LAB 4–Socket 和网络编程

张煌昭，1400017707，元培学院

摘要—本次 Lab 使用 Python3 的 socket 包编程，采用 TCP 通信，实现了一对简单的 Echo Client-Server。并在此基础上，将 Server 运行与远程服务器上，并对其进行速率、丢帧率等测试。本次报告使用 Overleaf L^ATEX 在线平台编写[1]，实验源码开源于 GitHub 平台[2]。

I. Socket 编程

Socket 由 IP 地址和端口两部分组成。网络上的两个进程通过一条双工连接实现数据交换时，这一条连接的一段便为一个 Socket（也称套接字）。粗略的理解下，一个 Socket 便对应了一个计算机进程，计算机进程通过自己的 Socket 与其它进程的 Socket 连接，实现数据交换。

目前网络主要使用 TCP 和 UDP。其中 TCP 要求可以提供面向连接的可靠服务，一个 TCP 连接被其两端的 Socket 唯一确定，即进程 A 与进程 B 的 TCP 连接一一对应于 $\{Socket_A, Socket_B\} = \{(IP_A, Port_A), (IP_B, Port_B)\}$。而 UDP 则提供无连接的不可靠服务，进程 A 与进程 B 进行数据交换时，必须检查消息的地址是否正确。

基于 **UDP** 的 Client-Server 的工作流程如图 1(a)所示。由于 UDP 不需要建立连接，Server 和 Client 端不需要建立连接，但需要在发送和接收时指明发端和收端。UDP 的 Server 端简单易于实现，并且 UDP Server 的并发也易于进行实现。

基于 **TCP** 的 Client-Server 的工作流程如图 1(b)-1(c)所示，其中图 1(b)为单线程的 TCP Server 的工作流程图，图 1(c)为多线程的 TCP Server 的工作流程图。TCP 要求建立连接，因此 Server 端必须 Bind 到相应的端口上并监听该端口；Client 端建立 Socket 后通过 Connet 尝试连接 Server 端监听 Socket。单线程 Server 一次只能支持一个 Client 连接并提供服务，当该 Client 的服务完成后，双方关闭 Socket，Server 回到监听状态继续等待别的 Client 连接。多线程 Server 可以支持多个 Client 同时连接，其可以分为 Master Server 和 Slave Server，Master Server 进行监听，接受 Client 的连接并建立 Socket 连接；当 Socket 连接建立后，Master Server 通过创建线程，该线程作为 Slave Server 完成对该 Client 的服务。为了防止连接出错，或用户连接后长时间不进行操作占用 Server 资源，在 Server 端会设置计时，当计时到且 Client 没有操作时，产生 Timeout 异常，Server 可以选择在接到 Timeout 异常时将该连接关闭以防止资源浪费。

[1]本报告源码可通过以下 git 命令获得，
git clone https://git.overleaf.com/15642711mpgntxxpnxdn

[2]实验源码可通过以下 git 命令获得，
git clone git@github.com:LC-John/dummy_chat.git

II. Simple Echo Client-Server

本节在第 I节的基础上，介绍和讨论实现的 Simple Echo 程序。该对程序实现简单的 Echo 服务，Client 同 Server 建立连接后，Client 端发送一段用户输入的消息，Server 端接收后原样发回，完成一次服务后，双方约定断开 Socket 连接，不支持连续多次的输入。详细代码请见开源目录下/servers/simple_server.py（对应 Server 端）和/clients/simple_client.py（对应 Client 端）文件。

