网络应用程序设计与开发实验

任务一：头歌平台编程实现WEB服务器

任务二：TCP 与 QUIC 协议性能对比分析

任务要求：

1. 基于 TCP 的客户端-服务器程序实现，TCP服务器功能包括：监听指定端口；接受客户端连接；接收客户端发送的消息；向客户端返回响应（包含接收数据的长度）。客户端功能包括：连接到TCP服务器；向服务器发送指定消息；接收服务器响应并显示。
2. 基于QUIC的客户端-服务器程序实现：使用quiche库实现QUIC服务器与客户端，功能与第一条类似。

实验环境：

操作系统：Windows或Linux；

编程语言：C语言（其他语言也可，能实现实验功能即可）

依赖库：socket库、quiche库

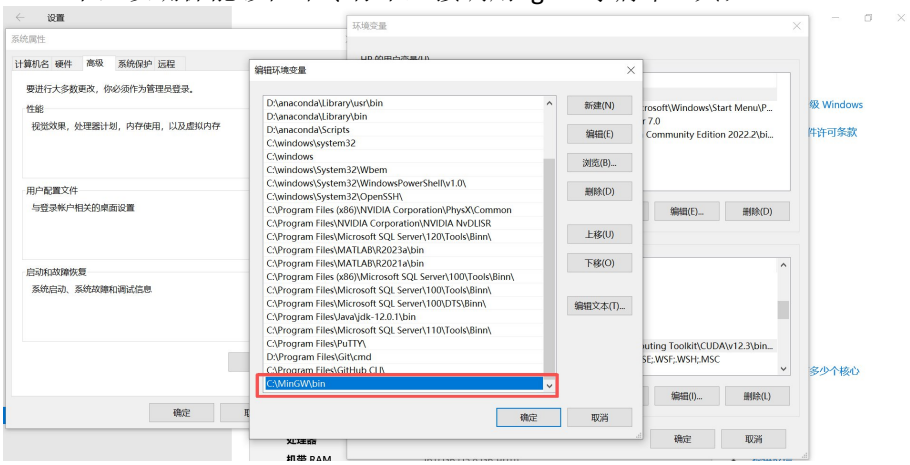
实验步骤：

1. 基于TCP的客户端-服务器程序实现

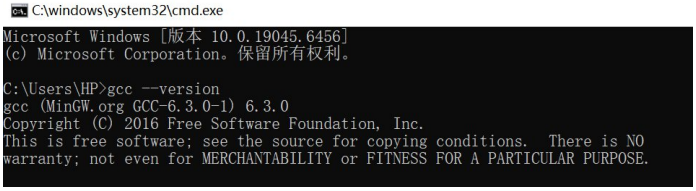
（1）环境配置

此处以Windows系统下使用VSCode运行c语言代码为例（不同的系统、运行软件、编程语言之间会有差异，自行决定选用工具，实现功能即可）。

在 Windows 系统中使用 VSCode 运行 C 语言代码之前，需要先配置 C 语言开发环境。首先，在官网下载并安装 MinGW（下载链接：https://sourceforge.net/projects/mingw/）。安装完成后，需将 MinGW 的 bin 目录添加到系统的环境变量 Path 中，以确保能够在命令行中直接调用 gcc 等编译工具；

