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： | 蒋仕彪 |
| 学　　院： | 计算机学院 |
| 系： | 竺可桢学院求是科学班（计算机） |
| 专　　业： | 计算机科学与技术 |
| 学　　号： | 3170102587 |
| 指导教师： | 董玮 |

2020 年　5　月　19 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：蒋仕彪，李广林　实验地点：计算机网络实验室

## 一、 实验目的

- 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
- **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
- Visual C++、gcc 等 C++集成开发环境。

## 四、 操作方法与实验步骤

- 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()，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：调用 send()将获取时间请求发送给服务器，**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：调用 send()将获取名字请求发送给服务器，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：调用 send()将获取客户端列表信息请求发送给服务器，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，然后调用 send()将数据发送给服务器，观察另外一个客户端是否收到数据。
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，并调用 send()发给客户端
2. 请求类型为获取名字：调用 GetComputerName 获取本机名，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、端口等数据通过调用 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，将要转发的消息组

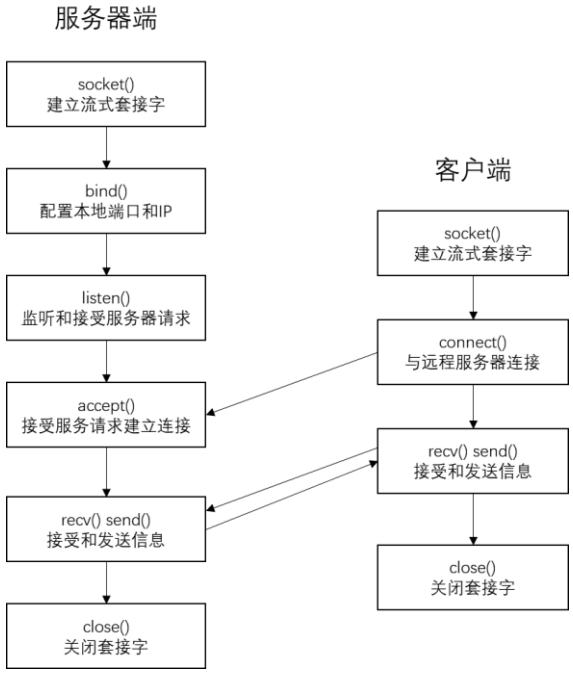
装通过调用 send()发给接收客户端（使用接收客户端的 socket 句柄）。

- 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

## 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**
- 客户端和服务器端框架图（用流程图表示）




- 客户端初始运行后显示的菜单选项

```
jiangshibiao@LAPTOP-F9VPKPAM:~$ ./client
**********************************
connect [IP] [PORT] : connect to the server, e.g. connect 0.0.0.0 2587
time : show the server time
name : show the client name
clients : show the clients connected to the server
close : close the connection to the server
send [ID] [MSG] : send [MSG] to the client [ID], input _exit_ to stop sending message
exit : exit the program
**********************************
```

- 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）

```
void* listening()
{
        while(1){
                int ok=recv(cid,recbuf,size,0);
                if(recbuf[0]==0){
                        pthread_mutex_lock(&mreceived);
                        memcpy(buf,recbuf,size);
                        pthread_cond_signal(&received);
                        pthread_mutex_unlock(&mreceived);
                }else{
                        if(strncmp(recbuf+1,"close",5)==0){
                                close(cid);
                                break;
                        }
                        else{
                                int sid = *(int *)&recbuf[1];
                                printf("[MSG] client %d send:%s\n",sid,recbuf+5);
                        }
                }
        }
}

int recmessage(char *cbuf)
{
        if(connected==0){
                printf("[MSG] No connection is running.\n");
                return 0;
        }
        int sid = *(int *)&cbuf[4];
        pthread_mutex_lock(&mreceived);
        send(cid,cbuf,size,0);
        pthread_cond_wait(&received,&mreceived);
        pthread_mutex_unlock(&mreceived);
        return 1;
}
```