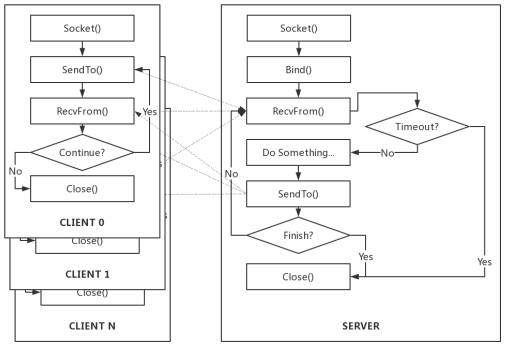
A. Simple Echo Client

Client 端严格按照图 1(c)实现即可，实现的详细情况请见开源目录下/clients/simple_client.py 文件。

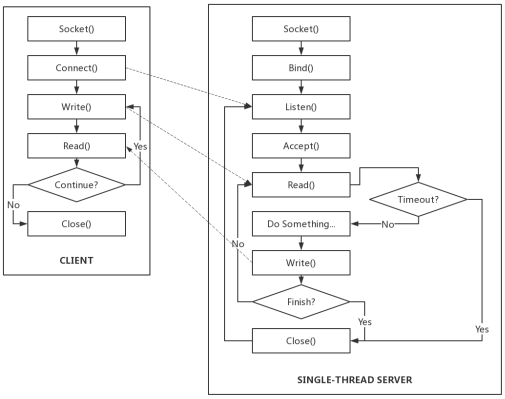
在 Simple Echo Client 中，实现了 Client 端的基于 TCP 的单次 Socket 通信，可以通过参数对 Client 连接的 Server 的 IP 地址和端口，超时时间（即等待 Server 端 Echo 的时间上限）和缓冲区大小进行设置，具体的参数如表 I所示。

表 I
Simple Client 参数设置

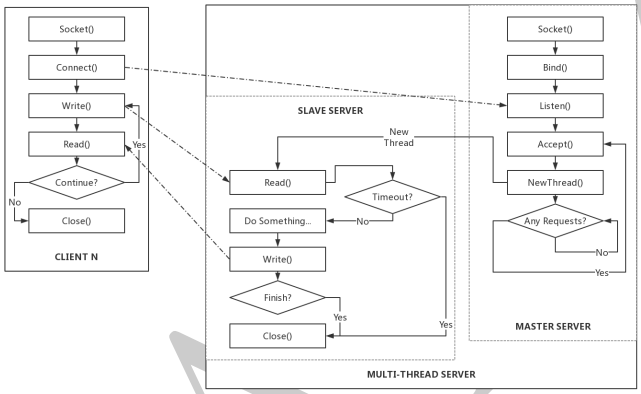
参数	含义	默认缺省值
-h	打印帮助菜单，不可与其余参数同时使用	–
–ip	设置连接的 Server 端的 IP 地址	localhost
–port	设置连接的 Server 端的端口	2333
–timeout	设置超时时间（单位秒），-1 表示不设置	-1
–buffer	设置读取时的缓冲区大小	1024



(a) Socket 实现基于 UDP 的 Client-Server 的流程



(b) Socket 实现基于 TCP 的 Client-Server(单线程) 的流程



(c) Socket 实现基于 TCP 的 Client-Server(多线程) 的流程

图 1. 使用 Socket 编程实现的各类 Client-Server 应用的服务流程。图 1(a)为 UDP 的 Client-Server 流程；图 1(b)为 Server 端单线程的 TCP 的 Client-Server 流程；图 1(c)为 Server 端多线程的 TCP 的 Client-Server 流程。

使用命令 'python3 simple_client.py' 并通过命令行参数进行设置，运行 Simple Echo Client，打印配置参数并连接 Server 端，若连接成功则等待用户进行输入。结果展示请见第II-C节。

表 II
Simple Server 参数设置

参数	含义	默认缺省值
-h	打印帮助菜单，不可与其余参数同时使用	–
–port	设置 Server 监听的端口	2333
–listen	设置最大监听数	100
–timeout	设置超时时间（单位秒），-1 表示不设置	10
–buffer	设置读取时的缓冲区大小	1024

B. Simple Echo Server

Server 端按照图 1(c)中所示进行实现。Server 开始运行后会创建 Socket 并 Bind 在相应的端口之上，之后 Listen 监听等待连接；一旦有 Client 试图连接，则 Accept 建立连接；之后创建新的子线程，传递 Socket 连接并运行 echo_server 函数，子线程作为 Slave Server 进行 Echo 服务，而监听的父线程作为 Master Server 继续监听端口等待连接。详细代码请见开源目录下/servers/simple_server.py 文件。

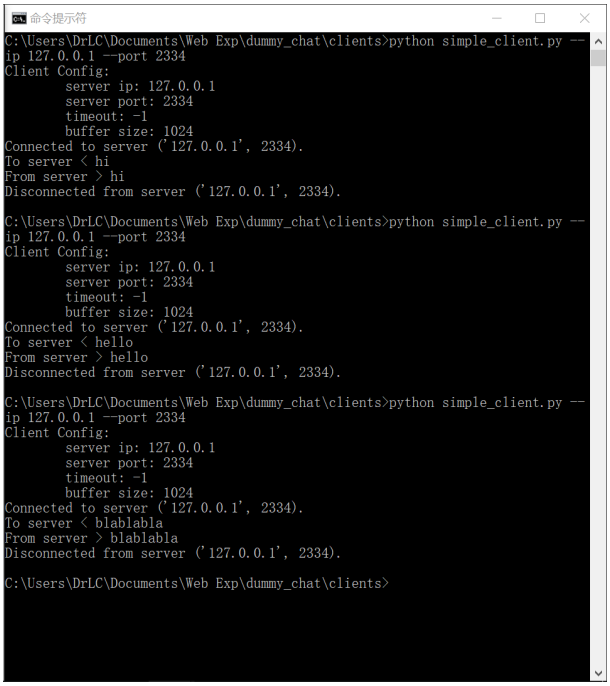
在 Simple Echo Server 中，实现了 Server 端的基于 TCP 的单次 Socket 通信，可以通过参数对 Server 监听的端口，最大监听数目，超时时间（即等待 Server 端 Echo 的时间上限）和缓冲区大小进行设置，具体的参数如表 II所示。

