FlyTrap

Decentralised Blockchain Authorization & Auditing Architecture for IoT and MQTT Brokers

Konrad M. Dryja

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master in Science
of the
University of Aberdeen.

Department of Computing Science

2020

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

[illegible]

Date: May 6, 2020

Abstract

In this dissertation I present a novel approach introducing Authentication, Authorisation and Accountability to MQTT Brokers with the main target being IoT sensors. IoT (thus MQTT) and Blockchain are emerging technologies that are gaining popularity in a lot of industries due to their unique features and design, including potential 100% uptime, transparent & immutable transaction history and attached cryptocurrency opening doors for monetisation of data collected by the sensors. Recent changes in legislation regarding personal data has also been a motivator for this project.

FlyTrap - introduced in this project - is a framework that aims to combine distributed ledgers with MQTT protocol to enhance it with a middleware capable of improving the authentication and accountability - which at the moment is very limited with the standard implementation. It is composed of four layers: Blockchain Layer, Broker Layer, Presentation Layer and Client Layer - loosely coupled with each other. It upgrades the standard MQTT protocol with features such as: monetising data access, securing particular topics with public / private key pairs, persistent audit logging on blockchain, daily usage reports and fending off primitive denial-of-service type of attacks - all deployed on a distributed Ethereum network.

Several evaluation scenarios were considered to determine how significant is the performance loss due to the extra layer of security, the total operational cost on the public blockchain and the potential to scale up with an increased number of connections. The performance loss was observed to be around 20% (compared to vanilla MQTT Broker), cost of operations ranged from \$0.02 to \$2.10 and FlyTrap was having constant impact with increased number of connections, indicating great potential to scale up. The tests also reflected back to the use-case scenarios, to determine whether the stated problems are appropriately addressed.

At the end, the dissertation includes a discussion between proof-of-authority and proof-of-work networks, suggesting that the former might be more interesting for business more interested in data's secrecy than transparency and the latter for the opposite scenarios - where data transparency plays the biggest role. The project should also be followed up with an additional future work concerning the overall impact of TLS encryption, usage of WebSocket (as opposed to plain TCP/TLS) and looking into verification of devices authenticity.

The paper is concluded with promising results, encouraging further testing & usage, pointing out that the combination of the blockchain technologies with IoT might be able to provide robust architectures with distributed data layer and transparent transaction logs. Even though the produced software is a proof-of-concept, it is ready for deployment in production systems.

Acknowledgements

First and foremost, I would like to thank Prof Peter Edwards and Dr Milan Markovic for their incredible help supervising my project. They were always happy to answer my questions and always found time to accommodate any additional requests. I could always count on their accurate and critical feedback. I could not have imagined to have anyone better guide me in this final year of my University adventure.

I would also like to acknowledge Computing Science class 2019, which I was part of until the very end (before departing on my placement) along with class 2020. The countless days spent on studying for Robotics, Algorithmic Problem Solving or Security.

To my parents who sacrificed a lot to give me the future I am living in right now. I am beyond grateful and probably will never be able to repay.

My entire team from Google Zurich: Caroline Muthambiri, Joe Coyne, Patrick Oberholzer - and especially Edu Campillo - where I was working during my industrial placement. You made me feel like part of the bigger effort and you inspired me to keep pushing beyond my limits. If it wasn't for you, I'd still be stuck trying to make Python2.7 happen, instead of moving forward. I look forward to working alongside you.

Dr Liana Marinescu for being an incredible help, always supportive regardless of circumstances and always ready to sacrifice the day to cheer others up. I don't think I would be able to push through my conversion and my final year of University without you. You're a treasure.

Last but not least, Mikko. For the late-night tech discussions. For the takeaways in Meston. For countless failed projects. For struggling through our internships. I bid you farewell as my University colleague and will welcome you very soon as a coworker.

And to everyone else I forgot to mention, but were part of my journey - thank you!

Contents

Word count: 17382

List of Figures

Abbreviations

ABI Application Binary Interface

ACL Access Control List

API Application Programming Interface

BTC Bitcoin

CCPA California Consumer Privacy Act

CLI Command Line Interface

DPA Data Protection Act

ECDSA Elliptic Curve Digital Signature Algorithm

ETH Ether

EVM Ethereum Virtual Machine

GDPR General Data Protection Regulation

IoT Internet of Things

JSON Javascript Object Notation

MQTT Message Queuing Telemetry Transport

PII Personal Identifiable Information

PoA Proof-of-Authority

PoS Proof-of-Stake

PoW Proof-of-Work

QoS Quality of Service

RFID Radio-Frequency Identification

TCP Transmission Control Protocol

TLS Transport Layer Security

Chapter 1

Introduction

In the introductory chapter, we introduce FlyTrap as a solution to insufficient authentication and accountability options for MQTT protocol. Main motivators are also discussed along with outlining the benefits provided over state-of-the-art & main goals and no-goals of the dissertation. At the end, project's structure can also be found.

1.1 Overview

This section outlines the main concepts discussed in the project.

1.1.1 Internet of Things

Internet of Things, also known as IoT, is a growing field within technical industries and computer science. It is a notion first coined in **?**] where the main focus was around RFID (radio-frequency identification) tags - which is a simple electromagnetic field usually created by small-factor devices in the form of a sticker capable of transferring static information, such as a bus timetable or URL of a website (e.g. attached to a poster promoting a company or an event). Ashton argued the concern of data consumption and collection being tied to human presence at all times. In order to mine information, a human was first required to find the relevant data source, which then could be appropriately evaluated. However, as it was accurately pointed out, people have limited resources & time, and their attention could not be continuously focused on data capture. Technologists suggested delegating the task to the machines themselves; altogether remove the people from the supply chain. A question was asked, whether "things" could collect data from start to finish. That paper is known to be the first mention of IoT and building stone, de facto defining it as an interconnected system of devices communicating with each other without the need for manual intervention.

With time and ever-expanding presence of smartphones, personal computers and intelligent devices, the capabilities of those simple RFID tags were also growing beyond just simple static data transmission functionalities. Following the observation by **?**], the size of integrated circuits was halving from year to year, allowing us to put more computational power on devices decreasing in size. They were now not only capable of acting as a beacon, but actively processing the collected information (for example, temperature) and then passing it along to a more powerful computer which then could make decisions on whether to increase or decrease the strength of radiators at home - all without any input from the occupants. Eventually, IoT found its way to fields and areas such as households (smart thermostats or even smart kettles), physical security (smart motion sensors and cameras) or medicine (smart pacemakers). The scope is expected only to grow in the future. Data from Juniper Research [?] and CISCO [?] suggests that the total number of IoT devices might reach 50 billion by 2023.

1.1.2 Security of Data

The growing presence of smart-devices significantly increased the convenience and capabilities of "smart-homes" - at the same time, IoT also started handling more and more sensitive data - especially considering the last example from the previous paragraph. Scientists from the University of Massachusetts successfully performed an attack on a pacemaker [?], reconfiguring the functionality, which - if performed with malicious intents - could have tragic consequences. Nevertheless, even less extreme situations, such as temperature readings at home, are nowadays heavily regulated by data protection laws. Examples being the General Data Protection Regulation (GDPR) introduced by ?] or California Consumer Privacy Act by ?]. Collection of data is required to be strictly monitored and frequently audited in case of a breach - which also includes restrictions on the collection of Personal Identifiable Information (PII, as per GDPR). Those and more put an obligation on every company willing to exchange user data to govern the data appropriately and ensure its security - which includes data collected by Internet of Things devices.

1.1.3 MQTT

IoTs are usually low-power with limited computational capacity - mostly to decrease the required maintenance and ensure long-lasting life, without the need of replacing the power source (which is often a fixed battery) - meaning that only a minimum amount of work should be performed on the "thing" itself, in favour of sending it off to a centralised structure (e.g., a server hosted on the cloud) for further processing. One of the popular choices includes an intermediary, a broker, relaying communication between clients connected to it. That way, Peer-to-Peer connection is not required and can be wholly delegated to a separate backend server. A popular choice for the broker is the MQTT (Message Queuing Telemetry Transport)[1] standard defining the exact shape and form of TCP packets, handling unexpected timeouts & reconnects along with distributing channels of communication onto different topics containing separated information. From there, clients can either subscribe (i.e. consume) or publish (which can also be used for issuing commands) the data. Unfortunately, the OASIS standard introduces limited security capabilities (offering only username/password authentication) and no auditing or logging.

1.1.4 FlyTrap

This project also aims to develop a novel approach - further referred to as **FlyTrap** - for handling security in systems utilising MQTT brokers and their implementations, focusing on a platform-agnostic solution, capable of working with any MQTT-ready brokers - regardless of the implementation. FlyTrap does not depend on the specific software implementing the broker but instead aims to work with any broker that fully implements MQTT v5.0 standard. Furthermore, to ensure decentralised operation resistant to data breaches, downtime and full transparency, the Ethereum[2] platform is used as a data layer: capturing relevant interaction as publicly available transactions. In order to limit the quantity of data put on the blockchain (as computational and storage power there is limited), we also introduce several rules dictating logging of only specific events. The

[1]https://mqtt.org/
[2]https://ethereum.org/

system's purpose is to incorporate an **Authentication, Authorisation** and **Accountability** (AAA) framework to IoT devices communicating through MQTT.

In addition, to maximise the offered features of a blockchain architecture, FlyTrap will offer a solution to monetise distributed data and allow people using FlyTrap to put access to the data behind a fee, which would then be transferred back to the data owner, providing a potential business model to profit from.

1.2 Motivation

This section outlines the main problems that FlyTrap is attempting to address, along with motivation on why particular pieces of technology were used.