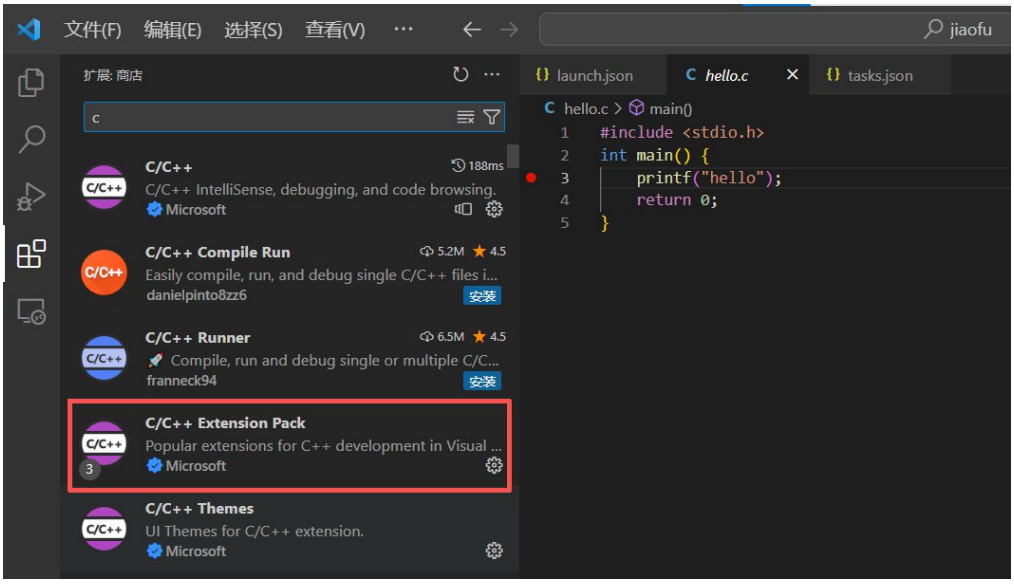


添加完成之后使用Windows+R命令打开cmd窗口，运行命令gcc --version可验证是否成功安装。



成功安装MinGW后，在VSCode中下载扩展包，并选择MinGW中的gdb作为编译器，而后需要配置 tasks.json 和 launch.json 来确保能够通过快捷键编译和调试 C 代码（配置教程可参考：https://www.bilibili.com/video/BV1UC4y127Wr/?vd_sou

rce=c8b8e386001896affe4ff18d6d987034）。



在完成.c文件编写后，必须编译并生成对应的.exe可执行文件，才能启动调试过程。

以上图中的hello.c为例（其余代码的运行调试过程相同，修改相应命令即可），首先在终端运行命令gcc hello.c -o hello.exe生成对应的hello.exe文件，之后在对应的路径下运行./hello.exe即可得到输出。

```
PS E:\jiaofu> gcc hello.c -o hello.exe
PS E:\jiaofu> ./hello.exe
hello
```

（2）代码书写

① 服务端代码

a. 引入相关头文件，链接 ws2_32.lib 库，对服务器监听的端口号和缓冲区大小进行宏定义

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>

#pragma comment(lib, "ws2_32.lib")

#define PORT 8081
#define BUFFER_SIZE 1024
```

b. 定义需要用到的变量

```c
WSADATA wsaData;            // 用于存储 Winsock 库的初始化信息。
SOCKET serverSock, clientSock;  // 定义服务器和客户端的套接字变量。
struct sockaddr_in serverAddr, clientAddr;  // 定义服务器和客户端的地址结构体，存储 IP 和端口信息。
int clientAddrSize = sizeof(clientAddr);  // 用于存储客户端地址的大小。
char buffer[BUFFER_SIZE];    // 定义缓冲区，用于接收客户端发送的消息。
int bytesReceived;           // 用于存储实际接收到的字节数。
```

c. 使用Windows套接字编程，首先需要初始化 Winsock 库。如果初始化失败，打印错误信息，返回 -1 值，结束程序运行

```c
// 初始化 Winsock 库
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    printf("WSAStartup failed\n");
    return -1;
}
```

d. 使用socket()函数创建服务器套接字，如果创建失败，打印错误信息，清理 Winsock 资源并返回 -1 表示程序异常结束。

e. 使用memset将 serverAddr 结构体清零，确保没有残留数据，设置地址族为 IPv4（AF_INET），绑定到所有可用的网络接口（INADDR_ANY 表示接受任意

IP 地址的连接），设置端口号，使用 htons() 函数确保端口号以网络字节序（大端）传输。然后通过bind()绑定套接字，如果绑定失败将打印错误信息，释放资源后结束程序运行。

f. 使用listen()命令开始监听客户端连接，输出服务器正在监听的端口号。

g. 等待客户端连接，输出客户端已连接的信息。

h. 从客户端接收数据，存入 buffer 中，输出接收到的消息，向客户端返回响应，包含接收到数据的长度。在通信完成后，关闭套接字，并清理资源。

② 客户端代码

a. 引入相关头文件，链接 ws2_32.lib 库，定义客户端连接的目标端口号，必须与服务器端口一致。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>  // Windows 网络编程库
#include <windows.h>   // 用于 Sleep 函数

#pragma comment(lib, "ws2_32.lib")  // 链接 Winsock 库

#define PORT 8081
```

b. 定义相关变量。

```
WSADATA wsaData;  // 用于存储 Winsock 库的初始化信息。
int sock = 0;     // 用于存储客户端套接字的变量。
struct sockaddr_in serv_addr;  // 用于存储服务器地址的结构体，包括服务器的 IP 地址和端口。
char *message = "Hello from client!";  // 客户端发送给服务器的消息。
char buffer[1024] = {0};  // 用于存储从服务器接收到的数据。
int valread;  // 用于存储实际接收到的字节数，确保变量声明
```

c. 初始化Winsock库，确保在使用套接字之前初始化 Winsock。

```
// 初始化 Winsock 库
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    printf("WSAStartup failed\n");
    return -1;
}
```

d. 创建客户端套接字。设置地址族为 IPv4（AF_INET），设置目标服务器的端口号，使用 htons() 转换为网络字节序。使用inet_addr() 将字符串格式的 IP 地址转换为 in_addr 结构体中可使用的二进制格式。此处使用的是本地回环地址（localhost），意味着客户端将连接到本地运行的服务器。

e. 连接到服务器。

f. 和服务器进行通讯。

send() 函数将消息从客户端发送到服务器。参数 sock 是套接字，message 是要发送的消息，strlen(message) 是消息的长度，0 表示没有使用附加标志。

recv() 函数从服务器接收数据，将其存储在 buffer 中。valread 存储实际接收到的字节数。如果接收失败，返回值为负数，程序将打印错误信息。否则，打印服务器的响应内容。

