53 75 72 66 61 63	@.....50 12Surfac
0040	65 2d 4b 61 7a 65	e-Kaze

*画线部分为交互字符串：绿色为类型编码（1位，“5”表示名字响应数据包），红色为内容长度（3位），黄色为内容

相关的服务器的处理代码片段：

```
else if (strcmp(pactype, REQ_NAME) == 0) //如果是请求服务器名字的数据包，系统调用获取名字
{
    char hostname[100] = { 0 };
    if (gethostname(hostname, sizeof(hostname)) < 0)
    {
        //错误并返回
    }
    printf("Hostname on server:%s\n", hostname);
    length = strlen(hostname);
    sprintf(slength, "%03d", length);
    strcat(message, RES_NAME);
    strcat(message, slength);
    strcat(message, hostname);
    printf("Client Number %d requestes hostname on server!\n", pM->clientnumebr); //server上打印哪个客户端请求了服务器主机名
    printf("*****\n");
    ret = send(pM->sServer, (char *)&message, strlen(message), 0);
}
```

- 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截图。

客户端：

```

* * * * *
* 客户端状态：已连接...
* 1. 连接
* 2. 断开连接
* 3. 获取时间
* 4. 获取名字
* 5. 获取客户端列表
* 6. 发送消息
* 7. 退出
* * * * *
> 输入选项序号以进行选择（按回车键结束）：5

[服务器响应] 与本客户端连接到相同服务器的客户端列表如下：
编号      IP地址      端口
0          10.110.6.33    14111

[系统消息] 打印完毕！

```

服务器：

```

Client Number 0 requestes clientlist on server!
* * * * *

```

Wireshark 抓取的数据包截图（展开应用层数据包，标记请求、响应类型、返回的客户端列表数据对应的位置）：

0000	00 1c 42 41 04 ca 00 1c 42 00 00 18 08 00 45 00	..BA....B.....E.
0010	00 41 c7 16 00 00 80 06 e5 02 0a b4 42 12 0a d3	.A.....B...
0020	37 05 01 8c f6 56 dc 36 95 85 16 d7 b5 62 50 18	7....V.6.....bP.
0030	40 00 11 e5 00 00 36 30 32 31 31 2d 31 30 2e 31	@.....60 211-10.1
0040	38 30 2e 37 30 2e 31 35 38 2d 35 37 38 38 33	80.70.15 8-57883

*画线部分为交互字符串：红色为类型编码（1位，“6”表示客户端列表响应数据包），绿色为内容长度（3位），黄色为内容

相关的服务器的处理代码片段：

```

else if (strcmp(pactype, REQ_LIST) == 0)
{
    //如果是请求连接列表的数据包，列出所有的连接服务器列表
    int ccount = 0; //活着的主机数
    strcat(fmessage, RES_LIST); //标志列表响应数据包
    for (int i = 0; i < MAX_NUM; i++)
    {
        if (saClientlist[i].isalive == true)
        {
            //如果是存活的客户端，打包客户端信息
            if (ccount == 0) { //如果是第一个要打印的
                sprintf(newmsg, "%d-%s-%d", saClientlist[i].clientnumebr, inet_ntoa(saClientlist[i].saClient.sin_addr), ntohs(saClientlist[i].saClient.sin_port));
                strcat(message, newmsg);
            }
            else { //如果前面已经有了一些字段
                sprintf(newmsg, "%d-%s-%d", saClientlist[i].clientnumebr, inet_ntoa(saClientlist[i].saClient.sin_addr), ntohs(saClientlist[i].saClient.sin_port));
                strcat(message, newmsg);
            }
            ccount++;
        }
    }
    sprintf(slength, "%03d", strlen(message));
    strcat(fmessage, slength);
    strcat(fmessage, message);
    printf("Client Number %d requestes clientlist on server!\n", pM->clientnumebr); //server上打印说哪个客户端请求了连接服务器列表
    printf("*****\n");
    ret = send(pM->sServer, (char *)&fmessage, strlen(fmessage), 0);
}

```

- 客户端选择发送消息功能时，客户端和服务端显示内容截图。

发送消息的客户端：

```

* * * * *
* 客户端状态：已连接...
* 1. 连接
* 2. 断开连接
* 3. 获取时间
* 4. 获取名字
* 5. 获取客户端列表
* 6. 发送消息
* 7. 退出
* * * * *
> 输入选项序号以进行选择（按回车键结束）：6
> 请输入你想要发送消息的目标客户端序号：0
> 请输入你想要发送的消息内容：hello
[服务器响应] Send Succeed!