1.2.1 MQTT

MQTT v5.0 (as per the specification[3]) does not dictate nor specify any requirements regarding the security. It does offer an option of restricting some topics only to specific users, defined in access control lists (ACLs). The users then are required to provide a password when initiating a connection with the broker. Although, the basic username/password authentication is known to be cumbersome, only offering limited security. This also puts a burden on system administrators to maintain those ACLs in some centralised system, which then again is at risk of breaches or leakage. Moreover, placing the burden on a singular MQTT broker creates a single point of failure, where system downtime could halt the entire architecture.

1.2.2 Blockchain

By decentralising the data layer of the AAA framework and in the process, placing it on a distributed ledger, we can ensure maximised uptime and complete transparency of performed transactions. Events such as permission changes, failed authentication attempts can be recorded as a separate transaction which then could be audited by anyone knowing the public address of the system. This then could be handed over to authorities or auditing corporations to ensure that data is passed lawfully. Utilising Blockchain technologies also opens an opportunity to require payment (in the form of cryptocurrency) from potential consumers of data effectively expanding the business model.

Since here Blockchain is used just as a data layer, one could ask why not just use a centralised solution, such as SQL database. There is a lot of research (such as [**?** **?**]) into combining edge computation on IoT with Blockchain. Blockchain also offers 100% uptime, possibility to monetise the access to the data and full transparency of stored information, enhancing its trust, making it an interesting area to research. Data trust is also a problem that industries handling sensitive information are facing. With centralised solution, data integrity is completely up to the data owner, whereas on blockchain, it becomes immutable as soon as its submitted, strongly enhancing its transparency, as users are certain that no one has tampered with it.

1.2.3 Legislature

The rise of awareness of the necessity of data protection has also encouraged governments to introduce legal requirements (such as GDPR or CCPA) of data governance and face heavy fines in case of non-compliance. The MQTT standard and their implementation at the moment would

[3]https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

be considered non-compliant, due to effectively no way to trace past operations. General Data Protection Regulation requires entities handling user data to maintain proper retention of data and purge if requested by the data owner. MQTT at its current state is not capable of either setting data retention or purging all messages (whether already consumed or not), as messages are removed from the broker as soon as they are consumed, leaving no trace of “who” accessed “what” along with “why” they accessed it.

1.3 Goals

The project can be divided onto four main goals and two extras, leaving some field for manoeuvring in case of roadblocks or difficulties resulting from the challenges faced in the dissertation. By having soft targets, I was be able to stop sooner in case of overestimating the schedule, or carrying on with extra work, if I were to find myself meeting the targets quicker than expected.

Main Goals:

1. Design structure of blockchain network, relevant data models that would be placed on the blockchain and deployed on the Ethereum platform, capable of recording transactions and allowing for modification of ACLs, i.e. which wallets are permitted to access specific resources on the MQTT brokers.

2. Design rules that would be used for capturing the transactions. For example, a rule stating that if the client makes more than five consecutive, failed authentication attempts would be placed on a blacklist and that action would be added onto the blockchain as a transaction.

3. Design software acting as a secure proxy between brokers and connecting clients. This handles both authentication and log a performed action as an immutable transaction on a blockchain network. Logging is only be performed if the requested operation triggers some pre-defined rules.

4. Perform evaluation of the designed solution using an off the shelf MQTT broker and a range of experimental scenarios with a simulated network of MQTT clients.

Extra Goals:

1. Create public frontend for the auditors to freely access the contents of blockchain and thus transactions containing information about suspicious operations.

2. Generalise the implementation of the framework so it can be deployed with any broker following the MQTT standard, acting as an extension.

What this project is not trying to be:

- Design a new blockchain platform from scratch.

- Write / modify operating system of IoT devices.

- Design a new MQTT Broker.

1.4 Report Structure

The dissertation is divided onto seven chapters, each describing the following aspects of the project:

Chapter 1 **Introduction** chapter outlines the main motivation behind the project and introduce the notions used as building blocks in the design. It also list goals and no-goals defining success.

Chapter 2 In **Background & Related Work**, similar research and state of the art is described along with outlining the differences between them and this project. Furthermore, a thorough explanation of used software is also attached, such as what is blockchain, Ethereum, MQTT.

Chapter 3 **Requirements** includes analysis of both functional and non-functional requirements, main use-cases that are driving the project and provide stories inspired by real-life scenarios on the kind of problems that this project is trying to address.

Chapter 4 **Design** provides a thorough overview on the design of the project, explaining what components the software is composed of and detailing their operation & inter-connectivity.

Chapter 5 **Implementation** talks about the process of implementation of the design into software. It includes notions such as followed processes, used frameworks and sample code snippets.

Chapter 6 Inside **Evaluation** a comparison between state-of-the-art software, vanilla and FlyTrap are performed. Tests checking for performance impact, cost of operation on public blockchain, and whether common attacks can be detected/stopped are also run.

Chapter 7 **Discussion & Future Work** includes conclusions of the project, elements that were left-over, but beneficial for future iteration and all difficulties encountered throughout the dissertation.

Chapter 2

Background & Related Work

In this chapter, we discuss the main technologies used in the project including the MQTT protocol, existing MQTT brokers and Ethereum blockchain implementation. In addition, we overview selected related works which have attempted to address security of IoT networks through combining MQTT brokers and blockchains.

2.1 Relevant Legislation

The biggest motivator for this project are the recent changes in laws concerning how companies should be managing user data, its storage & treatment. Prior to 2018, data privacy in European Union was regulated by Data Protection Directive from 1995 [**?**]. With time, this piece of legislation started to become less and less relevant, as the Internet evolved. Moreover, the various data protection laws were often inconsistent across EU member states (as each country usually had their own legislation), there was no harmonised limitations or regulations and companies often suffered no major consequences (other than loss of trust by the customers) in case of leaking their information. This has changed severely in 2018, when General Data Protection Regulation (GDPR) [**?**] was introduced by the European Commission. Ever since the announcement of the planned legislation, lot of companies, such as Google rushed to ensure their compliance - but in the end to no avail, as both Facebook and Google suffered fines as high as $8.8 billion on the very first day when the bill became enforceable [**?**], indicating that it remains an unsolved challenge, calling for more robust solutions.

The regulation changed the perspective on how digital industry should be managing the information of their customers in a substantial way. As an example, Article 17 of GDPR, titled Right to erasure (or, “right to be forgotten”) introduced one of the most significant changes. This particular point requires all companies to allow their customers to request data erasure at any point, with no questions asked (with small exceptions, such as banking, where the data is retained because of money laundering or terrorism laws)[1]. Then, the company is required to guarantee erasure of all logs - and if it’s determined that it was not the case - they could face heavy fines.

The notion of Personal Identifiable Information was also put into law with special regulations on how such data should be handled; article 4 of GDPR (titled ‘Definitions’) describes it as:

> “‘personal data’ means any information relating to an identified or identifiable natural person (‘data subject’); an identifiable natural person is one who can be identified,

[1]Also where the data is essential for business continuity, for example debt history or retention required by other directives, such as “Anti-money laundering directive” adopter on July 2019.

directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person;”

Such data should either be anonymized before storing or access should be heavily audited and monitored, to minimise chance of leakage - and again, in case of non-compliance - face large fines.

Though, GDPR was only the beginning and other jurisdictions followed suit, aiming to introduce similar protections. The UK Government decided to ratified slightly modified version of GDPR into their own law and called it Data Protection Act (DPA) 2018. Most recent example being state of California, introducing California Consumer Privacy Act (CCPA) put into law on January 1st, 2020 [**?**]. It shared lot of similarities with GDPR and DPA, such as right to erasure, defining personal information, setting penalties and aiming to protect consumers and their data.

2.2 MQTT

When designing architecture with the main target being the communication of many (even thousands a second) clients continously exchanging data, scalability and availability needs to be kept in mind. The first and obvious solution would be to directly connect data consumers and data producers, by making them communicate in Peer-to-Peer fashion, removing the need for any extra infrastructure. This might work perfectly fine with small systems, however, as the number of clients requesting access to data increases, the total capacity of the sensor would eventually be capped - since IoT usually are of limited power and computation capacity. To illustrate, imagine a scenario where a single temperature sensor is continually getting flooded with requests for current readings. It might be able to cope with up to 5 incoming requests every second, everything else would cause malfunction or significantly slower response times.

Then there is also an issue of authentication. By allowing clients to connect to IoT devices, we are opening a new attack vector. What if the client does not want only to access the temperature readings, but perhaps inject a worm which would intercept other sensors (such as cameras)? For example, recently baby monitors, responsible for alerting the parents when the child is crying and also relieving the adults from having to be always nearby, gained popularity. A direct camera feed could be accessed via a smartphone, from any location. This eventually led to exploitation, as it was found that many of those devices were vulnerable to remote access by third parties[**?**].

MQTT aims to address those issues, by moving the communication to a separate entity, which operates in a publish-subscribe fashion. This would mean that IoT devices only have to publish information that is available to them (e.g. temperature readings), allowing to remove remote direct access, effectively mitigating this particular attack vector. Furthermore, the MQTT brokers can be further placed behind load balancers which can enhance their availability further.