使用命令 'python3 simple_server.py' 并通过命令行参数进行设置，运行 Simple Echo Server，打印配置参数并等待连接和提供 Echo 服务。结果展示请见第II-C节。

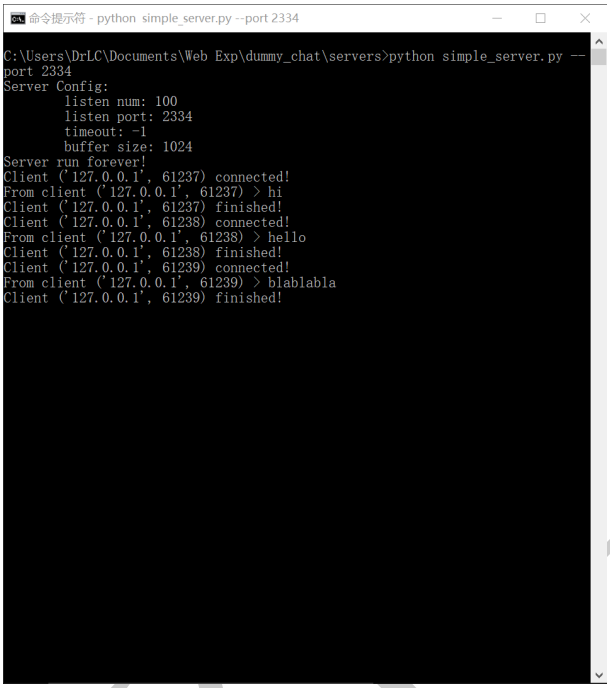
C. Simple Echo 本地连接测试结果

首先，测试单一 Client 连接 Server 的场景下，Client 端和 Server 端的行为，如图 2(a)和 2(b)所示。Client 每次连接 Server，发送一条用户输入的消息后，迅速收到 Echo 回复并打印，之后断开连接，若需要连续输入，则必须连接多次；Server 端接收 Client 连接后，Slave Server 收到一条 Client 端的消息，便原样发回进行 Echo 服务，之后断开连接。结果表明 Simple Echo 程序符合设计预期。

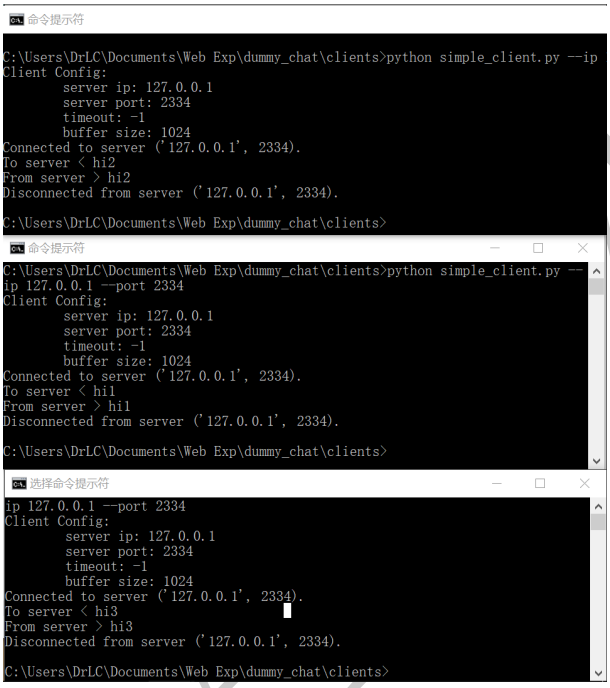
之后，对多 Client 连接 Server 的场景下，Client 端和 Server 端的行为，进行测试，如图 2(c)和 2(d)所示。共有 3 个 Client 端同时连接至 Server 端，之后三个 Client 按顺序发送消息（消息中的标号顺序），各个 Client 端均接收到正确的 Echo 回复；Server 端，由于多线程机制的作用，三个 Slave Server 相互独立，彼此