客户端不断接受来自服务器端的消息，服务器端的消息在开始位置标志了该信息是客户端请求的返回信息还是其他客户端发来的信息，如果是请求返回的信息，则利用条件变量机制，让等待结果返回的主线程收到信息继续向下运行，否则直接将其他客户端发来的信息输出。

- 服务器初始运行后显示的界面

```
jiangshibiao@LAPTOP-F9VPKPAM:~$ ./server
[OK] Build socket 3 successfully.
[OK] Bind (IP: 0.0.0.0 , port 2587) successfully.
Listen successfully, waiting clients...
```

- 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）

```
void *solveClient(void *clientID){
    int id = *((int *) clientID);
    printf("[MSG] Client %d starts!\n", id);
    char buf[BUFFER_SIZE+10], sd[BUFFER_SIZE];
    for (;;){
        memset(buf, 0, BUFFER_SIZE);
        memset(sd, 0, BUFFER_SIZE);
        int message = recv(ConnectFD[id], buf, BUFFER_SIZE, 0);
        if (message == -1 || same(buf, "close")){
            if (message == -1)
                printf("[FAIL] Client %d receive fails!\n", id);
            else
                printf("[MSG] Client %d shutdown.\n", id);
            memcpy(sd,"1close",6);
            send(ConnectFD[id],sd,BUFFER_SIZE,0);
            shutdown(ConnectFD[id], SHUT_RDWR);
            close(ConnectFD[id]);
            used[id] = 0;
            return;
        }
        if (same(buf, "time")){
        if (same(buf, "name")){
        if (same(buf, "clients")){
        if (same(buf, "send")){
    }
}
```

每当新连接一个 client 的时候，开一个新的线程叫做 solveClient，把对应的 client ID 传进去。然后在此函数里无限循环，处理对应的 time、name、clients、send 等用户请求。当收到 close 后，会往 client 发一个确认信息，并中断该线程。

- 客户端选择连接功能时，客户端和服务端显示内容截图。

```
connect 0.0.0.0 2587
[MSG] connected to the server 0.0.0.0.
```

```
Listen successfully, waiting clients...
[MSG] Client 0 starts!
```

- 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。

```
time
[MSG] time is 2020-05-19 11:45:42
```

- 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。

```
name
[MSG] name is LAPTOP-F9VPKPAM
```

相关的服务器的处理代码片段：

```
if (same(buf, "name")){
    sd[0] = 0; gethostname(sd+1, BUFFER_SIZE);
    send(ConnectFD[id], sd, BUFFER_SIZE, 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。

```
clients
[MSG] The 2 clients informations are as follows...
[ID] 00  [IP] 127.0.0.1  [PORT] 27098
[ID] 01  [IP] 127.0.0.1  [PORT] 43482
```

相关的服务器的处理代码片段：

```
if (same(buf, "clients")){
    int offset = 5, cnt = 0;
    for (int i = 0; i < MAX_QUEUE_LENGTH; i++)
        if (used[i]){
            memcpy(sd + offset, (char*)&i, sizeof(int));
            offset += sizeof(int);
            memcpy(sd + offset, (char*)&client_address[i], sizeof(struct sockaddr_in));
            offset += sizeof(struct sockaddr_in);
            cnt++;
        }
    assert (sizeof(int) == 4);
    sd[0] = 0; memcpy(sd + 1, (char*)&cnt, sizeof(int));
    send(ConnectFD[id], sd, BUFFER_SIZE, 0);
}
```

```
printf("[MSG] The %d clients informations are as follows...\n", cnt);
int offset=5;
for (int i=0;i<cnt;i++){
        int id=*((int*)&buf[offset]);
        offset+=sizeof(int);
        struct sockaddr_in t=*((struct sockaddr_in*)&buf[offset]);
        offset+=sizeof(struct sockaddr_in);
        printf("[ID] %02d\t [IP] %s\t [PORT] %u\n",id,inet_ntoa(t.sin_addr), ntohs(t.sin_port));
}
```

服务器把对应数据压缩到 char*里传输。第 1 位是标识位 0，2~5 位是一个 int 描述目前在线的 clients 总数 cnt，随后一共有 cnt 段内容，每段由 id 和 sockaddr_in 组成，描述 client 的 id 和地址。最终由客户端负责解压并展示给用户。

- 客户端选择发送消息功能时，两个客户端和服务端（如果有的话）显示内容截图。

  发送消息的客户端：

