南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

实验 1：利用流式套接字编写聊天程序

黄尚扬

年级：2023 级

专业：计算机科学与技术

指导教师：徐敬东 张建忠

2025 年 11 月 11 日

摘要

关键字：计算机网络 TCP 协议 socket 编程

本报告是计算机网络课程的第一次实验。

在这一部分，笔者将根据所学内容进行流式套接字聊天系统的实验及分析。并且，笔者将完成具体实验要求以获取更加深刻的实验理解。

目录

一、 实验目的

(1) 设计聊天协议，并给出聊天协议的完整说明。

(2) 利用 C 或 C++ 语言，使用基本的 Socket 函数进行程序编写，不允许使用 CSocket 等封装后的类。

(3) 程序应有基本的对话界面，但可以不是图形界面。程序应有正常的退出方式。

(4) 完成的程序应能支持英文和中文聊天。

(5) 采用多线程，支持多人聊天。

(6) 编写的程序应结构清晰，具有较好的可读性。

(7) 在实验中观察是否有数据包的丢失。

二、 项目结构

实验的 git 链接为：Hsy23333/NKU_ComputerNetworks

本次实验存放在 l1 文件夹中，分别包含 server.cpp、user.cpp 以及其分别的可执行文件。正常运行中，需要先启动 server.exe，后启动一个或多个 user.exe 即可。

三、 原理说明/协议设计

(一) 原理介绍

本次实验的基石是 TCP 传输协议和流式套接字 socket。接下来笔者先对它们进行简要说明。

TCP 协议全称是 Transmission Control Protocol 传输控制协议，是一种面向连接的、可靠的字节流服务协议。[1] 在数据传输前必须先建立连接，并通过序号、确认应答、超时重传、流量控制和拥塞控制等机制，确保数据无差错、不丢失、不重复、按序到达。在通信过程中，TCP 通过“三次握手”建立连接，通过“四次挥手”断开连接。它在应用层之下、IP 层之上，为应用程序提供可靠的字节流服务，是 HTTP、FTP、SMTP 等常见网络协议的基础。

而 socket 是网络通信中应用层与传输层之间的编程接口，用于实现不同主机间的进程通信。笔者在先前接触过 Java 下的 socket 编程，而本次在 C++ 环境下的 socket 也类同。

(二) 协议设计

笔者的通信框架分为服务端和客户端两个部分，接下来先进行忽略细节的分别介绍。

1. 服务端

对于服务端而言，其会先打开 serversocket 并将其绑定在主机硬编码的 127.0.0.1:8080 上，而后开始循环监听客户端是否连接。服务器会维护一个 vector 容器和无数后台线程。每个客户端连接后会在容器中记录并开辟新线程，前者用于 broadcast，而后者用于 handle 每个客户端传来的信息。

监听时采用非阻塞方式，便于服务端随时按下 ESC 退出。

服务端在每次接收到信息或有新客户端加入时会弹出提示，并且支持中英文；此外，对于接收到的所有信息，广播时会附带时间戳。

2. 客户端

客户端会创建一个 clientsocket，并试图连接到服务器指定的端口。一旦连接成功就会随即创建一个后台的消息接收线程，接收服务端的广播；而主线程永远保证对话框可用，在输入信息后 ENTER 即可发送至服务器。输入 QUIT 即可正常退出。

客户端支持中英文输入输出。

四、 源码说明

(一) 初始化

按照我们的功能需求，初始化分为 socket 初始化和语言环境初始化两部分。

1. socket 初始化

这部分在客户端和服务端基本一致。程序会先调用 WSADATA 相关方法来初始化 socket 前置的 winsock 库；而后创建 socket。下面是 server.cpp 中的例子；而在 user.cpp 中仅需改动 socket 名称即可。

server.cpp

```
bool Initsocket(){//初始化socket
    WSADATA wsaData;
    if(WSAStartup(MAKEWORD(2,2), &wsaData)){
        wcout << L"启动失败" << endl;
        return false;
    }
    wcout<<L"启动成功" << endl;
    return true;
}

bool Createsocket(){//创建socket
    serversocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if(serversocket == INVALID_SOCKET){
        wcout << L"创建socket失败" << endl;
        Cleansocket();
        return false;
    }
    wcout << L"创建socket成功" << endl;
    return true;
}
```