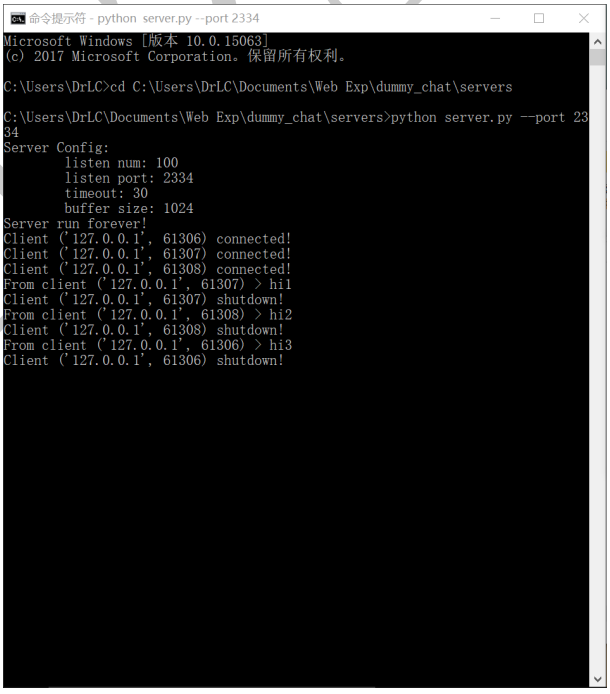
(a) 单一 Client 场景下 Simple Echo Client



(b) 单一 Client 场景下 Simple Echo Server



(c) 多 Client 场景下 Simple Echo Client



(d) 多 Client 场景下 Simple Echo Server

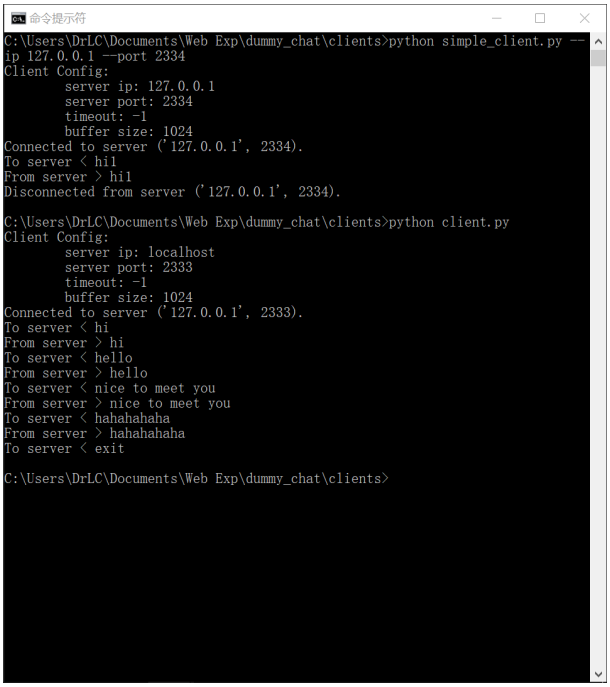
图 2. Simple Echo Client-Server 本地连接结果。图 2(a)和图 2(b)为同一组测试中的结果，该组测试展示了 Simple Echo 的运行结果，图 2(a)为 Client 端的输出结果，用户在 Client 端每输入一个消息，便迅速收到 Server 端的 Echo 回复；图 2(b)为 Server 端地输出结果，每有 Client 连接，断开或发送消息，都有相应的输出结果。图 2(c)和图 2(d)为同一组测试中的结果，该组测试展示了多 Client 场景下的运行结果，图 2(a)为 3 个 Client 端的输出结果，3 个用户在 Client 端按顺序输入消息，迅速收到 Server 端的 Echo 回复；图 2(b)为 Server 端的输出结果，每有消息顺序的输出结果正确。

