UNIVERSITY OF PADOVA

Department of Department of Information Engineering

Master Thesis in ICT For Internet and Multimedia

Performance Analysis of QUIC Protocol Used for Real-World Automated Meter Reading Application

Supervisor
Prof. Ombretta Gaggi
University of Padova

Co-supervisor
Engr. Massimo Cesaro
Inkwell Data Limited

Master Candidate
Bogdan Milovanovic

Student ID
2004970

Academic Year
2021-2022

"I suppose therefore that all things I see are illusions; I believe that nothing has ever existed of everything my lying memory tells me. I think I have no senses. I believe that body, shape, extension, motion, location are functions. What is there then that can be taken as true? Perhaps only this one thing, that nothing at all is certain."
— Rene Descartes

Acknowledgments

I would like to express my sincere gratitude to my supervisor Prof. Ombretta Gaggi for her invaluable patience, valuable guidance, and precious feedback. I would like to extend my sincere thanks to Eng. Massimo Cesaro for sharing his wide and valuable experience and giving a new perspective to this work from an engineering point of view.

Finally, I would like to thank my family who has always supported and encouraged me to step into the unknown bravely and without hesitation. And of course, without friends by my side, all of this would be impossible.

Abstract

The QUIC is a new encrypted transport protocol, designed on the basis of decades of transport and security experience, with the aim to find a valid, more performing substitute to TCP protocol in all its internet implications. Originally designed by Google, QUIC has taken all the best qualities of TCP connections and TLS encryption and implemented them over UDP. The goal of this thesis is to provide an in-depth study of implementing QUIC protocol for automated meter reading applications which enables automatic collection of consumption data, eliminates manual meter reading, improves efficiency, and saves costs. We conduct a comprehensive measurement study about the performance of QUIC when compared to TCP in Android devices measuring latency for various object sizes, in different network conditions, and for both up-link and down-link traffic. The benchmark tests show that the QUIC protocol should be taken into consideration for future AMR applications. Results acquired proved better Request Completion Time performance both for Wi-Fi and Mobile Network connection, but also for all packet sizes tested. It can be noticed that QUIC significantly outperformed TCP protocols in bandwidth-limited scenarios and smaller packets exchange. Moreover, the results obtained also give us new ideas for future work and improvements. In this regard, we consider this work as a good starting point for the further investigation and adoption of QUIC as an important player protocol in the IoT world and especially in Android devices for smart water metering tasks.

Contents

Listing of figures

Listing of tables

Listing of acronyms

TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
OS	Operating system
CDN	Content Delivery Network
SSL	Secure Sockets Layer
TLS	Transport Layer Security
ISP	Internet Service Provider
IETF	Internet Engineering Task Force
IoT	Internet of Things
AMR	Automated Meter Reading
AMI	Advanced Metering Infrastructure
AEAD	Authenticated Encryption with Associated Data
RFC	Request for Comments
UDP	User Datagram Protocol
OSI	Open Systems Interconnection
VM	Virtual Machine
CPU	Central Processing Unit
RCT	Request Completion Time

1 Introduction

1.1 Motivation

The world today has come close like never before. The development of communication technologies, especially the Internet, has had a profound impact on communication. Ever since its creation, the Internet has experienced an impressive rise both in the number of users and the volume of data traffic. In order to support the rapidly increasing mobile and desktop traffic over the Internet, not only that network infrastructure must be upgraded, but also existing communication protocols should be refined, or new protocols developed.

Nowadays, TCP is the most widely adopted transport protocol that provides a reliable message delivery, dealing with the complexities of network congestion and link-layer losses. The widespread use of TCP for HTTPS traffic is mainly attributed to the fact that almost every operating system includes TCP, and its availability on a wide range of infrastructures, such as load balancers, HTTPS proxies, content delivery networks, etc. Moreover, with the rising need for secure communication, TCP along with SSL/TLS makes communication secure but at the cost of additional delay.

In the last few years, the communication bandwidth has tremendously increased which has also changed the communication pattern over the Internet. The web is now being used as a platform for many latency-sensitive applications, and in that sense, using TLS over TCP protocol brings a lot of challenges since TCP has inherent issues like handshake delay, head-of-line blocking delay, difficulties in implementation of changes at the protocol level since TCP is part of kernel space, and problems in the deployment of new changes in the network due to middleboxes.

Since today the complexity of the Internet is so high, significant updates on the TCP/IP stack at the base of the Internet's core would require a huge effort to reach an agreement between ISPs that change is needed, as well as to accomplish coordination and interoperation between them. The result is a technological standoff lasting almost two decades and causing the necessity of finding new ways and workarounds to use the Internet for every modern purpose.

To address these challenges, the Internet research community has developed an intense interest in the design and implementation of novel transport protocols. In 2013, Google introduced QUIC (Quick UDP Internet Connections), a new encrypted transport layer protocol recently standardized by the IETF via the RFC 8999 [1], 9000 [2], 9001 [3], and 9002 [4] published in May 2021. The aim of this protocol is to find a valid, more performing substitute for TCP protocol in all its Internet implications finding then a new base for modern HTTP2 and the upcoming HTTP3 protocol versions.

1.2 Thesis structure

Chapter 2

The purpose of this chapter is to give a theoretical background and an overview of related work, in order to understand the thesis work exposed in the subsequent chapters. First of all, it deals with IoT technologies for smart water metering and explains smart meter design, giving the context of the field that this thesis is applied for, then it explains the slow evolution of existing Internet communication protocols and describes the need for a new protocol. Subsequently, it provides a description of the QUIC protocol, highlighting its salient points and advantages. Finally, the network performance measurements used as key points to perceive the quality of a network are explained, as well as the latency spin bit which allows end-to-end RTT measurements of a QUIC connection.

Chapter 3

Chapter 3 provides more details on the methodology as well as the architecture and technologies used to implement the client-side Android mobile application as well as the server-side Elixir application. Furthermore, it describes the challenges we faced during the implementation phase, and how some of them were solved or alternatives found.

Chapter 4

This chapter outlines the testing environment used to perform a practical analysis of the functioning of individual protocols. In this respect, it describes a testbed, network environment, experimental scenarios as well as metrics used. Additionally, it gives an overview of the results obtained and their analysis for all test scenarios conducted.

Chapter 5

This chapter aims to do an overall conclusion of the work done and some final observations. It also highlights the ideas for future work and adoption of QUIC in smart water metering systems and AMR Android mobile applications.

2 Background

2.1 IoT Technologies for Smart Water Metering

Internet of Things (IoT) is a network of devices, data, and other technological features to run any operation more intuitively with the minimal manual intrusion. IoT has gradually spread across everyday life and a commercial landscape, enabling various applications like smart cities, smart homes, industrial internet, wearable devices, etc. The fourth industrial revolution (Industry 4.0) concept is hugely based on IoT's intuitive adaptation and intelligent automation in every industrial activity. Moreover, in the last few years, IoT has entered water metering management, which solutions are now able to collect, analyze and distribute data in a way that no technology has done before.

2.1.1 Problems with Manual Meter Reading

While utility companies understand the power of tracking water usage metrics, it has not always been easy to collect this data from smart water meters quickly and be scalable. Even with recent advancements, utility companies still employ Meter Readers, to physically go and collect meter data from individual properties. In addition to being extremely tedious and costly, there are other problems that can occur:

- Meter Readers are often unable to access the water meter due to weather conditions or the meter's inaccessible location.
- This high cost per reading means companies typically check meters only once a month. This limited access to data drastically limits the power of insights such as usage trends and identifying potential outages.
- Any time humans are introduced into a process, it means a higher possibility of human error such as rough estimations on a meter level that can lead to inaccurately charging customers.

2.1.2 Smart Water Metering

Smart water metering enables automatic collection of consumption data, eliminates manual meter reading, improves efficiency, and saves costs. It also provides an opportunity to detect leaks and abnormal consumption more efficiently than manual methods.

A typical smart system relies on electronic sensors and bidirectional communication networks to remotely read, store, and transfer data for analysis and feedback. The transmitter attached to the water meter uploads the consumption data to the processing server for analysis, billing, and other processes. Usually, the automated meter reading and transmission frequency are flexible and can be daily, hourly, real-time, etc.

As more utility companies transition from the conventional manual meter reading practices, they are increasingly adopting Automated Meter Reading (AMR) and Advanced Metering Infrastructure (AMI) to automate the reading and billing processes.

2.1.3 Automated Meter Reading and Advanced Metering Infrastructure

A smart water metering system relies on several technologies to automate the collection and analysis of meter data. A typical AMR system, as shown in Figure 2.1, comprises a water meter with a data logger to capture the information, a communications technology to transmit the captured data, and a server to process the information.

Upgrading the communication water meter is usually done by adding a radio frequency transmitter that collects the consumption data with a sensor attached to the meter and transmits the reading to a collection system, which can be a mobile application that has an antenna connected as a receiver.

The AMR transmitters continuously broadcast the consumption data that are acquired by the receiver. Since the AMR transmitters have limited power, a receiver needs to be in the transmitters' vicinity (up to 100 meters), with the interference mostly harmless. The readings are collected by a meter reader using a handheld or vehicle-based radio device, or by a fixed network system.

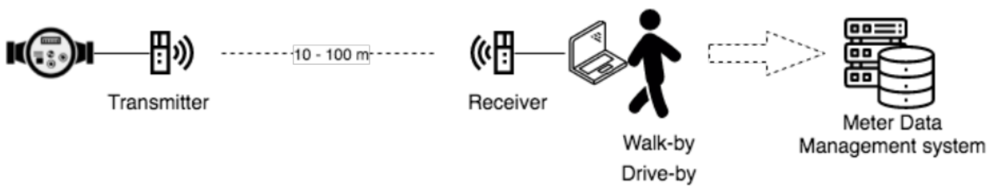


Figure 2.1: Automated Meter Reading

While AMR provides periodic water consumption, AMI (illustrated in the following Figure 2.2) collects customer water usage data in real-time. It represents an integrated system of water meters, communication networks, and data management systems that enable two-way communication between meter endpoints and utilities. Unlike AMR, AMI does not require utility personnel to collect the data. Instead, the system automatically transmits the data directly to the utility at predetermined intervals.

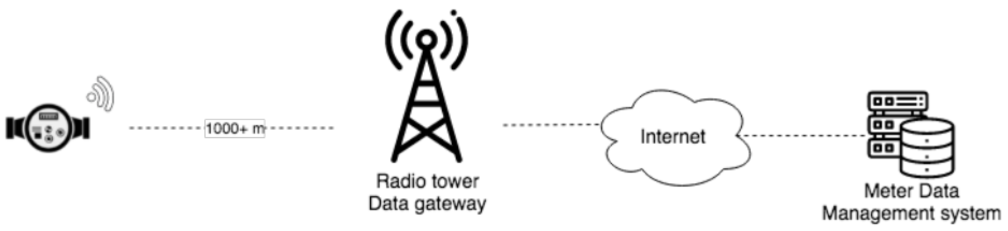


Figure 2.2: Advanced Metering Infrastructure

AMI solutions are preferred to AMR for large roll-outs, as they typically prove more cost-competitive than walk-by/drive-by meter reading, offering almost the ultimate in features and benefits.

2.1.4 Smart Meter Design

It has been estimated that about four to six percent of the gross national income in industrial countries is accounted for by measuring instruments. This also includes smart water meters. Regarding that, manipulations of measuring instruments' software could have far-reaching financial consequences. Clearly, special measures should be considered to secure such instruments.

Today, the measuring instruments are separated into legally relevant and legally non-relevant parts, as shown in Figure 2.3.

Only the legally relevant application code of the measuring instrument firmware is subject to legal control. After the relevant application has been approved the manufacturer cannot modify it without re-approval. The legally relevant application code ensures that billing quantities are measured, post-processed, displayed, printed, and transformed into encrypted data packets. This application also maintains billing information, log files, and load profiles in non-volatile memory. Certain information must be stored at predefined times so the operation of the Real-Time clock module is controlled by the legally relevant application.

The second part represents a legally non-relevant layer, and it performs all remaining software tasks. In contrast to legally relevant code, manufacturers can modify the application without re-approval, gaining flexibility and significant cost savings. It represents a layer above containing its own application logic, storage, and the capability of measuring instruments to share informative data using various protocols and formats as well as an over-the-air software updating.

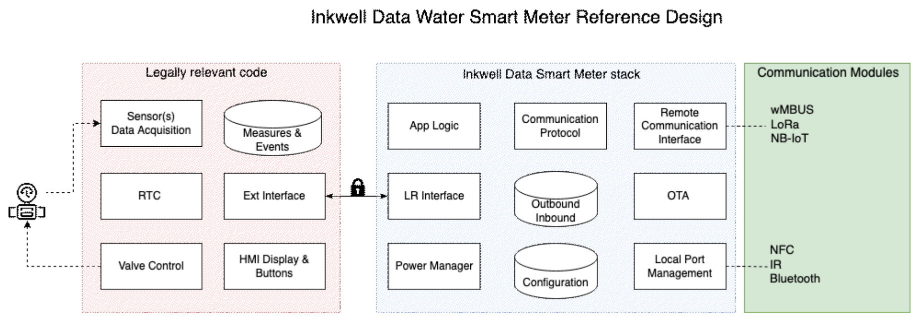


Figure 2.3: Smart Meter Design

2.2 Slow Evolution of Internet Communication Protocols

Ever since its creation, the Internet has experienced an impressive rise both in the number of users and the volume of data traffic. According to a report issued by Cisco, this trend is expected to continue in the following years. The report foresees that nearly two-thirds of the global population will have Internet access by 2023. There will be 5.3 billion total Internet users by 2023, up from 3.9 billion in 2018. Additionally, the number of devices connected to IP networks will be more than three times the global population by 2023.

Even though in the last decades the Internet has changed and grown so quite outstandingly, the same cannot be said for its core protocols. Since 1993, looking at the TCP/IP protocol suite, no significant changes have been seen in the Internet architecture. This does not mean that nothing has been done to adapt the protocols to today's requirements but, for many reasons, it has been preferred to bypass the known problems by introducing a series of minor changes rather than modifying the protocols at their core.

The main motivation behind this evolutionary stagnation is to be attributed to the ever-increasing economic and political interests which characterize today's network. In fact, in a commercial network, new technologies are usually introduced when a remarkable economic return can be made by ISPs. Since the Internet infrastructure is disseminated through middleboxes, i.e., devices that manage the known TCP version at a hardware level. It means that changing the TCP packet's format would mean changing the physical hardware stack of almost all these devices with a giant economic effort and many difficulties in coordinating [5].

For these reasons, almost all improvements and new features made for the Internet protocols have been introduced as optional extensions of the already present core functionalities. Consequently, their adoption, except for rare cases, has never been such as to make them extensively deployed in the whole network.

To address this challenge, the Internet research community has developed an intense interest in the design and implementation of novel transport protocols. In 2013, Google introduced QUIC (Quick UDP Internet Connections), a novel transport protocol proposed that aims at replacing TCP in the near future.

The next section gives a small overview of the main critical issues at the base of such a decision.

2.3 Downsides of Existing Protocols and the Need for a New Approach

TCP has proved as a very reliable transport protocol for HTTP. TLS has fulfilled the requirement of secure transportation of data. TCP along with TLS is becoming the essential component for data transfer securely and reliably. These protocols have undergone multiple changes to cope with the increasing demand for latency-sensitive applications and security. Despite its widespread usage, TCP faces some limitations, discussed below, which became the way for the evolution of QUIC.

First, TCP headers are not encrypted or authenticated, anyone with network access can manipulate them, with good or harmful intentions. Secondly, the space inside the header for current and future options is limited to 40 bytes, more than half of which are almost always used; there is, therefore, little scope for introducing new additional features. Coupled with TLS, handshake time is not satisfactory, especially over long distances where a 3-way handshake increases the connection establishment latency significantly, especially in the latency-sensitive application where it affects the user experience.

From the HTTP/2 version, IETF delivered the ability to multiplex different HTTP requests within the same TCP connection, allowing smarter and more efficient management of the network's bandwidth. Moreover, multiplexing would theoretically allow a single TCP connection can be used for multiple streams multiplexed together to transfer data, but when one of those streams suffers a packet loss, the whole connection (and all its streams) experience latency, until TCP re-transmits the lost packet. All the packets, even if they are already transmitted and waiting in the buffer of the destination node, are being blocked until the lost packet is re-transmitted. This problem is known as head-of-line blocking, illustrated by Figure 2.4.

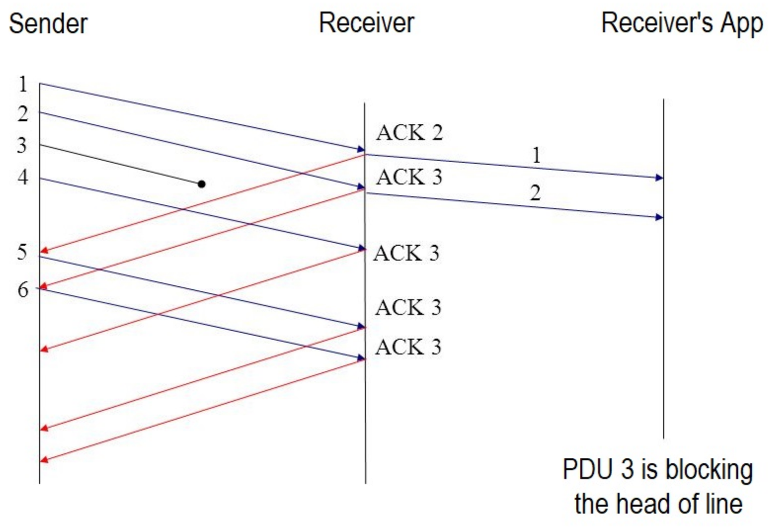


Figure 2.4: Head-Of-Line Problem

All the above factors led to the development of QUIC transport protocol which is implemented in user space, provides multiplexing, low latency handshake and better congestion control.

2.4 QUIC Transport Protocol

The QUIC is a new encrypted transport protocol, designed on the basis of decades of transport and security experience, with the aim to find a valid, more performing substitute to TCP protocol in all its Internet implications. Originally designed by Google, QUIC has taken all the best qualities of TCP connections and TLS encryption and implemented them over UDP. Starting as an experiment at Google, QUIC was developed through a collaborative and iterative standardization process at the IETF after almost five years. Finally, In May 2021, the IETF standardized QUIC in RFC 9000, supported by RFC 8999, RFC 9001, and RFC 9002.

Main characteristics

One of the main goals of QUIC is its sustainability. Regarding that, QUIC has been designed to run on top of the User Datagram Protocol (UDP), in the way that QUIC packets are encapsulated in UDP datagrams. This is necessary because many middlebox devices across the Internet process packets originating only from TCP and UDP. Since UDP is a connectionless protocol, QUIC handles all of the logic needed to guarantee a reliable connection between two endpoints. Moreover, QUIC is developed as an application-layer transport protocol, meaning that any updates on its working principles, even substantial ones, do not require OS changes which would take a long time to be deployed. By doing so, whenever a new feature or protocol revision will be introduced, an update of servers and application clients will be enough. Figure 2.5 compares a common HTTP stack with the QUIC stack in terms of OSI layering.