In short, MQTT, fully expanded to Message Queuing Telemetry Transport is an open protocol, certified by OASIS and ISO[**?**], responsible for the publisher-subscriber communication between client. It is important to point out that MQTT is not a piece of software or a server, but rather a set of standards defining what potential clients can expect (what kind of responses and data) while connection to brokers following the standard. Figure 2.1 briefly shows how an

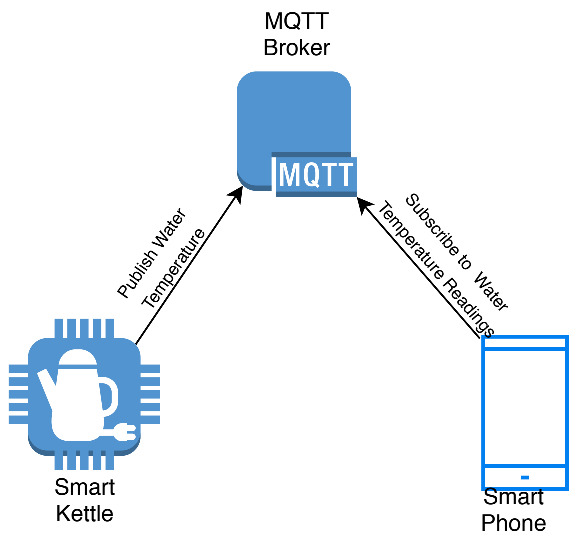

MQTT
Broker
MQTT
Publish Water Temperature
Subscribe to Water Temperature Readings
Smart
Kettle
Smart
Phone

Figure 2.1: MQTT Broker Architecture

MQTT-compatible broker can relay information between clients. Smart Phone and Smart Kettle do not have to be online at the same time in order to receive information, nor is Smart Phone even permitted to initiate a direct connection to Smart Kettle. The broker's responsibility is to track connected subscribers (which must specify the topic of their interest) and maintain the connection until subscribers advertise session termination or abruptly disconnect (e.g. due to loss of power or unreliable connection).

In MQTT architecture, Client ID identifies each of the connecting entities (publisher/subscriber) and topic identifies a bridge between publishers and subscribers connected to the same topic. For example, a Smart Kettle could be publishing temperature readings under a topic called "UK/Aberdeen/Kettle" - then, a smartphone would need to subscribe to the same topic to receive those readings.

2.2.1 MQTT Brokers

MQTT by itself is only a collection of standards instructing implementers on what patterns should be followed and the structure of particular messages. Therefore it is not shipped with any piece of software. It assumes operation on the TCP layer of the network (although newer versions also allow for WebSocket support [**?**]), thus also allowing for encrypted connection via Transport Layer Security. Every exchanged message is a TCP packet, following a strict convention, which in case of deviation is discarded as corrupted.

Two of the implementations that were considered during this project are Mosquitto[2] by Eclipse and Moquette[3]. The former is written in C and the latter in Java, although there are many, many more. In a paper by **?**], scientists compare Mosquitto and RabbitMQ, arguing their choice by the offered cloud infrastructure with more significant scalability opportunities. Moreover, some solutions are paid, whereas the considered approaches are free and open-source, allowing for a better understanding of operations. The paper is concluded with the finding that hardware and network latency has a far more significant impact on the performance, rather than the choice of

[2]https://mosquitto.org/
[3]https://github.com/moquette-io/moquette

the individual broker, which leaves the decision mostly down to offered extra features.

Mosquitto also offers a Docker container [?] in which the broker can be run, allowing for further isolation and removal of extra dependencies. Due to this reason, often scoring best response times and being written in C[4], Mosquitto was used as state-of-the-art broker throughout this dissertation.

2.2.2 Publishing

The most popular method of passing MQTT messages is still under the Transport Layer of OSI Model[5], as TCP packets. This allows for slightly higher freedom (compared to stricter protocols, such as HTTP), at the cost of more sophisticated parsing. MQTT standard is composed of several message types with the most important being:

- CONNECT - used to initiate the connection
- PUBLISH - used by the client to publish messages and by the broker to publish messages to subscribers
- SUBSCRIBE - used by the client to request a subscription to a given topic
- UNSUBSCRIBE - used by the client to request removal of subscription to given topics
- Along with relevant *ACK counterparts, for example CONNACK, used to indicate the successful transmission of the message

As shown in figure 2.2, the publishing flow starts with the CONNECT messages. Inside, there are several flags included, such as Quality of Service requested (MQTT can send heartbeat pings to clients to check if they are still alive), requested version of MQTT protocol (at the moment, v5.0 and v3.1). This part is also referred to as “Variable header”. The second part, known as “Payload” consists of the client ID.

Then, once the client has established its identity to the broker, the broker responds with CONNACK message, which contains a bit informing whether a further connection is allowed or not. From this point, the client is cleared to start the publishing session.

Usually, for every message to be published, there is one PUBLISH packet. The PUBLISH packet contains mostly two properties - topic to be published on and the actual payload. Each of the properties is prepended with 8 bytes indicating the length. From this fact, we can derive the maximum possible size of individual payload - 65535 characters (pure ASCII, no Unicode, which may take more than 1 bytes per character). Same as with CONNECT, each message is responded to with PUBACK, acting as a receipt for receiving the payload.

The client can continue to publish new messages without having to connect again, as long as the TCP session has not been terminated. If the client wants to disconnect, it should follow the standard TCP flow, i.e. issue FIN/ACK packet to the broker. For situations, where the connection has been terminated abruptly, there are options such as Will flag (message to pass in case of sudden disconnection) or Keep Alive (to indicate how long should the connection be kept alive for before assuming the client has lost connection).

[4]Which ensures maximised performance

[5]https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/

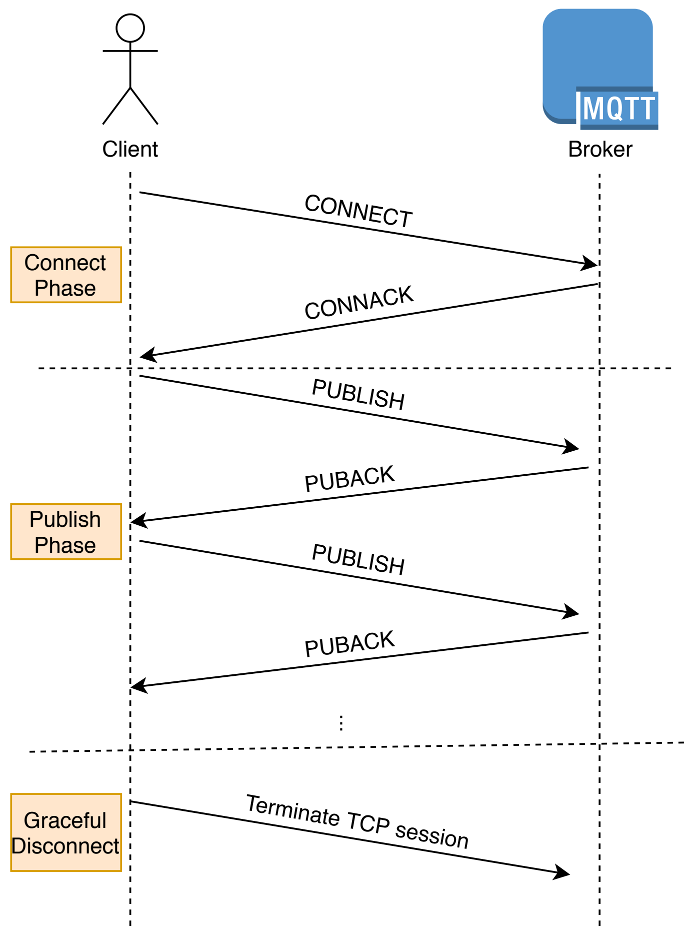


Figure 2.2: Publishing flow with MQTT

2.2.3 Subscribing

Subscribing flow is quite similar to Publishing, with some minor differences. Following figure 2.3, first and foremost, a connection needs to be established by instating a standard TCP/TLS session and then sending a CONNECT packet. The contents follow the same standard, i.e. containing information such as Client ID or even optional parameters in the form of "key: value" (particularly useful for this project).

After a successful connection, the client can proceed to send a request for subscription. Similar with a PUBLISH packet, the client specifies the type of the packet in the variable header and then the requested topic for subscription in the payload. The extra element is the QoS flag - Quality of Service. MQTT has three levels of QoS:

0. No response to PUBLISH messages

1. PUBLISH messages will be followed by PUBACK

2. More granular control over PUBLISH, with extra packets such as PUBREC (Publish Received), PUBREL (Publish Release) and PUBCOMP (Publish Complete).