完成数据交换后，关闭套接字并清理 Winsock 库。

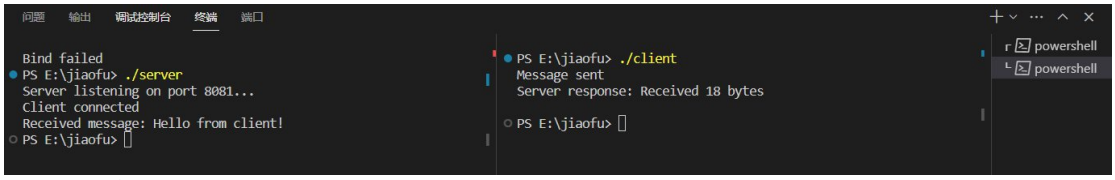
```
// 关闭套接字
closesocket(sock);

// 清理 Winsock 库
WSACleanup();

return 0;
```

（3）实验结果示例

编写好.c文件后，先在终端使用gcc命令生成.exe文件（每次对代码进行改动后都需要重新生成.exe文件）：gcc server.c -o server.exe -lws2_32，其中-lws2_32表示链接Winsock 库；生成并运行 server.exe 和 client.exe 后，可以看到以下实验结果。



（4）总结

本实验通过实现基于 TCP 协议的客户端-服务器通信，帮助理解网络编程的基本原理。要求完成服务器端和客户端的代码编写，并通过实际运行验证通信功能。

2. 基于QUIC的客户端-服务器程序实现

（1）实验环境

本实验需要使用quiche库实现QUIC服务器与客户端，除了要配置好VSCode的C语言环境之外，还需要安装quiche库，确保它支持C语言的绑定。

安装quiche库的步骤如下：

① 安装 Rust 和 Cargo

首先，需要安装Rust开发环境，Rust 包括了 cargo，这是构建和管理 Rust 项目的工具。按照官网提供的下载地址和下载步骤下载Rust和cargo：https://rust-lang.org/zh-CN/tools/install/

② 克隆并构建quiche仓库

a. 使用git工具进行克隆（这一步需要确保自己的主机安装了git工具，如果没有安装，可以通过https://git-scm.com/install/windows下载安装），使用Windows＋R快捷键打开cmd窗口运行克隆命令：

git clone https://github.com/cloudflare/quiche

cd quiche

如果上述方法遇到问题，可以通过访问quiche库的github主页下载并解压其源代码，网址为https://github.com/cloudflare/quiche。

b. 准备好quiche仓库后在cmd窗口运行命令cargo build --release --features ffi构建quiche的C语言接口（注意，这一步需要再下载好的quiche文件夹下完成）运行这条命令没有报错后，在quiche相应的文件夹下查找是否生成以下文件：

Your local quiche Path\target\release\quiche.lib

Your local quiche Path\target\release\quiche.dll.lib

Your local quiche Path\target\release\libquiche.d

Your local quiche Path\target\release\libquiche.rlib

Your local quiche Path\quiche\include\quiche.h

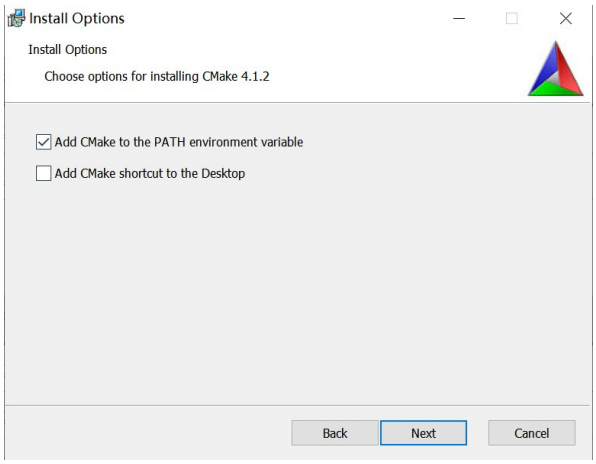
确认存在就说明已经配置好quiche库的C语言静态库。

如果在构建quiche库的C语言接口时出现问题可以尝试检查以下问题：

- 系统缺少 Visual Studio 或 Visual Studio Build Tools。访问https://visualstudio.microsoft.com/zh-hans/visual-cpp-build-tools/下载并安装Microsoft C++生成工具，在安装过程中，确保选择了 C++ 构建工具（包括 Visual C++ 编译器和链接器）。安装完成之后，找到编译工具链link.exe所在地址（一般位于D:\Program Files (x86)\Visual Studio 2022\Build Tools\VC\Tools\MSVC

