

实验一 在一台主机上通过回环地址实现客户-服务器间的 **socket** 通信

一、 实验要求

分别建立服务器和客户机，实现客户机和服务器的连接和通信，从客户机向服务器传输你的个人信息和任意文件信息，在服务器上显示你发送的信息和文件信息。

二、 实验内容

客户-服务器之间的通信基于 TCP 协议，先在客户与服务器之间建立 TCP 连接，然后进行对话。下面从服务器和客户端分别阐述。

服务器（`network_lab1/src/ThreadedServer.java`）：

服务器采用多线程，主函数用于监听客户端的连接请求，每当一个客户端请求建立连接时，便创建一个 TCP 套接字并为其建立一个线程，在该线程中处理信息的收发。

```
Runnable r = new ThreadedHandler(incoming, client_map);  
var t = new Thread(r);  
t.start();
```

子线程的建立

在每个客户端发送请求后，服务器会给它们自动分配一个 ID。服务器中的哈希表 `client_map`, key 值为 `SocketAddress`, value 值为对应的 ID，用来记录每一个客户端请求对应的 ID。客户端断开连接时 `client_map` 中对应记录也删除。`client_map` 在每个线程间共享，用于设备 ID 的查找以及统计在线设备数。

```

Socket incoming = s.accept();

var add :SocketAddress = incoming.getRemoteSocketAddress();
var ip_add :InetAddress = incoming.getInetAddress();
var port_add :int = incoming.getPort();

if(client_map.containsKey(add)) continue;
client_map.put(add, i);

```

处理客户端请求并更新 client_map

服务器的输入数据流采用 Scanner 处理，输出数据流采用 PrintWrite 处理（实验一和实验二都采用这种方法，后面不在赘述）。

```

try (InputStream inStream = incoming.getInputStream();
    OutputStream outStream = incoming.getOutputStream();
    var in = new Scanner(inStream, StandardCharsets.UTF_8);
    var out = new PrintWriter(
        new OutputStreamWriter(outStream, StandardCharsets.UTF_8),
        autoFlush: true)) {

```

输入输出流的处理

每个客户端连接后，服务端都为其建立一个线程用于处理收发，服务器在接收到信息时会先将其显示在屏幕上（客户端向服务器发送 BYE 后视为断开连接）。而对于服务器向客户端的响应信息，我设计了两种模式，一种为服务器将客户端发来的信息回射给客户端（每次收发一条消息）；另一种模式为等客户端发送完全后，在服务端手动输入信息并发送（每次可收发多条数据，直到输入空行视为输入结束）。

（第一种模式对应程序 61-79 行，第二种对应 81-106 行，使用时注释其一）

```

while (true) {
    String line = in.nextLine();
    System.out.println("客户端" + clientID + ": " + line);

    if (line.trim().equalsIgnoreCase( anotherString: "HELLO")) {
        out.println("Hello! Enter BYE to exit");
        continue;
    }

    if (line.trim().equalsIgnoreCase( anotherString: "BYE")) {
        out.println("BYE");
        client_map0.remove(incoming.getRemoteSocketAddress());
        System.out.println("客户端" + clientID + "断开连接!");
        System.out.println("当前在线设备数: " + client_map0.size());
        break;
    }

    out.println("Echo: " + line);
}

```

服务器向客户端回显信息

```

while (true) {
    String line;
    line = in.nextLine();
    boolean flag = false;
    while(!line.trim().equals("")){
        System.out.println("客户端" + clientID + ": " + line);
        if (line.trim().equalsIgnoreCase( anotherString: "BYE")) {
            flag = true;
            out.println("BYE");
            client_map0.remove(incoming.getRemoteSocketAddress());
            System.out.println("客户端" + clientID + "断开连接!");
            System.out.println("当前在线设备数: " + client_map0.size());
            break;
        }
        line = in.nextLine();
    }
    if(flag) break;
    System.out.println("向客户端"+clientID+"发送消息:");
    Scanner sc = new Scanner(System.in);
    String str;
    do {
        str = sc.nextLine();
        out.println(str);
    } while (!str.trim().equals(""));
    System.out.println("消息已发送, 等待客户端回应...");
}

```

服务器手动输入应答信息

客户端（network_lab1/src/Client.java）：

客户端在完成与服务器的连接后进入 `HandlerClient` 函数开始与服务器进行通信,收发消息的两种模式与服务端相同,再次不作赘述。

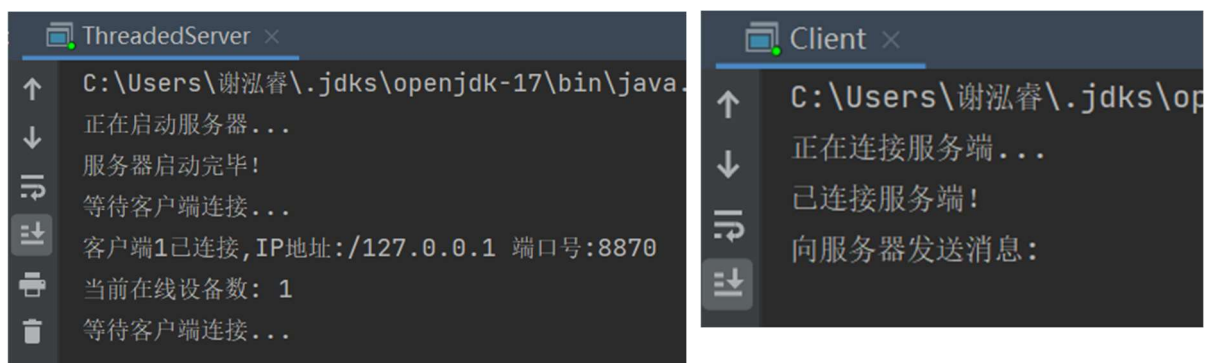
```
try(Socket socket = new Socket( host: "127.0.0.1", port: 8189)) {  
    System.out.println("已连接服务端!");  
    TCPClient client = new TCPClient(socket);  
    client.HandlerClient();  
    System.out.println("客户端已关闭!");  
}
```

客户端尝试与服务器连接

三、 实验结果与分析

将客户端代码中的目标 IP 设为 `127.0.0.1`, 使用回环地址实现通信, 结果如下:

1. 客户端 1 与服务器连接:



2. 客户端 1 与服务器通信:



3. 客户端 2 上线并通信:

```

消息已发送，等待客户端回应...
客户端2已连接,IP地址:/127.0.0.1 端口号:9506
当前在线设备数: 2
等待客户端连接...
客户端1: 你好
向客户端1发送消息:
你好!

消息已发送，等待客户端回应...
客户端2: 你好
向客户端2发送消息:
你好!

消息已发送，等待客户端回应...