Once the SUBSCRIBE message has been processed and approved by the broker, it will issue a SUBACK message and remain connected to the client. From this point, any message that is published on the topic specified in the SUBSCRIBE packet will be published (as a PUBLISH

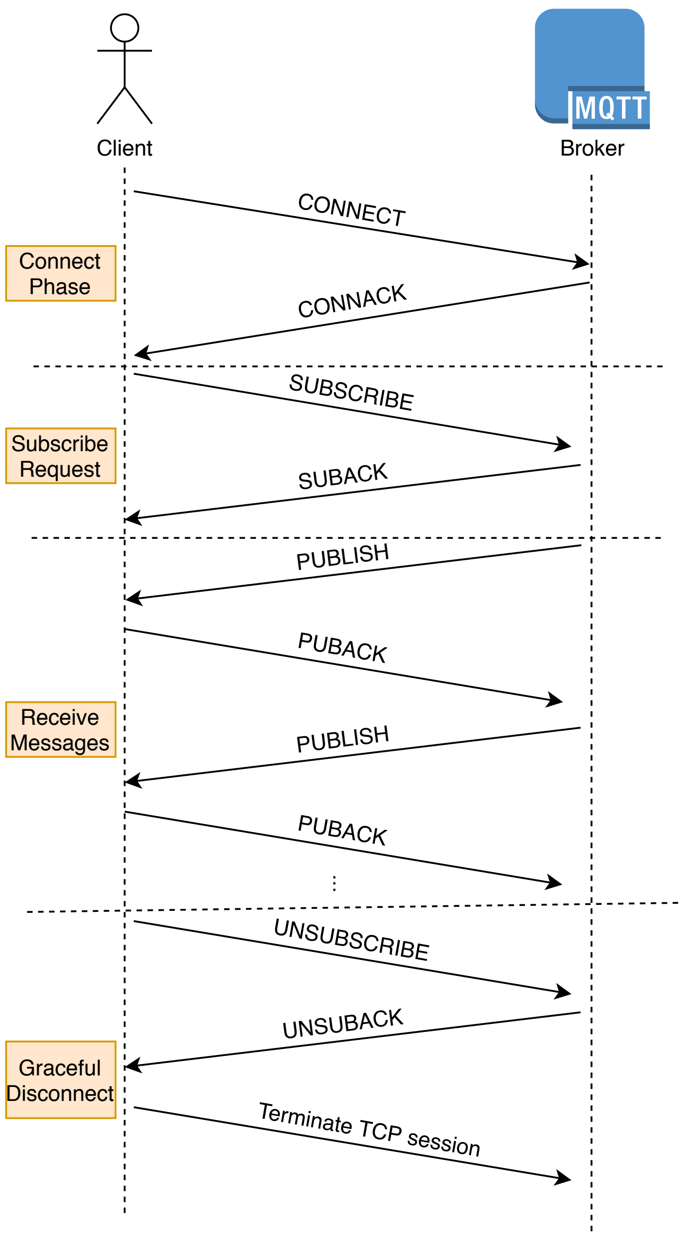


Figure 2.3: Subscribing flow with MQTT

packet) to every client currently subscribed to it. Of course, depending on requested QoS, the broker might then await for a PUBACK message (or even issue other messages such as PUBREC, PUBREL, PUBCOM). Figure 2.3 demonstrates a simple exchange with QoS set to 1.

To close off MQTT, Table 2.1 shows a dissection of an example MQTT packet with an explanation of each bit of the message. 2.1

1	Control field, specifies the type of the message (CONNECT, SUBSCRIBE etc.)
2	Remaining length of the message. Can be expanded to 2 bytes.
3-4	Packet ID
5-6	Payload length

82	04	00	01	00	07	F	L	Y	T	R	A	P	00
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Table 2.1: Hexdump of a SUBSCRIBE packet to topic FlyTrap packet

7-13 Payload. Corresponding hex encoding of characters, replaced with actual characters for clarity

14 Requested QoS

2.3 Blockchain

Lots of concepts in this paper involve blockchain methodologies, which, by itself, is an expansive area. As part of this section, we cover only the most relevant topics necessary to understand the design choices taken within this project.

2.3.1 Architecture

Blockchain often goes by its infamous name of simply overly complicated linked-list, and in fact, it is not very far away from being true. The concept was first introduced and popularised by **?**] in a paper introducing a highly controversial notion of digitalising and decentralising currency, by moving it into a structure called a blockchain. Blockchain network was meant to operate on a peer-to-peer basis, with different peers validating each other's transactions and holding a copy of the entire block. Then, each block's header contains a hash of the previous block, allowing us to traverse the chain.

This removes the need for a central authority governing the currency (for example, central banks), by placing a copy of all records on every participant's computer. But, one problem remained, and that was trust. How do we trust other peers that they do not inject fraudulent transactions? Let us first focus on figure 2.4, which outlines the difference between distributed and centralised ledgers. With the current economic model (for fiat currencies), usually there exists some central authority (in this example, a central bank) which is responsible for tracking, verifying and authorising all transactions between participants. Compare it with a decentralised ledger, where there is no such central entity. Instead, each participant verifies all transactions that happen between nodes. They no longer have to trust Central Bank to do their job currently, as they are free to confirm the authenticity of all transactions themselves.

2.3.2 Consensus Algorithms & Proof-of-Work

Before a participant can add their transaction ("block" from blockchain) to the public records ("chain" from blockchain), we need a cryptographically secure mean to verify whether this particular participant can add this block. Establishing trust between participants on a blockchain is often referred to as "consensus". Thus, several consensus algorithms exist. Currently, perhaps the most popular one, it is proof-of-work. In fact, Proof-of-Work (PoW) dates even before the paper by **?**], all the way back to **?**]. It utilises one of the most critical properties of hash functions, that is pre-image resistance. The blockchain will offer a cryptographic puzzle to the participant willing to add a new block. This puzzle would be based on reversing a hash, i.e. a hash would be generated, and the participant would be tasked with reversing it, thus getting the original value. This "puzzle"

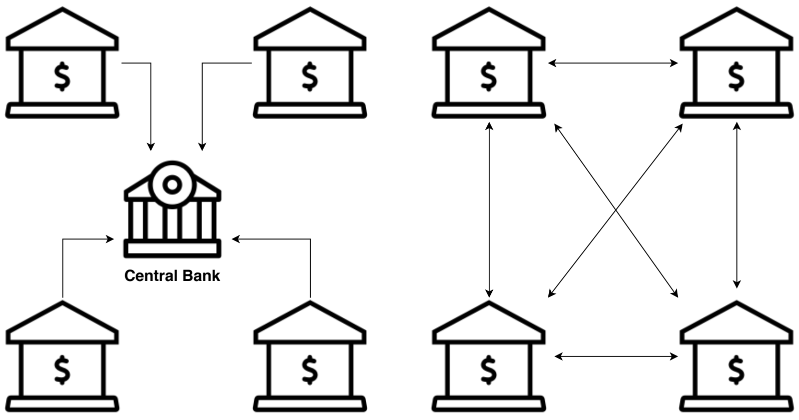
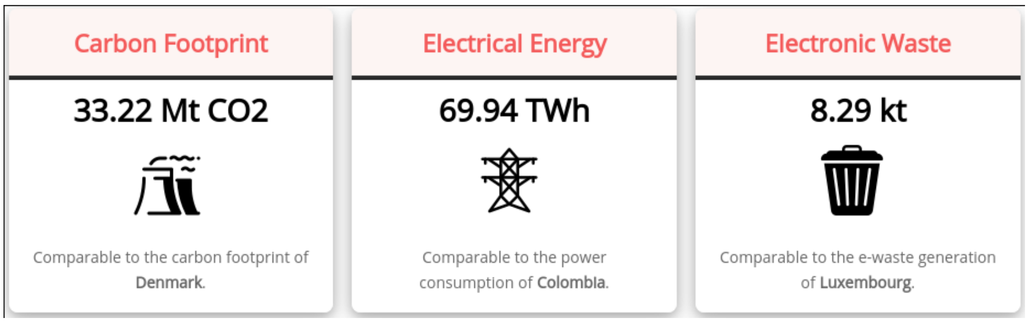


Figure 2.4: Centralised Ledger (on the left) vs Decentralised Ledger (on the right)

is also often referred as mining a new block, that is, finding a value that after passing through a specified hash function would produce the expected output (also known as cracking hashes), that is also part of the reason why modern mining requires much computational power.

Then, everyone starts a race towards reversing this hash. The first person to achieve the target is rewarded with a possibility to add new a block to the network (along with the found value). In the future, any peer can verify the authenticity by feeding the attached value through the hash function and verifying whether the obtained value matches the expected hash. All of it is possible since computing hashes is relatively fast and not a very computationally expensive operation. At the very end, when attaching the new block to the chain, the miner is usually rewarded with cryptocurrency, which can then later be exchanged with other participants.



Carbon Footprint	Electrical Energy	Electronic Waste
33.22 Mt CO2	69.94 TWh	8.29 kt
Comparable to the carbon footprint of **Denmark**.	Comparable to the power consumption of **Colombia**.	Comparable to the e-waste generation of **Luxembourg**.

Figure 2.5: Annualized Total Footprint of Bitcoin network [**?**]

Of course, this approach has several downsides. First of all, all the computational power is effectively wasted on this cryptographic puzzle, with no real end-use - especially if you take part in the race to crack the next hash and someone ends up being faster than you - all your effort went for nothing. This was widely discussed by scientists in **?**], who currently point out a negative impact on the environment. As reported by portal Digiconomist [**?**], as of 2020, the annualised carbon footprint of the Bitcoin network can be compared with the carbon footprint of the entire country of Denmark, with extra samples such as electrical consumption or electronic waste in Figure 2.5

Another problem that Proof-of-Work algorithms create is a 51% attack. Nowadays, setting up your Bitcoin node and starting to mine is not very feasible since people with higher hash rates (the speed at which a person can crack hashes) usually form organisations, that share this power amongst each other and then once they can crack the individual hash, reward each of the members with only a small portion. This might seem sufficient for individuals since now they are guaranteed a payout (rather than risking taking part in the race and losing, winning nothing), but it effectively defeats the decentralised concept of blockchain. If one organisation holds more than 51% of the hash rate of the entire blockchain, it can start authorising fraudulent blocks and adding them to the chain. Since they hold the majority of the network's hash rate, nobody can defy them.