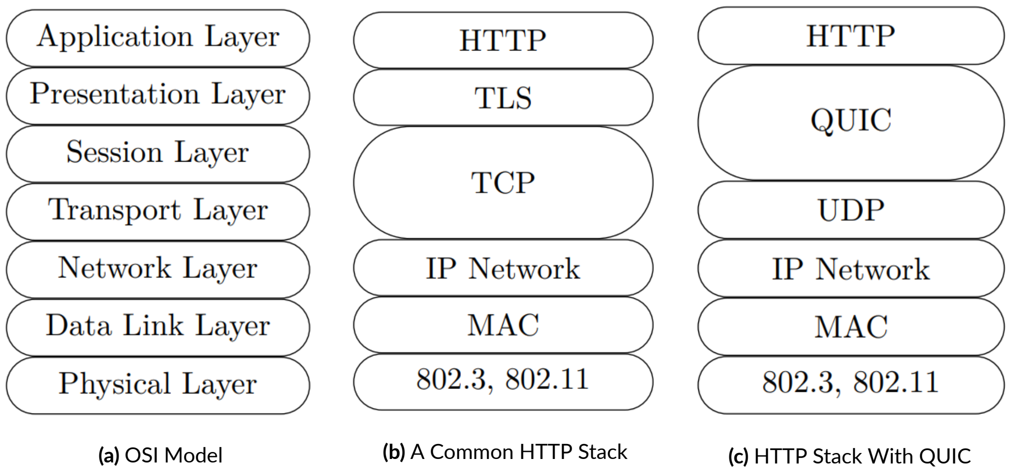


Figure 2.5: The OSI Model Compared a Common HTTP Stack and a QUIC Stack

Authentication and encryption

Another key aspect of QUIC is the security-by-default approach, radically different to the principle of TCP. Indeed, the TLS 1.3[3] cryptographic protocol is an integral part of the QUIC protocol. Used in AEAD configuration, it provides authentication and confidentiality of the entire transmission, with the exception of a few fields in the header which are simply authenticated to maintain visibility and ensure protocol functionality.

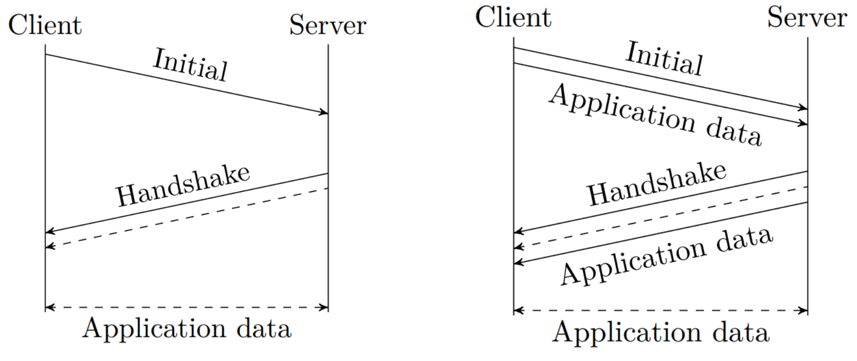
Faster connection

QUIC reduces the latency of TLS by embedding the encryption layer into the transport layer and by reworking the connection establishment mechanism. In standard HTTP+TLS+TCP, TCP needs a handshake to establish a session between server and client, and TLS needs its own handshake to ensure that the session is secured.

QUIC only needs a single handshake to establish a secure session. The default connection establishment is performed in a single round-trip, as illustrated by Figure 2.6a After the Handshake packets are sent, the server is able to exchange application data. After their receipt, the client is able to do it as well.

Connection migration

QUIC also addresses the rise of mobile devices by making connection resumption less costly in terms of latency. A client that has connected at least once with a given server can resume a connection to it along with application data without waiting for any round trip, as illustrated by Figure 2.6b. The underlying assumption is that mobile devices tend to go offline and switch back online very frequently. But the services they offer may still be running throughout. When connecting back using QUIC, these services will be able to resume their connections without having to wait any round-trip time before sending application data. This is known as 0-RTT connection resumption.



(a) 1-RTT Connection Establishment

(b) 0-RTT Connection Establishment

Figure 2.6: Connection Establishment in QUIC

Multiplexing without head-of-line blocking

Because the parallel nature of HTTP/2's multiplexing is not visible to TCP's loss recovery mechanisms, a lost or reordered packet causes all active transactions to experience a stall regardless of whether that transaction was directly impacted by the lost packet.

However, the QUIC transport protocol uses the new HTTP/3 [5] that incorporates stream multiplexing and per-stream flow control, similar to that provided by the HTTP/2 framing layer. A single connection can contain multiple independent streams that can be unidirectional or bidirectional, and each of them can carry an arbitrary amount of data divided into frames. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

Taking advantage of this feature, it is easy to understand how the problem of head-of-line blocking is solved at the core of the issue. In fact, in case of packet loss, only the streams whose frames were contained in the lost packet will be blocked. As a result, streams that have not suffered any loss will be able to continue their delivery process to the application.

QUIC Header Format

In order to minimize the amount of data used in each packet for control information, QUIC uses two different formats for its header. Packets with the long header are used during connection establishment. Packets with the short header are designed for minimal overhead and are used after a connection is established and encryption keys are available.

Long header

Packets with long headers, indicated in Figure 2.7, are used during the handshake procedure. In this category, there are the packets: Initial, Handshake, 0-RTT, and Retry.

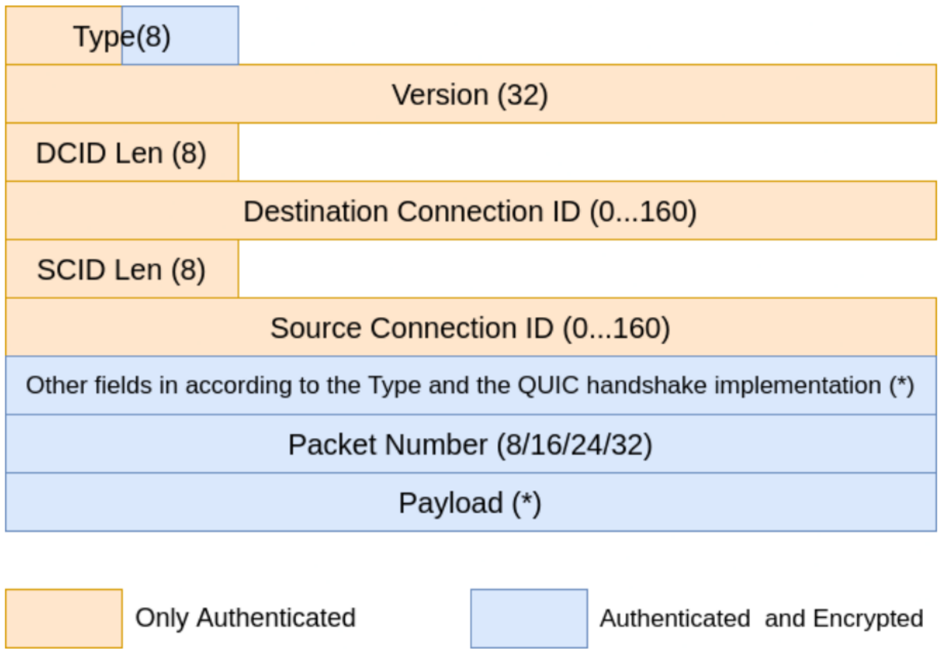

Type(8)
Version (32)
DCID Len (8)
Destination Connection ID (0...160)
SCID Len (8)
Source Connection ID (0...160)
Other fields in according to the Type and the QUIC handshake implementation (*)
Packet Number (8/16/24/32)
Payload (*)
Only Authenticated
Authenticated and Encrypted

Figure 2.7: QUIC Long Header Format

- **Flags Byte**: This is the first byte in a long header packet. The MSB is set to 1. The second bit is the fixed bit with a value of 1. The third and fourth bits are called Type bits and represent the long header packet type. The last four bits are encrypted and their values depend on the packet's type.
- **Version**: This field contains the QUIC version information and determines how the following packets byte need to be interpreted.
- **Destination Connection ID Length**: This field contains the length of the next field.
- **Destination Connection ID**: Field containing the unique ID value given to the destination endpoint in a QUIC connection.
- **Source Connection ID Length / Source Connection ID**: Same things as the previous two fields.
- **Payload**: The encrypted payload, containing the data and control frames.

Short header

Packets with a short header, shown in Figure 2.8, are sent after the QUIC connection is created and the session encryption keys are available. The Official QUIC version defines only the 1-RTT packet with the short header, used to exchange data frames and ACK frames.

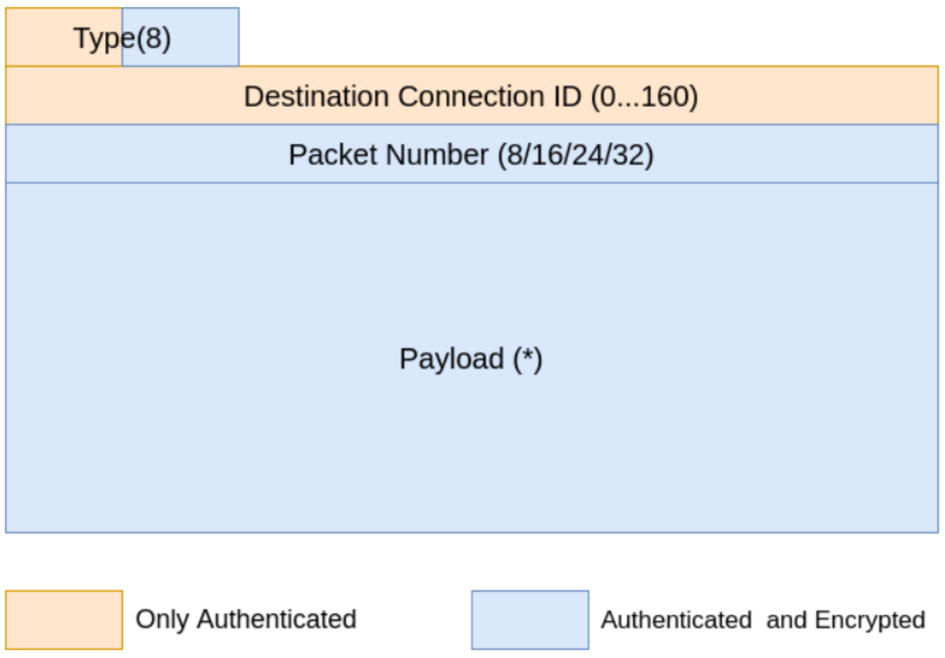

Type(8)
Destination Connection ID (0...160)
Packet Number (8/16/24/32)
Payload (*)
Only Authenticated
Authenticated and Encrypted

Figure 2.8: QUIC Short Header Format

- **Type Byte**: The counterpart of the long header's Flags Byte. It follows the pattern: 01SRRKPP. The most significant bit, as already said, identifies a short header. The next bit — also called the fixed bit — is set to 1 and packets containing a zero value for this bit must be discarded. The third bit (S) has been assigned to the SpinBit. The next two bits (RR) are reserved for future uses. The sixth one (K) is the Key Phase bit and it is used to help a packet recipient to identify the used packet protection keys. Finally, the last two bits (PP) provide the packet number length.
- **Packet Number Length**: Contains the encoded packet number field's length, that can be from 1 to 4 Byte long.
- **Destination Connection ID**: Field containing the unique ID value given to the destination endpoint in a QUIC connection (same thing as long header).
- **Packet number**: Field reporting the current packet number.
- **Payload**: The encrypted payload, containing data and control frames.

2.5 Is QUIC Considered a Better Choice Than TCP in Automated Meter Reading Applications?

As previously explained, Automated Meter Reading technology collects consumption, diagnostic, and status data from water meters and transfers the data to a central database for billing, troubleshooting, and analytics. It saves the expense and potential errors associated with manually reading meters. The Automated Meter Reading is usually implemented using vehicle-mounted or hand-held receivers to capture the signals transmitted from the advanced meters at frequent intervals.

The first solution is called the Drive-By and it provides efficient water meter reading coverage over large areas in very short time frames. Furthermore, those systems usually include dedicated navigation software for effective reading tasks. Regarding that, the reading application must be able to constantly update the server with its current location. It is important not only for keeping track of the device in real-time but also for designing efficient reading routes, which save fuel and labor costs and lowers carbon footprints. According to that, replacing TCP protocol with QUIC would significantly decrease latency, and improve communication.

On the other side, Walk-By solutions are designed for smart water meter data collection in locations that are inaccessible to vehicles. Often those places have poor network connectivity, which is a challenging environment for communication between mobile devices that are used for the collection of data, and the server. The results show that QUIC performs well under high latency conditions, in particular for low bandwidth that can be caused by bad network infrastructure in rural areas, as well as in congested locations when a lot of packets are dropped by the network.

The use of QUIC could also have a benefit above mentioned solutions allowing sending collected readings in real-time, together with specific properties of the device and their changes. Those properties can be related to a mobile device (carrier status, phone status, battery level, temperature, etc.), antennas, or android car statistics, and they can be useful for future processing and analysis.

2.6 Network Performance Measurements

One of the main points for the suitable functioning of the network is performance measurement. There are three main factors affecting the quality of a network, they are latency, throughput, and packet loss rate. Therefore, monitoring these parameters is very important not only in order to detect and fix issues, but also to understand if the infrastructure is able to support the data traffic, and decide if the improvement is needed.

2.6.1 Network Metrics

Networks' performances are affected mainly by three aspects [6]:

- **Latency**: Also called network delay, it represents the time required to transmit a packet across a network. More in detail, this metric is the sum of different delay types:
  - Processing delay: the total time required to process a packet from all the network devices between source and destination endpoints.
  - Transmission delay: the time required for a packet to cross the link between the two endpoints, depends on the link's bandwidth and the data traffic volume.
  - Propagation delay: the time duration taken for a signal to reach its destination.

- **Throughput**: The current amount of data transmitted over an Internet connection link in a given amount of time.
- **Packet loss rate**: Percentage of packets lost during data transmission, i.e., packets sent by a source host but never received by the destination host.

2.6.2 Measurement approaches

Currently, there are two main methods used for performance measurement: active, by insertion of test traffic, or passive by observing user generated traffic [6].

- **Active measurement**: During active measurement probe packets are sent through the network in addition to nominal data traffic. The biggest advantage of this method is the possibility to have packets flow also in absence of spontaneous traffic. It's also possible to mark the probe packets in order to easily recognize them between the other packets. At the same time, additional packets increase traffic volume and cause the consumption of more network resources. These two factors can degrade the network's performance and affect negatively the measurement samples retrieved.
- **Passive measurement**: It refers to the process of measuring a network, without creating or modifying any traffic, but merely observing certain characteristics exhibited by the various protocols. For example, RTT values can be measured by means of the handshake's packets of a TCP connection, i.e., taking into account the sending and receiving packet's timestamps on the endpoints. Although the traffic volume is not increased, is more difficult to use the nominal transmission packets. More complex algorithms and more expansive hardware are needed for the purpose.

Both measurement methods rely on information freely readable from the protocol header. Considering that QUIC encrypts essentially the whole transmission including control data, some explicit information, inevitably, must be exposed on the wire in order to allow network operators to measure more or less accurately RTT and loss rate of a connection.

2.6.3 Network Time Protocol

The Network Time Protocol (NTP) is a protocol that allows the synchronization of system clocks on computer networks to the universal coordinated time (UTC). Having synchronized clocks is not only convenient but required for many distributed applications. Furthermore, clock synchronization between Internet hosts is important in a variety of applications including measurement.

The basic operation of time synchronization in a distributed environment assumes the presence of a high-precision reference clock, which serves as the source of accurate time for other systems. Remote systems interact with the reference clock periodically to synchronize (discipline) their local clock. [7]. The primary challenges in such a distributed environment are the variation in crystal oscillator quality and environmental conditions, characteristics of the network paths that separate the remote host from the reference clock, and the protocol that is used to discipline the clock on the remote host.

The efforts of David L. Mills led to the standardization of the Network Time Protocol in 1985 [8]. NTP is widely used on the Internet today. At its core are high-precision reference clocks (typically atomic clocks or GPS-based clocks) that form the foundation of a widely distributed hierarchy of public time servers. The protocol mechanisms are based on periodic exchanges of information between clients and servers. However, the specific behavior of any particular client and its level of synchronization with a server can vary widely. Furthermore, for devices that do not require all the performance and accuracy benefits of NTP, a subset of NTP called Simple NTP (SNTP) [9] is used.

In order to get precise results in terms of measurements and benchmarking, it is very important that all test instances have a very similar idea of time. For that reason, it is highly recommended that all benchmark instances are configured to obtain time from some number of NTP servers.

2.7 The Latency Spin Bit

Because QUIC radiates far less information about its operation to devices on the path than TCP does, the QUIC Working Group has been working on a solution for measuring RTT. Brian Trammel describes the addition of a so-called "latency spin bit" in the QUIC header [2], which allows end-to-end RTT measurements of a QUIC connection.

The latency spin bit is a single bit value — placed in the third most significant bit of the QUIC short header. The algorithm works in the following way: Client and server contain an internal spin bit value used to set the spin bit on outgoing packets for that connection.

At the beginning of every QUIC connection, both endpoints set this value to 0. Then, the client and server manage the spin bit in different ways:

- Client: when it receives a short header packet with a higher sequence number than the last received, it toggles its spin bit value to the opposite value stored inside the received packet header.

- Server: when it receives a short header packet with a higher sequence number than the last received, it toggles its spin bit value in according to the value stored inside the received packet header.

The resulting spin value is therefore used for the following outgoing packets until an incoming packet with the opposite spin bit arrives changing the configured value. The result is the generation of a periodical square wave signal, as illustrated by Figure 2.9, observed by an observer placed between the two endpoints and able to measure the RTT value by measuring the length of this detected signal.

Moreover, an observer can work in one-way mode (it can only detect packets traveling in one direction, from client to server or vice versa) or in two-way mode (it is placed symmetrically on both upload and download directions).

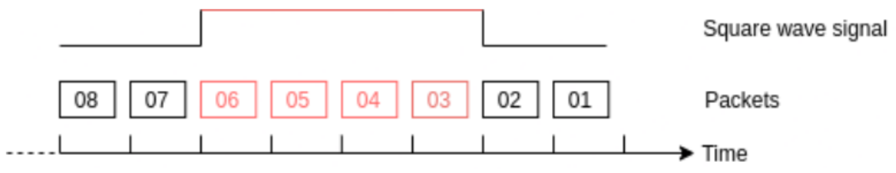


Figure 2.9: The Square Wave Generated by the Marked Packets

2.7.1 Limitations of the algorithm

Because of the simplicity behind the algorithm, it's only possible to retrieve RTT values with sufficient accuracy in optimal network conditions. In fact, there are some factors, called network impairments, that would degrade seriously the measurement system's accuracy.

Delays

RTT and half-RTT values are computed via the distance in a time of two square wave's edges. What could happen is that an observer overestimate the RTT value due to a longer square wave's length (the two wave's edges are more distant in time due to the additional delay). It could happen due to the activation of the QUIC congestion control or in presence of traffic holes, as indicated in Figure 2.10.