```

4. 两客户端依次断开:

```

消息已发送，等待客户端回应...
客户端1: BYE
客户端1断开连接!
当前在线设备数: 1
客户端2: BYE
客户端2断开连接!
当前在线设备数: 0
|

```

```

向服务器发送消息:
BYE
消息已发送，等待服务器回应...
服务器: BYE
已与服务器断开连接!
客户端已关闭!

Process finished with exit code 0

```

5. 服务端回显模式示例(服务端和客户端的收发模式应相同):

```

正在启动服务器...
服务器启动完毕!
等待客户端连接...
客户端1已连接,IP地址:/127.0.0.1 端口号:10121
当前在线设备数: 1
等待客户端连接...
客户端1: Hello
客户端1: 学号: 201981096
客户端1: 姓名: 谢泓睿
客户端1: BYE
客户端1断开连接!
当前在线设备数: 0

```

```

正在连接服务端...
已连接服务端!
向服务器发送消息: Hello
服务器: 你好,输入BYE以退出
向服务器发送消息: 学号: 201981096
服务器: Echo: 学号: 201981096
向服务器发送消息: 姓名: 谢泓睿
服务器: Echo: 姓名: 谢泓睿
向服务器发送消息: BYE
服务器: BYE
已与服务器断开连接!
客户端已关闭!

Process finished with exit code 0

```

四、 讨论与感想

实验 1 基于 TCP 协议，使用回环地址完成了客户端-服务器之间的通信，实现了基本需求。

实验 1 主要存在 3 点不足：1. 通信时的收发过程存在阻塞，发出信息后必须等待对方回信才可再发，做不到实时接收与实时发送；2. 只实现了服务器与客户端之间的通信，无法做到在线客户之间的通信；3. 若多个客户端都在等待服务器应答，则服务端的各线程对标准输入缓冲区的使用会出现竞争现象。以上不足在实验 2 中得到解决。

总而言之，实验 1 的完成让我对 TCP 的 socket 编程有了初步了解，熟悉了 Java 网络编程中一些基本语句的使用，加深了对 TCP 协议的理解，也为实验 2 以及今后的发展打下基础。

附：参考文献：

(美)Cay S. Horstmann 著 Java 核心技术卷 2-高级特性 机械工业出版社 2019.12

实验二 实现两台主机在局域网内的客户-服务器 socket 通信

一、实验要求

实现 TCP 通信，实现能够在局域网中进行两台主机之间的 socket 通信。

二、实验内容

经实验，将实验 1 客户端代码中的目的 IP 地址改为服务器的 IP 地址即可满足实验 2 的基本要求。

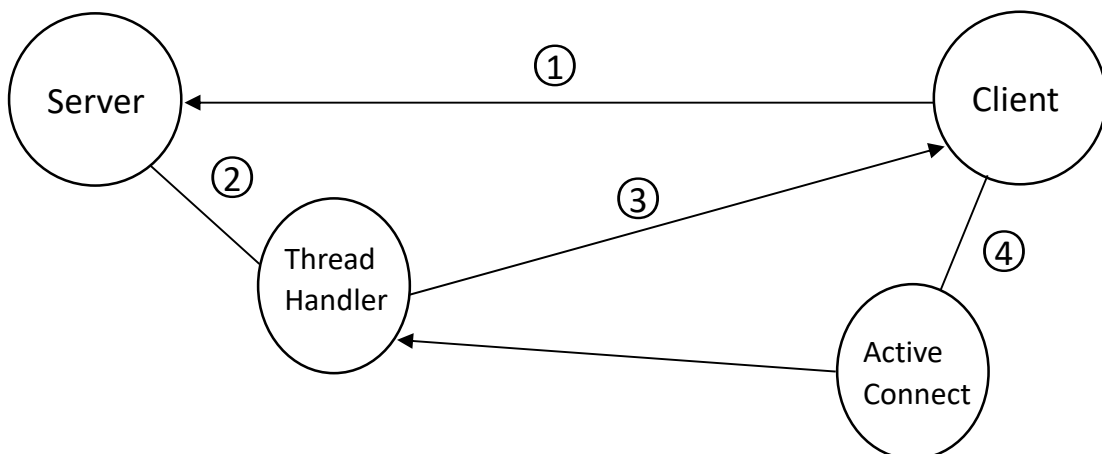
下面主要针对实验 1 中的不足进行改进，从而实现简易聊天室。

实验 1 的 3 点不足主要源于未进行合理的多线程设计，于是我对服务器和客户端的连接建立、对话建立、对话过程、对话中止和连接中止过程中的线程分配进行了设计，具体如下（代码详见 network_lab2/src）：

1. 连接建立

(1) 客户端向服务器端请求建立连接

(2) 服务器建立线程 ThreadHandler 用来为客户提供服务。



(3) ThreadHandler 向客户端发送第一条序号为 0 的数据（序号不会显示在客户端屏幕上）。

(4) 客户端收到序号 0 后建立子线程 ActiveConnect，该线程只接收客户输入并发送，接收服务端信息的工作仍由主线程完成（如上图 3）

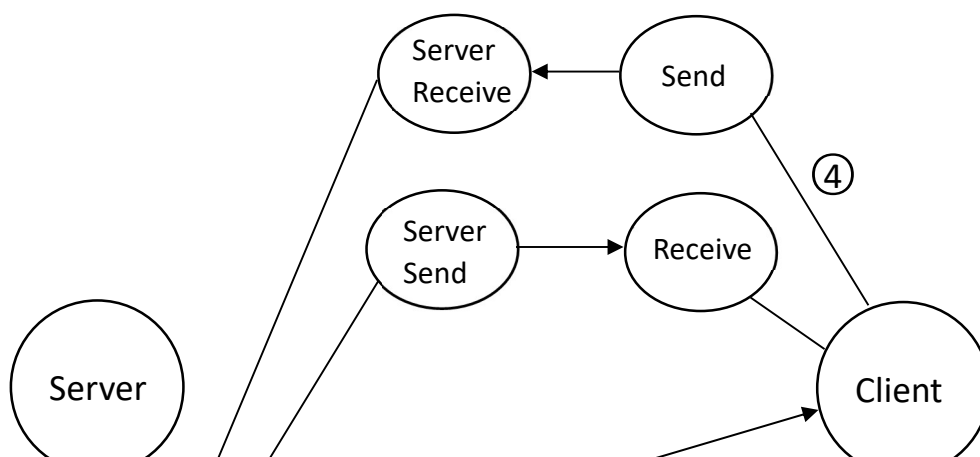
ThreadHandler 第一次向客户端发送的信息用于提醒客户端建立新线程，同时将当前在线设备提供给客户。ThreadHandler 后续可响应 ActiveConnect 线程发出的请求，包括更新在线设备、同意进行对话、断开连接等。

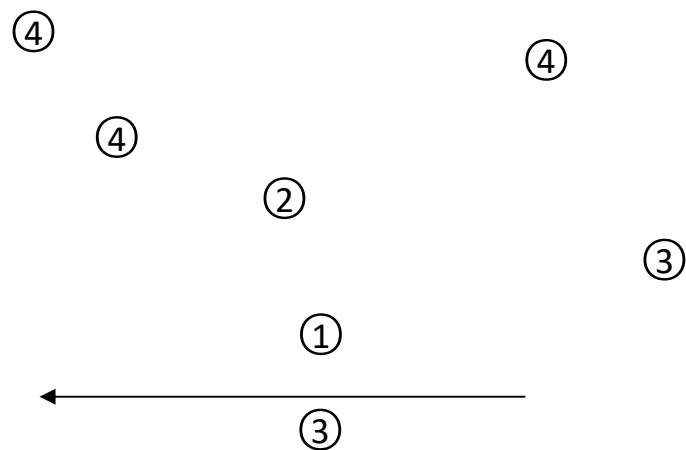
ThreadHandler 类的成员变量如下，其余详见代码：

```
private final Socket incoming;
private final HashMap<SocketAddress, Integer> client_map0; // 地址 to ID
private final HashMap<Integer, Socket> client_map1; // ID to 套接字
private final HashMap<Integer, ThreadedHandler> thread_map; // ID to 线程
private final HashSet<Integer> busy_map; // 忙碌客户
private final int ClientID;
private final AtomicBoolean exit; // 原子量，协调send和receive
private final HashMap<Integer, Scanner> client_in; // ID to 输入流
private final HashMap<Integer, PrintWriter> client_out; // ID to 输出流
private final Scanner in; // ThreadedHandler所服务的客户的输入流
private final PrintWriter out; // ThreadedHandler所服务的客户的输出流
private final AtomicBoolean CanTalk; // 原子量，协调请求对话客户和被请求对话客户
```