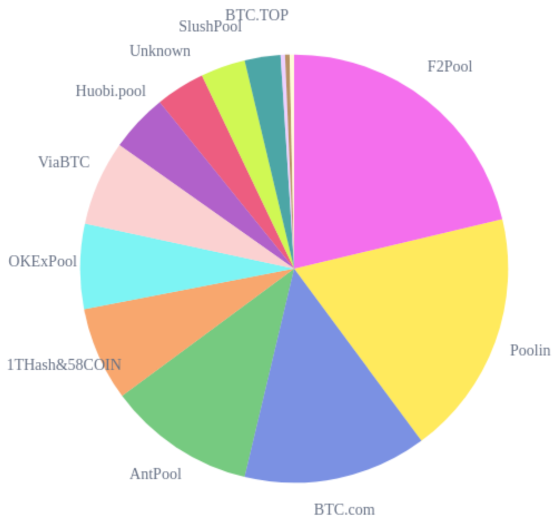


Figure 2.6: Summary of Mined Blocks as of 2020-03-30, per blockchain.com

Figure 2.6 shows the approximate split of total mined blocks in the past 48hrs since March 30th. Now, imagine a situation where organisations BTC.com, Proolin and F2Pool started collaborating, taking over 51% of the market. They would be able to add arbitrary blocks, self-verifying them - and since they hold the majority, nobody could oppose it.

2.3.3 Proof-of-Stake

A slightly different approach to verifying the transaction is called proof-of-stake, first introduced by [**?**] - which does not involve cryptographic puzzles nor requires high computational power and is all based on some pre-defined amount of cryptocurrency that is being put on hold, while delegates are selected. Every participant can bet any amount of crypto - which is returned to them after the validator is selected. The process can be outlined in the following steps:

1. Participants define their stake.
2. Network selects one participant that is going to be acting as a validator for the next block. The higher the stake, the higher the chance of getting selected. Losers get their stake back.
3. Validator goes ahead and verifies the next block that gets added to the chain, there is no block reward, although they receive network fees for the transaction.
4. After a couple of days (once other participants verify the transaction was not fraudulent), the stake is released and goes back to the validator.

This partially eliminates the issue of 51% attack mentioned above. First, because hash rate nor computational power no longer matters (and thus forming organisations loses the point) and secondly, even if the fraudster held more than half of the entire markets' cryptocurrency, they would still be at risk of losing the stake, as their chance of getting selected as a validator is not 100%.

Unfortunately, this approach also brings many challenges. Contrary to what was mentioned above, it creates a bias towards participants with more significant wealth, which can put more value on stake. This might create situations where the rich get richer - though there is always a non-zero chance of getting selected. Furthermore, while they might not have malicious intents, they would be at a higher chance of getting selected as validator and thus collecting more network fees.

As this field is still expanding, more work is published, suggesting refined approaches. At the current day, both Bitcoin and Ethereum use Proof-of-Work, though the latter aims to move towards Proof-of-Stake in the future iterations [**?**].

2.3.4 Proof-of-Authority

What if we do not care about full decentralisation and want to avoid extra operational costs through proof-of-work or proof-of-stake algorithms? A simpler solution, called proof-of-authority [**?**] can also be used. This approach offers no rewards for adding new blocks to the chain, so it is not used in public blockchains. The validators are pre-selected and are responsible for vetting new blocks. This has more uses in situations where data does not have to remain secret, and we do not mind the lack of decentralisation. In fact, if the validators become compromised, they would be able to start allowing malicious blocks.

2.3.5 Ethereum

Proof-of-Work proved itself to be a tremendous waste of energy and resources, with Bitcoin using it solely for authorising the transaction and nothing beyond it. That particular period was also a time when a lot of different currencies started showing up, as Bitcoin's source code was open, everybody was allowed to host their network. Ethereum was one of them, but it was also the first to introduce a concept known as smart-contracts - a way to put the proof-of-work energy to some use (though still a lot of was wasted), introduced in 2015 by **?**]. The network also gave birth to so-called decentralised applications (or Dapps for short), through smart-contracts. Smart-contract can be understood as pieces of code which can get executed on the blockchain, written in a specialised language called Solidity inside EVM (Ethereum Virtual Machine). The transactions were no longer limited to the information about transferring currency between accounts, but also could execute code and act as a persistent database, which could not be altered by anyone and change history was publicly available.

2.4 Related Work

In this section, we would like to point towards recent research that also had in mind improving the security of IoT by looking into combining it with the blockchain platforms. Below, two papers are described - both concluding with positive aspects of such approaches.

2.4.1 IoT, Hyperledger and GA

Attempts at combining IoT authorisation with blockchain have been made in the past. One of the examples is a recent work by scientists from Khon Kaen University in Thailand. In ?], researchers look into Authorization Architecture for IoT (using MQTT broker as an intermediary entity). They are arguing about the benefits of combining any solutions for low-power devices and distributed architecture, which ultimately enables much better scalability and removes the single point of failure.

They are also utilising Hyperledger Fabric - another blockchain-based ledger. Compared to Ethereum, Hyperledger [?] is used mostly for Business-to-Business scenarios, as it does not feature any reward for mining, i.e. adding extra blocks to the chain. Transparency is also limited, as the information is no longer placed on a publicly available platform but rather depends on trusting the nodes connect to the network. We spend some more time discussing the differences within the implementation and discussion chapters of this report.

Moreover, the focus of that paper is on finding optimised consensus algorithm, such that any latency caused by permission lookup is minimised. Scientists suggest using Genetic Algorithms to compose Optimal Consensus. Their experiments were executed on Kafka MQTT [?]. Thai researchers were able to achieve the performance of their solution called GA Kafka improved by 69.43% compared to standard Kafka.

Their approach, however does not take into consideration other parts of the AAA framework, as they focused solely on authorisation. It has no mention of authenticating connecting clients or making them accountable by keeping audit crumbs of the most sensitive operations conducted on the chain. In my work, we focus also less on the performance of the blockchain network, leaving this down to the blockchain itself. As mentioned in the previous section, Ethereum has had some rapid movements in terms of improving their consensus algorithms and moving away from Proof-of-Work instead aiming to implement Proof-of-Stake.

2.4.2 IoT Communication Using Blockchain

In a related paper, scientists from Insitut Teknologi Bandung [?] offer a solution of replacing MQTT protocol with smart contracts, arguing increased level of security and resilience to the outside threats. The approach is somewhat different, as they are comparing the MQTT protocol with the option of using Ethereum for communication. This differs from the architecture proposed in this paper, as FlyTrap still operates through MQTT and requires a broker to function, i.e. it's only a middleware with regards to the vanilla implementation - whereas the ?] removes MQTT and instead moves the communication to Ethereum.

The evaluation consisted of performing attacks on the networks. They also experimented with the SHA-256 [?] algorithm commonly used for producing message hashes and compared its avalananche effect with Keccak-256 used by Ethereum. Their research is concluded with findings that Ethereum's algorithms are more resilient than direct hashing + encryption approach and suggests further study in combining blockchain with IoT technologies, suggesting great security gains.

Chapter 3

Requirements

In this chapter, we outline the base requirements for the project along with sample stories that would later dictate the workflow.

3.1 Requirements

The project can be divided into a set of both functional and non-functional requirements (FR and NFR). Those provide success criteria for the produced software.

3.1.1 Functional Requirements

Set of requirements defining desired functionality of the system.

(FR1) The system will provide an interface to manage access to the topics along with inspecting the audit trails.

(FR2) The system can connect to any Ethereum node, be it a public endpoint or a locally running, closed network. This will provide the flexibility of either using transparent and with 100% uptime resource or a closed node with reduced costs.

(FR3) The system should provide a way to collect payments in ETH from clients attempting to gain access to relevant resources. This payment would then in the process, be transferred to the resource owner's Ethereum wallet.

(FR4) The system should offer an option to specify an exact amount of ETH required to publish or subscribe - with the possibility of separating the costs and also setting the cost to 0 (free).

(FR5) The system should be capable of fending off primitive denial-of-service attacks by blocking continuous, failed attempts to connect.

(FR6) The operations performed by clients will be of limited complexity, such that they can be executed on devices with limited computational power.

(FR7) The system can answer crucial GDPR questions, such as who accessed a given resource, why did they have access, when they accessed it and what exactly was accessed.

(FR8) The system should offer an option to restrict the client's country that can access the resource, which will be verified using GeoIP lookup, as various countries have various data protection laws.

3.1.2 Non-functional Requirements

In addition to the functional, it is also vital to mention the following non-functional requirements, as the system is intended for end-users (potentially non-technical) and due to incorporation with blockchain can introduce performance overhead.

(NFR1) The system should provide **an overhead of no more than half a second** per MQTT session. This is important, as the intention is to provide an add-on on top of the existing MQTT brokers. This might further compromise the current efficiency, so the system should aim to minimise the added latency.

(NFR2) The system should be agnostic of the used MQTT broker, as long as the broker **fully implements the MQTT v5.0 standard**. As pointed out earlier, there is a variety of brokers available to use, such as Mosquitto or Moquette. FlyTrap should not rely on the implementation of a broker, but rather only on the standard utilised.

(NFR3) The system should be capable of extending any MQTT broker with **Authentication, Authorisation, Accountability** framework. This is to ensure that data can only be accessed by authenticated entities, which are authorised to access requested resources - and in case of a breach or other disaster, keep them accountable to their actions.

(NFR4) The system should only be based on **Open-Source Software**. Since the ultimate aim is to provide increased security, keeping the source open would allow any potential users to inspect its operation. Furthermore, third party security audits can happen without the system owner's intervention.

3.1.3 User stories

User stories are further aiming to formulate the requirements into more concise sample situations:

1. As a government regulator, I would like to overview access history to specific MQTT topics, to make sure the data is handled in a GDPR compliant manner.
2. As a government regulator, I would like to verify why / when / who accessed a given resource at a specific time, such that I can issue fines for potential non-compliance and inspect data breaches.
3. As a topic owner, I would like to restrict people that can publish / subscribe to them, to maintain their confidentiality.
4. As a topic owner, I would like to collect payments from people willing to access my data.
5. As a topic owner, I would like to block access to my information from requests coming from outside the requested country, to comply with GDPR requirements.
6. As a broker owner, I would like to collect payments from people willing to publish their data on my system, to keep the system profitable.
7. As a broker owner, I would like to secure a distributed network of brokers (with varied implementations), to increase the system's availability.