在 Createsocket() 方法调用后，服务端会 bind 到指定的 127.0.0.1:8080 端口以便后续连接；而客户端的初始工作已经做完，下一部分再继续解释连接模块。

下面是服务端 bind 的相关函数:

server.cpp

```
bool Bindsocket(){//绑定socket
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(8080);
```

```
    inet_pton(AF_INET, "127.0.0.1", &serveraddr.sin_addr);//转换IP地址为可用格式后准备绑定
    if(bind(serversocket, (SOCKADDR*)&serveraddr, sizeof(serveraddr))==SOCKET_ERROR){
        wcout<<L"绑定socket失败" << endl;
        Cleansocket();
        return false;
    }
    wcout<<L"绑定socket成功" << endl;
    return true;
}
```

此外，笔者将上述几个函数定义为 bool 类型，方便在初始化阶段调用并返回相应错误码，提升了调试的可行性。

2. 语言环境初始化

由于任务要求支持中文信息传输，笔者统一使用了 UTF-16 编码，同时也用 wcout 代替 cout 进行宽字符输出避免乱码。下面是相关的设置：

server.cpp

```
// 设置控制台输出为 UTF-16
    _setmode(_fileno(stdout), _O_U16TEXT);
    // 设置控制台输入为 UTF-16
    _setmode(_fileno(stdin), _O_U16TEXT);
...
//下面这部分在Handleclient()中，防止服务端信息接收乱码
   int wideLen = MultiByteToWideChar(CP_UTF8, 0, msg.c_str(), -1, NULL, 0);
            wstring wmsg(wideLen, 0);
            MultiByteToWideChar(CP_UTF8, 0, msg.c_str(), -1, &wmsg[0],
                wideLen);//防止接收消息乱码
```

客户端也类似，在此不再赘述。

(二) 服务端/客户端连接

1. 服务端监听

初始化结束后，服务端会率先开始监听是否有客户端试图连接到指定端口。这部分用一个 while(true) 循环实现。因为服务器会长期处于这个状态，为了使得其能够随时优美地退出，笔者在此考虑关闭了 socket 的阻塞，而使用 100ms 的轮询休眠来在降低 CPU 耗能的前提下持续监听。

由于不是阻塞态，所以必然会接收到无数 INVALID_SOCKET。于是在接收部分加入 if 判断，并在顺利接收的情况下将 socket 重新改回阻塞模式后，在对当前 clientsocket 上锁的条件下记录之并为其分配一个独立线程用于接收客户端信息。

顺利接收时短暂改回阻塞的原因是子线程中包含 recv 语句，若非阻塞态就会使得其在连接一建立就马上错误接收不存在的信息，使得客户端崩溃闪退。当然，我们同样需要将关闭阻塞移动到 while 循环的内部来保证 ESC 随时有效。下面是关键代码：

server.cpp

```
void Startlisten(){//监听socket
    listen(serversocket, SOMAXCONN);
    wcout<<L"监听socket中 (ESC退出) " << endl;
    sockaddr_in clientaddr;//局部变量, 每次连接完扔掉就行
    int tmp=sizeof(clientaddr);//accept需要
    u_long mode = 1;
    ioctlsocket(serversocket, FIONBIO, &mode);//将accept改为非阻塞, 持续检测
        esc

    while(true){//循环接受客户端连接
        if (GetAsyncKeyState(VK_ESCAPE)) {
            wcout<<L"退出监听"<< endl;
            Boardcastmessage("服务器即将关闭。");

            {
                lock_guard<mutex> lock(clientsmutex);
                for (auto client : clientsockets) {
                    closesocket(client);//关闭客户端socket
                }
                clientsockets.clear(); // 清空列表
            }

            Sleep(1000);
            break;
        }


        SOCKET clientsocket = accept(serversocket, (SOCKADDR*)&clientaddr, &
            tmp);
        if(clientsocket == INVALID_SOCKET) {
            int err = WSAGetLastError();
            if(err != WSAEWOULDBLOCK) {
                wcout << L"accept出错:" << err << endl;
            }
            this_thread::sleep_for(chrono::milliseconds(100));//非阻塞空轮询
                休眠
            continue;
        }


        u_long mode = 0;
        ioctlsocket(clientsocket, FIONBIO, &mode);//将客户端socket改回阻塞模
            式
        lock_guard<mutex> lock(clientsmutex);
        clientsockets.push_back(clientsocket);
        wcout<<L"有新的客户端连接"<<endl;
        thread t(Handleclient,clientsocket); //分配新线程处理该客户端
```

```
44          t.detach();//分离线程使其后台运行
45      }
46
47  }
```