2. 对话建立与通信

2.1 客户与服务器建立对话并通信



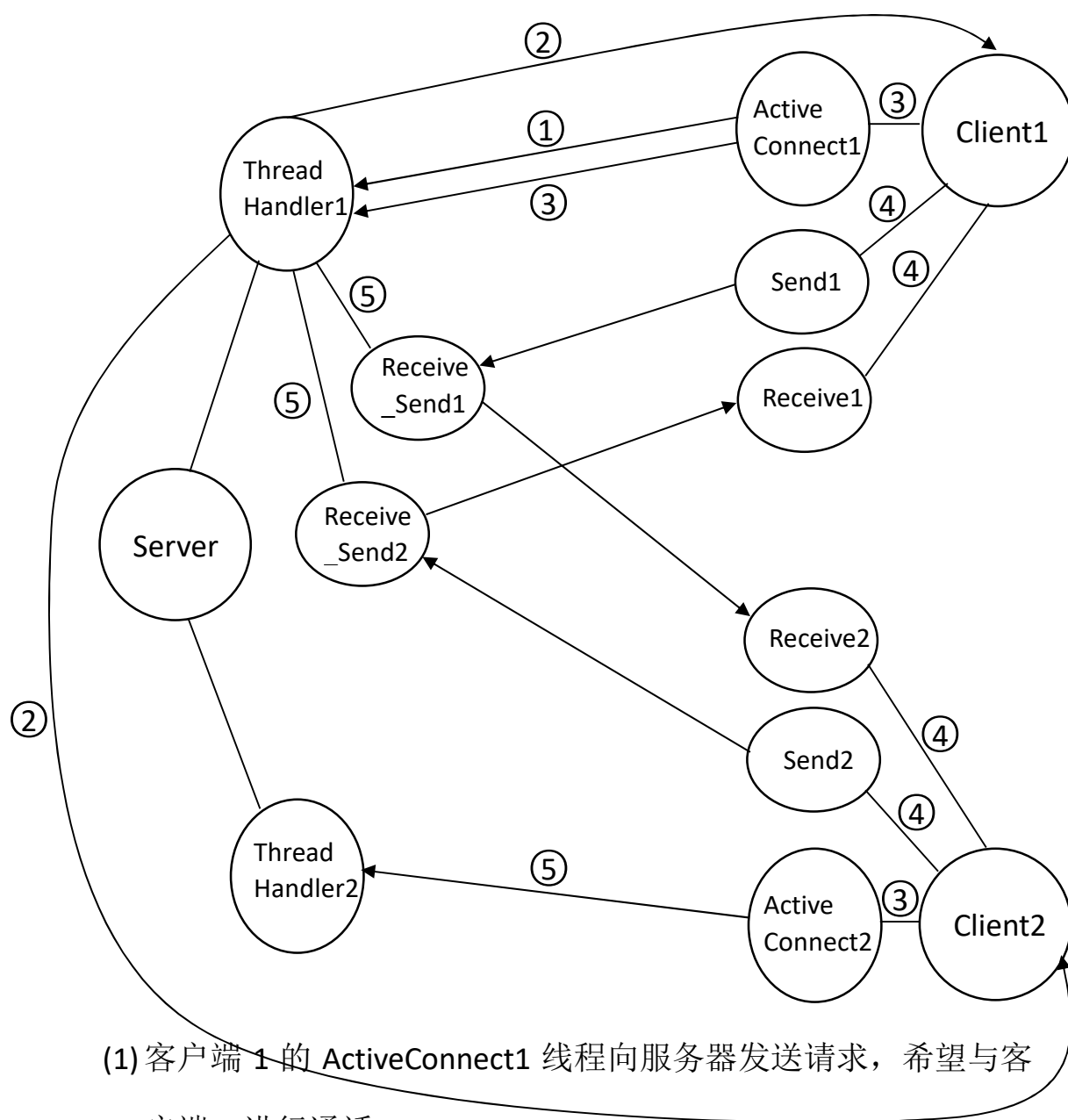


- (1) **ActiveConnect** 线程向服务器发送请求，希望与服务器通信。
- (2) **ThreadHandler** 线程收到请求后进行检查，确保客户进行请求通信的 ID 号合法，并且检查 ID 是否在 **busy_map** 中（服务器 ID 为 0，客户端 ID 为正整数），从而判断客户的请求是否可行。若请求可执行则向客户返回成功信号，并在 **busy_map** 中添加两端 ID（从检查到添加的这段程序是原子的），否则返回失败信号，并发送失败原因在客户端屏幕上显示。
- (3) 客户端若收到成功信号则要求客户输入 Y 并回发给服务器以确认开始（回发 Y 主要是客户端与客户端通信时，被请求对话的客户端用来防止在服务器端其对于服务器的输入流出现竞争情况。在这里也回发只是为了减小代码量，同时保持客户端行为一致。），同时设置 **flag** 原子布尔量，**ActiveConnect** 线程接收到客户输入的 Y 且根据 **flag** 进行判断确认可以开始对话后跳出循环，**ActiveConnect** 线程结束并关闭。
- (4) **ThreadHandler** 线程收到 Y 后创建两个线程 **Server_Receive** 和 **Server_Send** 负责收发，客户端 **ActiveConnect** 线程结束后创立

两个线程 **Receive** 和 **Send** 负责收发。

(5) 服务器和客户端通过(4)中创建的四个线程进行无阻塞通信。

2.2 客户端与客户端建立对话并通信



(1) 客户端 1 的 **ActiveConnect1** 线程向服务器发送请求，希望与客户端 2 进行通话。

(2) **ThreadHandler1** 线程接收到请求后检查是否可行，具体与 2.1 中(2)类似。若请求可行，**ThreadHandler1** 向客户端 1 发送成功信号，向客户端 2 发送被请求对话信号（向客户端 2 的输出流

从 `client_out` 表中根据 ID 获取)。若请求不可行, 向客户端 1 发送错误原因。

(3) 客户端 1 接到成功信号后等待用户输入 Y 并发送, 同时设置 `flag1` 原子布尔量, `ActiveConnect1` 线程接收到用户输入 Y 且根据 `flag1` 进行判断确认可以开始对话后跳出循环, `ActiveConnect1` 线程结束。客户端 2 接收到被请求对话信号后等待用户输入 Y, 同时设置 `flag2` 原子布尔量, `ActiveConnect2` 线程接收到用户输入 Y 且根据 `flag2` 进行判断确认可以开始对话后跳出循环, `ActiveConnect2` 线程结束。