8. As a broker owner, I would like to block access to the system to malicious clients performing denial-of-service attacks, to avoid system downtime.
9. As a data consumer, I would like to publish/subscribe my messages on low-power devices, such that I can utilise my IoT sensors.
10. As a data consumer, I would like to access the broker from over a hundred parallel sensors, each publishing data independently.

3.1.4 Use-case Scenarios

Below we have listed a collection of stories which relate to hypothetical scenarios and present various problems faced in different areas - both in industrial and commercial environments. FlyTrap is looking to address the issues surfaced in those short stories. In the Evaluation chapter of this paper, they are referred to and tested on whether it is indeed helpful and how the discussed solution solves the problems.

3.1.4.1 Scenario #1: Air Quality study in the UK

Scenario: Robert is working as a Research Fellow at a University located in Manchester. His research aims to issue air quality IoT sensors to staff across University, intending to capture information such as pollution or carbon dioxide level to analyse the contents of air in the state. Each sensor is issued to an individual taking part in an experiment (e.g. member of staff, lecturer, PhD student), which is based in a specific room on campus. Robert needs to be able to track the inventory, and thus every sensor must be traceable down to a person.

The budget allows to issue up to 1000 sensors, and Robert would like to use the MQTT broker to receive the data from the IoT devices. Additionally, he would like to share the dataset with researchers across the country; thus, he makes the MQTT broker public. As per GDPR, such data, containing full name, office location and detailed temperature readings, is fully protected and needs to adhere to various governance requirements within the European Union. This information should also be stored only within the jurisdiction, that is, only within EU, to ensure enforceability of GDPR. Robert needs to make sure that only researchers that are located in Europe can access the data and everyone else (for example, people from the USA or Russia) will be denied access.

What problem is addressed here: Data containing personal information of European citizens needs to be handled in a GDPR compliant way. This cannot be ensured if the information is accessible outside EU, as there would be no jurisdiction or enforceability once the data leaves the region. The person responsible for data storage needs to ensure that no people outside EU can access it.

Relevant requirements: FR1, FR6, FR7, FR8.

3.1.4.2 Scenario #2: Data breach in an oil drilling facility

Scenario: Bob is a Chief Information Technology in a company Chell handling processing of oil and gas in Scotland. Bob's company also contracts many smaller companies which provide staffing and direct drilling services. Many sensors are used in the company, which are responsible for collecting data such as air pressure, humidity, occupancy on drilling platform or temperature.

Those IoT sensors are utilising an MQTT Broker, which is restricted only to authorised Chell employees.

Unfortunately, due to unrelated reasons, access to the broker has been compromised thus allowing third parties to peek into the data flowing through potentially. Bob is approached by Judy, who works with the team responsible to trace the severity of leakage and determine the potential scope & damages. Judy asks Bob to outline who might have had access to the leaked information and what the leaked information contained. Judy also instructs Bob to inform all people and contractors that might have been affected by the breach.

What problem is addressed here: Companies that use MQTT brokers to store information which is confidential and contains trade secrets of high commercial value want to determine the range of leak that happened. This is part of disaster recovery procedure and is important for business continuity; i.e. determine how much of data has leaked and what kind of information was exposed to malicious actors.

Relevant requirements: FR1, FR7.

3.1.4.3 Scenario #3: Unsatisfied Customer

Scenario: Mary recently purchased a smart assistant, which comes with several smart sensors to be placed around the house. Unfortunately, Mary decided to return all the sensors and cease further usage, she reaches out to Moogle's representative - Matt.

Matt knows that Moogle is using MQTT brokers to connect their smart sensors and then use the phone app to issue commands back to them through the broker. Although the phone app is not the only piece of software that has access to the data from the smart sensors. Analytics teams also consume those in order to help Moogle create better products. Those servers have their own data storage capabilities and replicate the data that they consume. Matt is now tasked with identifying which internal analysis services might have accessed Mary's sensors in order to erase this information since it is one of the GDPR requirements, also called "right to be forgotten".

What problem is addressed here: Again, GDPR comes into action here, in particular Article 17 - Right to erasure. Moogle needs to permanently erase all trail coming from Mary's sensors, that includes any analytics datasets. Since those services are using MQTT brokers, there is no access trail and without proper infrastructure, it is impossible to go in the past and track which services were accessing the data.

Relevant requirements: FR1, FR7.

3.1.4.4 Scenario #4: Monetization of data

Scenario: Frank owns several farm fields in Scottish Highlands. As part of the ongoing digitalisation, he decided to purchase some IoT sensors to learn more about the environment his farms are located in. Some of the sensors include specialised soil acidity meters capable of measuring pH or determine how moist it is.

He decided to connect all sensors to one MQTT broker for ease of access and started to consume the information through the provided mobile application. As the equipment was quite expensive, Frank would like to also sell access to the sensors to the interested parties, such as other land owners or University researchers looking to learn more about the soil quality in the area. No personal information is included, though the data is highly dependant on the location (which currently only Frank can access) and thus might be also useful for other people.

What problem is addressed here: Selling data is a very lucrative business, especially if the data is valuable and has potential audience. MQTT does not have provisions to accommodate such requirements - anything like that would need to either be stored in some persistent location (and then processed for commercial purposes) or routed through an extra layer of verification.

Relevant requirements: FR1, FR3, FR4.

3.1.4.5 Scenario #5: Securing access to the broker

Bob works for a start-up company contracted by the local council to deploy smart sensors across the city to measure various statistics, such as air quality, traffic intensity, parking spot occupancy. All of those sensors are connected to the MQTT broker, which later can also be accessed by third parties - providing they have a legitimate and approved reason. The MQTT broker is accessible by people that might not have overlapping permissions, for example company A can access only data set Y and company B can access only data set X - both X & Y are published on the same MQTT broker.

Bob is aware of the vanilla implementation of username/password authentication for MQTT brokers, but is concerned about the insufficient security in case of leakage. He also already is using Ethereum within the company to hold internal data, so can issue more wallets - if necessary.

What problem is addressed here: Security capabilities of vanilla MQTT Brokers can be found lackluster and not sufficient for many needs. Using symmetric password authentication can be very damaging if the said password leaks to third parties. For people that are already using related blockchain technologies, the transition to FlyTrap would be even simpler, as it can make use of any and every wallet compatible with ETH.

Relevant requirements: FR1, FR2, FR5.

Chapter 4

Design

In this chapter, there is a discussion about the software that was created during this project, its elements and how they interconnect. Design charts showcasing high-level overview are also included, alongside with the architecture split onto layers - which are loosely coupled with each other. Finally, the presentation of novel approaches such as verifying the authenticity of the connecting clients, the data model stored on the blockchain and how the connection is secured from man-in-the-middle attacks through TLS encryption is shown.

4.1 Architecture

This section includes the diagram of the system and two sample workflows demonstrating how packets are flowing through the framework.

4.1.1 Overview

Figure 4.1 presents an overview of the system, decomposing it into four layers, each responsible for a different part of the framework. It also demonstrates how those layers are coupled and the direction of data flowing between them.

The four layers are as follows:

Consumer Layer - Responsible for interacting with end-devices. To them, FlyTrap should be indistinguishable from a normal MQTT broker and thus accepts/responds with MQTT v5.0 compliant TCP/TLS packets.

Broker Layer - Here FlyTrap acts like a client for the MQTT Broker. Similar to the Consumer Layer, all packets sent by FlyTrap need to be compliant with the MQTT standard in order to receive valid responses from the broker. In this situation, the utilised broker is not relevant - as long as it implements the standard and responds with standarised MQTT packets.

Blockchain Layer - Here FlyTrap performs communication with the Ethereum node through a separate CLI. FlyTrap is capable of either reading past contracts/transactions or submitting new ones. This also provides an opportunity to amend the state of the blockchain (such as access control lists) - given the private master key has remained secret.

Presentation Layer - Used by FlyTrap's end-users to access an overview of the state of the blockchain in a user-friendly way, easily extracting most relevant information such as reports from sensistive topics or audit trail for major operations on the chain.