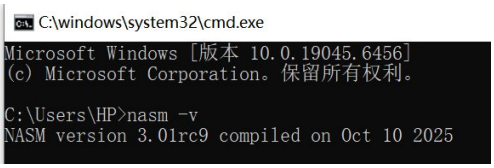
  \14.44.35207\bin），将bin文件地址添加到系统环境变量中。
- 未安装cmake工具。访问Cmake官网（https://cmake.org/download/）下载并安装适用于 Windows 的安装程序，选择 Windows x64 Installer。安

装时，确保勾选了 "Add CMake to the system PATH" 选项，这样可以在命令行中直接使用 cmake。

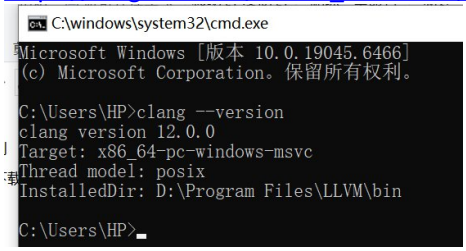


安装完成后可以在终端检查是否成功安装。

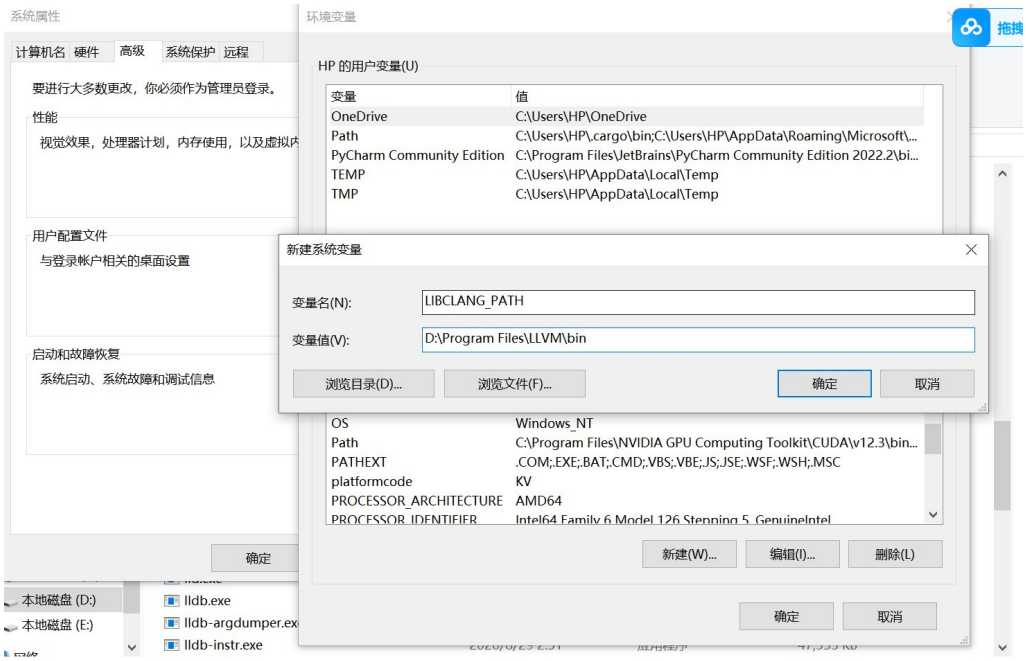
- 未安装NASM编译器。访问官网（https://www.nasm.us/）下载并安装适合你系统的版本。安装后，将 NASM 的路径添加到 PATH 环境变量中。在cmd窗口检验是否安装成功。



- 未安装clang编译器。可以参照这个博客下载并测试安装：https://blog.csdn.net/weixin_44565660/article/details/121758517



Clang安装后，需要设置LIBCLANG_PATH环境变量，指向libclang.dll的位置。该文件通常位于Clang安装目录的bin文件夹下，可以在该目录下找到这个文件。检查该文件确实存在之后打开系统环境变量设置，在系统变量部分点击新建然后添加一个新的环境变量，如下图所示。



