QUIC

Networked Systems (H) 2024-2025 – Laboratory Exercise 4
Dr Colin Perkins, School of Computing Science, University of Glasgow

1 Introduction

The laboratory exercises for Networked Systems (H) will introduce you to network programming in C using the Berkeley Sockets API, and help you understand the operation and structure of the network. Other exercises will illustrate key points in the operation of the network. The laboratory exercises are intended to complement the material covered in the lectures. Some expand on the lectures to give you broader experience in a particular subject. Others exercises cover material, such as network programming in C, that's better taught by doing than by lecturing.

This lab builds on the material discussed last week in Lecture 4 and explores the use of the QUIC Transport Protocol. **This is a formative exercise, and is not assessed.**

2 Formative Exercise 4: QUIC

QUIC is a new, general-purpose, network transport protocol. It operates in a client-server manner and provides a reliable, secure, multi-stream transport service to applications. QUIC can serve as an effective replacement for TCP and TLS in many cases, while providing lower connection establishment latency, improved security, multi-streaming, and a number of other new features.

Unlike TCP, which is implemented as part of the operating system kernel, the QUIC transport protocol runs in user space, using UDP as a substrate, and is typically implemented as a library that can be distributed with applications. The `aioquic` library (`https://github.com/aiortc/aioquic`) is an implementation of the QUIC transport protocol as a Python library.

You should write two Python programs that use the `aioquic` library:

`hello_quic_server.py` The server should listen for incoming QUIC connections on UDP port 5000. It should accept the first connection made, receive all the data it can from that connection, print that data to the screen, close the connection, and exit. If UDP port 5000 is in use on your systems, use another port number instead; the choice of port is not critical for this exercise.

`hello_quic_client.py` Your client should connect to the server on the port you have chosen, send the text "Hello, QUIC world!" on a QUIC stream, and then close the connection. The client should take the name of the host on which the server is running as its single command line argument. For example, if your server is running on host `stlinux03.dcs.gla.ac.uk`, you would run the client using the command:

```
python hello_quic_client.py stlinux03.dcs.gla.ac.uk
```

The client and server should ideally be run on two different machines. You can do this by running two instances of `ssh`, to connect to two of the student Linux servers (`stlinux02.dcs.gla.ac.uk` to `stlinux08.dcs.gla.ac.uk`) provided by the School, or you can use any other two machines that you have access to and between which communication is permitted (note that the University network has a firewall that blocks incoming UDP traffic, so you cannot run the client on your home machine and the server on one of the student Linux servers).

If you only have access to one machine, you can run client and server on the same machine. In this case, when running the client, use `localhost` as the name of the machine running the server (the `localhost` is an alias for "this machine").

Run your client and server, and demonstrate that you can send the text "Hello, QUIC world!" from one to the other. If possible, try this with client and server running on the same machine, and with them running on two different machines.

Once this is working, modify your client to send a much longer message and check that works too. When you have large messages working, modify the client and server so the client can send multiple messages, each on a separate QUIC stream, within the connection.

Finally, modify your client and server so that the server can send a message back to the client through the open connection. The client should send a message to the server, wait for, and display the message sent back by the server. The contents of the message returned by the server are unimportant.

3 Discussion

This is a formative exercise. **It is not assessed, and you do not need to submit your code.** The goals are (i) to give practise in network programming using protocols other than TCP and using interfaces other than the Berkeley Sockets API; and (ii) to complement the material covered in Lecture 4. Discuss with the lecturer or lab demonstrators when you have questions.

The `aioquic` library builds on the Python asynchronous I/O framework. If you're not familiar with this framework, the tutorial at `https://realpython.com/async-io-python/` may be helpful.

The `aioquic` library source code, available from GitHub, includes a number of examples. The DNS-over-QUIC examples (`doq_client.py` and `doq_server.py`) are perhaps the easiest to follow. It is suggested that you first get these working, then modify them to send the "Hello, world" text rather than DNS requests. For the client, this involves replacing the `DnsClientProtocol` class with a new class, perhaps called `HelloClientProtocol` that replaces the `query()` method with a `send_hello()` method.

QUIC is a secure transport protocol and cannot be used without TLS. This means your server needs a TLS certificate to prove its identity and your client needs a Certificate Authority (CA) certificate to validate that. In a real implementation you might use the CA certificates from your operating system or from the Python `certifi` package, and you would obtain the TLS certificate from a trusted CA such as Let's Encrypt (`https://letsencrypt.org`). For the purposes of this lab exercise, you may find it easier to use the `ssl_cert.pem`, `ssl_key.pem`, and `pycapert.pem` files from the `aioquic` testing directory instead (these are well-known TLS certificates used for testing purposes that provide no security and *must not* be used in real applications).

This is a difficult exercise that requires mastering several new skills: understanding QUIC, asynchronous programming in Python, the use of TLS certificates, and the (badly documented) `aioquic` library. Do not be concerned if you cannot get a solution to this lab working, and do not spend too much time on this exercise – it is not assessed.