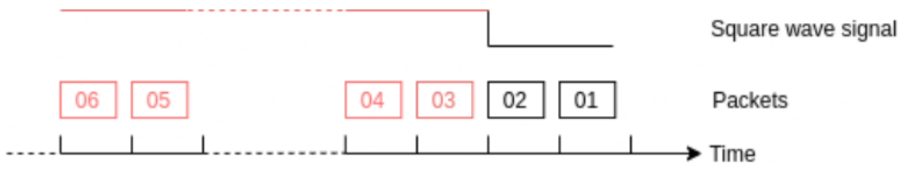


Figure 2.10: An Addition Delay Is Present Between the Marked Packets

Packet reordering

The client and server check the correctness of the packet's number every time a packet is received. If there are no reordering events, endpoints toggle the internal spin bit value accordingly to the algorithm's logic continuing to mark (or not) the packets as required. But the sequence number is readable only after removing the encryption layer over the packet's field where it is embedded, so the operation is possible only for the two endpoints involved in the connection. This means that an observer cannot access the packet number's field. Without any additional precaution, there is the possibility to detect one or more Spurious square wave edges with the result of a higher number of underestimated RTT samples, as shown in Figure 2.11.

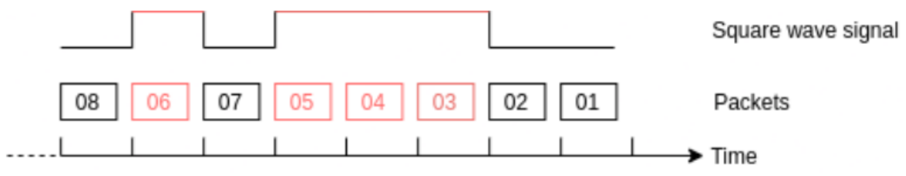


Figure 2.11: Spurious Edge Detected by the Observer

Packet loss

A packet could be lost somewhere between the source host and the destination host. Also, this event could affect negatively the latency spin bit accuracy. As can be seen from Figure 2.12, packets lost on the square wave edges would cause an underestimation of the RTT sample. In this case the square wave is shorter than the expected one.

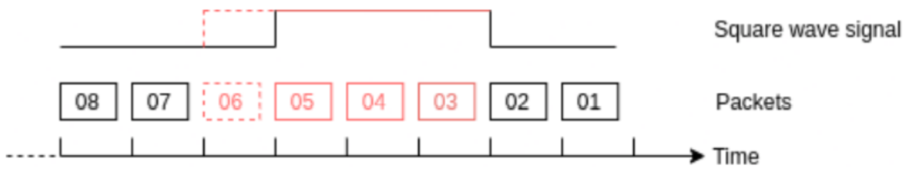


Figure 2.12: Underestimation of the Square Wave's Length

3 Methodology

3.1 Approach

The idea for this master thesis work originally started during my internship program at InkwelldData, after seeing the potential of mobile applications for AMR tasks, but also noticing potential improvements that can be made in this field.

For the purpose of this thesis, the first step was to build an application that handles the Industrial IoT work for smart water metering purposes, propose some new innovative solutions, and to implement all crucial features. The next big step was to improve the speed of network connection, and security while still maintaining reliability, using cutting-edge technologies, and switching from TCP to QUIC transport protocol, taking advantage of all benefits it brings. In the following sections, we are going to see more in detail the architecture as well as the technologies used for developing the client-side, which represents AMR Mobile Application, and the server-side which is the Elixir application.

3.2 Architecture

3.2.1 Client Side - AMR Mobile Application

On the client-side, a native android mobile application was developed using Kotlin programming language. This application is the mobile application for Android smartphones and tablets that allows users to operate simultaneously in regular (Manual and Photo-reading mode), and also in smart (Walk-By and Drive-By) reading mode. It is capable of acquiring and translating the received frames in real-time in the M-Bus wireless protocol in 868 MHz or 169 MHz transmitted by smart meters placed on the map screen based on their current location. Meters are clustered using Google Maps Android Marker Clustering Utility to avoid the map being hard to read in the case of a large number of meters and low zoom level, as illustrated in Figure 3.1. The application can receive data in the background, without requiring any interaction from the operator who can then devote himself to the execution of other tasks.

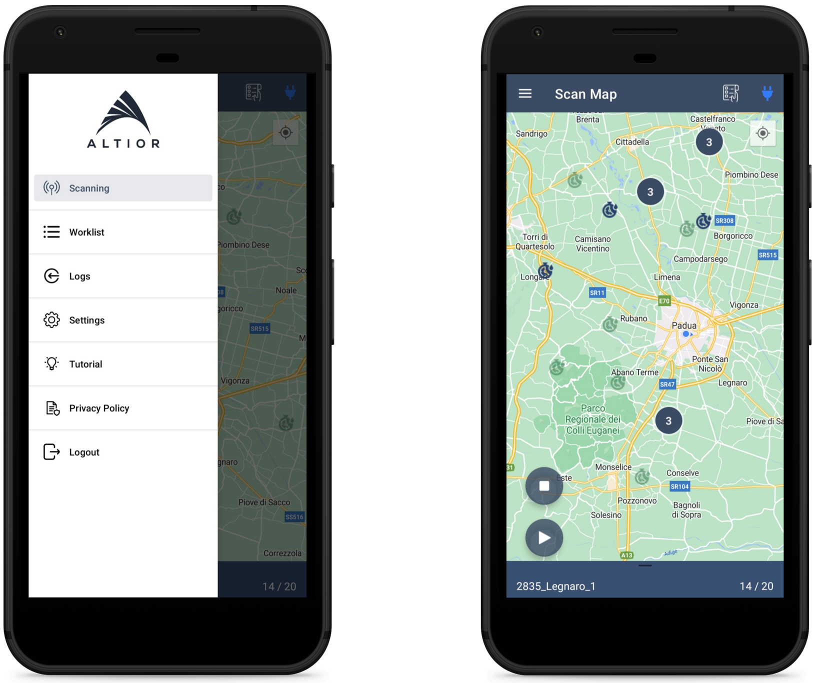
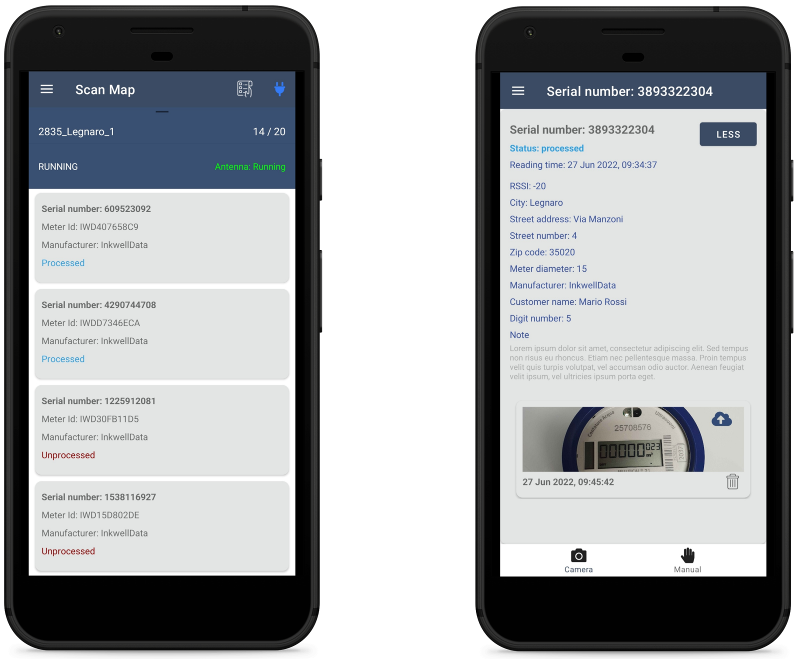


Figure 3.1: Map Screen

The application acquires the frames coming from the meters in WM-bus protocol in 868 MHz and 169 MHz, positioned around the operator during his movements, periodically sending the data to the InkwellData server which records and processes them, after which they become available on the platform.

In addition to the consumption of the meters detected, which is encrypted and not directly accessible, the water operator is able to monitor the number of readings acquired, acquiring time, type of the meter, its manufacturer, the signal strength of the received signal (RSSI), as well as the meter's location (Figure 3.2).


Scan Map

2835_Legnaro_1 14 / 20

RUNNING Antenna: Running

Serial number: 609523092
Meter Id: IWD407658C9
Manufacturer: InkwellData
Processed

Serial number: 4290744708
Meter Id: IWDD7346ECA
Manufacturer: InkwellData
Processed

Serial number: 1225912081
Meter Id: IWD30FB11D5
Manufacturer: InkwellData
Unprocessed

Serial number: 1538116927
Meter Id: IWD15D802DE
Manufacturer: InkwellData
Unprocessed

Serial number: 3893322304

Serial number: 3893322304 LESS
Status: processed
Reading time: 27 Jun 2022, 09:34:37

RSSI: -20
City: Legnaro
Street address: Via Manzoni
Street number: 4
Zip code: 35020
Meter diameter: 15
Manufacturer: InkwellData
Customer name: Mario Rossi
Digit number: 5
Note
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed tempus non risus eu rhoncus. Etiam nec pellentesque massa. Proin tempus velit quis turpis volutpat, vel accumsan odio auctor. Aenean feugiat velit ipsum, vel ultricies ipsum porta eget.

27 Jun 2022, 09:45:42

Figure 3.2: List of Meters and Meter Preview

Due to the fact that meters can be placed in locations without an internet connection, there is the feature for an offline reading mode that gives the possibility to the user to finish the reading tasks without any interruptions and upload all acquired data once the network becomes available.

MVVM Architecture Pattern

In order to separate the data presentation logic of the mobile application, from the core business logic part, we used Model-View-ViewModel (MVVM) architecture pattern, as shown in Figure 3.3, with the following characteristics:

- **Model**: This holds the data of the application. It cannot directly talk to the View. It exposes the data to the ViewModel through Observables.
- **View**: It represents the UI of the application, and it observes the ViewModel.
- **ViewModel**: It acts as a link between the Model and the View. It's responsible for transforming the data from the Model, and it exposes data streams that are relevant to the View.

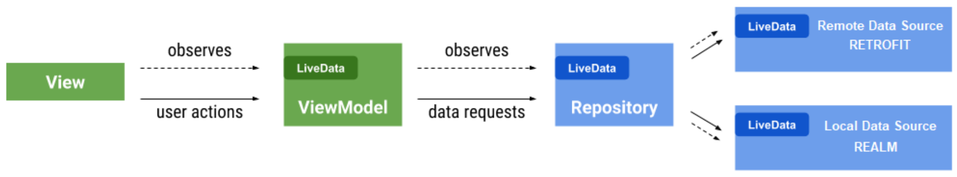


Figure 3.3: MVVM Architecture Pattern

3.2.2 Server Side - Elixir Application

We considered the Elixir language and the principles it brings because of its numerous advantages for server-side development in the IoT world. Elixir is a functional, dynamically typed language that is built on top of Erlang's VM and compiles down to Erlang bytecode. It allows one to create very scalable and reliable applications.

A Brief History of Erlang

Before explaining Elixir, we inevitably should start by providing a brief background of Erlang. Erlang is a functional, general-purpose language oriented towards building scalable, concurrent systems with high availability guarantees.

It was built at the end of the 1980s at Ericsson for handling telephone switches. At the time, telephone switching systems were one of the most complicated systems out there, like the internet is nowadays. For this reason, the language used to program them needed to support high concurrency and zero downtime. After going through multiple existing language options, three people at the company, Joe Armstrong, Robert Virding, and Mike Williams decided to create their own language – called Erlang.

Process-oriented

The main thing that distinguishes Erlang from other languages is its process-based computing model. It uses isolated, lightweight processes that communicate with each other through messages. These processes can receive messages and, in response to messages, create new processes, send messages to other processes, or modify their state. In other words, Erlang follows the actor model as shown in Figure 3.4.

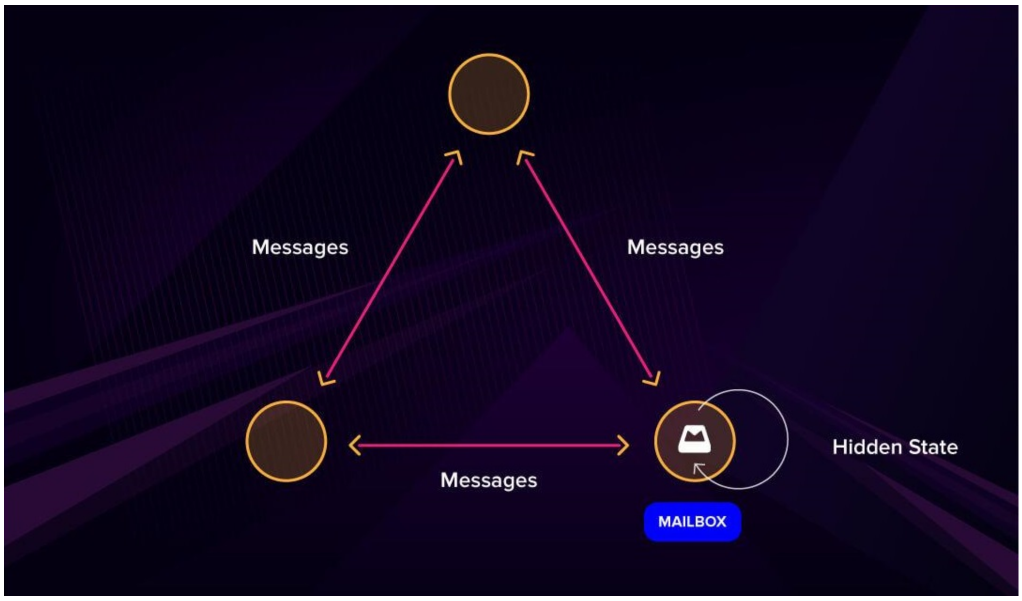


Figure 3.4: Actor Model in Erlang

The processes are isolated, fast to create, and take up only a small amount of memory. It is easy to expand the system by creating more of them. Since the processes do not discern whether the other processes are on the same core or in another place, scaling becomes easy, both horizontally (by adding more machines) and vertically (by adding cores).

Let It Crash Philosophy

The Elixir philosophy says not to worry much about code that crashes, instead, make sure the overall application keeps running.

Thinking of a typical application, if an unhandled error causes an exception to be raised, the application stops. Nothing else gets done until it is restarted, and in case it's a server handling multiple requests, they all might be lost. The main issue here is that one error takes the whole application down.

In the Elixir world, an application consists of hundreds or thousands of processes, each handling just a small part of a request. If one of those crashes, everything else carries on, and when that process gets restarted, the application is back running totally. That is why Erlang is known for its use in systems that are fault tolerant and reliable.

One of the ideas at the core of the Erlang runtime system's design is the *Let It Crash* error handling philosophy. The *Let It Crash* philosophy is an approach to error handling that seeks preserve the integrity and reliability of a system by intentionally allowing certain faults to go unhandled.

Basically, an Erlang app is a tree of processes (Figure 3.5). At the bottom leaves of the tree, there are worker processes – the ones doing most of the work. Up from them, there are supervisors, which launch the workers and check up on them. Supervisors themselves can be supervised by adding a Grand Supervisor on top of the tree here.

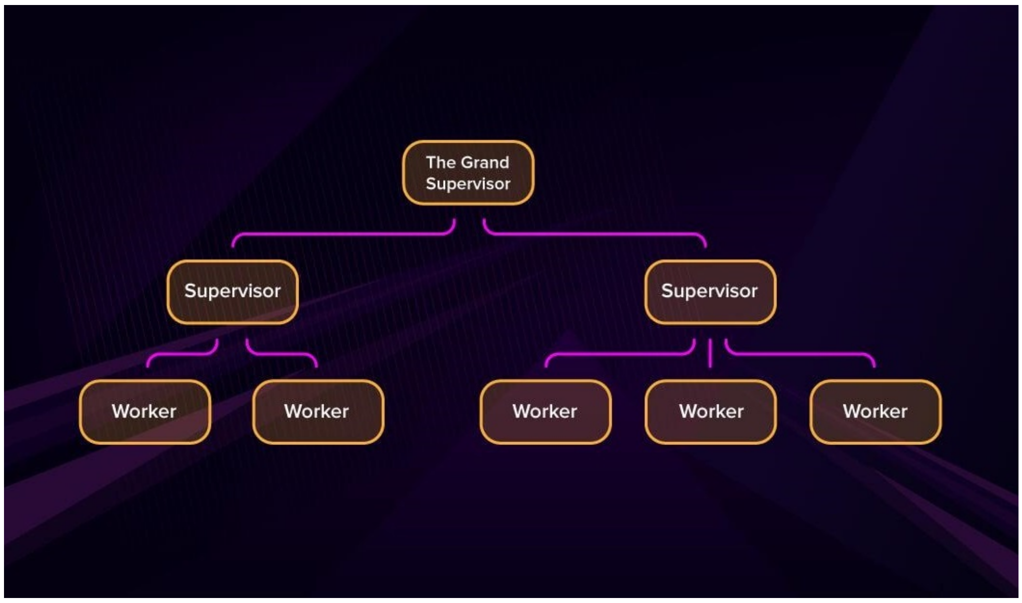


Figure 3.5: Erlang Three of Processes

In case a process crashes, it sends a message to its supervisor. Depending on the supervision strategy set, either just the process is restarted or all of the processes underneath its supervisor are. If restarting the connected workers does not solve the problem a given period of time, the supervisor will terminate all its children and then itself. At that point, the responsibility to try to handle the problem is pushed upwards to the next supervision layer (Figure 3.6).

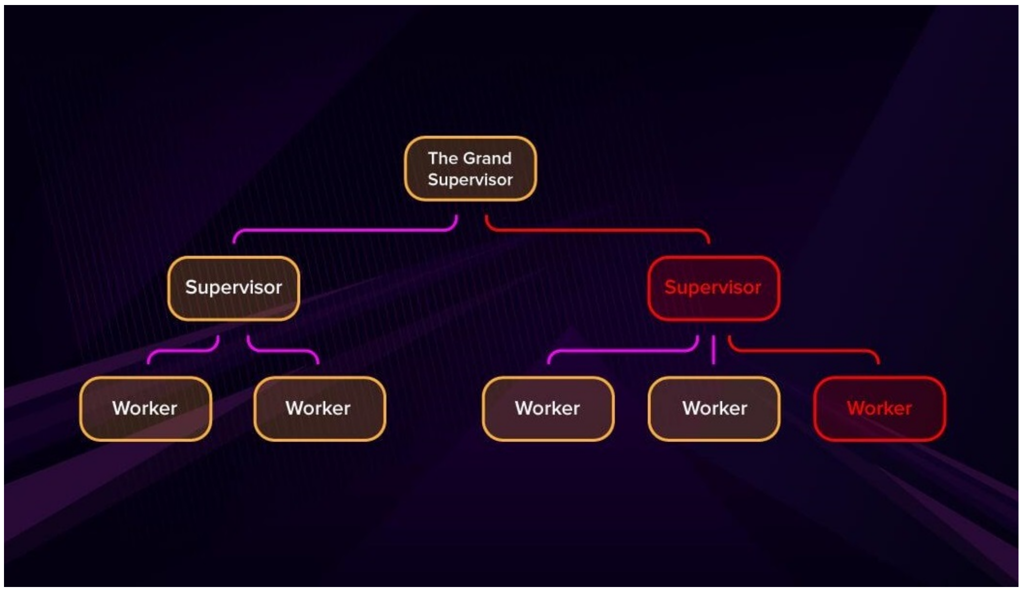


Figure 3.6: Crash of the process

Only if the top-level supervisor fails does it not get restarted and the application crashes.