之后验证通过运行命令echo %LIBCLANG_PATH%验证环境变量设置正确，正确的话会返回Clang安装目录的路径。

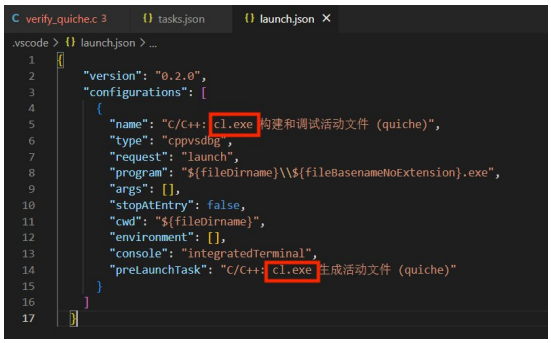
③ 书写C语言代码熟悉quiche库的使用

a. VSCode运行环境

使用MinGW编译工具运行quiche库难度比较大，不推荐新手使用，因此，本课程选用MSVC环境（VS Build Tools）运行C项目。打开“x64 Native Tools Command Prompt for VS 2022”，在窗口中cd到你的C工程目录，然后输入code .启动VSCode。

b. VSCode配置文件

为了链接到quiche库，需要改写相应C工程目录下.vscode的配置文件，通过args命令指定静态库的位置。



c. 最小验证程序

新建verify_quiche.c，先把quiche工具链跑通：

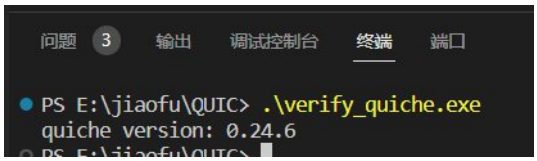
```c
#include <stdio.h>
#include <quiche.h>

int main(void) {
    const char *v = quiche_version();
    printf("quiche version: %s\n", v ? v : "(null)");
    return 0;
}
```

使用cl命令生成相应的.exe文件：

```
cl /nologo /Zi /EHsc /MD `
  /I E:\quiche\quiche-master\quiche\include verify_quiche.c `
  /Fe:verify_quiche.exe `
  /link /LIBPATH:E:\quiche\quiche-master\target\release `
  quiche.lib ws2_32.lib advapi32.lib `
  crypt32.lib userenv.lib ntdll.lib
```

然后在VSCode终端运行该.exe文件，输出quiche版本号就说明quiche的c语言接口构建完成：



（2）代码书写

① 服务端代码

a. 准备quiche配置

b. 创建UDP socket并监听端口：用socket(AF_INET/AF_INET6, SOCK_DGRAM, ...)创建 UDP socket，bind() 到指定端口（比如 5555），设置成 non-blocking，这样循环不会卡死在 recvfrom() 上，此时 server 已经在 “UDP 层”听包了，但还没有 QUIC 连接对象

c. 进行主循环（核心框架）

主循环反复执行：收UDP包。recvfrom()收到一个 QUIC UDP 包，同时拿到客户端地址 peer_addr（用于后续回包）

d. 解析包头：收到 UDP 包后，先用 quiche_header_info() 解析 QUIC 头部，拿到：version：QUIC 版本，dcid / scid：连接 ID（用于识别连接），token：实验里可以不深入理解

e. 第一次收到合法包时，创建 QUIC 连接（accept）

f. 把 UDP 包交给 quiche 处理（conn_recv）
g. 如果连接已建立（established），就读 stream 并生成响应
h. 把 quiche 产生的所有 UDP 包发出去（conn_send 循环）

② 客户端代码

a. 初始化 quiche_config
b. 创建 QUIC 连接 quiche_connect
c. 触发握手
d. recvfrom 后喂给 quiche（quiche_conn_recv）
e. 建立连接后（quiche_conn_is_established(conn)）发送一次消息
f. 调用quiche_conn_readable(conn) 函数获取可读 stream，对每个 stream id 调 quiche_conn_stream_recv，打印响应并设置 got_resp = true
g. 每轮循环 flush_egress，否则ACK / handshake / stream data 都可能“卡在内存里没发出去”，表现为建立慢或收不到响应
h. timeout 处理
i. 收到响应后关闭连接

③ 运行代码

首先需要在终端使用openssl命令生成证书，否则不能编译通过：

```
PS E:\jiaofu\QUIC> openssl req -x509 -newkey rsa:2048 -nodes -keyout cert.key -out cert.crt -days 365 -subj "/CN=localhost"
Generating a RSA private key
................+++++
.......+++++
writing new private key to 'cert.key'
-----
PS E:\jiaofu\QUIC> dir cert.crt,cert.key


    目录: E:\jiaofu\QUIC


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        2025/12/16     21:58           1134 cert.crt
-a----        2025/12/16     21:58           1732 cert.key
```

生成证书之后使用cl命令得到相应的.exe文件，cl命令参考：

```
cl /nologo /Zi /EHsc /MD  `
  /I E:\quiche\quiche-master\quiche\include quic_server.c   `
  /Fe:quic_server.exe   `
  /link /LIBPATH:E:\quiche\quiche-master\target\release  `
  quiche.lib ws2_32.lib advapi32.lib crypt32.lib userenv.lib ntdll.lib
```