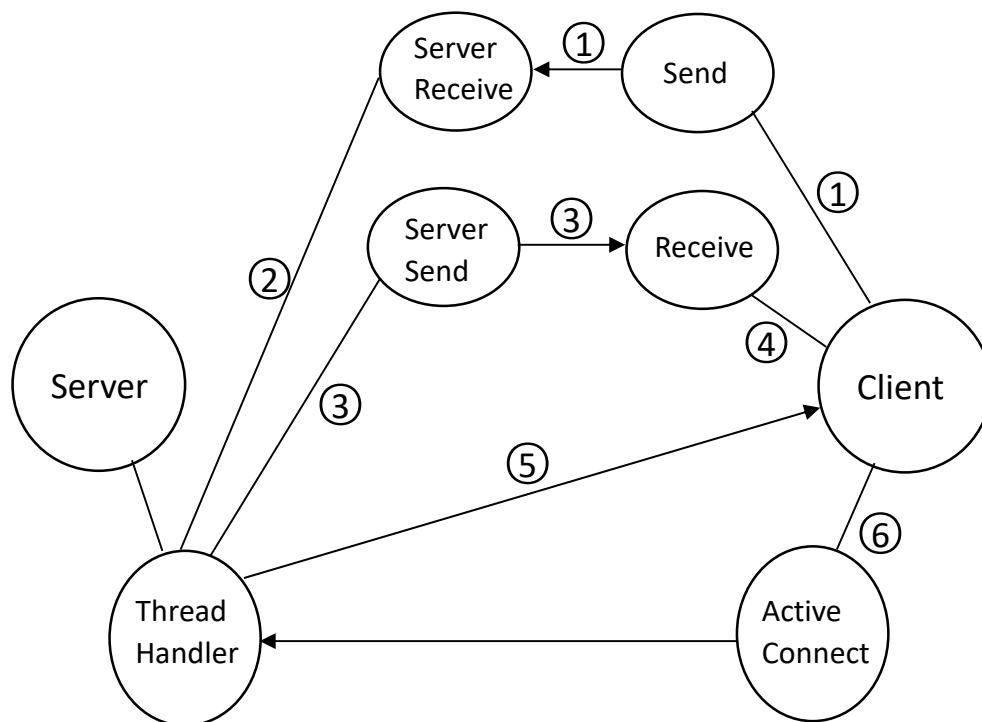
(4) 客户端 1 和客户端 2 在等待各自的 `ActiveConnect` 线程结束后创建各自的 `Receive` 和 `Send` 线程(必须先等待 `ActiveConnect` 线程结束, 这是为了防止输入缓冲区的线程竞争)。

(5) `ThreadHandler2` 接收到 Y 信号并检查发现自己已在 `busy_map` 中, 则设置原子布尔量 `CanTalk2` 同时开始阻塞, 让出对来自客户端 2 的输入流占用。`ThreadHandler1` 接收到 Y 信号并等待 `CanTalk2` 信号(`ThreadHandler1` 通过 `thread_map` 访问 `ThreadHandler2` 线程中的 `CanTalk`), 然后(此时 `ThreadHandler1` 才能获知 `ThreadHandler2` 线程不会占用输入流, 并且客户端 1 与客户端 2 已做好准备) `ThreadHandler1` 创建两个 `Receive_Send` 线程, `Receive_Send1` 接收来自客户端 1 的数据并发送给客户端 2, `Receive_Send2` 接收来自客户端 2 的数据并发送给客户端 1。

(6) 此时客户端 1 和客户端 2 可以开始无阻塞对话。

3. 对话中止

3.1 客户端与服务器对话中止



(1) 客户输入 BYE 并被 Send 线程接收到, 向服务器发出中止信号(服务器也可发送 BYE 来发出中止信号, 整体过程与下述类似), 同时设置客户端的原子布尔量 `exit`, 然后 Send 线程结束。

(2) ServerReceive 线程收到终止信号, 设置 ThreadHandler 中的原子布尔量 `exit`, 要求服务器管理员输入 Y, 然后 ServerReceive 线程结束。

(3) ServerSend 线程收到用户的 Y 并检查 `exit` 值, 若判断可以中止便向客户端发出中止信号 BYE, 然后 ServerSend 线程结束。

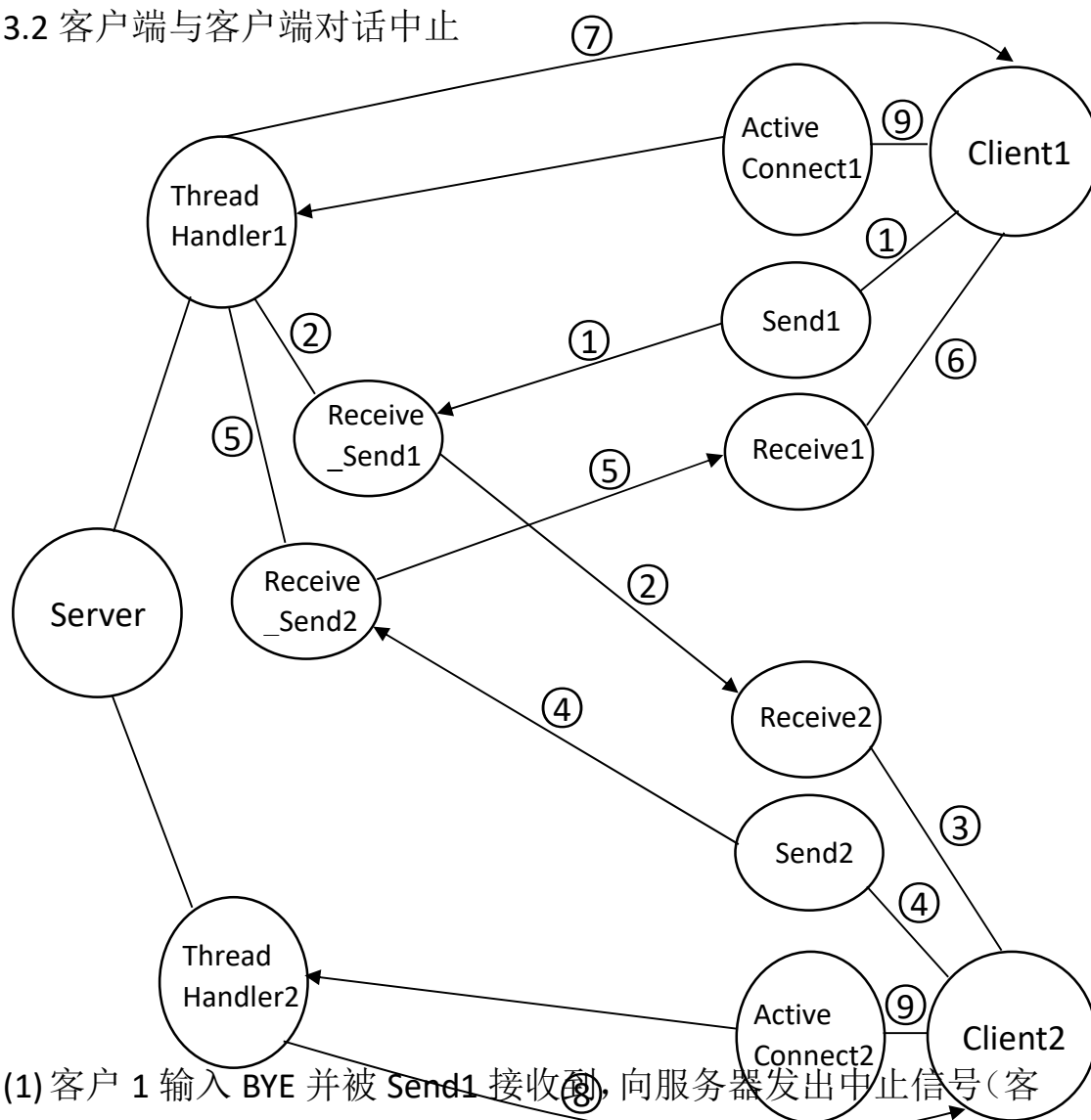
(4) 客户端 Receive 线程收到终止信号, 检查客户端的 `exit` 值, 发现 `exit` 值已被改变, 于是 Receive 线程直接结束。