不会影响，并且由于打印输出的系统调用对线程时间片具有原子性，因此其打印接收到的消息顺序也正确。

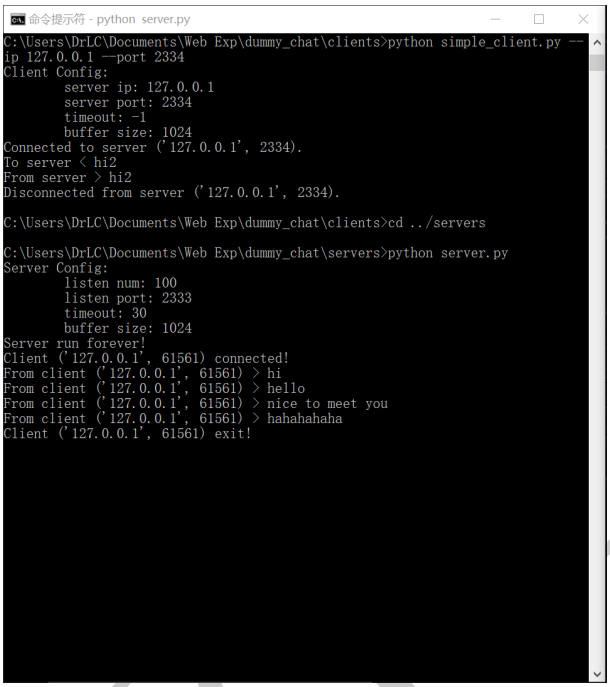
综合以上结果，判断多线程的基于 TCP 的 Simple Echo 程序，实现结果符合预期设计。

III. Echo Client Server

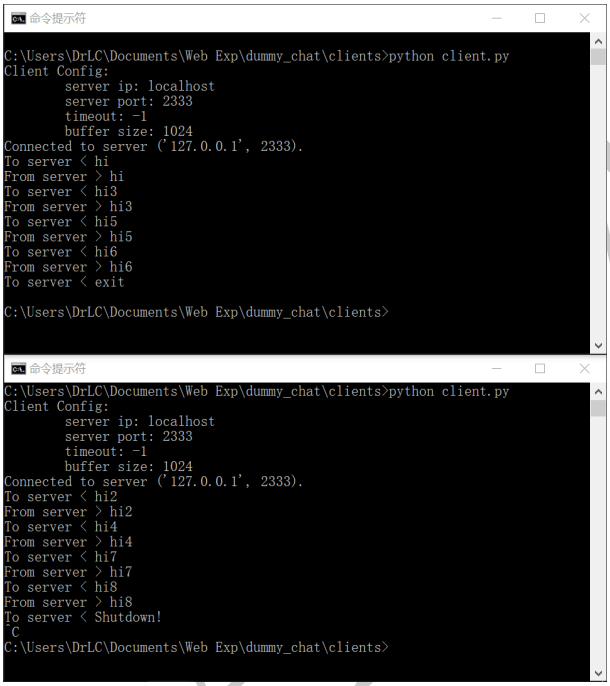
本节在第 I节和第 I的基础上，介绍和讨论实现的 Echo 程序，及对其进行性能测试的程序。该对程序实现完全的 Echo 服务，Client 同 Server 建立连接后，Client



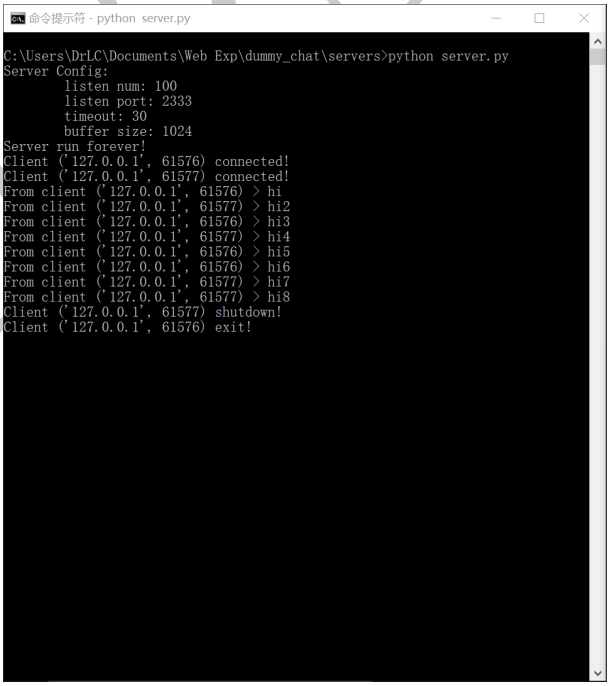
(a) 单一 Client 场景下 Echo Client



(b) 单一 Client 场景下 Echo Server



(c) 多 Client 场景下 Echo Client



(d) 多 Client 场景下 Echo Server

图 3. Echo Client-Server 本地连接结果。图 3(a)和图 3(b)为同一组测试中的结果，该组测试展示了 Echo 的运行结果，图 3(a)为 Client 端的输出结果，用户在 Client 端每输入一个消息，便迅速收到 Server 端的 Echo 回复，多次重复依然可用；图 3(b)为 Server 端地输出结果，Client 可以多次发送，并依然可用。图 3(c)和图 3(d)为同一组测试中的结果，该组测试展示了多 Client 场景下的运行结果，图 3(a)为 2 个 Client 端的输出结果，2 个用户在 Client 端按顺序轮番输入消息，均收到正确回复，一个 Client 使用 Ctrl+C 退出，另一 Client 使用 exit 退出；图 2(b)为 Server 端的输出结果，每有消息顺序的输出结果正确，Ctrl+C 退出和 exit 退出的 Client 均正常断开连接。

