Lab1: Socket Programming

常烁晨-521021910369

一、介绍

Socket编程，常被称为“套接字编程”，在网络应用领域扮演着至关重要的角色。套接字是一种网络通信的基础设施，它为应用层与传输层之间的信息交换提供了一个可编程的接口，允许独立运行的进程通过网络进行数据交换。这种编程实践使得不同的应用程序（具体指进程之间）能够实现数据的交流和信息的互换。

具体而言，套接字编程是一种使得进程间能够通信的机制。在Linux操作系统环境中，通过编写客户端和服务器端程序，并在相应的环境中运行这些程序，可以模拟套接字的实际工作方式。这些程序必须执行一系列操作，用于控制和管理进程之间的通信流程。本实验旨在展示一个具体的应用实例，即进程间的文件复制，通过这一实例，我们可以深入理解套接字编程的基本原理和应用方式。

此外在本实验中使用了Mininet工具。Mininet能够在单个Linux虚拟机或物理机上创建一个完整的虚拟网络，用户可以自定义网络拓扑结构，创建虚拟节点，包括交换机、路由器、主机等。这些虚拟节点能够模拟真实网络中的行为。

二、示例

实验指导文档中给出了一个简单的Socket Programming示例。将代码复制后编译运行。注意 ./client 要在新窗口中执行。

尝试使用Mininet环境模拟网络上进程通信。将给定的mininet脚本运行后，在mininet CLI中启动命令。注：&表示后台运行。

最后尝试多线程。多线程要求对服务器添加互斥锁mutex等机制，使得多个用户进程可以同时访问服务器。

三、作业

使用TCP套接字API来实现文件共享系统工作。

1、实现客户端/服务器模型。

首先是server.cc:

首先定义服务器处理单个客户端请求（以客户端套接字为接收参数）的函数。首先定义服务器上保存的文件（此处用一个文本文件为例）。此后服务器响应请求时打开文件，并通过 send 函数像客户端套接字发送文件内容。

在 main 函数中，首先对服务器的套接字等内容进行创建并初始化，设置并绑定 ip 地址到套接字上，并设置其的监听状态。接下来进行无限循环，不断接受客户端连接。每当接受到一个新的连接，就创建一个新线程来处理该连接，允许同时处理多个客户端连接。

接下来是client.cc:

在 main 函数中创建并初始化客户端的套接字，用于与服务器的通信。此处设置了服务器地址的相关信息，包括端口号等。之后通过套接字与服务器进行连接。

注意此处在命令行中传入一个名称，作为客户端的本地路径，并在该路径下新建一个文本文件，通过 read 函数从套接字中读取信息，并写入该文件中。

下面展示 Mininet 脚本文件的拓扑结构。本实验需要在 Mininet 虚拟环境中成功运行。首先模拟网络的拓扑结构。

然后启动脚本，在脚本中直接执行相关的 mininet CLI，观察实验结果。

此处多输入一个参数，是考虑到 client.cc 的程序设计架构，输入一个名称作为本地目录名。启动脚本后效果如图。（无需再输入命令）

同时可以观察到 lab01 路径下新增了三个客户端对应的文件夹，文件夹中的内容与 server 提供的文本文件内容相同，说明模拟成功。

2、评估下载时间：

对于 Mininet 启动脚本作出修改，用于统计文件下载的时间。首先检测存在 3 个客户端时下载的时间。

之前的脚本只能模拟网络通断的情况，无法模拟多个 client 同时访问时对带宽的限制。此外，考虑到在脚本中计算时间不够精确，此处还改动了 client.cc 代码，

首先给出脚本代码的修改：

```
# 添加链路
serverBw = 10
clientBw = serverBw / clientCount  # 平均分配带宽给每个客户端
self.addLink(server, switch, bw=serverBw, delay='10ms', loss=1, use_htb=True)
for client in clients:
    self.addLink(client, switch, bw=clientBw, delay='20ms', loss=2, use_htb=True)
```

下面是对 client.cc 的修改，主要是设置其在 log 中打印下载时间。

```
auto start = std::chrono::high_resolution_clock::now();

int bytesReceived = 0;
while((bytesReceived = read(sock, buffer, 1024)) > 0) {
    outFile.write(buffer, bytesReceived);
}
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed = end - start;
std::string logFilePath = folderName + "_client_log.txt";
std::ofstream logFile(logFilePath);
logFile << folderName << ":\tTime taken: " << elapsed.count() << " seconds" << std::endl;
logFile.close();
outFile.close();
close(sock);
```