2. 客户端连接

对于客户端而言，在连接部分就无需考虑那么多。它只需要对服务器一台主机进行交互即可，于是考虑直接访问 127.0.0.1:8080，并在成功连接的情况下启动消息接收的后台线程，同时将主线程转移到准备输入文本的模块即可。

user.cpp

```
1   bool Connectserver(){//连接服务器
2       sockaddr_in serveraddr;
3       serveraddr.sin_family=AF_INET;
4       serveraddr.sin_port=htons(8080);
5       inet_pton(AF_INET,"127.0.0.1",&serveraddr.sin_addr);
6
7       if(connect(clientsocket, (SOCKADDR*)&serveraddr, sizeof(serveraddr)) ==
            SOCKET_ERROR){
8           wcout<<L"连接失败"<<endl;
9           Cleansocket();
10          return false;
11      }
12      wcout<<L"成功连接"<<endl;
13      thread t(Receivemessage);//启动接收消息线程
14      t.detach();
15      return true;
16  }
```

(三) 信息交互

在顺利建立连接后，进入到重点的信息交互模块。对于服务端而言，其通过与所连接客户端一一对应的后台线程来接收信息，并遍历 clientsockets 的容器来进行广播或转发。而对于客户端而言，其通过接收服务端信息的后台线程将服务器的广播接收并打印在屏幕上，并维持主线程处于能够发送消息的状态。

下面是服务端的广播和 handle 方法。笔者通过锁保证了其是线程安全的，并在发送消息前获取了时间戳来优化用户体验。当然服务端本身也可以有时间戳显示，只需要简单地类似输出即可。这里目的已经达到了就没有做这个额外实现。

server.cpp

```
1
2   void Boardcastmessage(const string& msg){//广播消息给所有客户端
3       lock_guard<mutex> lock(clientsmutex);
4
5
6       auto now = chrono::system_clock::now();
```

```
7       time_t t = chrono::system_clock::to_time_t(now);
8       stringstream timeStream;
9       timeStream << "[" << put_time(localtime(&t), "%Y-%m-%d %H:%M:%S") << "] "
            ;
10      string fullmsg = timeStream.str()+msg;
11      //获取时间戳并转换
12
13
14      for(auto clientsocket : clientsockets){
15          send(clientsocket, fullmsg.c_str(), fullmsg.size(), 0);
16      }
17  }
18
19
20  void Handleclient(SOCKET clientsocket){//分配线程处理每个客户端
21      char buffer[1024];
22      while(true){
23          int ret = recv(clientsocket, buffer, sizeof(buffer)-1, 0);
24          if(ret > 0){
25              buffer[ret] = '\0';
26              string msg(buffer);
27
28
29              int wideLen = MultiByteToWideChar(CP_UTF8, 0, msg.c_str(), -1,
                    NULL, 0);
30              wstring wmsg(wideLen, 0);
31              MultiByteToWideChar(CP_UTF8, 0, msg.c_str(), -1, &wmsg[0],
                    wideLen);//防止接收消息乱码
32
33              auto now = chrono::system_clock::now();
34              time_t t = chrono::system_clock::to_time_t(now);
35              stringstream timeStream;
36              timeStream << "[" << put_time(localtime(&t), "%Y-%m-%d %H:%M:%S")
                     << "] ";
37              //同样是时间戳
38
39              wcout << L"收到消息: " << wmsg << endl;
40              Boardcastmessage(msg);
41          } else if(ret == 0){
42              wcout << L"客户端断开连接" << endl;
43              break;
44          } else {//错误处理（本来有区别，但不知为啥断开之后永远不走上面那条，
                遂改）
45              wcout << L"客户端断开连接" << endl;
46              break;
47          }
48      }
49      closesocket(clientsocket);
```

```
    lock_guard<mutex> lock(clientsmutex);
    clientsockets.erase(std::remove(clientsockets.begin(), clientsockets.end
        (), clientsocket), clientsockets.end());
    //收尾
}
```

