Università degli Studi di Napoli "Federico II"
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in
Ingegneria Informatica

Web and Real Time Communication Systems

# Pong-over-QUIC

Studente
Mauro Giliberti

Professore
Simon Pietro Romano

Anno Accademico 2024-2025

# 1 Abstract

The core of this project is in the network communication: a custom application layer protocol based on QUIC, the cutting-edge in transport layer protocols nowadays. The protocol is tailored to the famous pong game. The gameplay is simple, players can **create a match** or **join an existing one** making requests to a matchmaking server. When the match is ready, a **peer to peer** connection is established between the two players. The game part of the project is implemented in the iOS environment using Network and SpriteKit frameworks, while the matchmaking server is based on Java Servlets and Apache Tomcat.

# 2 QUIC

In order to be able to set up a QUIC connection between two devices is necessary that one acts like a server and the other one like a client. The first one has to offer the possibility to establish a connection, that in the iOS environment is accomplished using a **NWListener**, basically opening a local port and listening on it. One important step, maybe the most important for QUIC, is to provide a certificate to complete the handshake and build an encrypted connection without the need of TLS over TCP.

The client device has to create a connection to the right IP address and port, using **NWConnection**, and after the handshake is possible to use the `send` and `receive` methods.

The QUIC connection is customizable: is possibile to set the direction (unidirectional or bidirectional), the idle timeout, the messages characteristics like maximum and minimum length, and obviously the security properties.

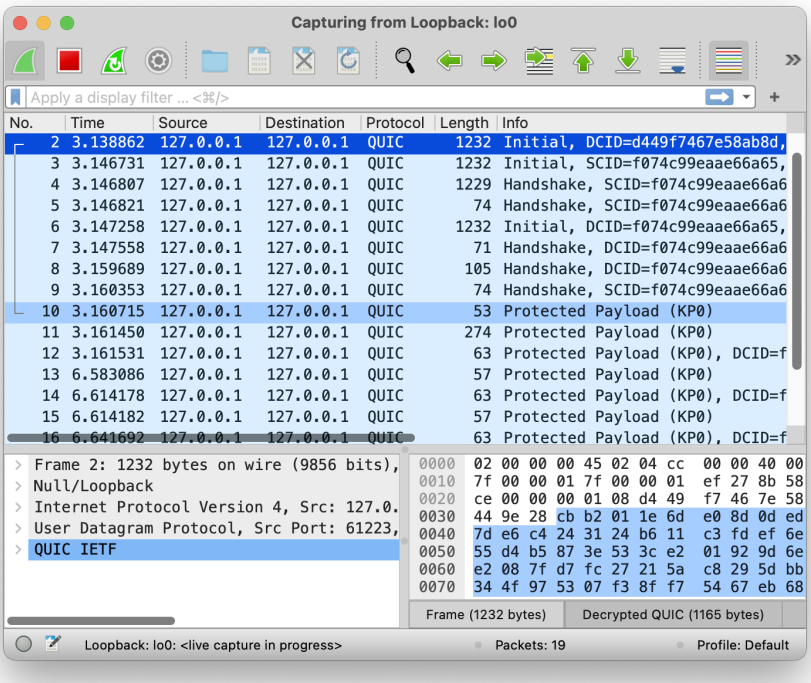
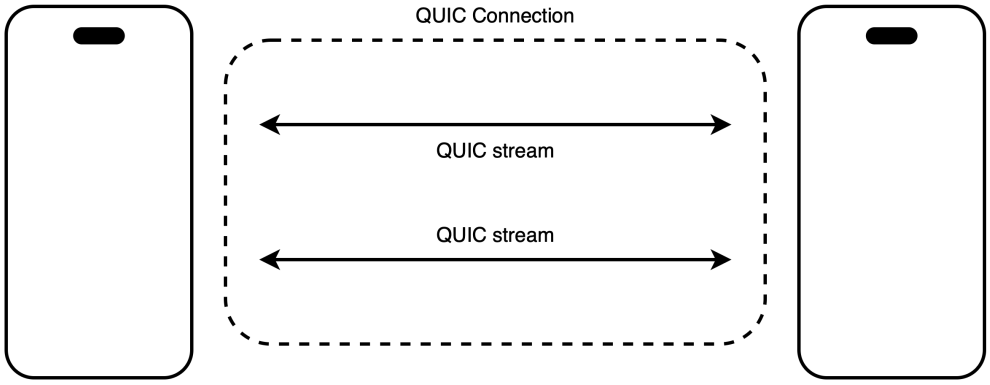



Figure 1: Wireshark capture of QUIC handshake

## 2.1 QUIC Connection

The game requires a communication layer which allows to send and receive data simultaneously and continuously from the devices. In a TCP-based set-up this would mean to establish two distinct connections, and similar would be for UDP.

With QUIC things are optimized, once the connection is established is possible to open one or more streams, unidirectional or bidirectional, to transmit data.




This can be made in iOS using the **NWMultiplexGroup** and **NWConnectionGroup** objects, useful when a multiple connections communication is needed. The group has to be initialized with the endpoint to connect to and the parameters of the connection, that describe the characteristics of the communication.