```

服务器：

```

* * * * *
Waiting for client connecting!
tips : Ctrl+c to quit!
* * * * *
Accepted Client: 10.211.55.5:50596
Server say Hello to client: 10.211.55.5:50596
* * * * *
Client number: 0 say hello to client number: 0

```

接收消息的客户端：

```

[系统消息] 接收到其他客户端消息：
编号：01
时间（年-月-日-时-分-秒）：2018-10-15-21-0-18
内容：hello

```

Wireshark 抓取的数据包截图（发送和接收分别标记）：

发送：

0000	00 1c 42 00 00 18 00 1c 42 41 04 ca 08 00 45 00	..B.....BA....E.
0010	00 33 41 f1 40 00 40 06 00 00 0a d3 37 05 0a b4	.3A.@.@.7...
0020	42 12 f6 56 01 8c 16 d7 b5 6d dc 36 95 ae 50 18	B.V.....m.6.P.
0030	01 00 8e c3 00 00 33 30 30 35 30 31 68 65 6c 6c300501hell
0040	6f	o

*画线部分为交互字符串：红色为类型编码（1 位，“3”表示消息请求数据包），绿色为内容长度（3 位），蓝色为客户端编号（2 位），黄色为内容

接收：

0000	00 1c 42 41 04 ca 00 1c 42 00 00 18 08 00 45 00	..BA....B.....E.
0010	00 48 c7 70 00 00 80 06 e4 a1 0a b4 42 12 0a d3	.H.p.....B...
0020	37 05 01 8c f6 56 dc 36 95 ae 16 d7 b5 78 50 18	7....V.6xP.
0030	40 00 bb 27 00 00 38 30 32 38 30 31 5f 32 30 31	@...'..80 2801 201
0040	38 2d 31 30 2d 31 36 2d 31 38 2d 34 38 2d 31 38	8-10-16- 18-48-18
0050	5f 68 65 6c 6c 6f	_hello

*画线部分为交互字符串：红色为类型编码（1 位，“8”表示指示消息数据包），绿色为内容长度（3 位），黄色为内容

相关的服务器的处理代码片段：

```
else if (strcmp(pactype, REQ_MSG) == 0) //如果是消息转发请求的数据包，需要长度字段和内容，继续读取后续字段
{
    getTime(timestring);
    length = strlen(timestring);
    pacTransMessage(pM, (char *)&message, &dest); //内容长度
    strcat(fmmessage, COM_MSG); //继续从sServer套接字接收消息，并存放放到message里面并发送
    sprintf(slength, "%03d", strlen(message) + 4 + strlen(timestring)); //标志为指示数据包
    strcat(fmmessage, slength); //3位打印消息长度
    sprintf(newmsg, "%02d", pM->clientnumebr); //字符串连接，补上长度项
    strcat(fmmessage, newmsg); //源客户端的编号
    strcat(fmmessage, "_"); //加上这个字节
    strcat(fmmessage, timestring); //用下划线分隔
    strcat(fmmessage, "_"); //时间字段
    strcat(fmmessage, message); //补上内容，补全数据包
    printf("Client number: %d say %s to client number: %d\n", pM->clientnumebr, message, dest); //服务器上打印一下发送了什么消息

    if (saClientlist[dest].isalive == true)
    {
        ret = send(saClientlist[dest].sServer, (char *)&fmmessage, strlen(fmmessage), 0); //发送数据包到目的主机
        if (ret == SOCKET_ERROR)
            printf("Send failed!\n");
        sprintf(message, "%s", "7013Send Succeed!");
        ret = send(pM->sServer, (char *)&message, strlen(message), 0); //返回消息给客户端表示发送成功
        if (ret == SOCKET_ERROR)
            printf("Send failed!\n");
    }
    else
    {
        sprintf(message, "%s", "7012Send Failed!");
        ret = send(pM->sServer, (char *)&message, strlen(message), 0); //返回消息给客户端表示发送失败
    }
}
```