```
clients
[MSG] The 2 clients informations are as follows...
[ID] 00  [IP] 127.0.0.1  [PORT] 53211
[ID] 01  [IP] 127.0.0.1  [PORT] 56795
send 1 Hello client1!
[MSG] **input _exit_ to stop sending message.**
 Can you hear me?
_exit_
[MSG] stopped sending message
```

  服务器端（可选）：

```
[MSG] From 0 to 1:  Hello client1!
[MSG] From 0 to 1:  Can you hear me?
```

  接收消息的客户端：

```
connect 0.0.0.0 2587
[MSG] connected to the server 0.0.0.0.
[MSG] client 0 send: Hello client1!
[MSG] client 0 send: Can you hear me?
```

  相关的服务器的处理代码片段：

```
if (same(buf, "send")){
    int sid = *(int *)&buf[4];
    if (sid < 0 || sid >= MAX_QUEUE_LENGTH || !used[sid]){
        char *msg = "FAIL";
        sd[0] = 0; memcpy(sd + 1, msg, 4);
        send(ConnectFD[id], sd, BUFFER_SIZE, 0);
    }
    else{
        char *msg = "DONE";
        sd[0] = 0; memcpy(sd + 1, msg, 4);
        printf("[MSG] From %d to %d: %s\n", id, sid, buf + 8);
        send(ConnectFD[id], sd, BUFFER_SIZE, 0);
        buf[3]=1;
        memcpy(buf + 4, (char*)&id, sizeof(int));
        send(ConnectFD[sid], buf + 3, BUFFER_SIZE, 0);

    }
}
```

相关的客户端（发送和接收消息）处理代码片段：

```
else if(strncmp(cmd,"send",4)==0){
    int id;
    scanf("%d",&id);
    printf("[MSG] **input _exit_ to stop sending message.**\n");
    fgets(text,size,stdin);
    while(strncmp(text,"_exit_",6)!=0){
        memcpy(sendbuf,cmd,4);
        memcpy(sendbuf+4,(char*)&id,sizeof(int));
        memcpy(sendbuf+4+sizeof(int),text,strlen(text));
        sendbuf[4+sizeof(int)+strlen(text)-1]=0;
        recmessage(sendbuf);
        if(strncmp(buf+1,"FAIL",4)==0){
            printf("[MSG] Failed in send.\n");
        }
        fgets(text,size,stdin);
    }
    printf("[MSG] stopped sending message\n");
}
```

```
void* listening()
{
    while(1){
        int ok=recv(cid,buf,size,0);
        if(buf[0]==0){
            pthread_mutex_lock(&mreceived);
            pthread_cond_signal(&received);
            pthread_mutex_unlock(&mreceived);
        }else{
            if(strncmp(buf+1,"close",5)==0){
                close(cid);
                break;
            }
            else{
                int sid = *(int *)&buf[1];
                printf("[MSG] client %d send:%s\n",sid,buf+5);
            }
        }
    }
}
```

为了区分**和服务器端的内容交互**还是收到了**由服务器转发的别的 client 的消息**，我们把服务器发送信息的第一位设为标识位：如果是 0，表示是和服务器正常进行信息交换；如果是 1，表示收到了一条由别的客户端发来的消息。

## 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？

  不需要，源端口是系统随机分配的，每次调用 connect 时，客户端的端口会随机指定一个。

- 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？

  可以连接成功，但是会有一定的连接数量限制。

  accept 是将已经完成握手的 socket 从握手完成队列里面取出来，如果不调用 accept 函数，当完成握手的 socket 数量到达队列大小上限时，就不能再建立新的连接。

- 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？

  虽然服务器的端口一样，但是与之连接的客户端的端口与 IP 地址不同，通过 IP 合端口可以区分是哪个客户端发来的信息。

- 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）