（3）实验结果示例

运行生成的.exe文件得到结果如下：

```
PS E:\jiaofu\QUIC> .\quic_server.exe 0.0.0.0 5555 cert.crt cert.
key
[server] listening on 0.0.0.0:5555 (UDP)
[server] new connection created
[server] got 5 bytes on stream 0
[server] connection closed
PS E:\jiaofu\QUIC>
```

```
PS E:\jiaofu\QUIC>
PS E:\jiaofu\QUIC> .\quic_client.exe 127.0.0.1 5555 "hello"
[client] connecting to 127.0.0.1:5555
[client] established, sending: hello
[client] response on stream 0: server received 5 bytes
[client] connection closed
PS E:\jiaofu\QUIC>
```

任务三：对比分析TCP与QUIC性能

任务 3.1：连接建立时间对比

任务要求：

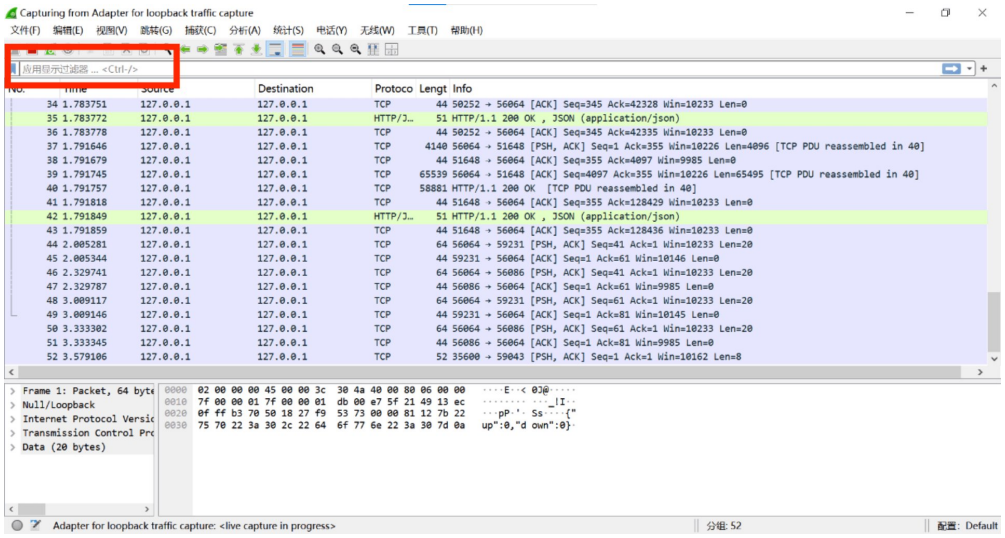
1. 测量 TCP 三次握手时间：使用 Wireshark 捕获 TCP 连接建立过程；记录从客户端发送 SYN 到收到服务器 ACK 的时间。
2. 测量 QUIC 连接建立时间：使用 Wireshark 捕获 QUIC 连接建立过程；记录从客户端发送初始数据包到完成握手的时间。
3. 对比分析：记录 3 次测试的平均值，比较两种协议的连接建立效率