相关的客户端（发送和接收消息）处理代码片段：

发送：

```
int SocketClient::sendMsg(const int id, const string& msg) {
    char strID[3];
    sprintf_s(strID, 3, "%02d", id); //格式化编号，2位
    char strLen[4];
    sprintf_s(strLen, 4, "%03d", msg.length()); //格式化消息长度，3位
    //根据规则组装得到要发送的字符串
    string msg2send = to_string(REQ_MSG) + string(strLen) + string(strID) + msg;
    int ret = send(sClient, msg2send.c_str(), msg2send.length(), 0);
    if (ret == SOCKET_ERROR) cout << "[系统消息] 消息请求发送失败" << endl;
    return ret;
}
```

接收处理：

```
auto iter = msgList.begin();
while (iter != msgList.end()) {
    if (iter->msgType == COM_MSG) {
        msg = iter->content;
        index_l = msg.find("_", index_f);
        id = msg.substr(index_f, index_l - index_f); //分离得到源客户端编号
        index_f = index_l + 1;
        index_l = msg.find("_", index_f);
        time = msg.substr(index_f, index_l - index_f); //分离得到时间
        index_f = index_l + 1;
        msg = msg.substr(index_f); //分离得到消息内容
        mxPrint.lock();
        cout << "\n[系统消息] 接收到其他客户端消息: " << endl;
        cout << "编号: " + id << endl << "时间(年-月-日-时-分-秒): " + time << endl << "内容:" + msg << endl;
        mxPrint.unlock();
        iter = msgList.erase(iter);
    }
    else iter++; //该消息不是指示消息，查看下一个
}
```

- 拔掉客户端的网线，然后退出客户端程序。观察客户端的 TCP 连接状态，并使用 Wireshark 观察客户端是否发出了 TCP 连接释放的消息。同时观察服务端的 TCP 连接状态在较长时间内（10 分钟以上）是否发生变化。

答：客户端断网并退出程序时。客户端方 TCP 连接状态没有和服务器端的连接。使用 Wireshark 观察到客户端试图发出 TCP 连接释放的消息，但由于断网没有成功发出，显示为红色状态。（如下图所示）

1189 70.307163		10.211.55.5		10.180.66.18		TCP		54 56993 → 396 [RST, ACK] Seq=1 Ack=11 Win=0 Len=0									
0000	00 1c 42 00 00 18 00 1c	42 41 04 ca 08 00 45 00	..B.....BA.....E.														
0010	00 28 42 6c 40 00 40 06	00 00 0a d3 37 05 0a b4	.(Bl@.@.7...														
0020	42 12 de a1 01 8c 75 7f	25 7b 0b 54 13 24 50 14	B.....u. %{.T.\$P.														
0030	00 00 8e b8 00 00																

经过十五分钟之后，服务器端 TCP 连接状态仍然不变，与客户端刚断网时相同，对应的 TCP 连接状态处于 ESTABLISHED

- 再次连上客户端的网线，重新运行客户端程序。选择连接功能，连上后选择获取客户端列表功能，查看之前异常退出的连接是否还在。选择给这个之前异常退出的客户端连接发送消息，出现了什么情况？

答：重新运行客户端并连接获取客户端列表后，之前异常退出的连接仍然在，且对其发送消息之后发送成功。Wireshark 抓取到服务器发送给目标客户端的数据包

	5875 1648.691612	10.180.66.18	10.180.70.158	TCP
0000	88 e0 f3 b2 60 ce bc 83	85 dc e1 db 08 00 45 00`... ..E.	
0010	00 53 36 62 40 00 80 06	26 2b 0a b4 42 12 0a b4	.S6b@... &+..B...	
0020	46 9e 01 8c e3 92 86 4a	26 6c d8 87 49 df 80 18	F.....J &l..I...	
0030	00 43 4c cd 00 00 01 01	08 0a 02 14 85 ff 0b af	.CL..... ..	
0040	3d ec 38 30 32 37 30 31	5f 32 30 31 38 2d 31 30	=.802701 _2018-10	
0050	2d 31 36 2d 31 39 2d 33	34 2d 37 5f 68 65 6c 6c	-16-19-3 4-7_hell	
0060	70			

- 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。服务器是否正常处理了 100 次请求，截取客户端收到的响应（通过程序计数一下是否有 100 个响应回来），并使用 Wireshark 抓取数据包，观察实际发出的数据包个数。


```

*****
Server time: 2018-10-16-20-16-10
Client Number 0 requestes system time on server!
*****
Server time: 2018-10-16-20-16-10
Client Number 2 requestes system time on server!
*****
Server time: 2018-10-16-20-16-10
Client Number 0 requestes system time on server!
*****
Server time: 2018-10-16-20-16-10
Client Number 2 requestes system time on server!
*****
Server time: 2018-10-16-20-16-10
Client Number 2 requestes system time on server!
*****
Server time: 2018-10-16-20-16-10
Client Number 0 requestes system time on server!
*****