下面则是客户端的 send/receive 方法。它支持 1024 字节以内的数据，并在输入 QUIT 的时候自动退出。

这里没有像前文服务器连接处一样做非阻塞，所以在服务器断开后并不会第一时间退出，而要再次 ENTER。

为了支持中文格式的转换，这里额外写了一个 wstring 的转换函数，不是重点就不赘述了。

user.cpp

```
void Sendmessage(){//发送信息给服务器
    wcout<<L"输入消息发送给服务器，输入QUIT退出"<<endl;
    wstring wmsg;
    while(server_running){
        getline(wcin, wmsg);
        if(wmsg == L"QUIT") break; //QUIT退出
        string msg = wstring_to_utf8(wmsg);
        send(clientsocket, msg.c_str(), msg.size(), 0);
    }
}

void Receivemessage(){//接收服务器消息
    char buffer[1024];
    while(true){
        int ret = recv(clientsocket, buffer, sizeof(buffer)-1, 0);
        if(ret > 0){
            buffer[ret] = '\0';
            wstring wmsg = utf8_to_wstring(buffer);
            wcout << wmsg << endl;
        } else if(ret == 0){
            server_running = false;
            wcout<<L"服务器断开连接"<<endl;
            break;
        } else {
            server_running = false;
            wcout<<L"接收消息失败"<<endl;
            break;
        }
    }
}
```

（四） 清理机制

在退出程序前，调用下面方法释放资源即可。客户端与服务端完全一致。

server.cpp

```
void Cleansocket(){//释放socket
    WSACleanup();
    closesocket(serversocket);
    wcout<<L"关闭成功" << endl;
    return;
}
```

五、 丢包测试

由于 TCP 是面向连接的可靠传输协议，其特性包括：自动重传丢失的数据包（由内核处理）；确保数据按顺序到达；发送端和接收端看到的是“无差错、无丢包”的字节流。所以在 TCP 层面，并不会看到“丢包”现象，因为它被 TCP 自动重传并掩盖了。

于是它变为了“消息是否被按照预期接收”的检测。

我们可以打开多个客户端窗口，并且发送一系列中英文文本，观察服务端以及这些客户端上的信息是否完整且有序。下面是实验结果：

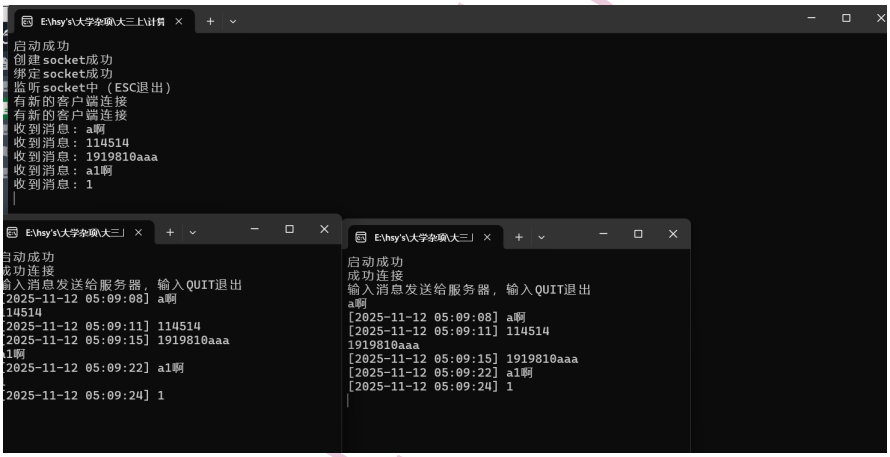

图 1: 丢包测试

可以看到并没有数据乱序或丢失的情况，故可以认为不存在显式的丢包现象。

六、 总结

本次实验中，笔者采用了流式套接字进行了 TCP 协议下的编程，并成功建立了服务端和用户端两个平台，使之能够进行可靠的数据交互。

实验所有要求的任务全部顺利完成。实现过程大体顺利，其中中文字符转换以及 socket 阻塞/非阻塞状态的切换稍稍费时。

本次实验加深了笔者对 socket 方法的理解，也使我对计算机网络课程有了更深刻的认识。

参考文献

[1] Java 码农. 有关 tcp 协议，这是我看过讲的最清楚的一篇文章了！. **知乎，** *https://zhuanlan.zhihu.com/p/1971223189712515523*, 2025.