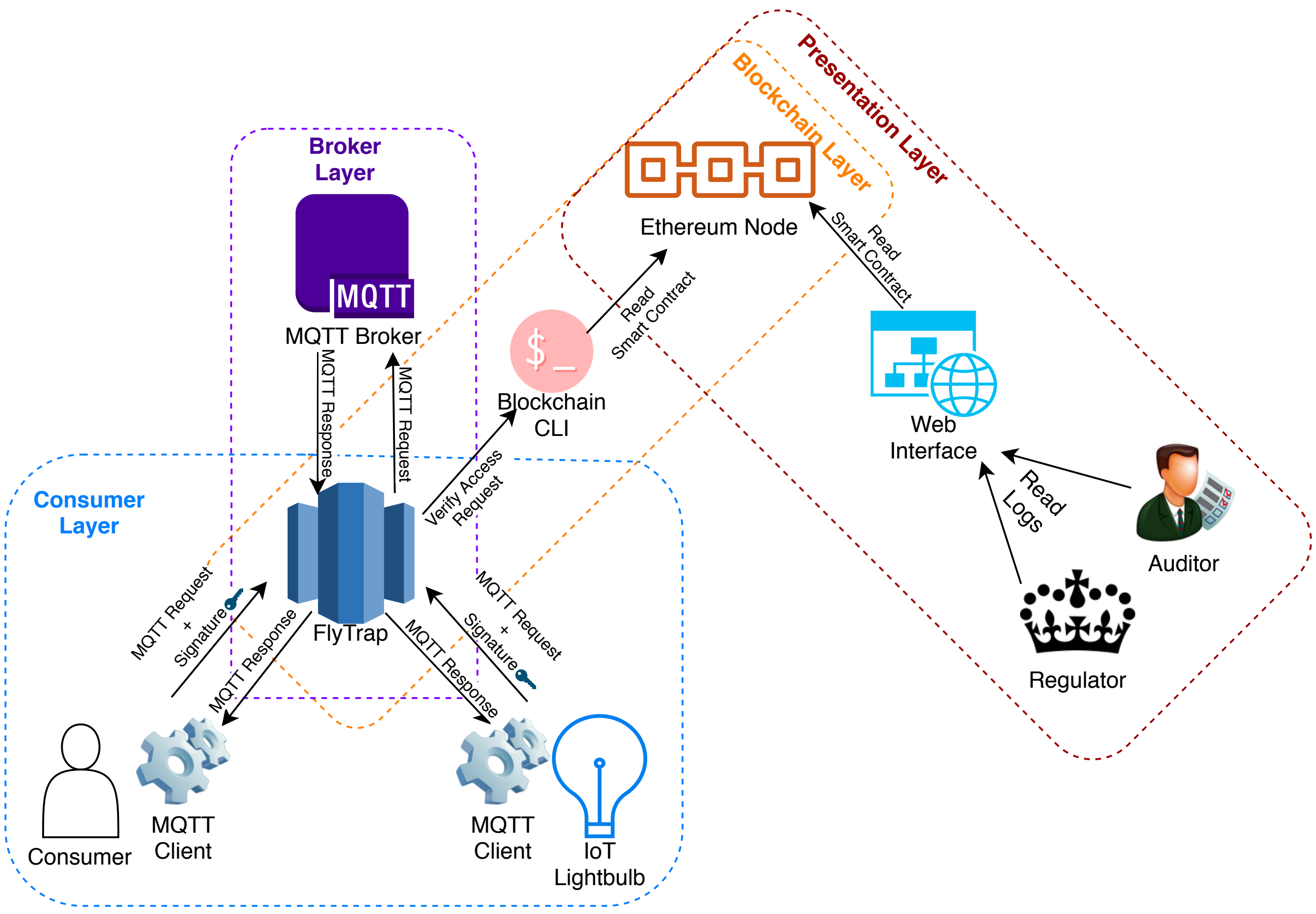


Figure 4.1: FlyTrap high-level architecture overview

4.1.2 Sample Successful PUBLISH Workflow

To provide further context, Figure 4.2 provides an example process in which a client wants to publish a message to the broker and is successful in doing so. The diagram shows step-by-step the logic performed at each stage of the connection, until the client receives the response. Each step is annotated with either **BL** - Blockchain Layer, **CL** - Consumer Layer or **ML** - MQTT Broker Layer to signify the layer in which this step takes places.

Explanation of each step:

0. Marked as optional since each client can accept a pre-computed signature, which can be loaded onto the device; this can be helpful for situations where there is not enough computational power for calculations. (**CL**)

1. Client sends CONNECT packet, including signature + public key in the optional fields of MQTT message. (**CL**)

2. FlyTrap extracts the signature from the optional field and then verify its integrity. It also checks if the client has not been attempting many unsuccessful connections.[1] Finally, FlyTrap responds with CONNACK, signalling to the client that it may now submit relevant

[1]By looking up in-memory dictionary, see Section 4.2.1.4 for details.

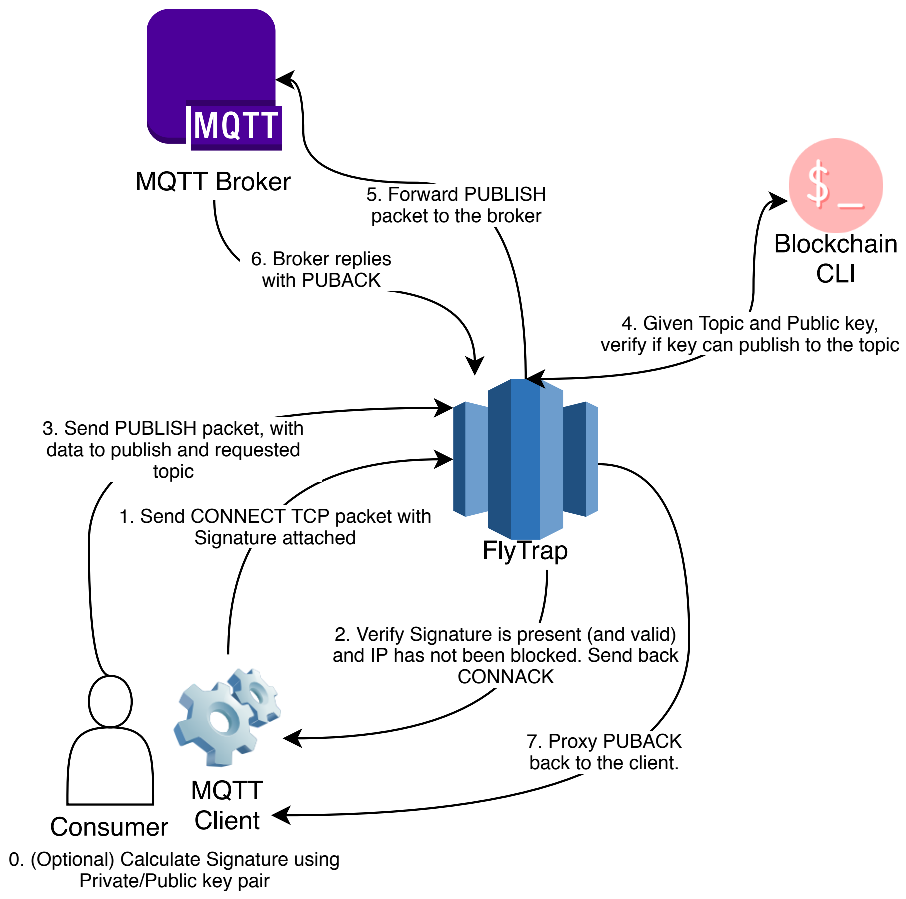

MQTT
MQTT Broker
5. Forward PUBLISH packet to the broker
6. Broker replies with PUBACK
$_
Blockchain CLI
4. Given Topic and Public key, verify if key can publish to the topic
3. Send PUBLISH packet, with data to publish and requested topic
1. Send CONNECT TCP packet with Signature attached
FlyTrap
2. Verify Signature is present (and valid) and IP has not been blocked. Send back CONNACK
7. Proxy PUBACK back to the client.
Consumer
MQTT Client
0. (Optional) Calculate Signature using Private/Public key pair

Figure 4.2: Example successful workflow to publish a message on the broker through FlyTrap.

payload packets. If the integrity check has failed, CONNACK would also have a flipped flag indicating rejected connection and cease further communication. (**CL**)

3. Client now forwards the relevant PUBLISH/SUBSCRIBE packet to FlyTrap (as it still believes that it is a regular MQTT broker). (**CL**)

4. FlyTrap now extracts the requested topic from the MQTT packet and communicate with the blockchain, presenting Public Key and requested topic to verify whether data can be accessed. For this example, the access check was successful. (**BL**)

5. FlyTrap proxies (unchanged) PUBLISH packet to the actual MQTT broker. (**ML**)

6. MQTT broker now responds with PUBACK, indicating successful PUBLISH. (**ML**)

7. Finally, FlyTrap proxies the same PUBACK packet back to the initial client to let them know that the operation was successful - and at the end, gracefully terminate the connection. (**CL**)

4.1.3 Sample Failed PUBLISH Workflow

In this workflow we demonstrate a similar operation, but with the difference being that this time the connection is not allowed, as the presented Public Key is not allowed to publish information on the given topic. For the sake of brevity, steps 0-3 are skipped - as they are identical to the successful scenario.

Below you can find extra details on each step outlined by Figure 4.3:

4-5. Having verified the authenticity of the Public Key, FlyTrap again tries to verify with the Smart Contract whether the client can access the topic. This time though, the response is negative, and the client is not allowed to publish on the requested topic.

5. FlyTrap sends PUBACK back to the client, setting reason code[2] to "Not authorised", at the same time terminating the connection with the client. To client, the situation is to that occurring when an invalid username/password is set to a vanilla MQTT broker.

6. Framework verifies with the cached values to check if the originating IP has not exceeded the maximum number of allowed tries. If it did, it could place it on a blacklist, and every subsequent connection is denied for the specified time. In this example, that is 5 minutes. This step is optional.

7. If the ban happens, FlyTrap also registeres a new transaction on the blockchain to persistently log this event to check potential attack spikes or generate reports w.r.t. captured data. This step is also optional.

Note: as can be seen from Figure 4.3, the MQTT Broker is never connected to nor contacted with the potentially malicious message. FlyTrap rejects the message and forbids the connection.