Elixir over Erlang

Elixir was intended to be a Ruby-like language that would give developers access to all the powerful tools that Erlang gives to developers for doing parallel and concurrent computation. Elixir was to include features that were necessary but missing in Erlang.

Main features of Elixir are:

- **Built on top of the Erlang VM**: Elixir has access to all the concurrency tools that Erlang has access to, making it one of the most powerful modern languages for building scalable, distributed systems.
- **Ruby-like syntax**: Ruby is known as one of the most concise and productivity-oriented languages, but it is lacking in performance and concurrency. Erlang VM solves both of these problems, and, therefore, Elixir is the best of both worlds.
- **Functional**: While Elixir and Erlang could be characterized as being in a group of their own, since they are both process-oriented, Elixir also has all the constructs expected from modern functional programming languages. In particular, immutable data structures help concurrency, and pattern matching is a great way for writing declarative code.
- **Dynamic typing**: Elixir has dynamic typing in contrast to other functional languages like Haskell and Scala. This means that types are checked in run-time, not during compilation. While this can be a downside when building critical systems, it also increases development speed for simple web applications.

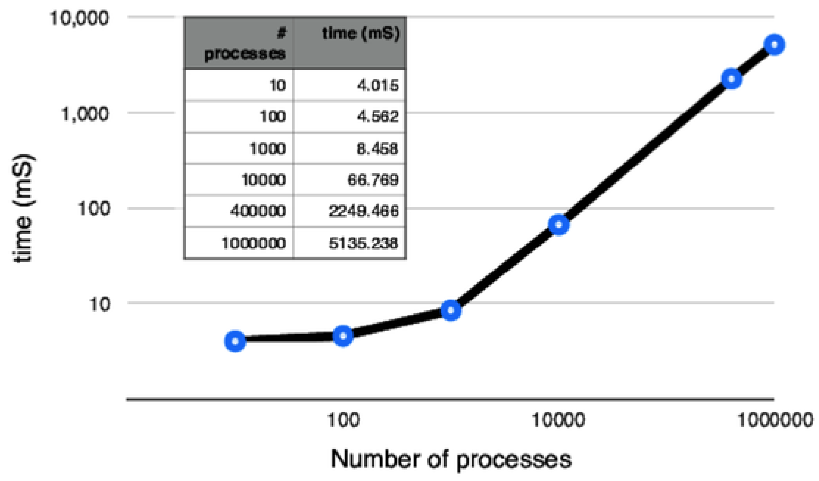
Benefits of Elixir

Elixir is built on top of BEAM, the Erlang VM, and it shares the same abstractions that have made Erlang one of the best choices for concurrent, high-volume applications.

With Erlang, Elixir shares these three characteristics:

- **Concurrency**: Elixir uses lightweight threads of execution (called processes). These are isolated, run across all CPUs, and communicate through messages. Together with the immutability of data that comes from the functional nature of the language, this makes it easier to write concurrent programs in Elixir.
- **Scalability**: These same processes allow to scale systems easily, either horizontally (adding more machines to the cluster) or vertically (using the available resources of the machine more efficiently).
- **Reliability**: Elixir and Erlang have a unique approach to fault-tolerance. While things sometimes inevitably fail in production, lightweight processes can be quickly restarted by the supervisor system.

As mentioned before, processes in Elixir are very low overhead. A benchmark test showed that we can run a million processes (sequentially) in just over 5 seconds. And, as Figure 3.7 shows, the time per process was pretty much linear once the startup time is overcome [10].



# processes	time (mS)
10	4.015
100	4.562
1000	8.458
10000	66.769
400000	2249.466
1000000	5135.238

Figure 3.7: Process Creation

This kind of performance is stunning, and it changes the way the code is designed. We can create hundreds of little helper processes. And each process can contain its own state—in a way, processes in Elixir are like objects in an object-oriented system (but they're more self-contained).

Functional programming

Functional programming is a programming paradigm that treats programs like evaluations of mathematical functions and avoids things like mutable data and changing state.

In contrast to mainstream programming languages like Java or Python, Elixir code is structured in functions and modules (groups of functions) instead of objects and classes. In addition, all the data types are immutable. For example, calling a function on a variable will produce a new variable, not change the variable in place.

Functional programming supports and takes advantage of things like pattern matching, higher-order functions, and declarative style of writing code, and that comes with multiple benefits such as: better maintainability, clearer testing and debugging and code that is simpler to write and understand.

3.3 Implementation

3.3.1 Client Side - AMR Mobile Application

Retrofit

To make communications with the server fast and reliable, the AMR mobile application uses the Retrofit library. Retrofit is a type-safe REST client for Android, Java, and Kotlin developed by Square [11]. The library provides a powerful framework for authenticating and interacting with APIs and sending network requests with OkHttp.

The main reason for using Retrofit was that it has many features like easy to add custom headers and request types, file uploads, and mocking responses, through which we can reduce boilerplate code in our apps and consume the web service easily. Moreover, Retrofit keeps updating with the latest trends such as compatibility with RxKotlin and coroutines also used in the application development.

To implement Retrofit we first create a Retrofit instance through the Singleton pattern in Kotlin using lazy initialization, as shown in Figure 3.8, specifying the URL that contains the data required and using the Retrofit Builder class (Figure 3.9). Here we also specify different converters depending on the JSON one would like to use.

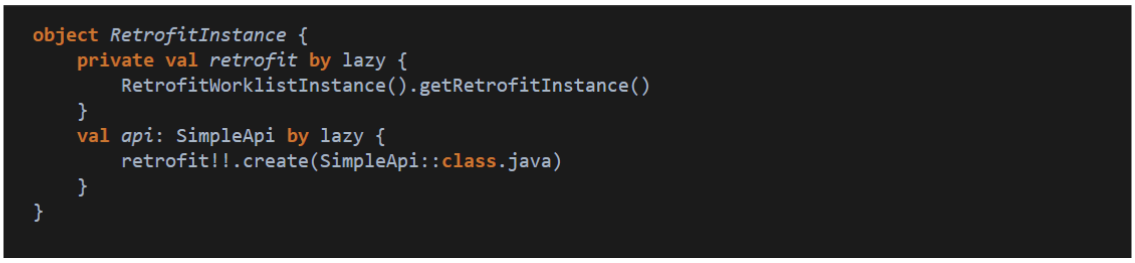
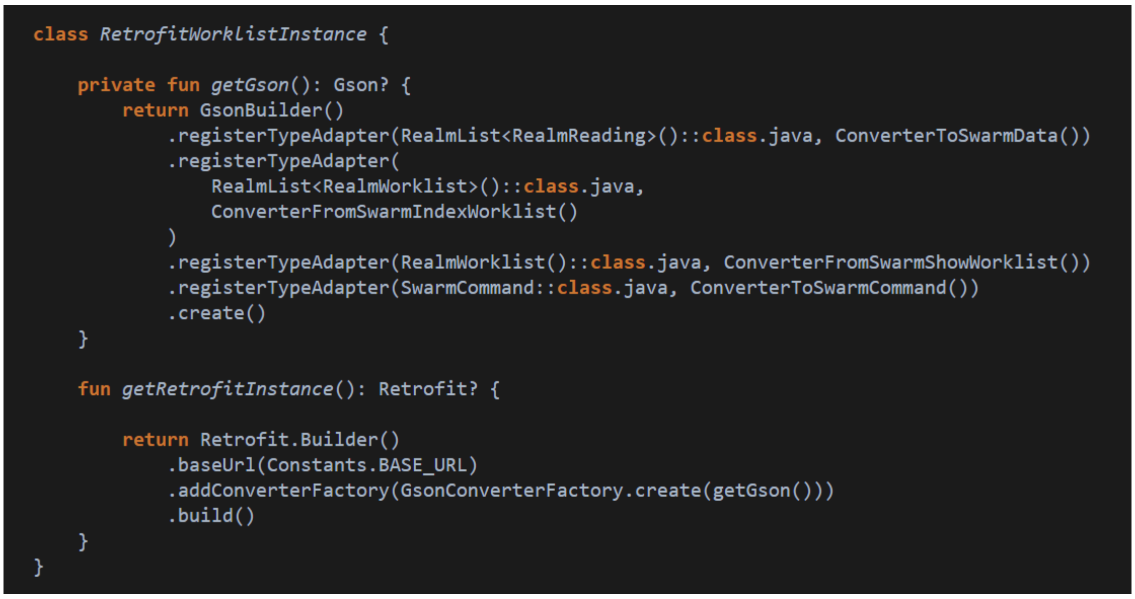

```kotlin
object RetrofitInstance {
    private val retrofit by lazy {
        RetrofitWorklistInstance().getRetrofitInstance()
    }
    val api: SimpleApi by lazy {
        retrofit!!.create(SimpleApi::class.java)
    }
}
```

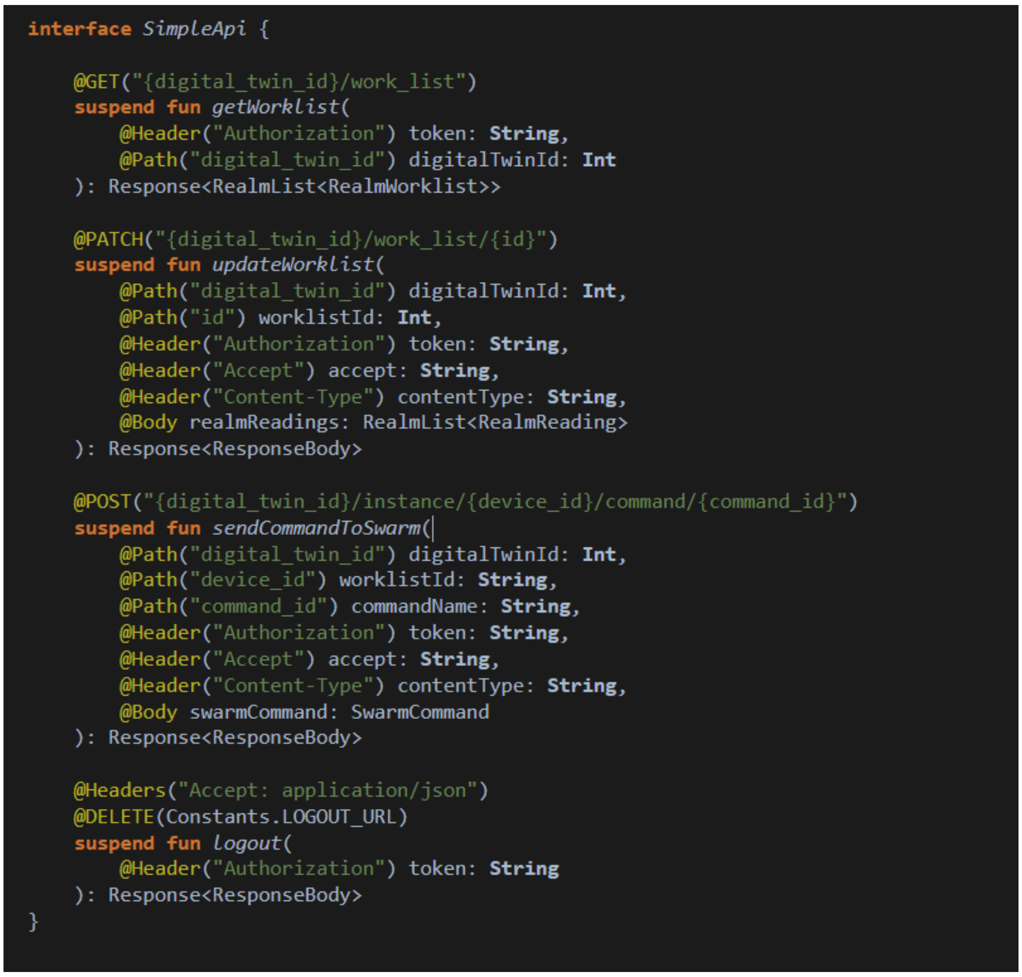
Figure 3.8: Retrofit Instance Created Through Singleton Pattern in Kotlin Using Lazy Initialization



```kotlin
class RetrofitWorkListInstance {

    private fun getGson(): Gson? {
        return GsonBuilder()
            .registerTypeAdapter(RealmList<RealmReading>()::class.java, ConverterToSwarmData())
            .registerTypeAdapter(
                RealmList<RealmWorklist>()::class.java,
                ConverterFromSwarmIndexWorklist()
            )
            .registerTypeAdapter(RealmWorklist()::class.java, ConverterFromSwarmShowWorklist())
            .registerTypeAdapter(SwarmCommand::class.java, ConverterToSwarmCommand())
            .create()
    }

    fun getRetrofitInstance(): Retrofit? {

        return Retrofit.Builder()
            .baseUrl(Constants.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create(getGson()))
            .build()
    }
}
```

Figure 3.9: Retrofit Instance Class

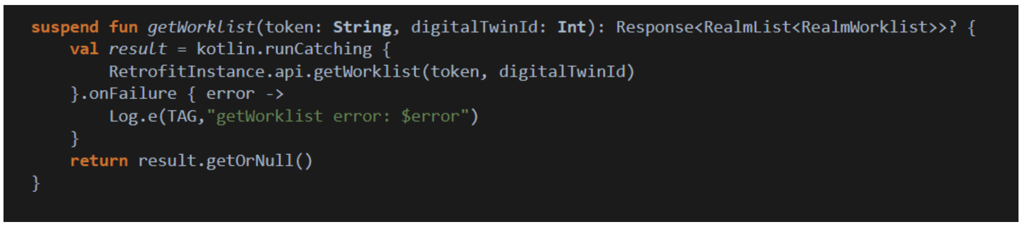
Subsequently, we specify the endpoints, which are usually defined inside an Interface class. An endpoint refers to the path where information is obtained. Figure 3.10 illustrates different types of annotations for different requests used inside the application.



```kotlin
interface SimpleApi {

    @GET("{digital_twin_id}/work_list")
    suspend fun getWorklist(
        @Header("Authorization") token: String,
        @Path("digital_twin_id") digitalTwinId: Int
    ): Response<RealmList<RealmWorklist>>

    @PATCH("{digital_twin_id}/work_list/{id}")
    suspend fun updateWorklist(
        @Path("digital_twin_id") digitalTwinId: Int,
        @Path("id") worklistId: Int,
        @Header("Authorization") token: String,
        @Header("Accept") accept: String,
        @Header("Content-Type") contentType: String,
        @Body realmReadings: RealmList<RealmReading>
    ): Response<ResponseBody>

    @POST("{digital_twin_id}/instance/{device_id}/command/{command_id}")
    suspend fun sendCommandToSwarm(
        @Path("digital_twin_id") digitalTwinId: Int,
        @Path("device_id") worklistId: String,
        @Path("command_id") commandName: String,
        @Header("Authorization") token: String,
        @Header("Accept") accept: String,
        @Header("Content-Type") contentType: String,
        @Body swarmCommand: SwarmCommand
    ): Response<ResponseBody>

    @Headers("Accept: application/json")
    @DELETE(Constants.LOGOUT_URL)
    suspend fun logout(
        @Header("Authorization") token: String
    ): Response<ResponseBody>
}
```

Figure 3.10: API Interface

In the following step, we call API endpoints defined in our interface class from the Repository, as illustrated in Figure 3.11. According to the answer that we receive, ViewModel updates its state that is observed by the UI that renders itself accordingly.



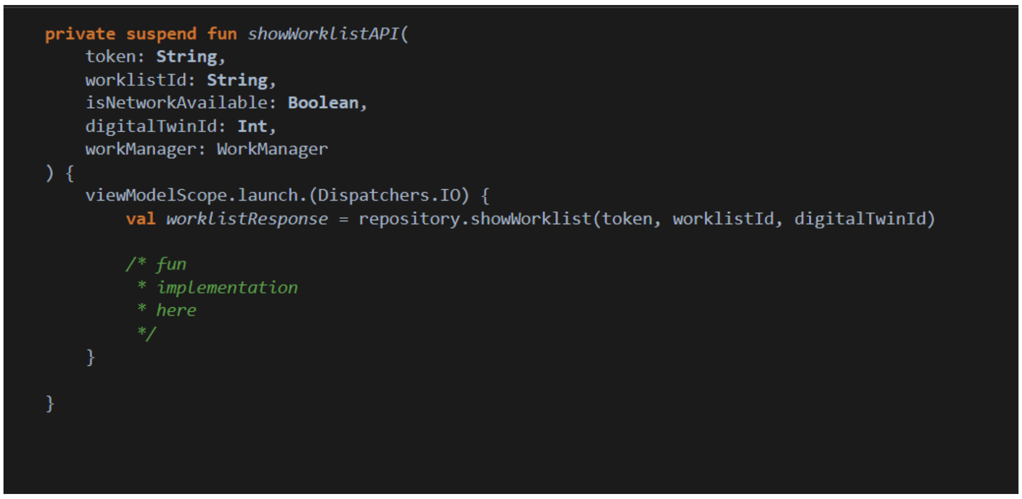
```kotlin
suspend fun getWorklist(token: String, digitalTwinId: Int): Response<RealmList<RealmWorklist>>? {
    val result = kotlin.runCatching {
        RetrofitInstance.api.getWorklist(token, digitalTwinId)
    }.onFailure { error ->
        Log.e(TAG,"getWorklist error: $error")
    }
    return result.getOrNull()
}
```

Figure 3.11: Retrofit Repository Function Call

Coroutines

In order to make the AMR mobile application responsive and to effectively deal with async work such as database or network access, as well as to manage long-running tasks that might otherwise block the main thread and cause the application to become unresponsive, we used coroutines, that Kotlin team defines as lightweight threads.

In general, making a network request on the main thread causes it to wait, or block, until it receives a response. Since the thread is blocked, the OS is not able to call *onDraw()*, which causes an application to freeze. For a better user experience, the solution is to move execution off the main thread, and to create a new coroutine and execute the network request on an I/O thread, as shown in Figure 3.12.



```kotlin
private suspend fun showWorklistAPI(
    token: String,
    worklistId: String,
    isNetworkAvailable: Boolean,
    digitalTwinId: Int,
    workManager: WorkManager
) {
    viewModelScope.launch.(Dispatchers.IO) {
        val worklistResponse = repository.showWorklist(token, worklistId, digitalTwinId)

        /* fun
         * implementation
         * here
         */
    }

}
```

Figure 3.12: Launching a Coroutine

There are a couple of important concepts worth mentioning that are used in the code above:

- **viewModelScope**: It is a predefined *CoroutineScope* that is included with the *ViewModel* KTX extensions. Note that all coroutines must run in a scope. A *CoroutineScope* manages one or more related coroutines.
- **launch**: It is a function that creates a coroutine and dispatches the execution of its function body to the corresponding dispatcher.
- **Dispatchers.IO**: Indicates that this coroutine should be executed on a thread reserved for I/O operations.

The *showWorklistAPI* function is executed as follows:

- The app calls the *showWorklistAPI* function from the View layer on the main thread.
- **launch** creates a new coroutine, and the network request is made independently on a thread reserved for I/O operations.
- While the coroutine is running, the *showWorklistAPI* function continues execution and returns, possibly before the network request is finished.

Since this coroutine is started with *viewModelScope*, it is executed in the scope of the ViewModel. If the ViewModel is destroyed because the user is navigating away from the screen, *viewModelScope* is automatically cancelled, and all running coroutines are cancelled as well.