(5) ThreadHandler 等到 ServerReceive 和 ServerSend 线程结束后在

busy_map 中清除两端 ID，重设各原子布尔量，并重新向客户端发送序号为 0 的信号，同时发送在线设备列表。

(6) 客户端接收到序号为 0 的信号后显示在线设备列表，重新建立 ActiveConnect 线程，往后过程又回到 2 中。

3.2 客户端与客户端对话中止



(1) 客户 1 输入 BYE 并被 Send1 接收到，向服务器发出中止信号（客户端 2 请求中止与下同理），同时设置客户端 1 的原子布尔量 exit1，然后 Send1 线程结束。

(2) Receive_Send1 线程接收到 BYE 信号，设置 ThreadHandler1 中的原子布尔量 exit，并向客户端 2 发出 BYE 消息，然后

Receive_Send1 线程结束。

(3) Receive2 线程接收到 BYE 消息，要求客户 2 输入 Y 以继续，同时设置客户端 2 的原子布尔量 exit2，然后 Receive2 线程结束。

(4) Send2 线程接收到用户输入的 Y，并检查 exit2 发现 Receive2 线程已结束，则向服务器发出 BYE 信号，然后 Send2 线程直接结束。

(5) Receive_Send2 线程接收到 BYE 信号，并检查 exit 发现 Receive_Send1 已结束，则向客户端 1 发出 BYE 信号，然后 Receive_Send2 线程直接结束。

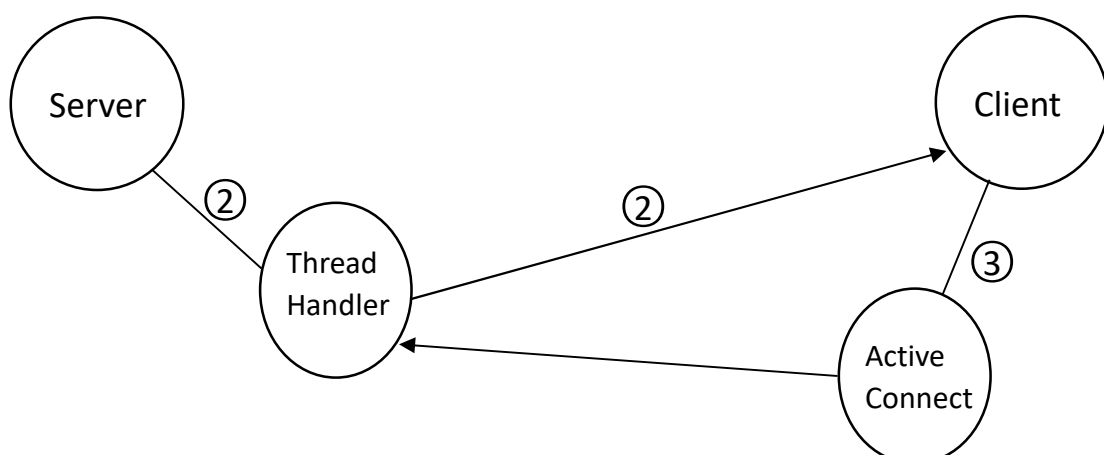
(6) Receive1 线程接收到 BYE 消息，并检查 exit1 发现 Send1 线程已结束，则 Receive1 线程直接结束。

(7) ThreadHandler1 等待 Receive_Send1 和 Receive_Send2 线程都结束后，重新设置各原子布尔量，两端 ID 从 busy_map 中移除，然后向客户端 1 重新发送序号为 0 的信号。

(8) ThreadHandler2 发现自己的 ID 在 busy_map 中消失后，重新设置各原子布尔量，然后向客户端 2 重新发送序号为 0 的信号。

(9) 客户端 1 和客户端 2 接收到序号为 0 的信号后显示在线设备列表，重新建立各自的 ActiveConnect 线程，往后过程又回到 2 中。

4. 连接中止



①

- (1) 客户端的 **ActiveConnect** 线程向服务器发出断开连接信号。并设置原子布尔量 **flag**。
- (2) **ThreadHandler** 线程收到信号，向客户端发送 **BYE** 信号，并从两个客户表（**client_map0** 和 **client_map1**）中删去该客户，随后 **ThreadHandler** 线程结束。
- (3) 客户端收到 **BYE** 信号后，提示客户已断开连接，提示用户输入 **Y** 确认，**ActiveConnect** 线程接收到用户输入 **Y**，并检查 **flag**，判断可以退出后 **ActiveConnect** 线程结束，随后客户端程序结束。

三、实验结果与分析

将客户端代码中的目的 IP 改为服务器 IP 进行测试。

1. 客户端 1 与客户端 2 上线，客户端 1 开始与客户端 2 通信：


```
ThreadedServer x
C:\Users\谢泓睿\.jdk\openjdk-17\bin\java.exe
正在启动服务器...
服务器启动完毕!
等待客户端连接...
客户端1已连接,IP地址:/10.9.110.84 端口号:57633
当前在线设备数: 1
等待客户端连接...
客户端2已连接,IP地址:/10.9.110.84 端口号:57646
当前在线设备数: 2
等待客户端连接...
客户端1开启了与客户端2的对话!
```

服务器

```
Client x
C:\Users\kkkw1\.jdk\openjdk-17.0.1\bin\java.exe "-javaagent:
正在连接服务端...
已连接服务端!
服务器: 请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):
[0] 服务器
-2
服务器: 请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):
[0] 服务器
[2] /10.9.110.84:57646
2
客户端2已准备好与您对话(请输入Y以开始)
Y
可以开始对话!
你好!
我是jack
客户端2: 你好,我是mike
how are you, mike?
客户端2: not bad!
```

客户端 1

```
Client x
C:\Users\kkkw1\.jdk\openjdk-17.0.1\bin\java.exe "-javaagent:
正在连接服务端...
已连接服务端!
服务器: 请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):
[0] 服务器
[1] /10.9.110.84:57633
客户端1准备与您对话(请输入Y以继续)
Y
可以开始对话!
客户端1: 你好!
客户端1: 我是jack
你好,我是mike
客户端1: how are you, mike?
not bad!
```

客户端 2

2. 客户端 3 上线并开始与服务器通信：

```
客户端3已连接,IP地址:/10.9.110.84 端口号:57691
当前在线设备数: 3
等待客户端连接...
客户端3开启了与您的对话!
可以开始对话!
客户端3: 你好, 服务器管理员!
客户端3: 现在设备情况如何?
你好, 设备一切正常!
```