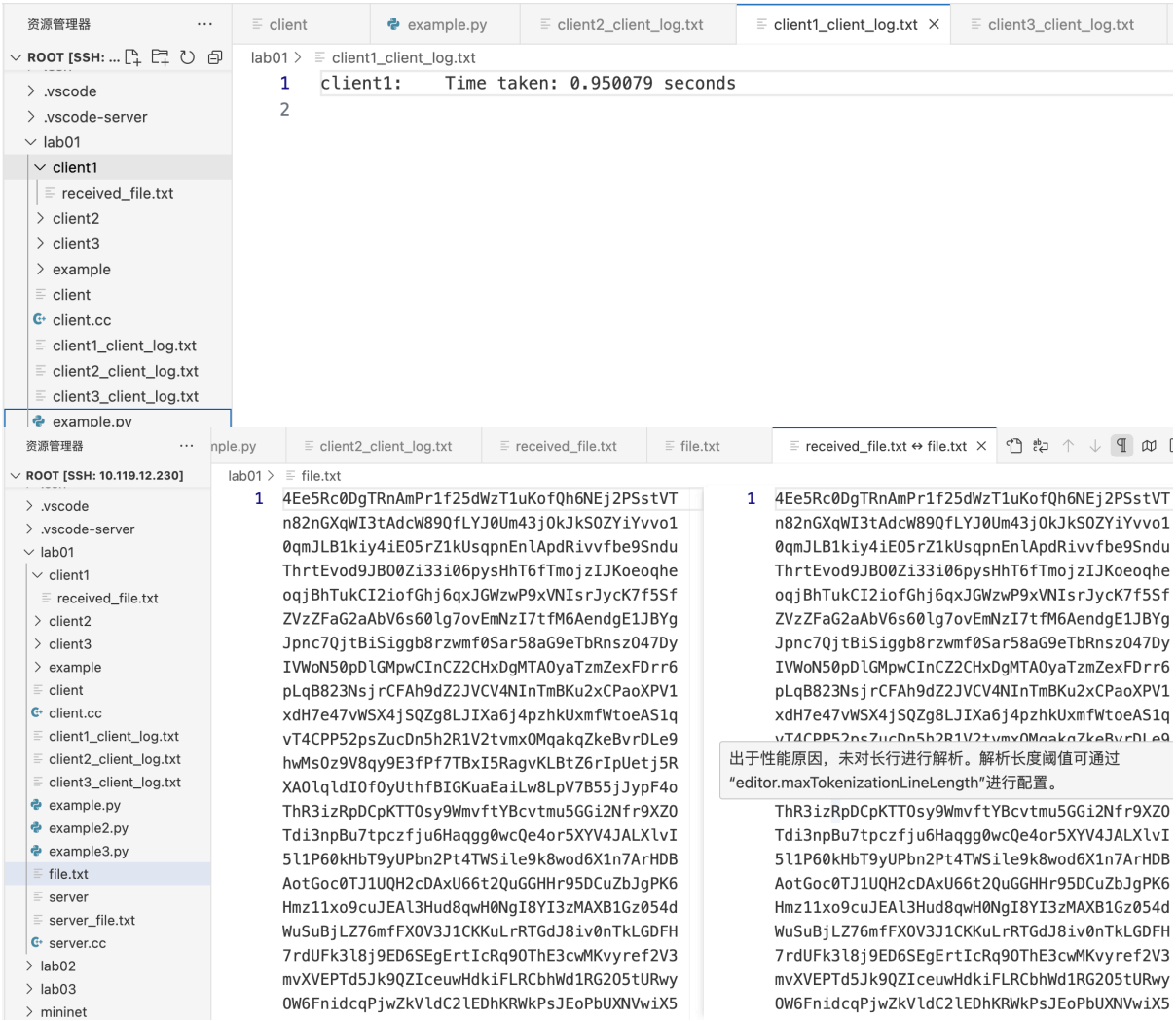
此外为了计算下载时间，此处需要生成较大的文本文件。采用

```
head -c 1024000 /dev/urandom | tr -dc 'a-zA-Z0-9' > file.txt
```

来产生较大的可读性文本文件。这条命令产生一个 1000KB （不是1M）的序列，筛选其中的字母和数字保存在 file.txt 中。此时 server.cc 也需要对文件目录做出相应修改。这样便于打开 txt ，验证复制是否正确。