# 3 Application Layer Protocol

The application protocol is really simple, the header has just two fields: message type and message length, both 8 bits unsigned integers. Messages can be of three types: *movement*, *name* and *invalid*. The first one is a message where the payload is a signed number that represents the position of the pong bar relative to the center of the screen. The second permits to send the player name in order to display it, and the last one is used in case errors occur when messages are built. The custom protocol has been implemented in iOS using the **NWProtocolFramer** class.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               +
|                            Payload                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

## 3.1 Protocol Stack

All data transmission in the app use this protocol stack 2. The game controller reads the position of the player bar as a *Double type* value, sends it to the network manager instance that encapsulate it in a message then sent over a QUIC connection.

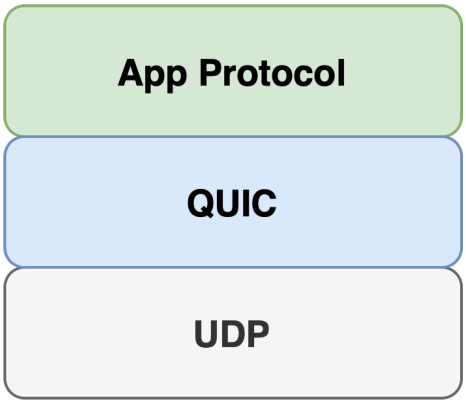



Figure 2: Protocol Stack

Unfortunately there is a known bug in the NWConnectionGroup implementation that does't allow to attach an application protocol over a transport

protocol for all the group, which in a QUIC based environment means that is not possible to use a application protocol as default for all the streams. Indeed is necessary to declare it for every stream.

The consequence of this bug is that is not possible for the *server peer* to create a stream that has on top an application layer protocol, because the connection group on the client side would not match the full protocol stack. For this reason only the *client peer* can create streams, in particular two bidirectional streams, which then will be assigned an *owner*, namely who will send data on that stream, client or server.

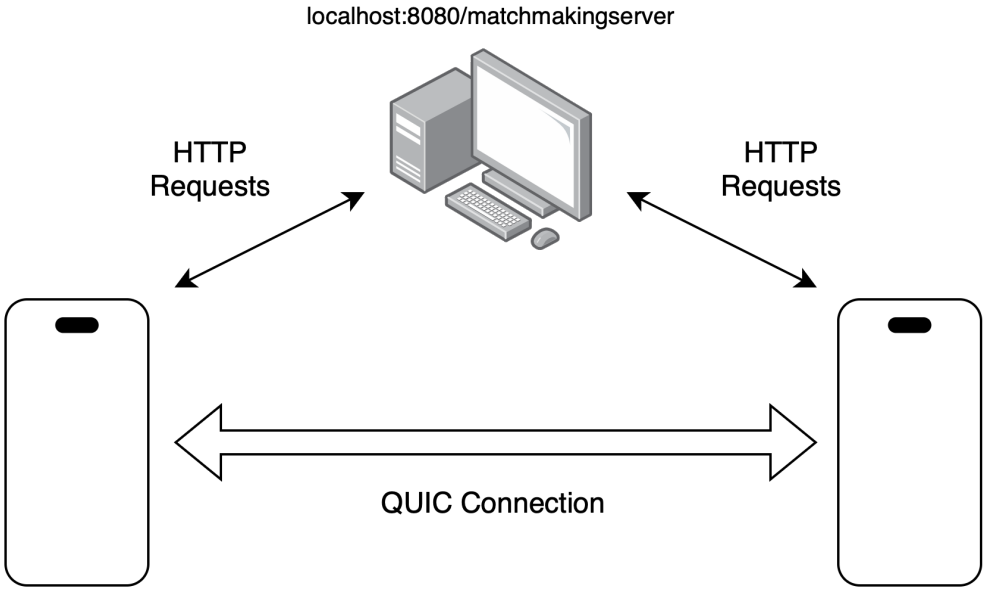
# 4 Matchmaking Server

A matchmaking server is needed to establish the communication between two players. It exposes three services:

- **createMatch**: allow a player to host a match choosing a unique identifier for it
- **getAllMatches**: allow a player to retrieve all the open matches available
- **chooseMatch**: allow a player to choose which match to join and so make it unavailable for others

When a match is created the player, that in this case will act as the server peer, has to provide the IP address and port to which the opponent will have to connect, as well as the match identifier. If a player wanted to know if is possible to join to an existing match the second service allows to get a list of open matches. The last service is used to confirm the attendance to a specific match, done providing to the server the identifier of that.

Only after the matchmaking phase the two devices have the informations required to create a peer to peer connection.