端发送一段用户输入的消息，Server 端接收后原样发回，这一过程持续多轮，直至出现连接错误或 Client 端中断连接。详细代码请见开源目录下/servers/server.py（对应 Server 端）和/clients/client.py（对应 Client 端）文件。测试程序使用作为 Client 端的特例，不再需要用户输入，而是自动连续发送，测试 Server

端回复的速率，以及发生错误的概率。详细代码请见开源目录下的/clients/rate.py（对应速率测试）和/clients/fail_rate.py（对应错误测试）。

A. Echo Client

Client 端实现同第II-A节中 Simple Client 的实现类似，仅仅是 Client 不再会在一次 Echo 后断开连接，而是不断进行循环，直至发生 KeyInterupt 中断（命令行中按下 'Ctrl+C' 终止 Client 进程），发生中断后，Client 端断开连接。实现的详细情况请见开源目录下/clients/simple_client.py 文件。Client 端程序的参数设置与 Simple Client 相同。

使用命令 'python3 client.py' 并通过命令行参数进行设置，运行 Echo Client，打印配置参数并连接 Server 端，若连接成功则等待用户进行输入。结果展示请见第III-C节。

B. Echo Server

Server 端实现同第II-B节中 Simple Server 的实现类似,Slave Server 进行重复的 Echo 服务，直至超时或者连接出现异常，一旦有异常发生，则中断连接并打印。详细代码请见开源目录下/servers/server.py 文件。Server 端程序的参数设置与 Simple Server 相同。

使用命令 'python3 server.py' 并通过命令行参数进行设置，运行 Echo Server，打印配置参数并等待连接和提供 Echo 服务。结果展示请见第III-C节。

C. Echo Result

同第II-C节，首先，测试单一 Client 连接 Server 的场景下，Client 端和 Server 端的行为，如图 3(a)和3(b)所示。Client 每次连接 Server，发送一条用户输入的消息后，迅速收到 Echo 回复并打印，该过程可以重复多次；Server 端接收 Client 连接后，Slave Server 每收到一条 Client 端的消息，便原样发回进行 Echo 服务，该过程也重复多次。当 Client 端输入 exit 或按下 Ctrl+C 后，双方断开连接。

之后，对多 Client 连接 Server 的场景下，Client 端和 Server 端的行为，进行测试，如图 3(c)和 3(d)所示。共有 2 个 Client 端同时连接至 Server 端，之后 2 个 Client 按顺序发送消息（消息中的标号顺序），各个 Client 端均接收到正确的 Echo 回复，最后一个 Client 先按下 Ctrl+C 中断连接，另一 Client 输入 exit 中断连接，二者均正常结束；Server 端收到消息的顺序正常，回复正常，对于 Ctrl+C 中断连接和 exit 中断连接的 Client 的处理均正常。

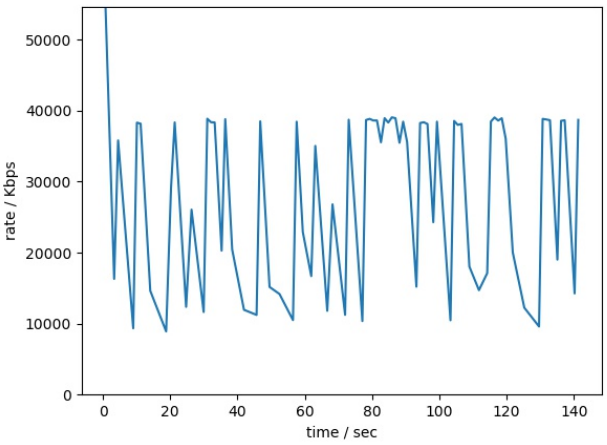


图 4. 本地测试 Echo 速率的曲线。由于本地连接不需要传输，因此速率极高。

综合以上结果，判断多线程的基于 TCP 的 Echo 程序，实现结果符合预期设计，可以提供多轮的 Echo 服务。

D. Echo 测试

编写两个 Client，用于对 Echo Server 的速率和错误率进行测试。

开源目录下的/clients/rate.py 为测试速率的 Client 端。该 Client 无需用户输入内容，在建立连接后，无间断地重复以下动作：1) 发送由 1000 个 0 组成的消息；2) 等待至收到消息，计时并检查回复是否正确；3) 若回复正确则计算本次 Echo 的速率并重新计算总的 Echo 速率，Echo 速率的计算公式如式 1所示。当用户按下 Ctrl+C 后，循环终止，测试 Client 端断开 Socket 连接，并绘制记录的速率曲线，如图 4所示。