The next step is to ensure that anything that calls *showWorklist* needs to remember to explicitly move the execution off the main thread. For that reason, *showWorklist* function is marked with the *suspend* keyword, that is Kotlin's way to enforce a function to be called from within a coroutine.

LiveData and Observables

To ensure a more interactive and dynamic user experience, and to have components to respond interactively with the changes and updates in related data, we used the concepts of LiveData and the observable pattern.

LiveData is an android observable data holder class, and unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state [12].

LiveData only notifies active observers about updates, while inactive observers registered to watch LiveData objects are not notified about changes. One can register an observer paired with an object that implements the *LifecycleOwner* interface. This relationship allows the observer to be removed when the state of the corresponding Lifecycle object changes to *destroyed*. This behaviour is especially useful for activities and fragments because they can safely observe LiveData objects and not worry about leaks—activities and fragments are instantly unsubscribed when their lifecycles are destroyed.

There are a lot of advantages of using LiveData in mobile applications:

- **Ensures UI matches data state**: LiveData follows the observer pattern, and it notifies Observer objects when underlying data changes. The code can be consolidated to update the UI in these Observer objects. That way, updating the UI is not needed every time the app data changes because the observer is in charge of that.
- **No memory leaks**: Observers are bound to Lifecycle objects and clean up after themselves when their associated lifecycle is destroyed.
- **No crashes due to stopped activities**: If the observer's lifecycle is inactive, such as in the case of an activity in the back stack, then it does not receive any LiveData events.
- **No more manual lifecycle handling**: UI components just observe relevant data and do not stop or resume observation. LiveData automatically manages all of this since it's aware of the relevant lifecycle status changes while observing.
- **Always up to date data**: If a lifecycle becomes inactive, it receives the latest data upon becoming active again. For example, an activity that was in the background receives the latest data right after it returns to the foreground.
- **Proper configuration changes**: If an activity or fragment is recreated due to a configuration change, like device rotation, it immediately receives the latest available data.
- **Sharing resources**: LiveData objects can be extended using the singleton pattern to wrap system services so that they can be shared in the application. The LiveData object connects to the system service once, and then any observer that needs the resource can just watch the LiveData object.

Realm

Realm Mobile Database is a cross-platform database solution that can be used as an alternative to SQLite and Room. Realm is a NoSQL database that makes it possible for programmers to declare relationships between objects, as developers can do it in an object graph of any programming language. For the purpose of this project, it was decided to use Realm because of the many advantages it brings. First, compared to SQLite and Room, Realm is easier to set up and use. To perform the same operation in Realm, usually fewer lines of code are needed, than using SQLite or Room. Second, it offers other modern features such as encryption, JSON support, and data change notifications.

Unlike a traditional database, objects in Realm are native objects. It is not needed to copy objects out of the database, modify them, and save them back, because we are always working with the "live" real object. If one thread or process modifies an object, other threads and processes can be immediately notified, therefore objects always stay in sync.

Also, on performance, Realm is considered as faster. Below there are several graphs comparing the speed of executing the CRUD operations on various datasets for SQLite, Room, and Realm. Thus, the benchmark tests show that Realm is faster than SQLite and Room [13].

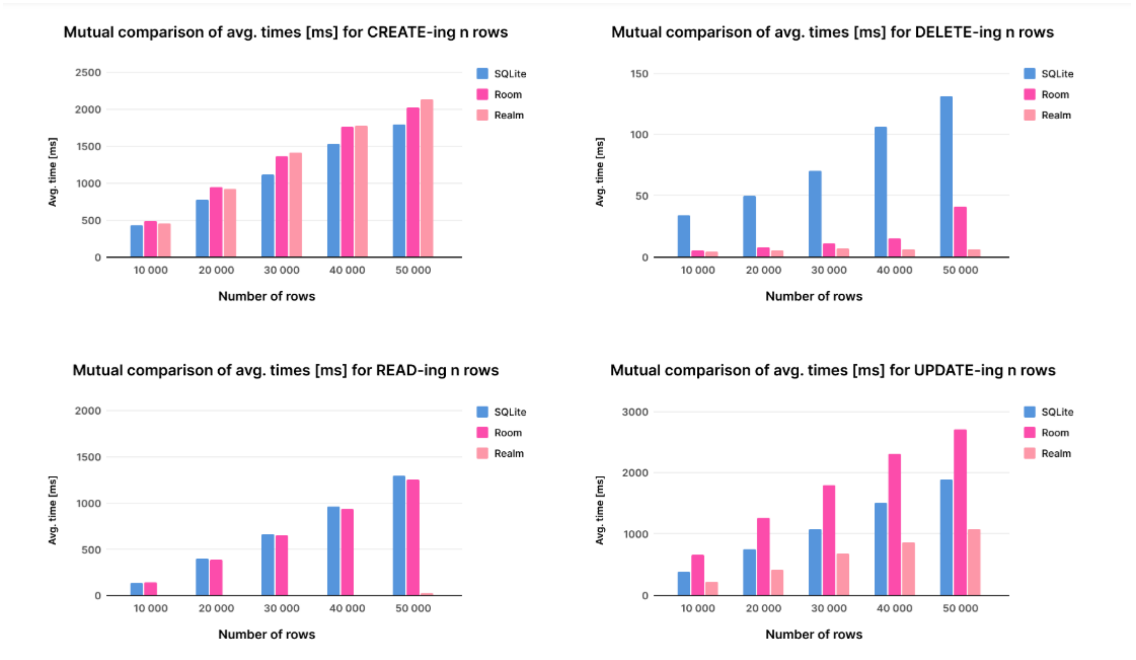


Figure 3.13: Databases CRUD comparison

Furthermore, unlike the SQLite database, Realm provides default encryption features for data security, which fulfills one of the key aspects of any Android app nowadays.

QUIC Integration – Cronet

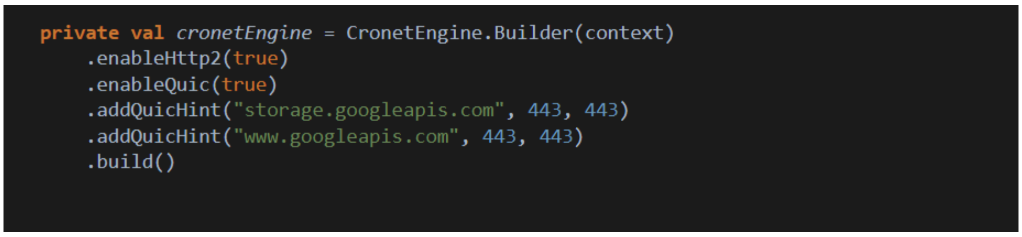
To successfully integrate QUIC and improve app performance we replaced initial networking stack (HTTP/2 over TLS/TCP) with the QUIC protocol. We leveraged the Cronet networking library from the Chromium Projects which implement a version of the QUIC protocol, and takes advantage of multiple technologies that reduce the latency and increase the throughput of the network requests that the app needs to work.

Cronet comes with a lot of features, such as:

- **Protocol support**: Cronet natively supports HTTP, HTTP/2, and HTTP/3 over QUIC protocols.
- **Request prioritization**: The library allows setting a priority tag for the requests. The server can use the priority tag to determine the order in which to handle the requests.
- **Resource caching**: Cronet can use an in-memory or disk cache to store resources retrieved in network requests. Subsequent requests are served from the cache automatically.
- **Asynchronous request**: Network requests issued using the Cronet Library are asynchronous by default, which means that worker threads are not blocked while waiting for the request to come back.
- **Data compression**: Cronet supports data compression using the Brotli Compressed Data Format.

Create a Network Request

The library provides a *CronetEngine.Builder* class that can be used to create an instance of *CronetEngine*. The following example in Figure 3.14 shows how to create *CronetEngine* object.



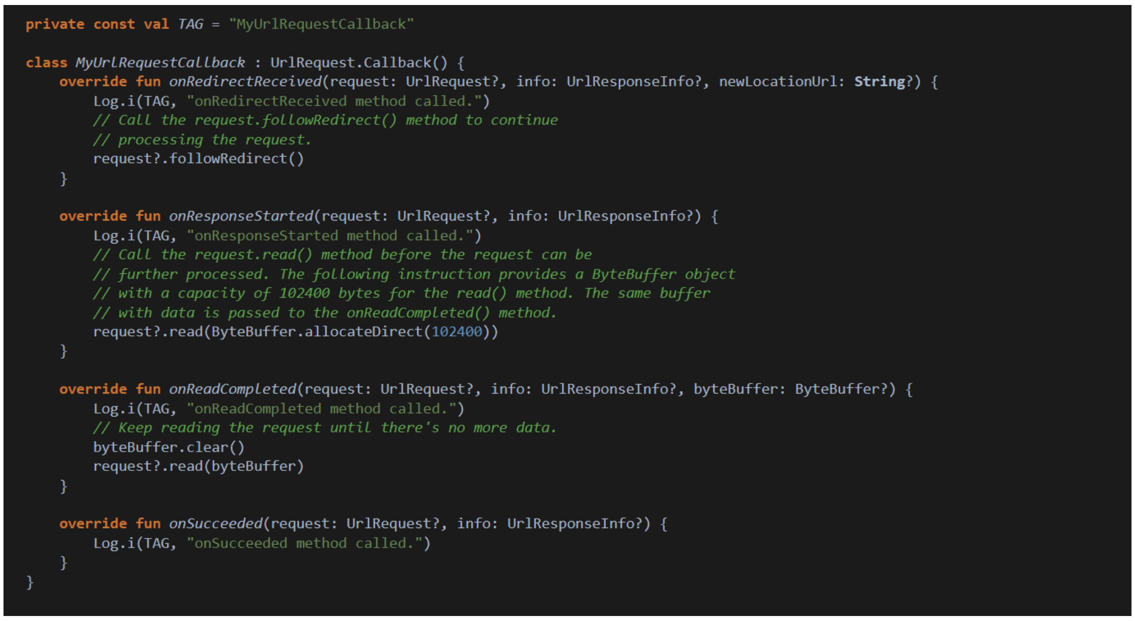
```
private val cronetEngine = CronetEngine.Builder(context)
    .enableHttp2(true)
    .enableQuic(true)
    .addQuicHint("storage.googleapis.com", 443, 443)
    .addQuicHint("www.googleapis.com", 443, 443)
    .build()
```

Figure 3.14: Cronet Engine Object

Also, we can use the Builder class to configure a *CronetEngine* object, for example we can provide options like cashing and data compression. HTTP2 and QUIC support is enabled by default. When both are enabled (and no hints are provided), Cronet tries to use both protocols and it's non-deterministic which one will be used for the first few requests. As soon as Cronet is aware that a server supports QUIC, it will always attempt to use it first.

Implementation of the Request Callback

To provide an implementation of the callback that handles events during the lifetime of the request, we create a subclass of *UrlRequest.Callback* and implement the required abstract methods, as illustrated in Figure 3.21.



```kotlin
private const val TAG = "MyUrlRequestCallback"

class MyUrlRequestCallback : UrlRequest.Callback() {
    override fun onRedirectReceived(request: UrlRequest?, info: UrlResponseInfo?, newLocationUrl: String?) {
        Log.i(TAG, "onRedirectReceived method called.")
        // Call the request.followRedirect() method to continue
        // processing the request.
        request?.followRedirect()
    }

    override fun onResponseStarted(request: UrlRequest?, info: UrlResponseInfo?) {
        Log.i(TAG, "onResponseStarted method called.")
        // Call the request.read() method before the request can be
        // further processed. The following instruction provides a ByteBuffer object
        // with a capacity of 102400 bytes for the read() method. The same buffer
        // with data is passed to the onReadCompleted() method.
        request?.read(ByteBuffer.allocateDirect(102400))
    }

    override fun onReadCompleted(request: UrlRequest?, info: UrlResponseInfo?, byteBuffer: ByteBuffer?) {
        Log.i(TAG, "onReadCompleted method called.")
        // Keep reading the request until there's no more data.
        byteBuffer.clear()
        request?.read(byteBuffer)
    }

    override fun onSucceeded(request: UrlRequest?, info: UrlResponseInfo?) {
        Log.i(TAG, "onSucceeded method called.")
    }
}
```

Figure 3.15: Cronet Request Callbacks

- **onRedirectReceived()**: Invoked when the server issues an HTTP redirect code in response to the original request.

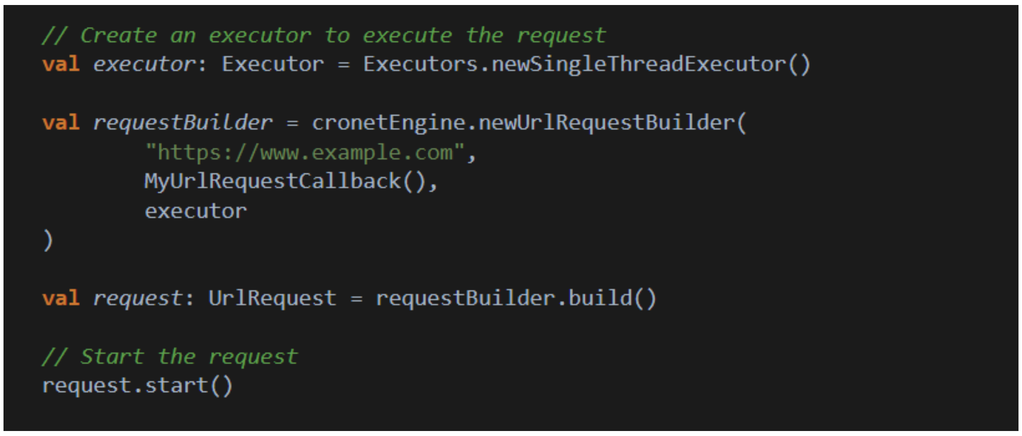
- **onResponseStarted()**: Invoked when the final set of headers is received. This method is only invoked after all redirects are followed.

- **onReadCompleted()**: Invoked whenever part of the response body has been read.

- **onSucceeded()**: Invoked when the network request is completed successfully.

Send a Network Request

Once we implement the request callback, we can then make a request by using a *UrlRequestBuilder* which will combine the URL, the callback, the executor, and the Cronet Engine. To start the network task, the *start()* method of the request is called (Figure 3.16).



```kotlin
// Create an executor to execute the request
val executor: Executor = Executors.newSingleThreadExecutor()

val requestBuilder = cronetEngine.newUrlRequestBuilder(
        "https://www.example.com",
        MyUrlRequestCallback(),
        executor
)

val request: UrlRequest = requestBuilder.build()

// Start the request
request.start()
```

Figure 3.16: Send a Network Request

Cronet Lifecycle

Before explaining the lifecycle of a Cronet request, there are a few concepts that are important to understand:

- **State**: A state is the particular condition in which the request is in at a specific time. *UrlRequest* objects created using the Cronet Library move through different states in their lifecycle. The request lifecycle includes an initial state, and multiple transitional and final states.
- **UrlRequest method**: Clients can call specific methods on *UrlRequest* objects depending on the state. The methods move the request from one state to another.
- **Callback method**: By implementing methods of the *UrlRequest.Callback* class, the app can receive updates about the progress of the request. The callback methods can be implemented to call methods of the *UrlRequest* object that take the lifecycle from a state to another.

The following list describes the flow of the *UrlRequest* lifecycle:

1. The lifecycle is in the **Started** state after the app calls the *start()* method.
2. The server could send a redirect response, which takes the flow to the *onRedirectReceived()* method. In this method, one of the following client actions may take place:
    - Follow the redirect using *followRedirect()*. This method takes the request back to the **Started** state.
    - Cancel the request using *cancel()*. This method takes the request to the *onCanceled()* method where the app can perform additional operations before the request is moved to the **Canceled** final state.
3. After the app follows all the redirects, the server sends the response headers and the *onResponseStarted()* method is called. The request is in the **Waiting for read()** state. The app should call the *read()* method to attempt to read part of the response body. After *read()* is called, the request is in the **Reading state**, where there are the following possible outcomes:
    - The reading action was successful, but there is more data available. The *onReadCompleted()* is called and the request is in the **Waiting for read()** state again. The app should call the *read()* method again to continue reading the response body. The app could also stop reading the request by using the *cancel()* method.

- The reading action was successful, and there is no more data available. The *onSucceeded()* method is called and the request is now in the **Succeeded** final state.
- The reading action failed. The *onFailed* method is called and the final state of the request is now **Failed**.

Figure 3.17 shows the lifecycle of a *UrlRequest* object.

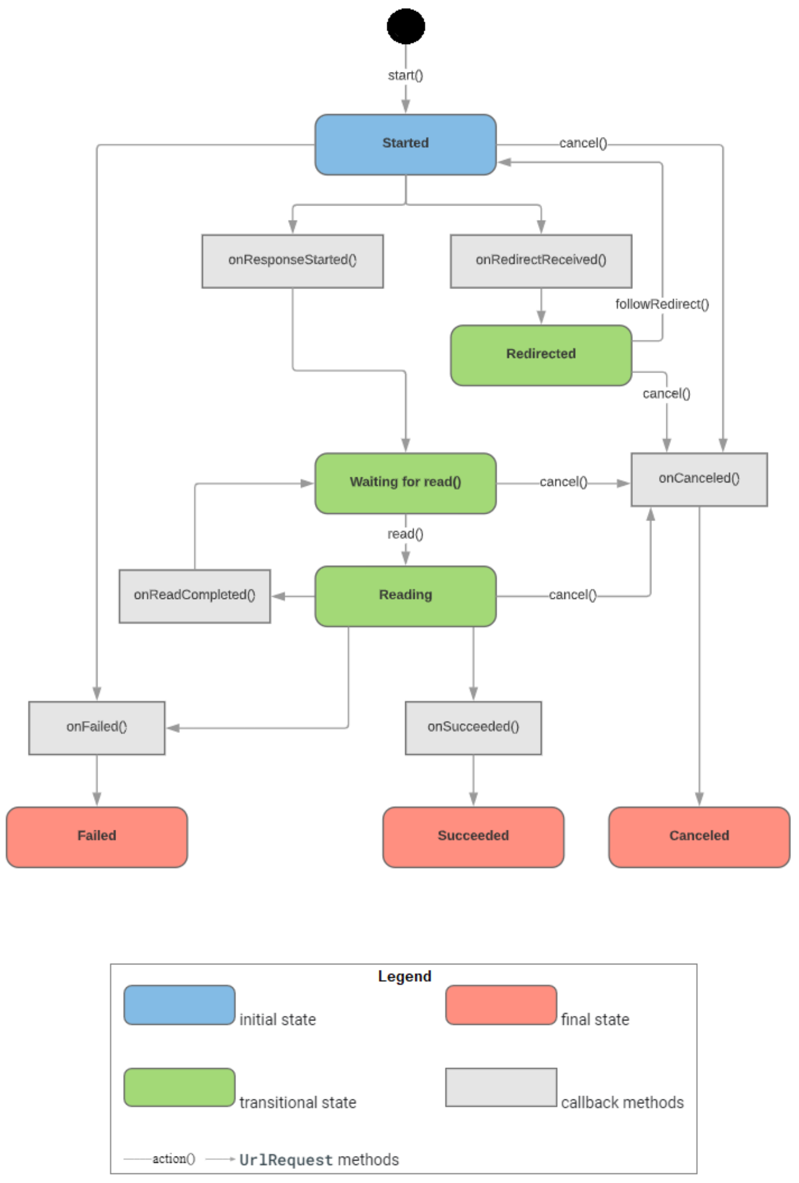


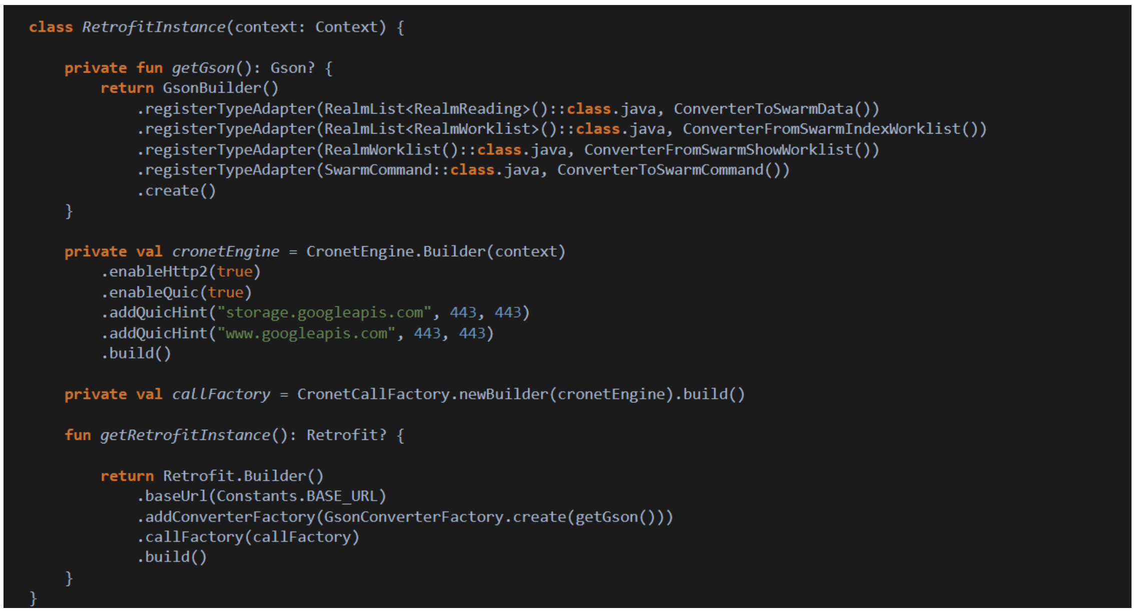
Figure 3.17: Cronet Lifecycle

Cronet with OkHttp and Retrofit

Cronet is a powerful and flexible tool also because it can be used in combination with other libraries, providing the best of utility, simplicity, and performance. The Cronet team provides a library that enables OkHttp users to use Cronet as their transport layer, benefiting from features like QUIC/HTTP3 support or connection migration. The library can also be used with other OkHttp-based libraries such as Retrofit, Coil, and others.