更改参数，分别统计当客户端有 3 个和 10 个的时候，客户端输出的 log 日志的下载时间是否发生明显变化。

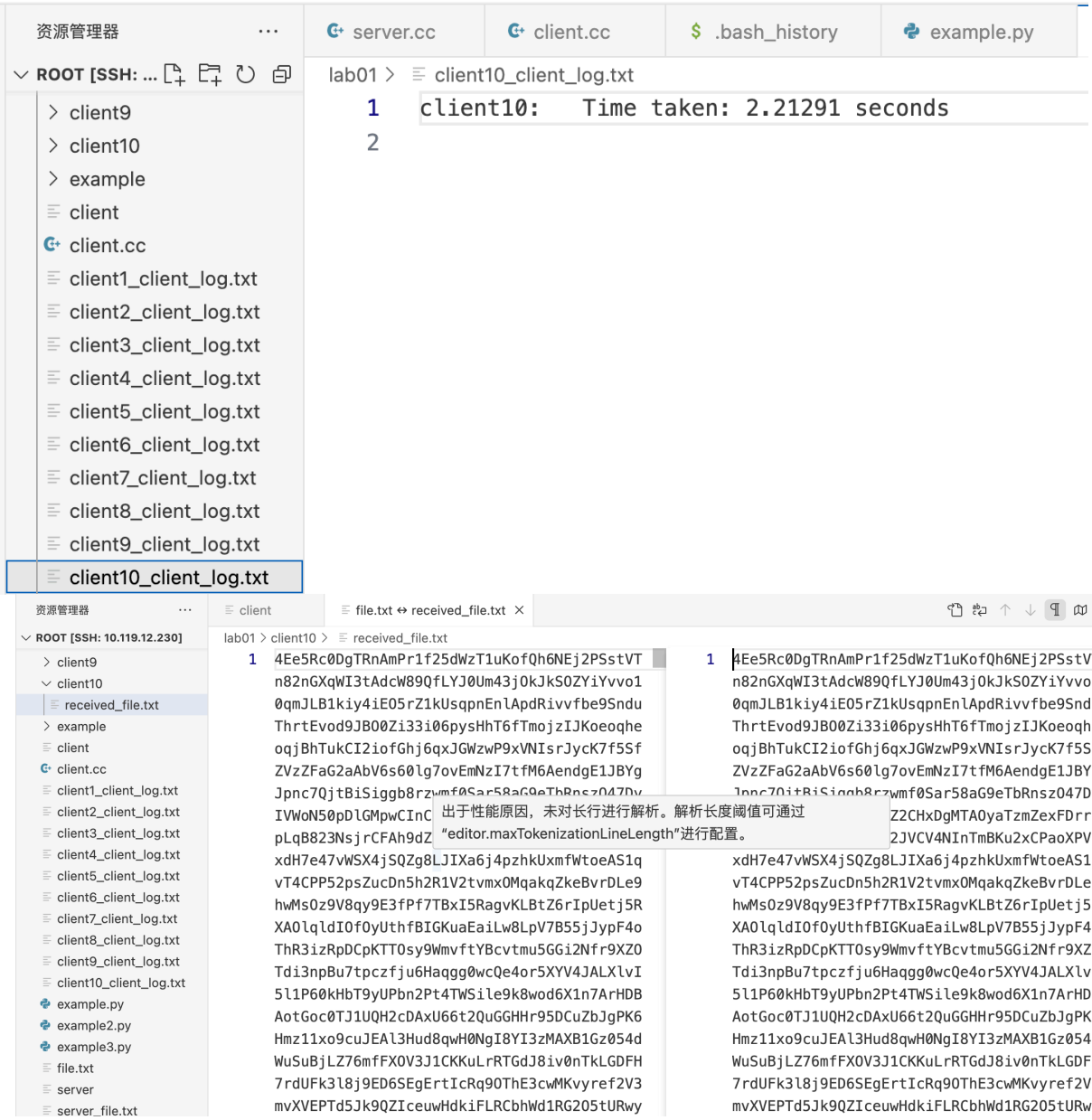
首先是clientCount = 3:



平均下载时间在 0.7s 左右。

其次是 clientCount = 10:

平均下载时间在 2.2s 左右。


资源管理器
server.cc
client.cc
.bash_history
example.py
ROOT [SSH: ...
lab01 > client10_client_log.txt
client10: Time taken: 2.21291 seconds
client9
client10
example
client
client.cc
client1_client_log.txt
client2_client_log.txt
client3_client_log.txt
client4_client_log.txt
client5_client_log.txt
client6_client_log.txt
client7_client_log.txt
client8_client_log.txt
client9_client_log.txt
client10_client_log.txt
资源管理器
ROOT [SSH: 10.119.12.230]
client
file.txt ↔ received_file.txt
lab01 > client10 > received_file.txt
出于性能原因，未对长行进行解析。解析长度阈值可通过"editor.maxTokenizationLineLength"进行配置。
received_file.txt
example.py
example2.py
example3.py
file.txt
server
server_file.txt