```
lgl@ubuntu:~$ netstat -an | grep 2587
tcp        0      0 0.0.0.0:2587            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:57362         127.0.0.1:2587          ESTABLISHED
tcp        0      0 127.0.0.1:2587          127.0.0.1:57362         ESTABLISHED
```

```
lgl@ubuntu:~$ netstat -an | grep 2587
tcp        0      0 0.0.0.0:2587            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:2587          127.0.0.1:57362         TIME_WAIT
```

```
lgl@ubuntu:~$ netstat -an | grep 2587
tcp        0      0 0.0.0.0:2587            0.0.0.0:*               LISTEN
```

  TIME_WAIT 状态持续了一分钟左右。

- 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？

```
lgl@ubuntu:~$ netstat -an | grep 57402
tcp        0      0 127.0.0.1:2587          127.0.0.1:57402         ESTABLISHED
tcp        0      0 127.0.0.1:57402         127.0.0.1:2587          ESTABLISHED
```

  没有变化，可以让服务器端每隔一段时间向客户端发送请求，如果连着多次没有收到回应，那么可以认为连接已经无效。

## 六、 讨论、心得

本次实验相比于之前的实验，步骤减少，重点放在了编程上。因为有客户端和服务器端两个不同的代码，我就和本门课程的同学李广林合作，他写 client 我写 server，最后合起来测试。

以前上 JAVA 课时写过简单的 socket 编程，本以为这次实验能很快做完。但是各种问题层出不穷，还是陆陆续续做了快一天才做完。记录一些经典的问题和耗时的地方。

① 本实验对 Windows 很不友好。不仅 socket 库需要额外安装、初始化，windows 也不能直接支持 C 的多线程编程。我们最终打算都在 Linux 环境下写代码并交互。

② 对于 C 的底层 socket API 需要足够熟悉。Socket、connect、bind、listen、accept、send、recv，这些函数看上去思路清晰，但还是有细微差别的。比如，有些函数的返回值是 0 或 -1 表示是否成功，但是有些是返回一个 int 表示该通道建立成功后的编号。

③ 分两个人写的话测试起来很麻烦。当我们互相连接后出现了各种问题：server 收不到请求、client 收不到结果、client 退出后 server 爆炸了……而且 bug 是涉及两个人的，彼此对于对方的代码又不熟悉。最后我们是轮流修改：先 A 改，把改好的代码全给 B，B 在此基础上继续修改、调试。

④ 在多线程的环境下，**clock()函数时有问题的**。网上说，clock() 会把所有 CPU 时间都算上；我们实际测试的时候，clock() 经常返回同一个值，或者返回 0，及其不稳定。最后我们换成了 Linux 的系统调用 gettimeofday() 才获得了稳定的时间戳。

⑤ 程序要用到多线程的思想，就很容易设计出问题（平时的编程环境都是单线程的）。比如，**在 server 端要想清楚开了子线程后，主线程和他会发生什么。**我之前有个顽固的 bug：在主线程每当接收到一个新的 client 后，我用一个 id 去循环找出空的编号给它，随后我把 id 地址作为线程参数带到新线程里去；但是此时此刻，在主线程里进入了下一次 accept，id 又被改掉了；可能刚才的线程要到这一步之后才做，相当于传入的 id 发生了变化。**在 client 端则要正确处理服务器消息和客户消息**。服务器传来的消息也要分为两种，一种是回复请求，一种是转发别的 client 消息，所以必要时要用一些锁的机制来保证多线程下的正确运行。

⑥ 如上一条，为了在 client 端正确区分服务器的消息（回复请求还是转发），我和同组同学还彼此约定：服务器发送消息时**额外增加一个标识位**。包括服务器发送消息时的其他内容，其 protocol 都要事先约定好，以尽可能地减少 bug。