$$rate = \frac{Length(Msg)}{RTT} \tag{1}$$

开源目录下的/clients/fail_rate.py 为测试错误率（失败率）的 Client 端。该 Client 无需用户输入内容，在建立连接后，无间断地重复以下动作：1) 发送由 1000 个 0 组成的消息；2) 等待至收到消息，计时并检查回复是否正确；3) 重新计算总的错误率。当用户按下 Ctrl+C 后，循环终止，测试 Client 端断开 Socket 连接。由于 TCP 本身是可靠连接，因此 TCP 从理论上，

从应用来看不会有错误发生，因而该测试实际上总是会得到错误率为 0 的结果；但若为 UDP，该测试即可检查出错误并计算错误率。

IV. 远程服务器部署和测试

A. 云服务器运行 *Server* 端

租用阿里云提供的轻量应用服务器用于部署 Simple Echo Server 和 Echo Server。首先通过控制台开放端口用于连接，之后通过 SSH 远程连接服务器配置 iptables 防火墙，开放端口。

将 server.py 和 simple_server.py 上传至服务器并运行，发现端口无法连接。发现代码中 Bind 的 IP 为 localhost，将其改为 0.0.0.0 重新运行，可以正常连接。

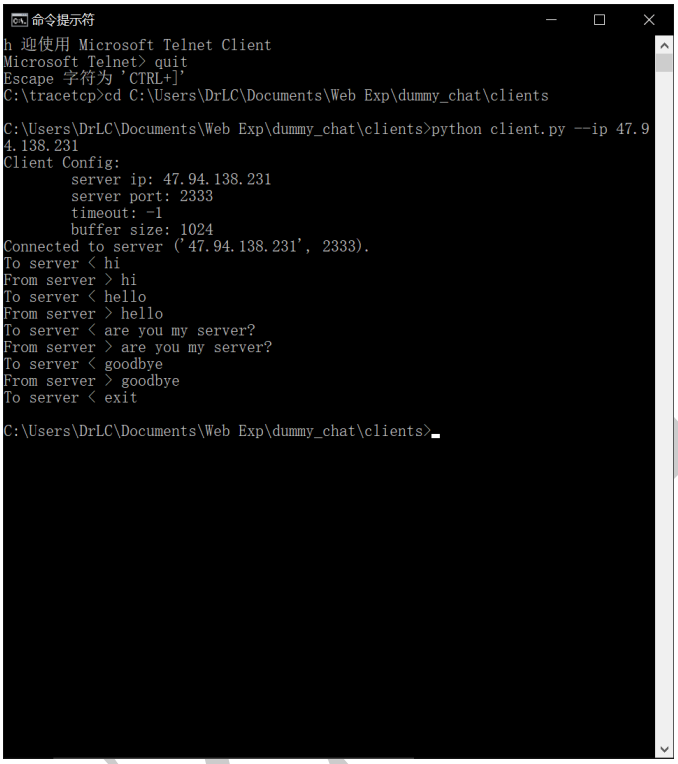
最终将 Simple Echo Server 运行于服务器 2334 端口，Echo Server 运行于服务器 2333 端口，服务器 IP 地址为 47.94.138.231，如 Server 端正常运行，则目前依然可以进行连接并测试。