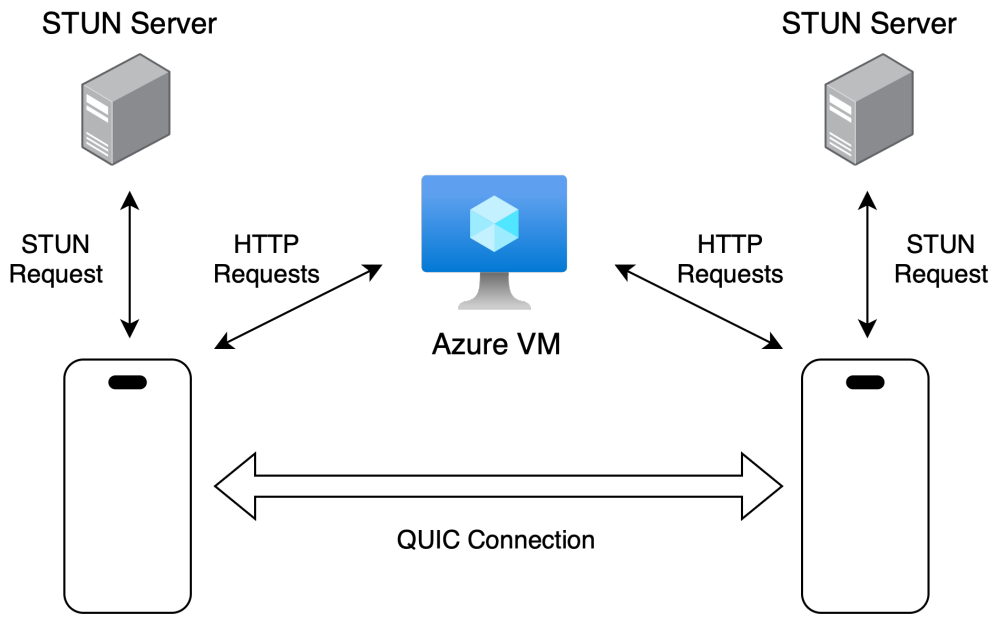


# 5 NAT Traversal

After implementing the peer to peer communication, the matchmaking server and the game, the natural next step is to make all of these available on the internet.

The very first thing to do (and maybe the easiest), is to expose the matchmaking server online, and an Azure virtual machine has been used to accomplish that. In this way the matchmaking is possibile anywhere a internet connection is available.

Though is not possible to do the same for the QUIC connection. The peers are, in most of the cases, not directly exposed online but are located behind a Network Address Translation, making necessary a different architecture.




Making a STUN request a peer can be aware of its *server reflexive* IP address, which is the way it is identified in the public internet, and then can provide this address to the other peer (through the matchmaking server) in order to establish a peer to peer connection.

Actually this is still not enough.

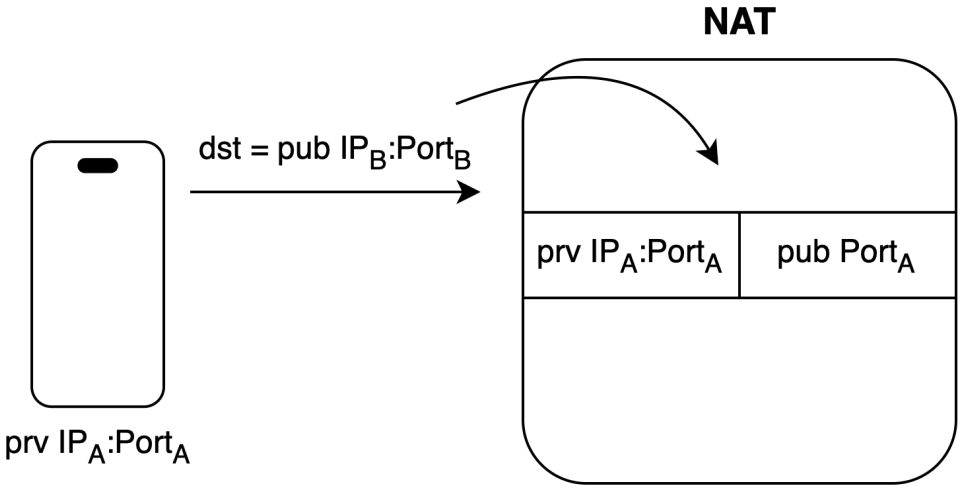
## 5.1 Hole Punching

A NAT translates the private IP address and port of a device to a public IP address and port (usually only the port varies between internal devices, it depends on how many interfaces the NAT has towards the internet).

This means that if the *server peer* is listening on a specific port and has a specific *server reflexive* address, the client peer can't just send a packet to this public address, due the lack of a NAT rule that maps the public address to the private one.

In a server-client architecture, this is not a problem, because is possibile to set a static rule on the NAT that maps a public port to a specific private IP address and port of the server, static too. This is called *port forwarding*.

In a peer to peer architecture instead, peers IP address and port are dynamically assigned so a static rule is not effective. To overcome this situation, a technique called *hole punching* is used. Basically the peers send packets to each other, using the server-reflexive addresses as destination, in order to create a NAT rule that maps the public addresses and port to the private ones.




This method is not always effective: if the NAT is symmetric, is not possible to take advantage of the *server reflexive* address. Actually is possible to send a STUN request but that IP:Port couple from the STUN response will be useless given that the NAT will assign a different port on the next connection. In that case a TURN server is needed, which relays the packets between the peers, acting like a broker, obviously public exposed.

Unfortunately, this method is not implementable, or better testable, in a

virtualized iOS environment, due to the fact that macOS will behave like a symmetric NAT for the virtual machines.