服务器

```
Client x
C:\Users\kkkw1\jdk\openjdk-17.0.1\bin\java.exe "-javaagent
正在连接服务端...
已连接服务端!
服务器: 请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):
[0] 服务器
[2] /10.9.110.84:57646
[1] /10.9.110.84:57633
0
服务端已准备好与您对话(请输入Y以开始)
Y
可以开始对话!
你好, 服务器管理员!
现在设备情况如何?
服务器: 你好, 设备一切正常!
啊, 那就好
```

客户端 3

3. 客户端 4 上线请求与客户端 2 通信：

```
客户端4已连接,IP地址:/10.9.110.84 端口号:57703
当前在线设备数: 4
等待客户端连接...
客户端3: 啊, 那就好
客户端1结束当前对话!
```

服务器

```
正在连接服务端...
已连接服务端!
服务器: 请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):
[0] 服务器
[3] /10.9.110.84:57691
[2] /10.9.110.84:57646
[1] /10.9.110.84:57633
0
对方正忙, 请稍后!
visdgi
输入格式错误, 请输入对应ID!
2
对方正忙, 请稍后!
3
对方正忙, 请稍后!
```

客户端 4

4. 客户端 1 与客户端 2 中止对话，客户端 2 请求与客户端 4 通信：

```
等待客户端连接...
客户端3：啊，那就好
客户端1结束当前对话！
客户端2开启了与客户端4的对话！
```

服务器

```
BYE
中止对话请求中，请稍后...
已结束当前对话！
服务器：请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):
[0] 服务器
[4] /10.9.110.84:57703
[3] /10.9.110.84:57691
[2] /10.9.110.84:57646
```

客户端 1

```
↑ 对方结束当前对话！（输入Y以继续）...
↓ Y
↺ 服务器：请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):
⏴ [0] 服务器
⏵ [4] /10.9.110.84:57703
⏴ [3] /10.9.110.84:57691
⏵ [1] /10.9.110.84:57633
4
客户端4已准备好与您对话(请输入Y以开始)
Y
可以开始对话！
你好，4号！
这是2号
客户端4：你好你好
```

客户端 2

```
🖨 客户端2准备与您对话(请输入Y以继续)
🗑 Y
可以开始对话！
客户端2：你好，4号！
客户端2：这是2号
你好你好
```

客户端 4

5. 客户端 1 断连：

```
客户端1断开连接！
当前在线设备数： 3
```

服务器

```
服务器：请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):  
[0] 服务器  
[4] /10.9.110.84:57703  
[3] /10.9.110.84:57691  
[2] /10.9.110.84:57646  
-1  
已与服务器断开连接! (请输入Y以结束)  
Y  
客户端已关闭!  
  
Process finished with exit code 0
```

客户端 1

6. 客户端 3 结束与服务器通话:

```
客户端3结束当前对话! (输入Y以继续) ...  
Y  
客户端3结束当前对话!
```

服务器

```
BYE  
中止对话请求中, 请稍后 ...  
已结束当前对话!  
服务器：请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):  
[0] 服务器  
[4] /10.9.110.84:57703  
[2] /10.9.110.84:57646
```

客户端 3

7. 客户端 2 结束与客户端 4 通话:

```
BYE  
中止对话请求中, 请稍后 ...  
已结束当前对话!  
服务器：请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):  
[0] 服务器  
[4] /10.9.110.84:57703  
[3] /10.9.110.84:57691
```

客户端 2

```
对方结束当前对话! (输入Y以继续) ...  
Y  
服务器：请在在线设备中选择一个进行通话(输入-1以结束连接,输入-2以刷新):  
[0] 服务器  
[3] /10.9.110.84:57691  
[2] /10.9.110.84:57646
```

客户端 4

8.客户端依次断连:

```
客户端3结束当前对话！  
客户端2结束当前对话！  
客户端3断开连接！  
当前在线设备数：2  
客户端4断开连接！  
当前在线设备数：1  
客户端2断开连接！  
当前在线设备数：0
```

服务器

四、讨论、感想

实验 2 针对实验 1 中的不足进行了改进，实现了无阻塞收发、客户-客户与客户-服务器通信。在多线程的设计耗费了大量时间，经过数次修改与重构，最终得以实现。

由于时间关系，未设计 UI 界面，这也是后续可以完善的地方。

通过实验 2，我加深了对 socket 通信、TCP 协议以及多线程的理解，并能逐步将理论用于时间，相信对今后的发展时大有裨益的。

实验三 本机/两台主机之间视频传输

一、实验要求

由客户端采集摄像头图像后经 **Socket** 传输到服务器端并在服务器端显示出来。

二、实验内容

视频传输基于 **UDP** 协议，下面从客户端的采集发送到服务器端的接收显示依次分析。

1. 客户端摄像头图像采集

摄像头图像的采集基于 Webcam 外部依赖（运行程序前需先确保 network_lab3/lib 中的 4 个 .jar 文件被添加为项目的依赖）。图像的采集有两种模式：1. 使用 VGA 模式采集图像（对应 Client.java 的 33 行）。VGA 是所有显卡都接受的基准分辨率，最大支持 640*480 分辨率。2. 使用 HD720 模式采集图像（对应 Client.java 的 34-41 行），支持 1280*720 分辨率。

```
// 初始化摄像头
Webcam cam = Webcam.getDefault();

cam.setViewSize(WebcamResolution.VGA.getSize());
Dimension[] nonStandardResolutions = new Dimension[] {
    WebcamResolution.PAL.getSize(),
    WebcamResolution.HD720.getSize(),
    new Dimension(1000, 500),
    new Dimension(800, 400),
};
cam.setCustomViewSizes(nonStandardResolutions);
cam.setViewSize(WebcamResolution.HD720.getSize());

cam.open();
```

摄像头的打开（默认以 VGA 模式）

2. 客户端图像数据处理

采集到图像后首先对图像的大小进行压缩，客户可自己设置压缩率（默认为 1，即不压缩）。压缩会缩小图片大小，增大传输速度。

```
BufferedImage image0 = cam.getImage();
int width = image0.getWidth();
int height = image0.getHeight();
width = (int) (width * SCALE);
height = (int) (height * SCALE);
image = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
image.getGraphics().drawImage(image0, x: 0, y: 0, width, height, observer: null);
```

图像捕获与压缩

压缩大小后，使用 JPG 标准对图像进行进一步压缩，将图片处理为字节数组，便于发送。

```
// 使用jpg格式将图像转换为字节流
ByteArrayOutputStream imageByteArrayOS = new ByteArrayOutputStream();
ImageIO.write(image, formatName: "jpg", imageByteArrayOS);

// 转字节数组
byte[] imageByteArray = imageByteArrayOS.toByteArray();
```

转化为字节数组

3. 数据分片与发送

尽管对图像进行了压缩处理，在常规情况下不会发生数据大小超过 UDP 报文段最大长度的情况，但考虑到若追求超高画质，仍可能会出现图片数据过大，无法一次发送的情况，故需要对数据进行分片，使每一片的大小不超过最大值。