```

六、 实验结果与分析

- 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？

答：不需要。操作系统会在客户端上自动选择一个未被占用的端口，分配为客户端的源端口。每次调用 connect 时客户端端口会发生改变，因为上一时刻连接所用的端口可能在下一次连接被占用，因此操作系统可能会选择另一个未被占用的端口用于创建 socket 通信。

- 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？

答：如果服务端连接队列未滿就可以连接成功。listen 函数将套接字变成被动的连接监听套接字之后，客户端调用 connect 就可以与服务端进行三次握手建立连接。服务端 accept 函数会从连接队列取出 establish 状态的连接。但是如果服务端的连接队列已滿，则不会再响应客户端的 connect 请求。

- 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？

答：一致。

- 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？

答：服务器根据数据包来源的 IP 地址和端口区分客户端。

- 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？（可以使用 netstat -an 查看）

答：客户端主动断开连接后，当时的 TCP 连接状态为 TIME_WAIT，状态持续了大概 2 分钟左右后该 TCP 连接消失。

- 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？

答：没有变化。为了检测类似的意外退出情况，服务器和客户端之间应该添加心跳包机制，客户端每隔固定时间发送在线确认数据包给服务器，若服务器长时间没有收到过某一客户端的心跳包则判断客户端掉线。

七、 讨论、心得

1、 客户端

最初设计客户端逻辑时使用两个子线程函数，一个负责接收服务器发送的数据包，另一个负责处理消息队列，而主线程仅负责处理界面交互。但这样实现的问题在于不能满足设计要求的阻塞等待响应消息，于是更改了消息处理逻辑：仍然保留实时处理消息队列的子线程，但仅处理指示消息，这样就能实时显示其他客户端发送的消息内容；其他类型的消息使用普通函数处理，每次发送消息之后调用，以此来满足阻塞需求。

消息队列对于多个线程是可访问、操作的，这样就可能导致迭代器失效问题（如线程 A 使用临时变量存储消息队列的某个迭代器之后，线程 B 对消息队列进行了添加/删除操作，就有可能导致迭代器失效）。为解决这个问题引入线程互斥锁 `mutex` 类，在某个线程获取迭代器之前进行锁定，直到操作完成后解锁，这样就能避免迭代器失效问题。

由于未开发图形界面，故所有消息均显示在终端窗口。主线程在打印菜单的同时，实时处理指示消息的子线程可能也在打印消息内容，这时输出内容就会混在一起。为了解决这个问题，同样可以使用线程互斥锁，在一方输出时进行锁定，输出完毕后解锁，该问题由此得到解决。

2、 服务器端

服务端逻辑是由主线程监听来自客户端的连接，每当成功建立连接则创建一个子线程负责与请求连接的客户端进行通信。然后再子线程里负责与收发数据包与客户端进行通信。一开始有考虑用异步模型，让服务端相应消息的效率更高，但是相对会降低服务端的可靠性，综合各方面考虑最后还是选择了能够可靠传输的同步模型。但是在实验过程中我们也发现了此模型的缺点，同步模型有时候比较难实现用户级交互，例如用户很难给主线程发消息，因为主线程大部分时间都会阻塞在监听客户端的连接请求，而无法响应用户的命令。

服务端遇到的另一个问题是难以验证客户端是否存活。我们设计的逻辑是当服务器要求客户端相应而客户端无法相应或收到客户端的终止连接信号时，则判断客户端已断开连接。但遇到拔网线的问题这种逻辑就出现了漏洞，服务端无法判断客户端是否仍然处于连接态。查阅相关资料后，我们学习到可以通过建立一种心跳包机制，让连接状态的客户端每隔一段时间给服务端发送数据包表示自己还活着。

整个合作过程并不一番风顺，开发期间遇到了许许多多的 **bug**，但最后都被我们一一解决。本次实验也锻炼了我们团队协作的能力。通过这次的实验，我们不仅学习到了 **Socket** 编程底层 **API** 的原理和应用方法。还对网络知识有了更加深入的了解。希望能够通过日后的实践，积累更多书本内和书本外的知识，做到学以致用。