实验环境：

安装好wireshark软件即可

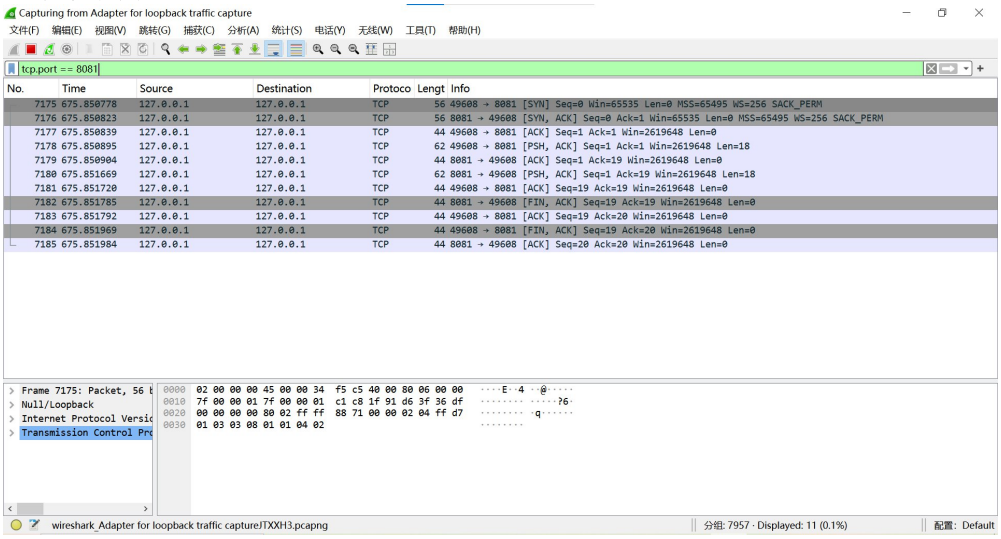
实验步骤：

1. 打开wireshark主界面之后根据书写的代码选择正确的网卡，之后开始捕获；
2. 根据代码端口或者协议类型设置wireshark的显示过滤器；



3. 运行书写的代码模拟通信过程即可得到结果并进行分析。

实验结果示例：



任务 3.2：吞吐量测试

任务要求：

1. 修改 TCP 和 QUIC 程序，实现大文件传输功能（如传输 100MB 的随机文件）
2. 在不同网络条件下测试吞吐量：

   正常网络（无丢包）

   使用 tc 模拟 5% 丢包率：sudo tc qdisc add dev eth0 root netem loss 5%

   使用 tc 模拟 100ms 延迟：sudo tc qdisc add dev eth0 root netem delay 100ms

   计算并对比两种协议的吞吐量（MB/s）

实验环境：

为了在Windows系统上实现第2点，需要准备一个网络故障模拟工具clumsy,当然，也可以在Windows系统下安装WSL2或者Linux虚拟机使用sudo tc命令运行相同的代码并测试吞吐量。

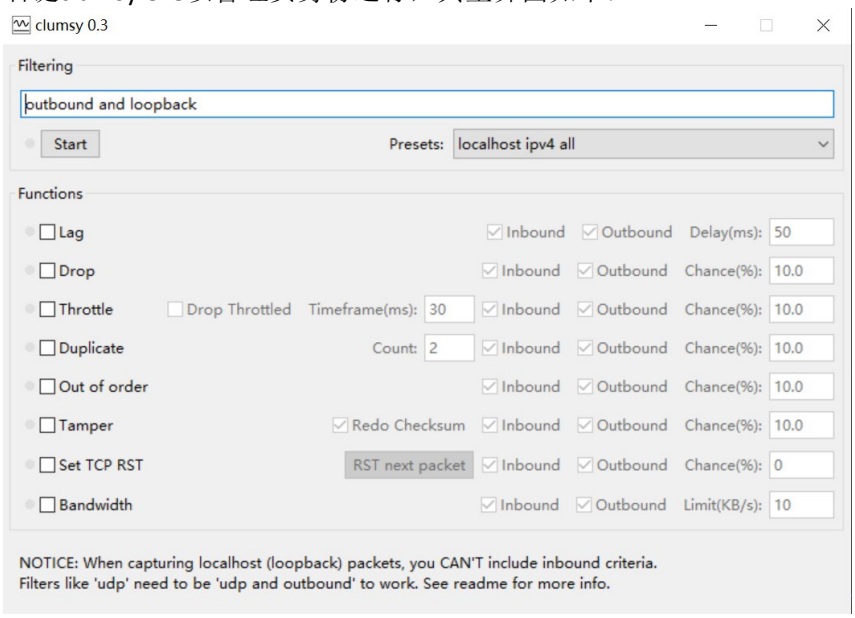
本手册以clumsy为例进行教学，使用WSL或虚拟机工具请自行学习。

首先，打开clumsy的下载网址（http://jagt.github.io/clumsy/download）根据系统

版本下载对应的软件压缩包。解压后得到一个文件夹：

名称	修改日期	类型	大小
clumsy.exe	2022/1/12 21:37	应用程序	1,399 KB
config.txt	2021/4/24 23:24	文本文档	2 KB
License.txt	2023/10/21 22:21	文本文档	2 KB
WinDivert.dll	2019/10/20 16:26	应用程序扩展	46 KB
WinDivert64.sys	2019/10/20 16:26	系统文件	89 KB