考虑到服务器端的分片重组，我对 UDP 报文段数据部分进行了设计，在数据部分新增了 14 字节的数据首部字段，便于服务器重组。

①	②	③	④	⑤	⑥
---	---	---	---	---	---

(1) 1 字节长度的开始标志，一个图片的第一段分片为 0x80，其余分片为 0x00，用来提示服务器新的图片已经到来。

(2) 1 字节长度的图像编号，所有相同图像的分片有同一个编号，服务器以此确认收到的分片是否属于同一图片。

(3) 1 字节长度的总片数，记录该分片所在图像被分为了几片。由于 1 字节整数最大值为 256，故最大可将一张图片分成 256 片，足以满足需求。

(4) 1 字节长度的当前片数，记录该分片使该图像的第几分片。

(5) 2 字节长度的当前片大小。

(6) 8 字节长度的时间戳，用来记录发送时的时间，精确到毫秒，服务器以此来计算时延。

客户端按序处理完一片后便向服务器发送。

```

// 首部数据处理
fragment[0] = (byte)flags; // 开始标志
fragment[1] = imageNumber; // 同一图像片段计数
fragment[2] = (byte)fragments; // 片数
fragment[3] = (byte)i; // 当前片段
fragment[4] = (byte)(size >> 8); // 需要2个字节来表示当前大小
fragment[5] = (byte)size;
fragment[6] = timestampByteArray[0]; // 需要8个字节来表示时间戳
fragment[7] = timestampByteArray[1];
fragment[8] = timestampByteArray[2];
fragment[9] = timestampByteArray[3];
fragment[10] = timestampByteArray[4];
fragment[11] = timestampByteArray[5];
fragment[12] = timestampByteArray[6];
fragment[13] = timestampByteArray[7];

// 复制图像的图像部分以及首部
System.arraycopy(imageByteArray, srcPos: i * DATAGRAM_MAX_SIZE, fragment, HEADER_SIZE, size);

// 创建要发送的包
DatagramPacket packet = new DatagramPacket(fragment, fragment.length, inet, PORTA);
// 发送数据
socket.send(packet);

```

分片与发送

客户端循环采集图片并分片发送，从而形成视频。

4. 服务器接收并处理

服务器从 UDP 报文中取出数据部分，并从数据部分提取出相关信息。

```

DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
// 从包中提取数据
byte[] fragment = packet.getData();
// 从数据中提取信息
byte flag = fragment[0];
short imageNumber = fragment[1];
int fragments = fragment[2] & 0xff;
short currentFragment = (short)(fragment[3] & 0xff);
int size = (fragment[4] & 0xff) << 8 | (fragment[5] & 0xff);
byte[] timestampByteArray = new byte[] {
    (byte) (fragment[6] & 0xff),
    (byte) (fragment[7] & 0xff),
    (byte) (fragment[8] & 0xff),
    (byte) (fragment[9] & 0xff),
    (byte) (fragment[10] & 0xff),
    (byte) (fragment[11] & 0xff),
    (byte) (fragment[12] & 0xff),
    (byte) (fragment[13] & 0xff),
};

```

提取信息

服务器根据提取到的信息进行如下操作：

- (1) 根据开始标志字段判断收到新图片的分片，则更新当前图片编号，并设置 **CanReceive** 布尔量为 **true**，让分片计数器清零。新建一个大小为总片数*每片最大长度的字节数组 **imagedata[]**来接收分片。未来只接收图片编号为当前图片编号的分片，否则直接丢弃。
- (2) 接收到图片编号为当前图片编号的分片且 **CanReceive** 为 **true**，则将图片部分的数据拷贝到 **imagedata** 中。同时让分片计数器加一。
- (3) 若已收到的分片等于总分片数，即已接收完所有分片，则向窗口更新图片，并设置 **CanReceive** 为 **false**，这意味着，直到有一个新图片的第一分片到来前收到的分片都将被丢弃。

通过上述操作，服务器可以正确接收并显示图像，因 **UDP** 的不可靠性导致的问题也可被正确处理。

5. 服务器计算时延等参数

服务器根据 **UDP** 报文中的 8 字节时间戳字段并结合服务器自己获取的时间可测出时延。服务器每接收到一个 **UDP** 报文即获取一次当前时间 t_1 ，上一次获取的时间为 t_2 ，报文段长度为 L ，则接收速率为：
$$u = \frac{L}{t_1 - t_2}。$$

三、实验结果与分析

首先使用回环地址，采用 **VGA** 模式发送，服务器页面如下：



延迟在 10ms 内变动，可以看出，使用回环地址基本无延时，接收速率在 0.4MB/s 左右。在服务器终端可输入 **q** 以结束接收，结束后关闭页面即可结束程序。

```
服务器端已经启动，等待客户端发送数据...
输入q以结束接收
q
监视结束！

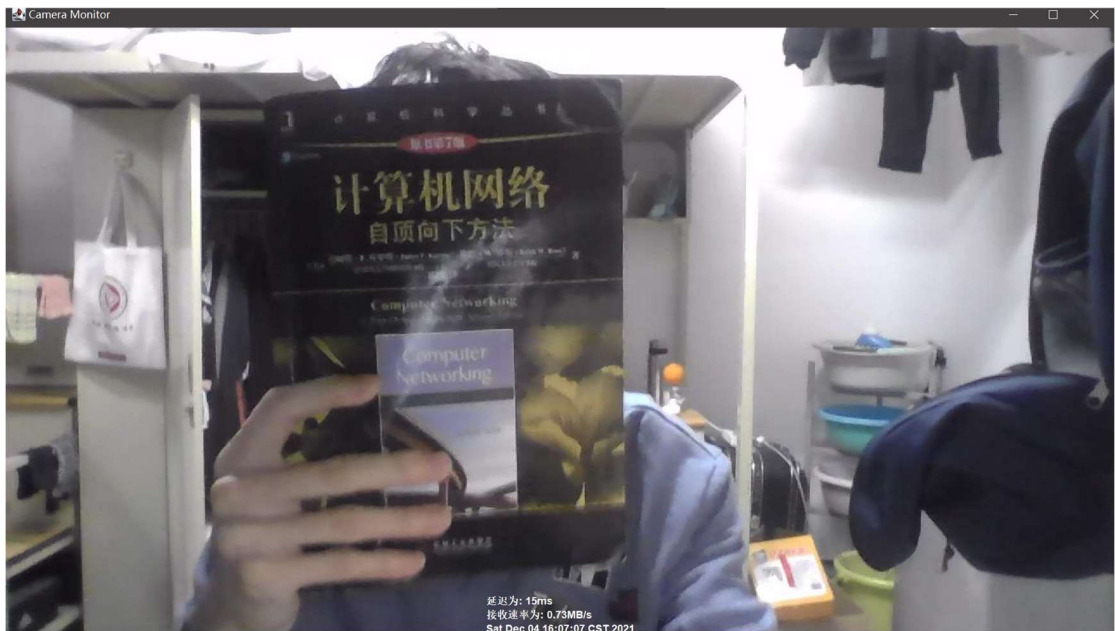
Process finished with exit code 0
```