That feature allowed us to integrate Cronet into our Android application and enable QUIC support with zero migration cost. Instead of completely replacing our network stack that used the OkHttp library, the idea was to integrate the Cronet library under the OkHttp API framework. By performing the integration in this way, we avoided changes to the API layer of our network calls, which rely on Retrofit.

The new Retrofit instance, that integrates Cronet and enables QUIC is demonstrated in Figure 3.18.



```kotlin
class RetrofitInstance(context: Context) {

    private fun getGson(): Gson? {
        return GsonBuilder()
            .registerTypeAdapter(RealmList<RealmReading>()::class.java, ConverterToSwarmData())
            .registerTypeAdapter(RealmList<RealmWorklist>()::class.java, ConverterFromSwarmIndexWorklist())
            .registerTypeAdapter(RealmWorklist()::class.java, ConverterFromSwarmShowWorklist())
            .registerTypeAdapter(SwarmCommand::class.java, ConverterToSwarmCommand())
            .create()
    }

    private val cronetEngine = CronetEngine.Builder(context)
        .enableHttp2(true)
        .enableQuic(true)
        .addQuicHint("storage.googleapis.com", 443, 443)
        .addQuicHint("www.googleapis.com", 443, 443)
        .build()

    private val callFactory = CronetCallFactory.newBuilder(cronetEngine).build()

    fun getRetrofitInstance(): Retrofit? {

        return Retrofit.Builder()
            .baseUrl(Constants.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create(getGson()))
            .callFactory(callFactory)
            .build()
    }
}
```

Figure 3.18: Cronet Integration with Retrofit

3.3.2 Server Side - Elixir Application

As already stated, in this paper we have examined Elixir programming because of the many principles and advantages it brings. Regarding that, this section will provide both TCP and QUIC protocols development in Elixir using a set of libraries and tools such as OTP and GenTcp that are used to build the server in this way.

OTP Servers

OTP stands for the Open Telecom Platform [14]. It was initially used to build telephone exchanges and switches. But these devices have the same characteristics that are needed in any large application nowadays, so OTP is now a general-purpose tool for developing and managing large systems.

OTP is a great set of tools and libraries that Elixir inherits from Erlang. It defines systems in terms of hierarchies of applications. An application consists of one or more processes. These processes follow one of a small number of OTP conventions, called behaviors. There is a behavior used for general-purpose servers, one for implementing event handlers, and one for finite-state machines. Each implementation of one of these behaviors will run in its own process (and may have additional associated processes). Our EchoServer implements the server behavior, called GenServer.

GenTcp

GenServer is a behavior module that is used to implement the server of a client-server relation. A GenServer is a process, just like any other Elixir process. It can be used to manage state and execute code asynchronously and includes practical functionality around tracing and error reporting. The advantages of using GenServer are:

- It comes with a standard set of functions to introduce the functionality of both error tracking and reporting.
- It can also become a part of supervision.

GenServer is also an OTP protocol [15]. OTP works by assuming that the module defines a number of callback functions. In the case of GenServer, there are six callback functions. If we wrote the GenServer in Erlang, our code would have to contain implementations of all six.

When we add the line *use GenServer* to a module, Elixir creates default implementations of these six callback functions. All we have to do is override the ones where we add our own application-specific behavior.

GenServer's callback functions are:

- *init(start_arguments)*:

  Called by GenServer when starting a new server. The parameter is the second argument passed to start_link. It should return {:ok, state} on success, or {:stop, reason} if the server could not be started.

  An optional timeout can be specified using {:ok, state, timeout}, in which case GenServer sends the process a :timeout message whenever no message is received in a span of *timeout* ms. The message is passed to the handle_info function.

  The default GenServer implementation sets the server state to the argument passed.

- *handle_call(request, from, state)*:

  Invoked when a client uses GenServer.call(pid, request). The from parameter is a tuple containing the PID of the client and a unique tag. The state parameter is the server state. On success it returns {:reply, result, new_state}.

  The default implementation stops the server with a :bad_call error, so handle_call needs to be implemented for every call request type the server implements.

- *handle_cast(request, state)*:

  Called in response to GenServer.cast(pid, request). A successful response is {:noreply, new_state}. It can also return {:stop, reason, new_state}. The default implementation stops the server with a :bad_cast error.

- *handle_info(info, state)*:

  Called to handle incoming messages that are not call or cast requests. For example, timeout messages are handled here. So are termination messages from any linked processes. In addition, messages sent to the PID using send (so they bypass GenServer) will be routed to this function.

- *terminate(reason, state)*:

  Called when the server is about to be terminated.

- *code_change(from_version, state, extra)*:

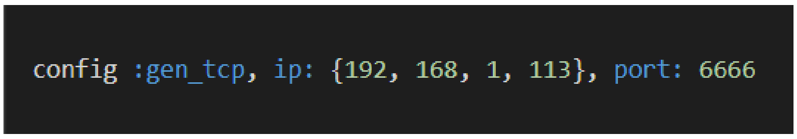
  Updates a running server without stopping the system. However, the new version of the server may represent its state differently from the old version. The code_change callback is invoked to change from the old state format to the new.

- *format_status(reason, [pdict, state])*: Used to customize the state display of the server. The conventional response is [data: [{'State', state_info }]].

TCP Acceptor Implementation

Before starting the TCP acceptor implementation, a small configuration should be added to set the IP and port numbers inside the elixir configuration file (Figure 3.19).

The value of the IP is set to a tuple with four integers, and the port number is also an Integer. These values are going to be used inside the *Acceptor* and *SimpleGenServer* modules for creating a socket, receiving packets and sending the same packets back to the client.



```
config :gen_tcp, ip: {192, 168, 1, 113}, port: 6666
```

Figure 3.19: IP and Port Configuration

The *start_link* function shown in Figure 3.20 acts like an entry point for initiating the GenServer. Here, *Application.gen_env(:tcp_server, :ip, {192,0,0,113})* will fetch the values from the configuration file *config/config.ex*. For safe side, when the configuration is not found, the default parameters are sent here as well. Also, if the configuration file is overridden with *config/dev.ex* or *config/prod.ex* then, the values from the respective environment will be loaded.

The line *GenServer.start_link(MODULE, [ip, port], [])* will callback the server init function with a list of parameters [ip,port].

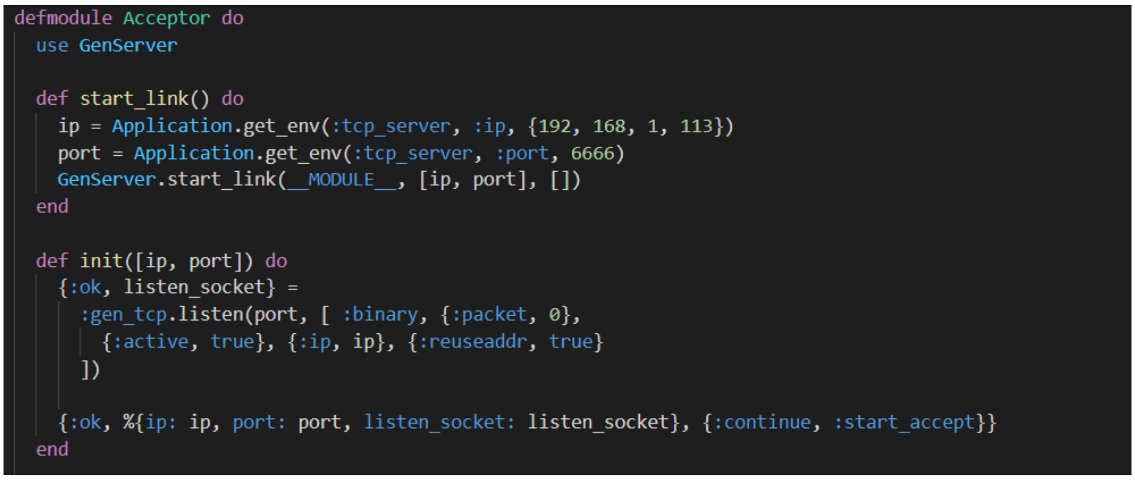
The line *{:ok, listen_socket} = :gen_tcp.listen(port, [:binary, {:packet, 0}, {:active, true}, {:ip, ip}, {:reuseaddr, true}])* inside *init()* function will set the socket on port.

- *:binary*: The received packet will be delivered as a binary . We also have a list option as an alternative for receiving the packets.

- *{ip, address}*: If the host has many network interfaces, this option specifies which one to listen on.

- *{:active, true}* If the value is true, which is the default, everything received from the socket is sent as messages to the receiving process.

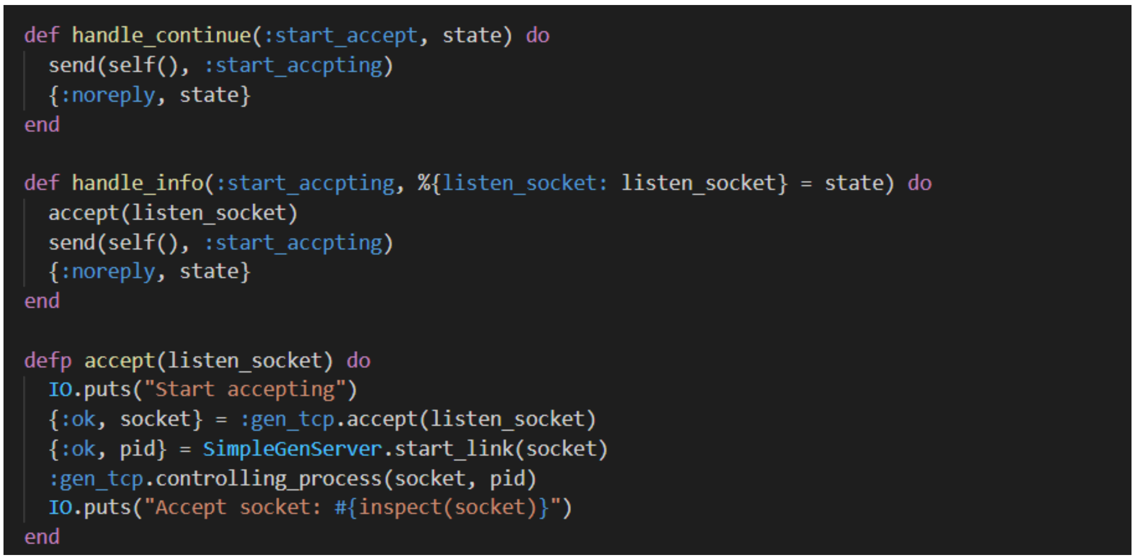
It is specified that the received packet will be delivered as a binary, and also there is a list option as an alternative for receiving the packets.



```
defmodule Acceptor do
  use GenServer

  def start_link() do
    ip = Application.get_env(:tcp_server, :ip, {192, 168, 1, 113})
    port = Application.get_env(:tcp_server, :port, 6666)
    GenServer.start_link(__MODULE__, [ip, port], [])
  end

  def init([ip, port]) do
    {:ok, listen_socket} =
      :gen_tcp.listen(port, [ :binary, {:packet, 0},
        {:active, true}, {:ip, ip}, {:reuseaddr, true}
      ])

    {:ok, %{ip: ip, port: port, listen_socket: listen_socket}, {:continue, :start_accept}}
  end
```

Figure 3.20: GenServer Acceptor

After the port, IP, and listening socket are defined, the *accept(listen_socket)* will accept the connection on the listening socket, and start a new process to handle the receiving packets - the SimpleGenServer, as illustrated in Figure 3.21.

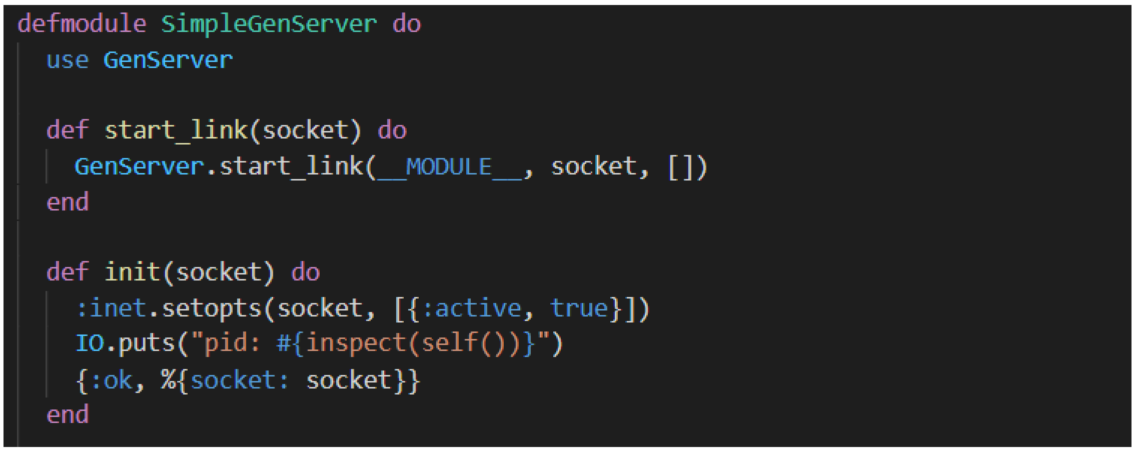


```
def handle_continue(:start_accept, state) do
  send(self(), :start_accpting)
  {:noreply, state}
end

def handle_info(:start_accpting, %{listen_socket: listen_socket} = state) do
  accept(listen_socket)
  send(self(), :start_accpting)
  {:noreply, state}
end

defp accept(listen_socket) do
  IO.puts("Start accepting")
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  {:ok, pid} = SimpleGenServer.start_link(socket)
  :gen_tcp.controlling_process(socket, pid)
  IO.puts("Accept socket: #{inspect(socket)}")
end
```

Figure 3.21: GenServer Acceptor

Sending the Packet to the Client

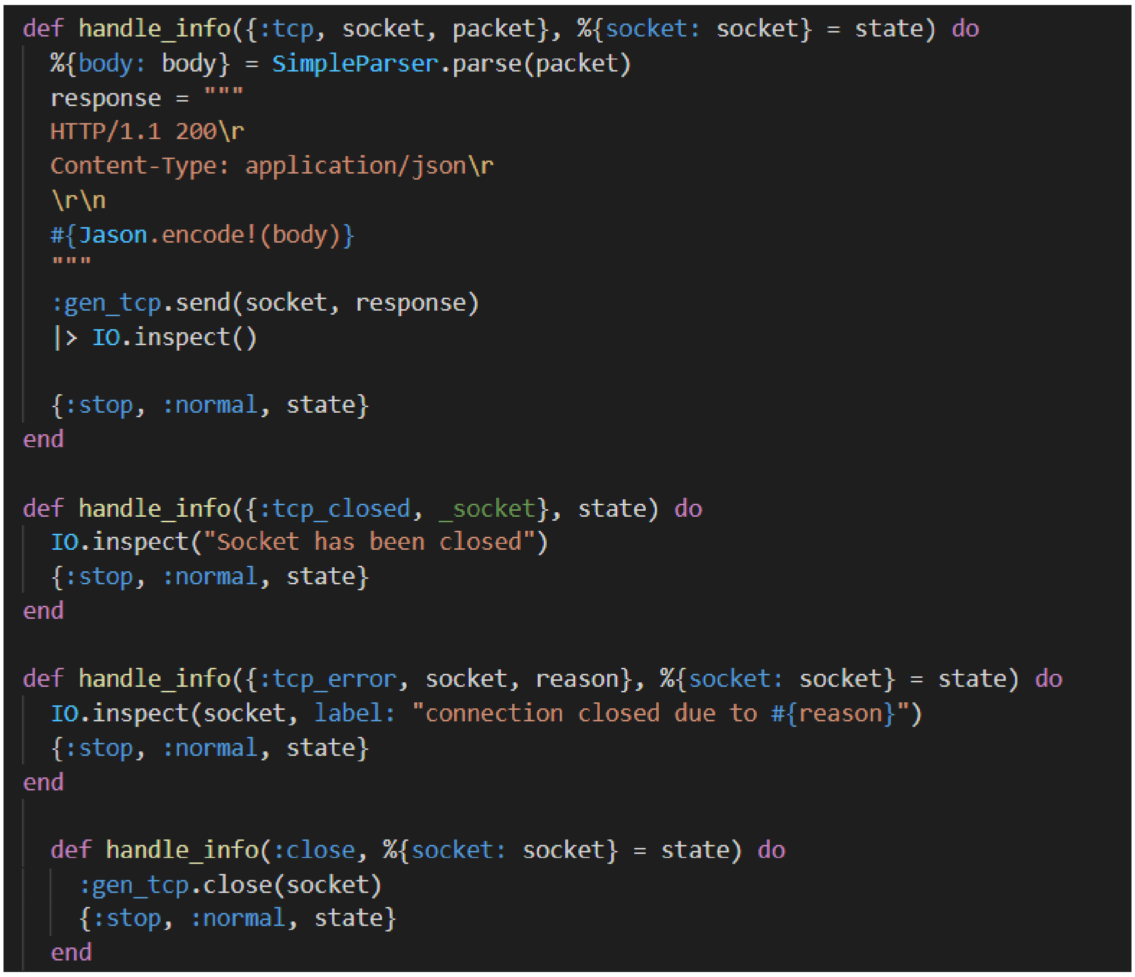
At the beginning, the SimpleGenServer fetch the socket passed and sets options needed (Figure 3.22).



```
defmodule SimpleGenServer do
  use GenServer

  def start_link(socket) do
    GenServer.start_link(__MODULE__, socket, [])
  end

  def init(socket) do
    :inet.setopts(socket, [{:active, true}])
    IO.puts("pid: #{inspect(self())}")
    {:ok, %{socket: socket}}
  end
```

Figure 3.22: SimpleGenServer Initialization

Furthermore, there are four *handle_info/2* functions defined and they are differentiated with pattern matching the actual message, as demonstrated in Figure 3.23. The first one is for receiving the actual messages with packets. Using *:gen_tcp.send* we are sending the return message to the client in the HTTP response format that we build using a simple parser module made for this purpose. The second one is for catching the socket close. The third *handle_info/2* function is used to catch the errors in sockets, while the last one closes the TCP socket when the error occurs.