右键clumsy.exe以管理员身份运行，其主界面如下：



有疑问可以参考这篇博客：

https://blog.csdn.net/hgftgfffg/article/details/147412888

实验步骤：

1. 实现大文件传输功能

（1） TCP程序

① 按照提示补全TCP程序即可

② 编译与运行

gcc -O2 tcp_server.c -o tcp_server.exe -lws2_32

gcc -O2 tcp_client.c -o tcp_client.exe -lws2_32

（2） QUIC程序

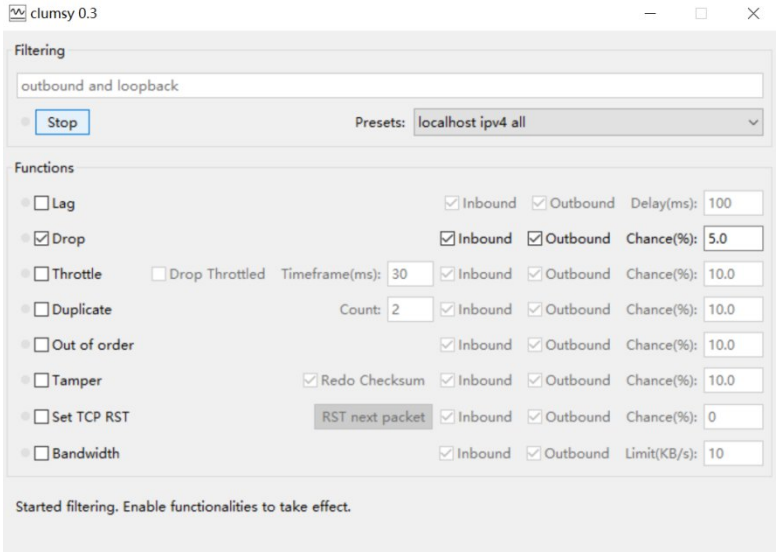
① 按照提示补全QUIC程序即可

② 编译与运行

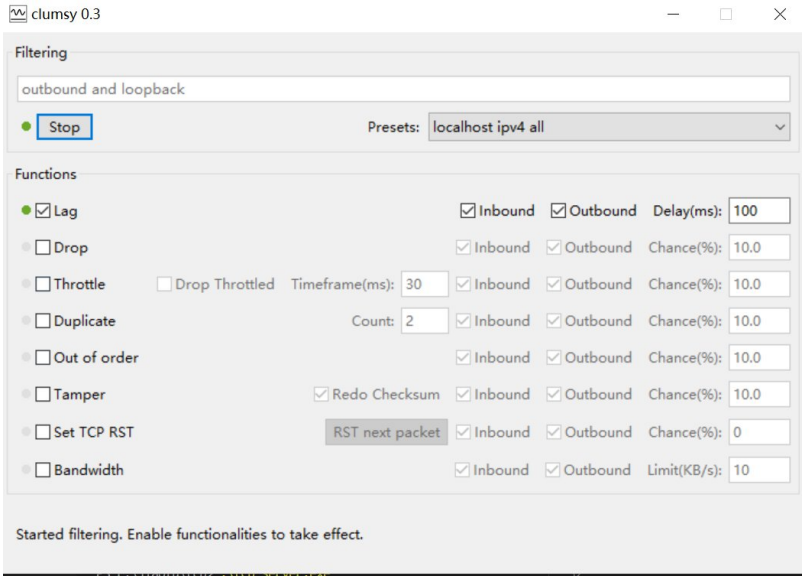
依旧参照之前的格式使用cl命令进行编译，得到可运行文件后运行观察结果。

2. 在不同网络条件下测试吞吐量

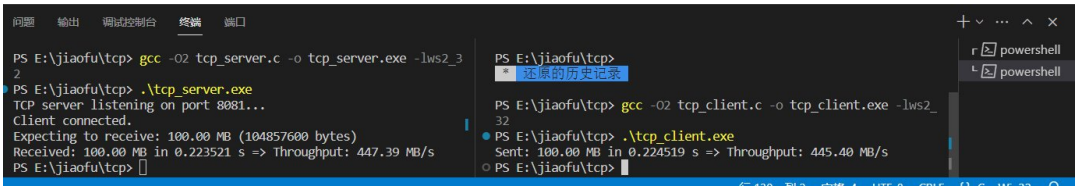
使用clumsy模拟 5% 丢包率：首先以管理员身份打开clumsy.exe，选择Lag=100ms，点击开始，然后再次运行程序，观察实验结果。



使用clumsy模拟 100ms 延迟：首先以管理员身份打开clumsy.exe，选择Lag=100ms，点击开始，然后再次运行程序，观察实验结果。



3. 实验结果示例：



此处仅提供了正常网络环境下tcp传输大文件的结果示例，并且是以本地loopback进行实验作为例子，建议在自己做实验的时候可以改用本级IP尝试实现实验，这样结果会更加接近真实网络，且clumsy对真实网卡路径更加稳定。

任务 3.3：多路复用性能测试

任务要求：

设计多流传输测试：同时建立 5 个 TCP 连接传输数据，在单个 QUIC 连接上建立 5 个流传输数据，测量并对比两种方式的总传输时间和资源占用，分析 QUIC 多路复用如何解决 TCP 的队头阻塞问题。

任务 3.4：网络异常恢复测试

任务要求：

模拟网络中断后恢复的场景：

建立连接并开始传输数据

使用tc qdisc add dev eth0 root netem loss 100%模拟网络中断

30 秒后使用tc qdisc del dev eth0 root恢复网络

对比两种协议的恢复能力和数据完整性

测试 QUIC 的连接迁移能力（服务器 IP 或端口变化后）