在客户端可显示数据大小及分片数：



可以看出 VGA 模式采集图片的大小并未超过最大值。

下面改用 HD720 模式：



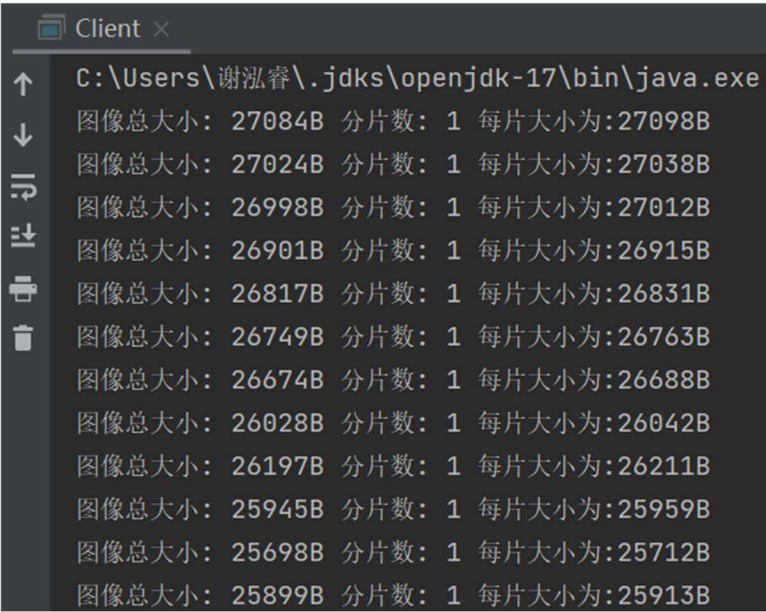
可以看出延迟依旧很低（约 15ms 左右），接收速率大幅度增加（在 0.8MB/s 左右）。而在客户端，可以看出 HD720 画质的图片大小较大，有些需要分片处理：

Client			
↑	图像总大小:	59045B	分片数: 1 每片大小为:59059B
↓	图像总大小:	64623B	分片数: 1 每片大小为:64637B
↕	图像总大小:	65181B	分片数: 1 每片大小为:65195B
↔	图像总大小:	64653B	分片数: 1 每片大小为:64667B
⇄	图像总大小:	64391B	分片数: 1 每片大小为:64405B
📄	图像总大小:	64653B	分片数: 1 每片大小为:64667B
🗑️	图像总大小:	63883B	分片数: 1 每片大小为:63897B
	图像总大小:	65559B	分片数: 2 每片大小为:65507B 80B
	图像总大小:	62037B	分片数: 1 每片大小为:62051B
	图像总大小:	64677B	分片数: 1 每片大小为:64691B
	图像总大小:	66189B	分片数: 2 每片大小为:65507B 710B
	图像总大小:	66872B	分片数: 2 每片大小为:65507B 1393B
	图像总大小:	67152B	分片数: 2 每片大小为:65507B 1673B
	图像总大小:	67114B	分片数: 2 每片大小为:65507B 1635B
	图像总大小:	67112B	分片数: 2 每片大小为:65507B 1633B
	图像总大小:	67559B	分片数: 2 每片大小为:65507B 2080B
	图像总大小:	67910B	分片数: 2 每片大小为:65507B 2431B
	图像总大小:	67678B	分片数: 2 每片大小为:65507B 2199B

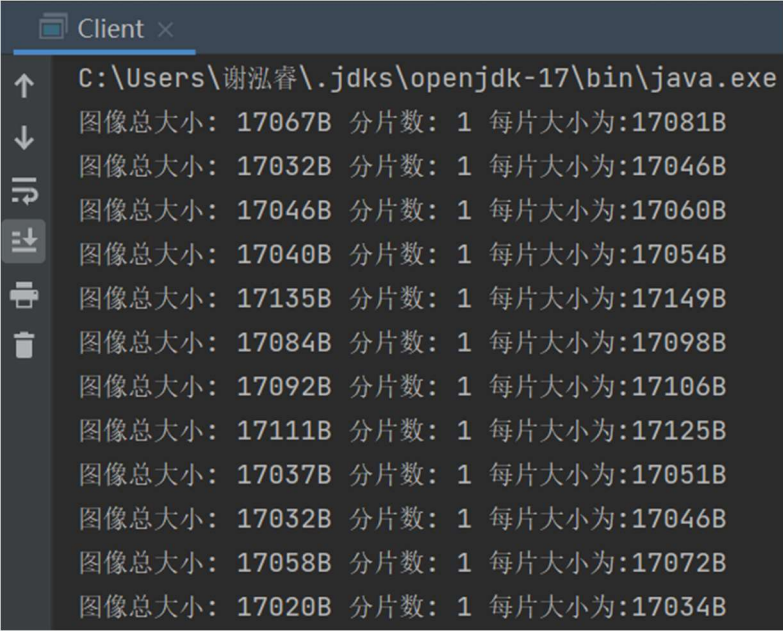
可以看出，数据的分片情况在服务器和客户端都能得到正确处理。

最后，测试一下压缩过程。采用 VGA 模式，压缩率分别使用 1.0

和 0.7，可以看出，压缩效果明显。



```
Client x
C:\Users\谢泓睿\.jdk\openjdk-17\bin\java.exe
图像总大小: 27084B 分片数: 1 每片大小为:27098B
图像总大小: 27024B 分片数: 1 每片大小为:27038B
图像总大小: 26998B 分片数: 1 每片大小为:27012B
图像总大小: 26901B 分片数: 1 每片大小为:26915B
图像总大小: 26817B 分片数: 1 每片大小为:26831B
图像总大小: 26749B 分片数: 1 每片大小为:26763B
图像总大小: 26674B 分片数: 1 每片大小为:26688B
图像总大小: 26028B 分片数: 1 每片大小为:26042B
图像总大小: 26197B 分片数: 1 每片大小为:26211B
图像总大小: 25945B 分片数: 1 每片大小为:25959B
图像总大小: 25698B 分片数: 1 每片大小为:25712B
图像总大小: 25899B 分片数: 1 每片大小为:25913B
```



```
Client x
C:\Users\谢泓睿\.jdk\openjdk-17\bin\java.exe
图像总大小: 17067B 分片数: 1 每片大小为:17081B
图像总大小: 17032B 分片数: 1 每片大小为:17046B
图像总大小: 17046B 分片数: 1 每片大小为:17060B
图像总大小: 17040B 分片数: 1 每片大小为:17054B
图像总大小: 17135B 分片数: 1 每片大小为:17149B
图像总大小: 17084B 分片数: 1 每片大小为:17098B
图像总大小: 17092B 分片数: 1 每片大小为:17106B
图像总大小: 17111B 分片数: 1 每片大小为:17125B
图像总大小: 17037B 分片数: 1 每片大小为:17051B
图像总大小: 17032B 分片数: 1 每片大小为:17046B
图像总大小: 17058B 分片数: 1 每片大小为:17072B
图像总大小: 17020B 分片数: 1 每片大小为:17034B
```

除了采用回环地址，我也使用两台主机进行了测试，由于两台主机的系统时间存在不一致的情况，故时延数据误差较大。从直观感受上来看，并无明显时延。

四、讨论、感想

通过实验 3，我实现了摄像头图像的采集、图像的压缩、数据的分片与重组、图像的显示。时延与速率的计算等功能，加深了对 UDP 协议的理解，熟练掌握了 Java 网络编程的基本方法，为未来的学习和工作打下了基础。

附：参考文献：

<https://stackoverflow.com/questions/38723393/>

<https://stackoverflow.com/questions/31324551/>

<https://stackoverflow.com/questions/14757775/>

<http://webcam-capture.sarxos.pl/>