B. *Client* 端连接测试

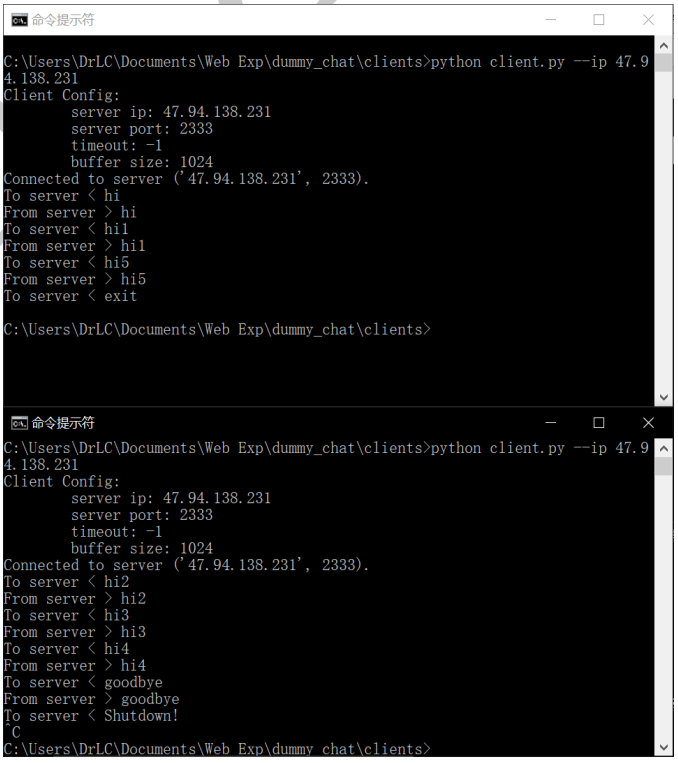
首先使用 tracetcp 工具检查链路以及服务器端口是否开放并有进程监听。结果如下，说明链路正常，服务器端口开放，可以用于连接。

```
$> tracetcp 47.94.138.231:2333
Tracing route to 47.94.138.231 on port 2333
Over a maximum of 30 hops.
1       8 ms    8 ms    8 ms    10.2.180.1
2       11 ms   5 ms    10 ms   162.105.252.46
3       9 ms    7 ms    5 ms    162.105.252.197
4       17 ms   13 ms   21 ms   202.112.41.185
5       9 ms    8 ms    10 ms   202.112.41.177
6       12 ms   11 ms   11 ms   101.4.117.82
7       4 ms    8 ms    14 ms   101.4.112.89
8       6 ms    44 ms   47 ms   101.4.112.2
9       39 ms   41 ms   43 ms   101.4.112.70
10      37 ms   38 ms   41 ms   101.4.116.117
11      37 ms   40 ms   38 ms   101.4.117.26
12      37 ms   *       52 ms   101.4.112.42
13      *       39 ms   *       101.4.135.202
14      42 ms   39 ms   43 ms   101.4.135.202
15      41 ms   35 ms   43 ms   219.224.103.226
16      45 ms   37 ms   40 ms   140.205.24.110
17      155 ms  57 ms   61 ms   140.205.25.26
18      51 ms   32 ms   43 ms   106.11.37.41
19      41 ms   39 ms   39 ms   11.218.196.225
20      Destination Reached in 35 ms.
Connection established to 47.94.138.231
Trace Complete.
```

接下来使用 telnet 工具，尝试对 Echo Server 进行连接。结果如下，每输入一个字符，telnet 便将其发送至 Server 端，之后收到相同的 Server 端的回复。说明 Server 端提供了正常的服务，每个字符作为一个消息发



(a) 单一 Client 场景下 Echo Client



(b) 多 Client 场景下 Echo Client

图 5. Client 端连接部署于云服务器的 Server 的测试。图 5(a)为单个 Client 连接下的测试；图 5(b)为 2 个 Client 连接下的测试。

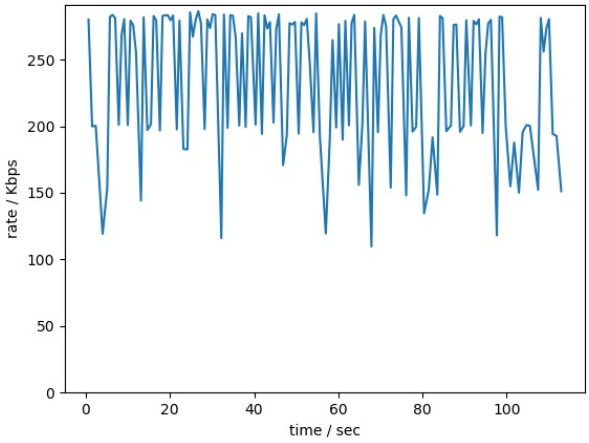


图 6. 云服务器测试 Echo 速率的曲线。

送至 Server 端，都收到了正确的回复字符（消息）。

```
$> telnet 47.94.138.231 2333
hheellllloo  tthhiiss  iiss  LLCC!!
```

最后,在本地运行 client.py,测试 Client 端与 Server 端的连接，如图 5所示。单个 Client 连接和多个 Client 连接结果均正确，说明部署于云服务器的 Server 端提供了正确的服务。

C. 性能测试

使用 rate.py 对远程 Server 端进行速率测试，得到 Echo 速率曲线，如图 6所示。由于错误率均为 0，未对其进行绘图展示。