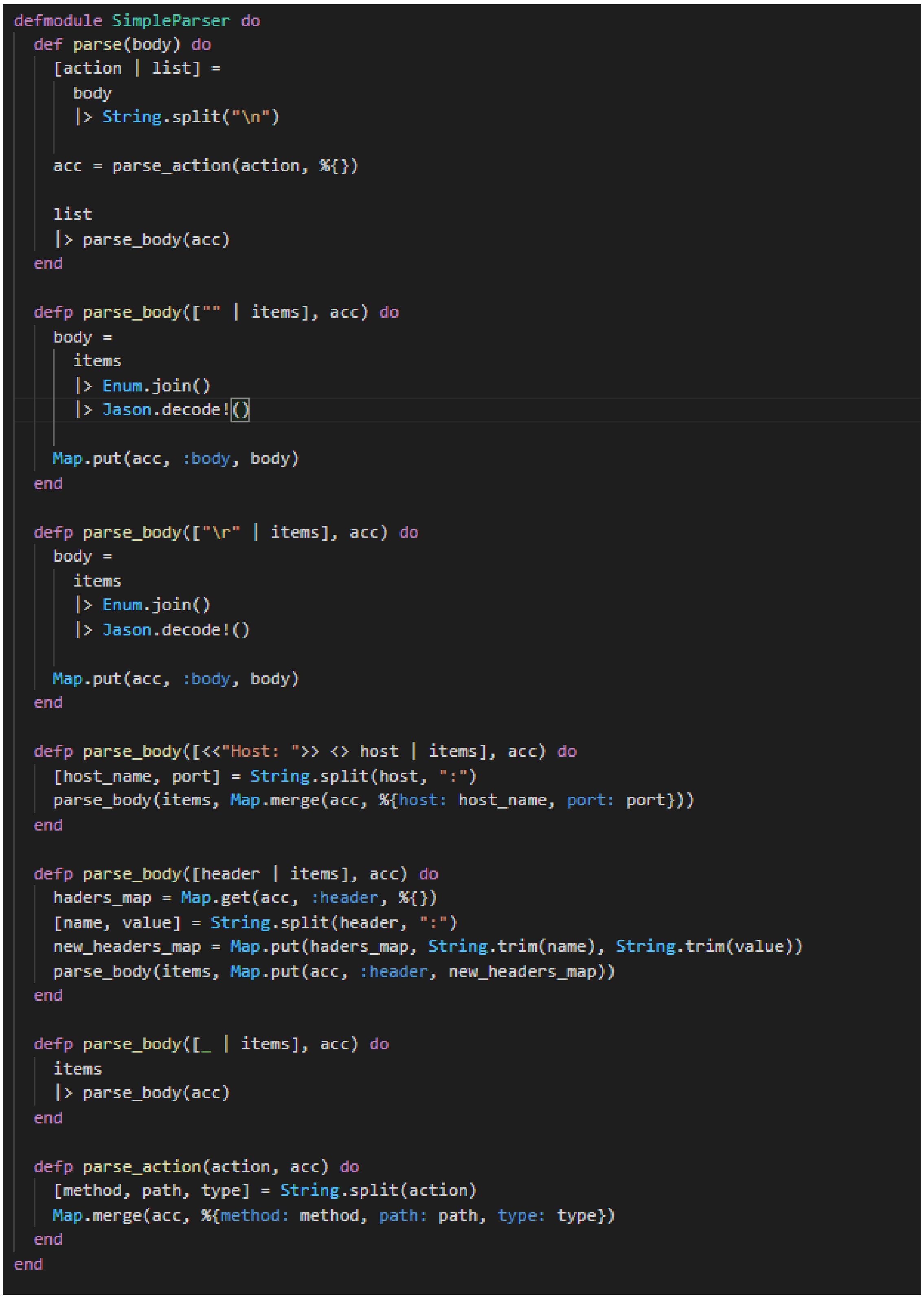


```elixir
def handle_info({:tcp, socket, packet}, %{socket: socket} = state) do
  %{body: body} = SimpleParser.parse(packet)
  response = """
  HTTP/1.1 200\r
  Content-Type: application/json\r
  \r\n
  #{Jason.encode!(body)}
  """
  :gen_tcp.send(socket, response)
  |> IO.inspect()

  {:stop, :normal, state}
end

def handle_info({:tcp_closed, _socket}, state) do
  IO.inspect("Socket has been closed")
  {:stop, :normal, state}
end

def handle_info({:tcp_error, socket, reason}, %{socket: socket} = state) do
  IO.inspect(socket, label: "connection closed due to #{reason}")
  {:stop, :normal, state}
end

  def handle_info(:close, %{socket: socket} = state) do
    :gen_tcp.close(socket)
    {:stop, :normal, state}
  end
```

Figure 3.23: GenServer Acceptor Callbacks

HTTP Body Parser

To enable the android application to properly communicate with the server and to understand the received message, that message has to be formatted in the right way. Since the android application implements a retrofit library for handling HTTP responses/requests, the message it receives must be structured like HTTP Response. Figure 3.24 shows the implementation of a simple parser, that parses the request, and takes the payload, which is then structured in HTTP repose and sent back to the client in JSON format.



```elixir
defmodule SimpleParser do
  def parse(body) do
    [action | list] =
      body
      |> String.split("\n")

    acc = parse_action(action, %{})

    list
    |> parse_body(acc)
  end

  defp parse_body(["" | items], acc) do
    body =
      items
      |> Enum.join()
      |> Jason.decode!()

    Map.put(acc, :body, body)
  end

  defp parse_body(["\r" | items], acc) do
    body =
      items
      |> Enum.join()
      |> Jason.decode!()

    Map.put(acc, :body, body)
  end

  defp parse_body([<<"Host: ">> <> host | items], acc) do
    [host_name, port] = String.split(host, ":")
    parse_body(items, Map.merge(acc, %{host: host_name, port: port}))
  end

  defp parse_body([header | items], acc) do
    haders_map = Map.get(acc, :header, %{})
    [name, value] = String.split(header, ":")
    new_headers_map = Map.put(haders_map, String.trim(name), String.trim(value))
    parse_body(items, Map.put(acc, :header, new_headers_map))
  end

  defp parse_body([_ | items], acc) do
    items
    |> parse_body(acc)
  end

  defp parse_action(action, acc) do
    [method, path, type] = String.split(action)
    Map.merge(acc, %{method: method, path: path, type: type})
  end
end
```

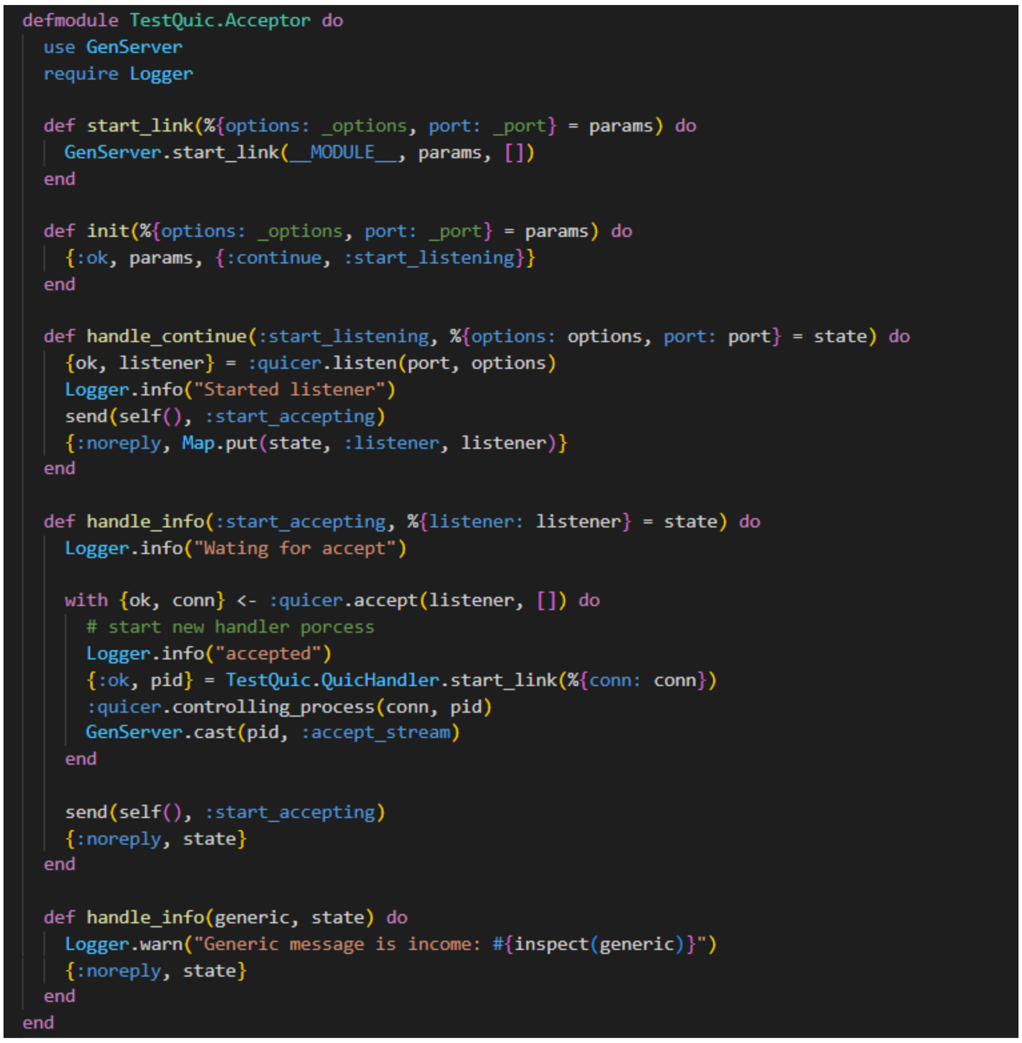
Figure 3.24: HTTP Parser

QUIC Server Implementation

To implement the QUIC protocol in Elixir, the best solution found so far is the Quicer[16] Erlang library. This library represents a wrapper over MsQuic[17], a Microsoft open source implementation of the QUIC protocol written in C, designed to be a cross-platform general purpose QUIC library optimized for client and server applications benefitting from maximal throughput and minimal latency.

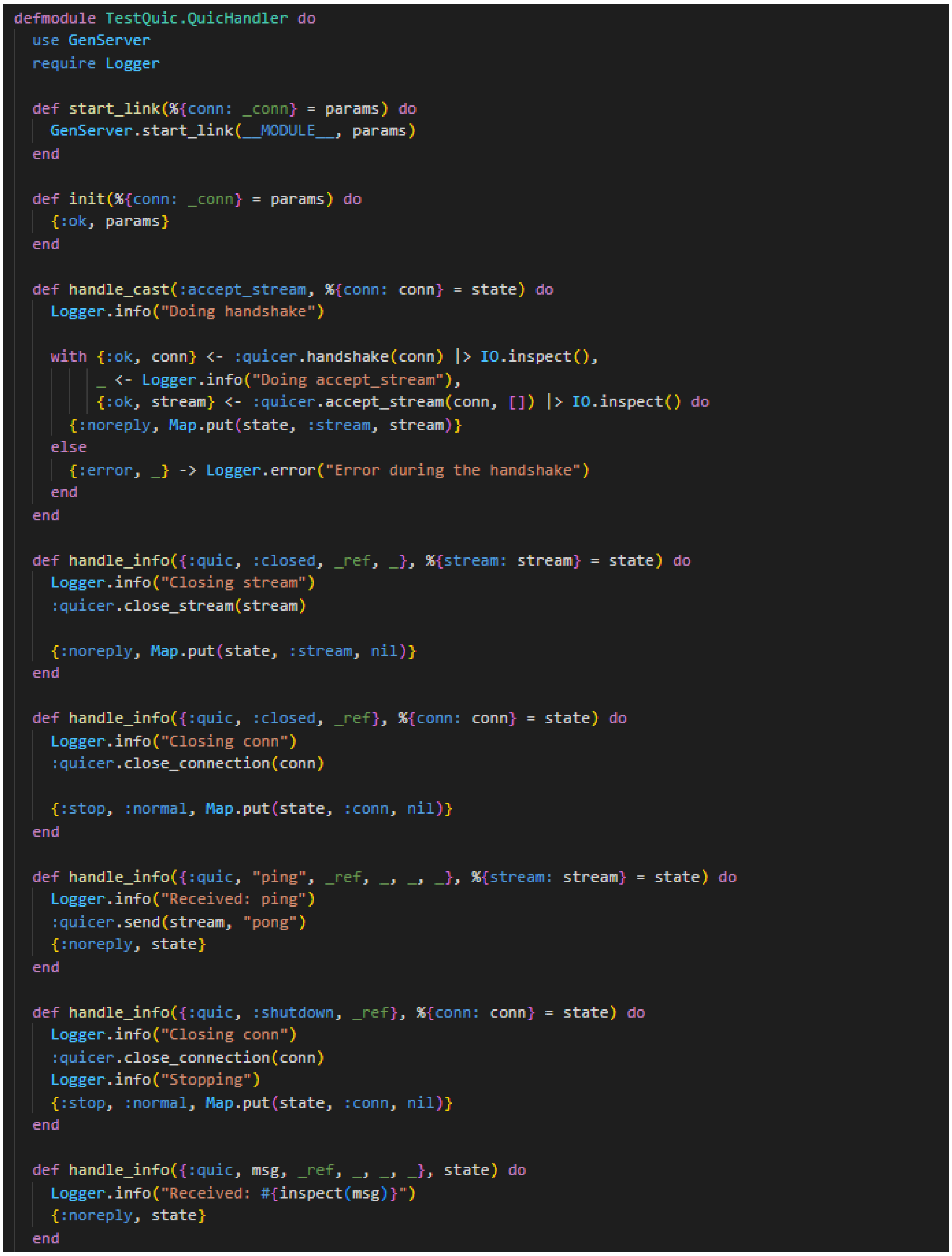
The QUIC server is split in two main modules:

- The **acceptor**, shown in Figure 3.25, accepts new connections.
- The **handler**, shown in Figure 3.26, manages the communication.



```elixir
defmodule TestQuic.Acceptor do
  use GenServer
  require Logger

  def start_link(%{options: _options, port: _port} = params) do
    GenServer.start_link(__MODULE__, params, [])
  end

  def init(%{options: _options, port: _port} = params) do
    {:ok, params, {:continue, :start_listening}}
  end

  def handle_continue(:start_listening, %{options: options, port: port} = state) do
    {ok, listener} = :quicer.listen(port, options)
    Logger.info("Started listener")
    send(self(), :start_accepting)
    {:noreply, Map.put(state, :listener, listener)}
  end

  def handle_info(:start_accepting, %{listener: listener} = state) do
    Logger.info("Wating for accept")

    with {ok, conn} <- :quicer.accept(listener, []) do
      # start new handler porcess
      Logger.info("accepted")
      {:ok, pid} = TestQuic.QuicHandler.start_link(%{conn: conn})
      :quicer.controlling_process(conn, pid)
      GenServer.cast(pid, :accept_stream)
    end

    send(self(), :start_accepting)
    {:noreply, state}
  end

  def handle_info(generic, state) do
    Logger.warn("Generic message is income: #{inspect(generic)}")
    {:noreply, state}
  end
end
```

Figure 3.25: QUIC Acceptor

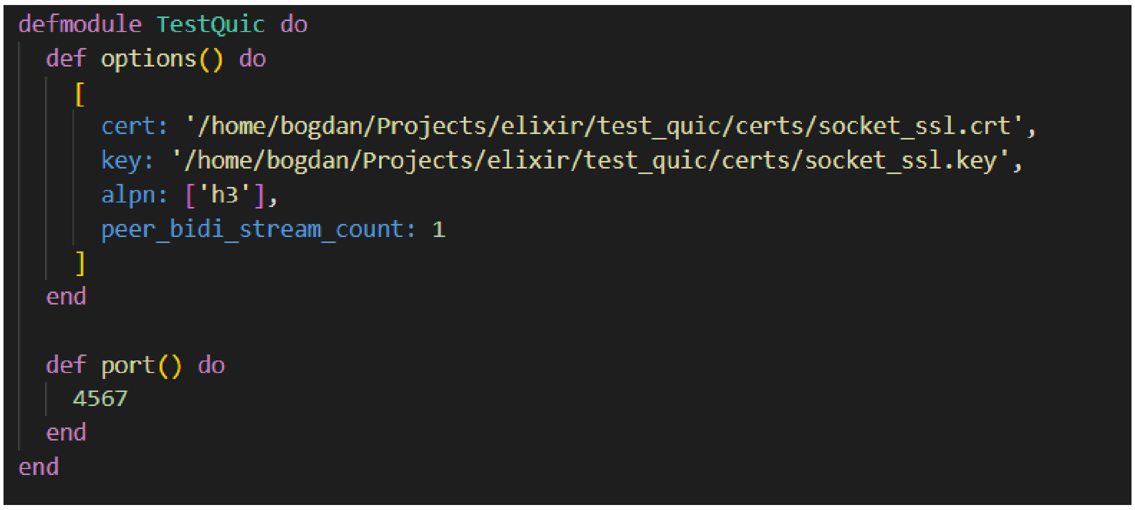


```elixir
defmodule TestQuic.QuicHandler do
  use GenServer
  require Logger

  def start_link(%{conn: _conn} = params) do
    GenServer.start_link(__MODULE__, params)
  end

  def init(%{conn: _conn} = params) do
    {:ok, params}
  end

  def handle_cast(:accept_stream, %{conn: conn} = state) do
    Logger.info("Doing handshake")

    with {:ok, conn} <- :quicer.handshake(conn) |> IO.inspect(),
         _ <- Logger.info("Doing accept_stream"),
         {:ok, stream} <- :quicer.accept_stream(conn, []) |> IO.inspect() do
      {:noreply, Map.put(state, :stream, stream)}
    else
      {:error, _} -> Logger.error("Error during the handshake")
    end
  end

  def handle_info({:quic, :closed, _ref, _}, %{stream: stream} = state) do
    Logger.info("Closing stream")
    :quicer.close_stream(stream)

    {:noreply, Map.put(state, :stream, nil)}
  end

  def handle_info({:quic, :closed, _ref}, %{conn: conn} = state) do
    Logger.info("Closing conn")
    :quicer.close_connection(conn)

    {:stop, :normal, Map.put(state, :conn, nil)}
  end

  def handle_info({:quic, "ping", _ref, _, _, _}, %{stream: stream} = state) do
    Logger.info("Received: ping")
    :quicer.send(stream, "pong")
    {:noreply, state}
  end

  def handle_info({:quic, :shutdown, _ref}, %{conn: conn} = state) do
    Logger.info("Closing conn")
    :quicer.close_connection(conn)
    Logger.info("Stopping")
    {:stop, :normal, Map.put(state, :conn, nil)}
  end

  def handle_info({:quic, msg, _ref, _, _, _}, state) do
    Logger.info("Received: #{inspect(msg)}")
    {:noreply, state}
  end
```

Figure 3.26: QUIC Handler

Both of them, the acceptor and message handler, are implemented using GenServer behaviour module, implementing its callbacks.

To start the QUIC server in Elixir we need to specify a set of options, such as a certificate, private hey, the HTTP protocol to use, and the port. Those options fields are demonstrated in Figure4.4b.



```
defmodule TestQuic do
  def options() do
    [
      cert: '/home/bogdan/Projects/elixir/test_quic/certs/socket_ssl.crt',
      key: '/home/bogdan/Projects/elixir/test_quic/certs/socket_ssl.key',
      alpn: ['h3'],
      peer_bidi_stream_count: 1
    ]
  end

  def port() do
    4567
  end
end
```

Figure 3.27: QUIC Options

3.4 Challenges Faced

At the beginning of the implementation phase, we had the idea to make benchmark tests between the client android application and the echo server developed in Elixir. Developed independently, both sides work as expected using TCP and QUIC protocols. In the case of the client side, the android application has been tested using already existing servers available supporting both protocols, and for the Elixir server, the behavior was tested using Quicer[16] client side.

The real challenge started when we tried to establish the communication between our android application and elixir server when the QUIC protocol is enabled. Due to the novelty of the protocol, the Cronet Android library that enables the QUIC protocol in Android devices has been poorly documented, without giving real insight into configuration parameters, and debugging tools. Furthermore, QUIC's secure-by-default feature makes our network analysis and troubleshooting much more difficult and time-consuming, or sometimes even impossible.

For example, to achieve secure communication, we initially used the OpenSSL library to generate private keys, create certificate signing requests, and install SSL/TLS certificates. The secure communication between the Android application and the Elixir server set up in this way is established by the TCP protocol, and data exchange takes place smoothly. However, the communication fails during the handshake phase when QUIC protocol is enabled returning the generic error message ERR_QUIC _PROTOCOL _ERROR. Further analysis using available network tools revealed that the Cronet library does not implement the network security configuration specified when using a non-public Certificate Authority (CA), nor does it provide official documentation regarding this. Finally, as a solution, we got a global CA and trusted certificates using Let's Encrypt [18].

Even when established, communication did not work properly. So, after much consideration, the decision was to change the approach and run the benchmark tests using our Android app and existing public servers providing REST APIs, supporting both TCP and QUIC protocols. Although it was not our initial idea, we believe that the new approach gives us even more realistic results. Using the public Internet and not in an isolated testbed, we ensured that we observe the quality of the connection in the real world and we got closer results that can be obtained in a real working environment. At the same time, we will continue developing and improving the existing client-server applications so we can conduct further experiments to get further insight, with more parameters and scenarios tested.

4 Experiments and Result Analysis

In this section, we present details about the testbed, the network environment, and the experimental scenarios that we have evaluated, as well as the results obtained. We conduct a comprehensive measurement study about the performance of QUIC when compared to TCP in Android devices measuring request completion time for various object sizes, in different network conditions, and for both up-link and down-link traffic.

4.1 Testbed

The testbed consists of an android smartphone, a desktop computer serving as a control, and a remote server that enables both TCP and QUIC protocols. The android smartphone is connected to both a Wi-Fi network and a cellular network, as shown in Figure4.4b. At the end of the testbed is a server machine, that provides both TCP and QUIC protocols, and that responds to GET and POST requests. The main functionality of the control mechanism is to provide an additional view into the network conditions and android device by logging the latency for each request to the server.

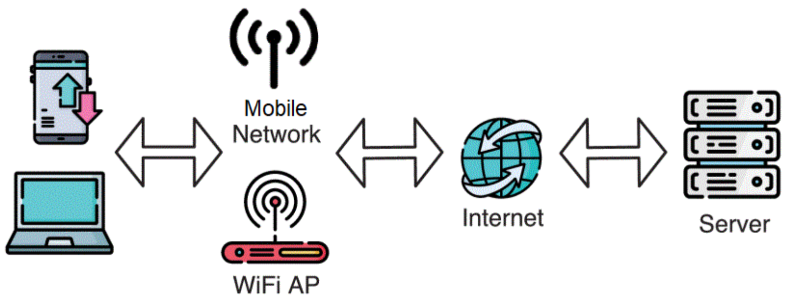


Figure 4.1: Testbed

4.2 Network Environment

We experimented with the two connectivity types found in smartphones, Wi-Fi, and mobile networks. The Wi-Fi network used for testing was an IEEE802.11ax, while for the mobile networks we chose a 2G connection with a very limited bandwidth of 64 Kbps, in order to simulate real-life work conditions of AMR android applications in rural areas and hard-to-reach places. Moreover, by conducting our experiments over the public internet and not in an isolated testbed, we ensured that we observed real-world connection quality.

4.3 Metrics and Experimental Scenarios

The primary metric for performance evaluation in this work is the Request Completion Time (RCT). As the name suggests, it is the total amount of time elapsed between calling the GET/POST method and the arrival of the complete response/acknowledgment to the application. We evaluate the RCTs for requests of different sizes in various network connectivity environments. Each experimental scenario starts with the control machine installing the android application on the Android device. The control machine then executes the application that, using the Cronet library, does a TCP or QUIC request. In both cases, the same Cronet library was used, with different configurations to enable or disable QUIC protocol, according to the test conducted. For POST requests, the application data was stored internally in JSON format and sent in the body of the request. On the other hand, the server machine was responding to GET requests from Android devices by sending data in JSON format, and in .jpg format.

4.4 Measurements

4.4.1 Google API

The first experiment was conducted using Google API, requesting images using both TCP and QUIC protocols. Each image size and results obtained using a Wi-Fi connection and mobile network can be seen in Table 4.1 and

Table 4.2 respectively, as are graphical representations in Figure 4.2. Images received are simple placeholders in the range from 7kB to 60kB, while the acquired RCT values represent the amount of time elapsed between calling the GET method and the arrival of the complete response to the application.

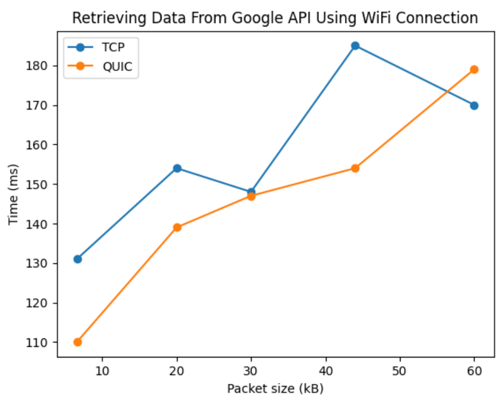
The tests show that in terms of Wi-Fi connection QUIC outperforms TCP in most cases and for most packet sizes tested. Also, it is interesting to note that the difference between the performance of the two protocols decreases as the packet size increases until it reaches the point of being equal when the packet size is 60 kB. Moreover, when the mobile device is connected to the mobile data network, the difference becomes more significant. In this regard, the reception of the smallest packet of 7kB is performed 32% faster using QUIC protocol, than TCP. On the other hand, for larger packet sizes, the values are getting more equal, and for the packets of 60kB TCP even outperforms QUIC around 9%.

Packet Size	TCP	QUIC
7kB	131ms	110ms
20kB	154ms	140ms
30kB	150ms	147ms
45kB	185ms	155ms
60kB	170ms	170ms

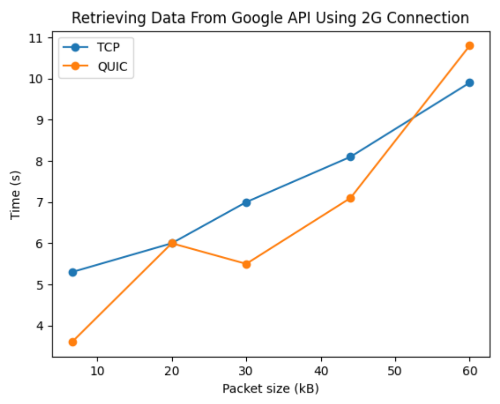
Table 4.1: Retrieving data From Google API Using WiFi

Packet Size	TCP	QUIC
7kB	5.33s	3.62s
20kB	6.12s	6.01s
30kB	7.11s	5.54s
45kB	8.24s	7.12s
60kB	9.93s	10.91s

Table 4.2: Retrieving data From Google API Using 2G



(a) Wi-Fi Network



(b) Mobile Network

Figure 4.2: Retrieving Images

4.4.2 JSON Server

The second set of experiments was conducted performing GET and POST requests to JSON Server that provides full REST API. In this case, the data was in JSON format and both TCP and QUIC protocols were compared.

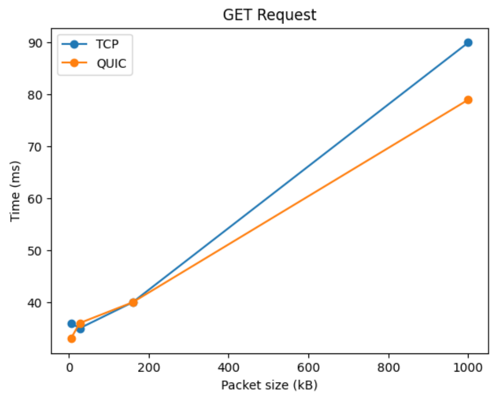
Table 4.3 and Table 4.4 give the insight of packets size and RCT obtained in different network environment when GET request is performed. Figure 4.3 graphically illustrates acquired results. Given values make us know that using a high bandwidth network during Wi-Fi connection, both protocols perform similarly, whereas QUIC gives better results for the smallest packet sizes around 8%. While on the other side, using a 2G connection the difference is much more remarkable and for the same packet size gives better results for up to 19%. Additionally, using mobile 2G network QUIC outweighs TCP for all packet sizes on an average of 16%.

Packet Size	TCP	QUIC
5.5kB	36ms	33ms
27.5kB	35ms	36ms
160kB	40ms	40ms
1Mb	90ms	79ms

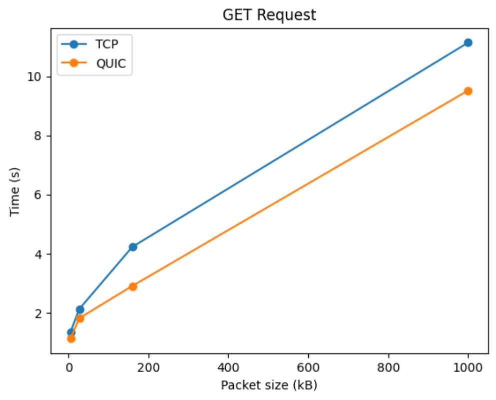
Table 4.3: Get Request Using WiFi

Packet Size	TCP	QUIC
5.5kB	1.35s	1.13s
27.5kB	2.13s	1.82s
160kB	4.23s	2.91s
1Mb	11.14s	9.52s

Table 4.4: Get Request Using 2G



(a) Wi-Fi Network



(b) Mobile Network

Figure 4.3: GET Request

The Table 4.5 and Table 4.6 give the insight of packets size and RCT obtained in different network environment when POST request was performed. The graphical representation of data is shown in Figure 4.4.

Although better in all cases using both network connectivity scenarios, QUIC gives particularly better results when the network bandwidth is limited. It means that for smaller packet sizes, using a 2G connection we acquire 17% better results using QUIC than TCP and compared to Wi-Fi RCT results for the same packet size where

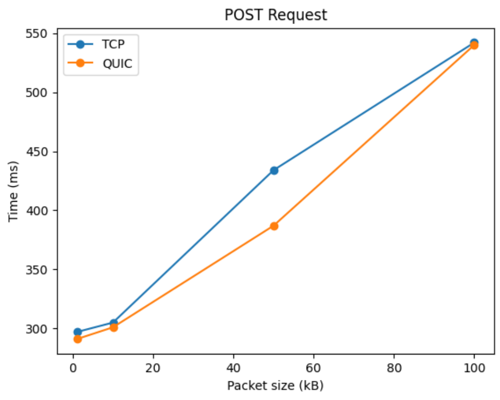
QUIC performs only 2% better. The same applies when larger packet sizes are used for POST requests. During Wi-Fi connection, QUIC performs better for no more than 0.3% in respect to 7% in 2G mobile connection.

Packet Size	TCP	QUIC
1kB	297ms	291ms
10kB	305ms	301ms
50kB	434ms	387ms
100kB	542ms	540ms

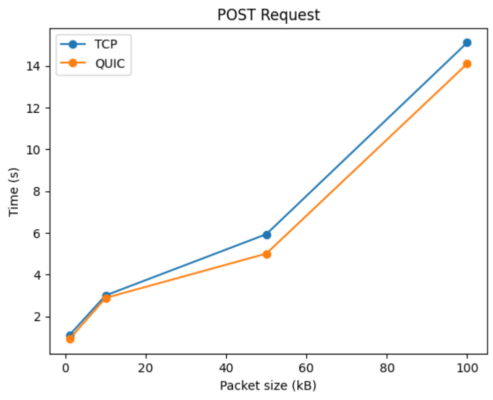
Table 4.5: Post Request Using WiFi

Packet Size	TCP	QUIC
1kB	1.11s	0.92s
10kB	3.01s	2.89s
50kB	5.94s	5.01s
100kB	15.11s	14.10s

Table 4.6: Post Request Using 2G



(a) Wi-Fi Network



(b) Mobile Network

Figure 4.4: POST Request

5 Conclusion and Future Work

QUIC is a new network protocol that resides in the application layer over UDP. Google developed QUIC as an alternative to TCP. It improved the performance of a lot of key functionalities offered by the existing TCP + TLS + HTTP/2 stack, such as reducing connection establishment latency and improving congestion control, along with introducing new features like connection migration and multiplexing without head-of-line blocking.

In this work, we wanted to test our hypothesis that adopting QUIC would bring a lot of benefits to both AMR solutions. As previously discussed, Automated Meter Reading can be implemented using vehicle-mounted or hand-held receivers to capture the signals transmitted from the smart water meters.

Our benchmark tests conducted show that QUIC outperformed HTTP/2 over TCP/TLS when it comes to smaller packets exchange between the Android client and the server. It can be directly applied to the Drive-By AMR solution, in which the android device is constantly exchanging data with the server, sharing location data, and device-specific properties, but also getting optimized navigation routes. It would result in improvement in communication and reducing the latency by up to 16% for performing GET requests, and up to 5% for performing POST requests, in the case of good network connectivity. On the other side, in the case of stable and reliable networks, the benefits of QUIC are not so obvious.

Regarding Walk-By solutions that are designed for smart water meter data collection in locations that are inaccessible to vehicles, the outcome is even more significant. Our results demonstrate that QUIC performs well under high latency conditions, in particular for low bandwidth which can be experienced in rural areas with bad network infrastructure, reducing RCT in POST requests by 17% compared to TCP for small packet sizes and 15% for performing GET requests for packets size of 1Mb. In practice, it would mean a significant improvement for downloading necessary data from the server as well as regularly updating it when operating in a challenging environment, with poor network connectivity.

The results we obtained give us a better idea of QUIC adoption in AMR applications, and new ideas for future work. First of all, in order to apply QUIC to the existing system and apply it in a real application environment, it is necessary to enable client-server communication with the existing Cronet library on the Android side and the

Quicker Erlang library on the server. After that, all the conducted tests will be done again and the results will be compared.

Furthermore, the current work should be further extended in the form of testing a larger range of packet sizes and network properties, as well as comparing scenarios with different levels of bandwidth, packet loss, and delay. Also, we plan to evaluate the performance of QUIC in terms of connection migration and conduct several experiments to investigate the role of QUIC and TCP in overall RCTs for this aspect.

In addition to this, knowing that the client application is installed on an Android device, one of the more important parameters to consider before implementing the protocol in existing AMR solutions is battery consumption. In this regard, our field of interest for future testing will be to compare the two protocols during high request frequency.

With all results obtained so far and ideas for future work, we consider this research as a good starting point and a solid baseline for the further investigation and adoption of QUIC as an important player protocol in the IoT world and especially in Android mobile devices for smart water metering tasks.

References

[1] M. Thomson, *Version-Independent Properties of QUIC.* IETF, 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8999.txt

[2] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport.* IETF, 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9000.txt

[3] M. Thomson and S. Turner, *Using TLS to Secure QUIC.* IETF, 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9001.txt

[4] J. Iyengar and I. Swett, *QUIC Loss Detection and Congestion Control.* IETF, 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9002.txt

[5] E. A. M. Bishop, *HTTP/3.* IETF, 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9114

[6] K. Ervasti, *A Survey on Network Measurement: Concepts, Techniques, and Tools*, 2017. [Online]. Available: https://www.cs.helsinki.fi/u/kervasti/projects/A%20Survey%20on%20Network%20Measurement%20-%20Concepts,%20Techniques,%20and%20Tools%20-%20Kim%20Ervasti%20-%2031-12-2016.pdf

[7] P. B. J. S. Sathiya Kumaran Mani, Ramakrishnan Durairajan. Mntp: Enhancing time synchronization for mobile devices. https://dl.acm.org/doi/abs/10.1145/2987443.2987484.

[8] D. Mills. Network time protocol. https://www.rfc-editor.org/rfc/rfc958.

[9] D.Mills. Simple network time protocol. https://www.rfc-editor.org/rfc/rfc1769.

[10] D. Thomas, *Programming Elixir 1.6.* The Pragmatic Programmers, 2018. [Online]. Available: https://vdoc.pub/documents/programming-elixir-16-65g4gq8ttip0

[11] Retrofit. https://square.github.io/retrofit/.

[12] Livedata. https://developer.android.com/topic/libraries/architecture/livedata.

[13] Android dbms performance comparison. https : / / github . com / Luja93 / AndroidDBMSPerformanceBenchmark.

[14] H.-C. H. Feng-Cheng Chang. A design approach for software robustness. https://ieeexplore.ieee.org/document/9391977.

[15] Genserver elixir. https://hexdocs.pm/elixir/1.13/GenServer.html.

[16] Emqx. Quicer. https://github.com/emqx/quic.

[17] Microsoft. Msquic. https://github.com/microsoft/msquic.

[18] Letsencrypt. https://letsencrypt.org/.