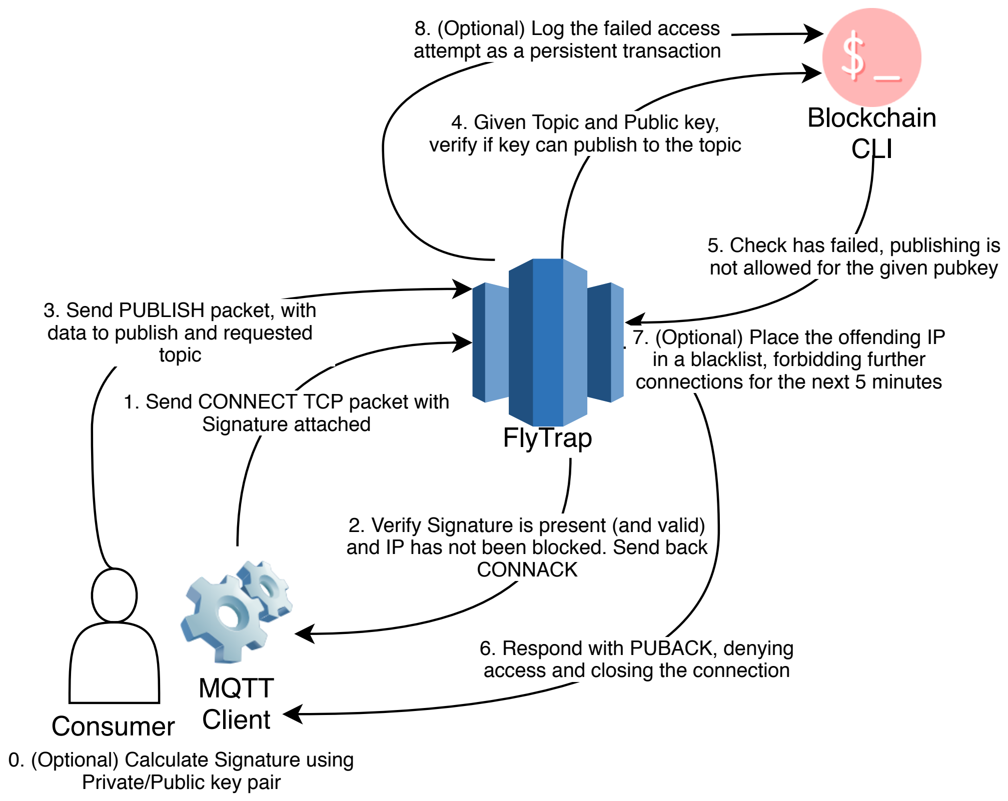

8. (Optional) Log the failed access attempt as a persistent transaction
$_
Blockchain CLI
4. Given Topic and Public key, verify if key can publish to the topic
5. Check has failed, publishing is not allowed for the given pubkey
3. Send PUBLISH packet, with data to publish and requested topic
7. (Optional) Place the offending IP in a blacklist, forbidding further connections for the next 5 minutes
1. Send CONNECT TCP packet with Signature attached
FlyTrap
2. Verify Signature is present (and valid) and IP has not been blocked. Send back CONNACK
6. Respond with PUBACK, denying access and closing the connection
MQTT Client
Consumer
0. (Optional) Calculate Signature using Private/Public key pair

Figure 4.3: Example failed workflow to publish a message on the broker through FlyTrap

4.2 Layer Decomposition

This section overviews each of the layers discussed in Section 4.1 and as shown in Figure 4.1.

[2]https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#_Toc3901124

4.2.1 Consumer Layer

This part of the framework handles all connections between clients attempting to PUBLISH or SUBSCRIBE to data. To them, FlyTrap should be indistinguishable from a vanilla MQTT broker, which should be accepting all regular MQTT payloads. Although the MQTT client produced in this project can connect with every MQTT Broker, a specific client is required for connection with FlyTrap (due to extra authentication fields) - which is further explained in the section below. Moreover, an approach of identifying whether the client holds a Private Key for the presented Public Key is here explained.

4.2.1.1 MQTT Client

FlyTrap makes use of the User Properties[3] introduced in MQTT v5.0, allowing clients to include key-value pairs in the MQTT packets, which then can be utilised by the brokers (or other middleware). That is also where the signature and public key is being placed by the client when attempting connection with FlyTrap - and that is also the need for a custom client since regular clients are not capable of producing highly specialised signatures to connect with Ethereum blockchain.

It is important to point out that the client is only slightly altered to provide (and compute if needed) the signature from a public/private key pair. It is not a critical part of FlyTrap, as any client capable of setting User Properties for an MQTT message would be sufficient, though, for the purposes of this dissertation, a custom implementation has also been designed and included. The User Properties are also ignored by regular brokers.

FlyTrap only expects two custom properties, set on the CONNECT message (meaning that other packets, such as PUBLISH or SUBSCRIBE remain untouched). The properties are: plain-text public key mapped to key “pub” and public key signed with the private-key mapped to “sig” (as can be seen in Figure 4.4).

4.2.1.2 Authenticity of Public Keys

Public Keys from the Ethereum wallet are used as an identifier when determining whether a client can access a given resource or not. Public keys, by definition are, public, meaning that anyone could impersonate legitimate holders of the public/private key pairs. This calls for an approach similar to the Certificate Authority problem when attempting encrypted HTTPS connections. Unfortunately, FlyTrap cannot expect every IoT device to have its own set of certificates, which would then need to be trusted by the framework in order to become recognisable - as those devices are often of limited storage and power.

In order to solve the problem of establishing whether the person presenting a public key also holds a corresponding key, a novel authentication method utilising signatures is used. Figures 4.4 and 4.5 outline client-side and FlyTrap side of the signing/verification process.

Figure 4.4 shows how a client - in possession of public/private key pair produces a signature which then is attached to the final MQTT CONNECT packet. First, it hashes the public key using Keccak-256 algorithm [**?**] (commonly used in Ethereum, e.g. for block hashes), then it uses the private key to sign this hash and attaches the obtained signature to the user properties part of the MQTT CONNECT message. A plain-text version of the public key is also attached in another

[3]https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#_Toc3901068

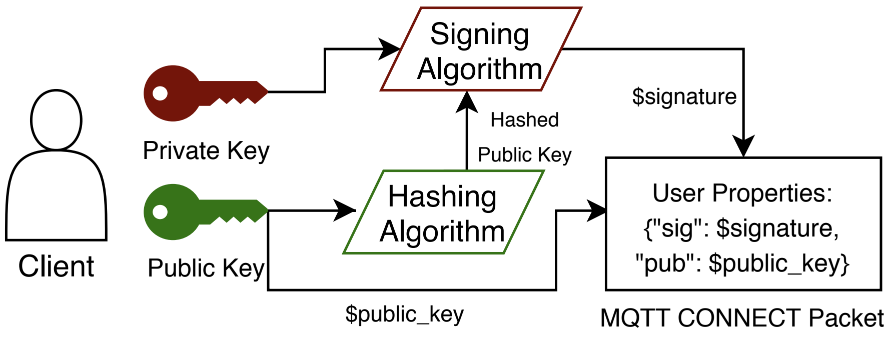


Figure 4.4: Client signing public key and attaching it to the message.

field. Ethereum signatures are created by signing arbitrary bytes through a generated private key - which then can only be verified using the corresponding public key. It is also possible to extract signed bytes from the same signature in the process.

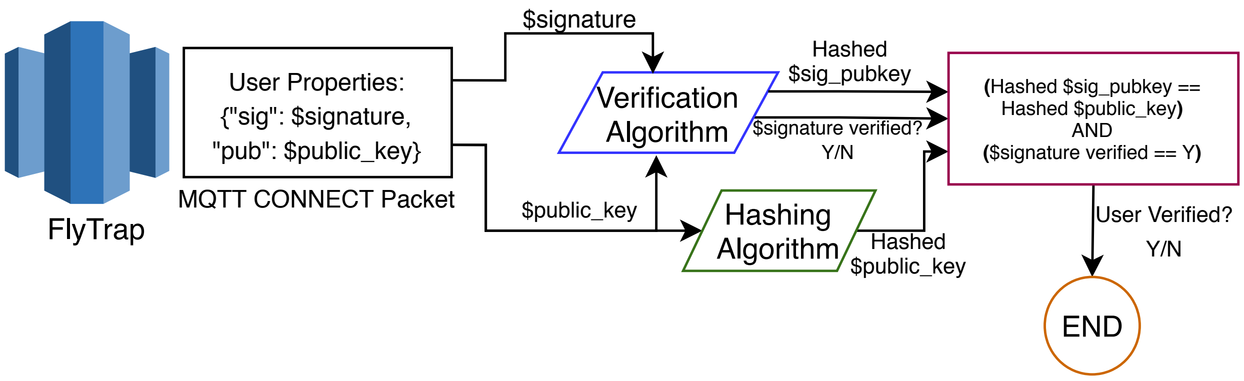


Figure 4.5: FlyTrap verifying the signature.

Then, as per Figure 4.5, FlyTrap receives the CONNECT packet and extracts both the signature & public key from the message. As the corresponding private key for the public key was used to produce the signature, the framework verifies this through Verification Algorithm[4]. The output is a binary yes/no value - determining whether the public key was used to create the signature - along with the signed value (in this case, hashed public key) by the client. Then, the framework can verify whether signature was indeed created with the private key corresponding to the attached public key and (by hashing it again) compare the extracted value with the attached public key. This gives a definite answer that the person presenting the public key also holds the private key and thus is successfully authenticated.

For Ethereum, both the Verification Algorithm[5] and Signing Algorithm are part of the Elliptic Curve Digital Signature Algorithm [**?**], where a public key has exactly 160 bits (which, coincidentally, is also used for ETH addresses [**?**]). The hashing Algorithm - as mentioned above - is Keccak-256.

[4]Here, it is an Elliptic Curve Digital Signature Verification
[5]See footnote 4

4.2.1.3 Secure Proxy

In order to enable FlyTrap to make decisions on whether the requests for publishing or subscribing should be accepted or denied, a secure proxy needs to be established between the clients and the MQTT Broker. As the communication between the broker and the consumers happens via the Transport Layer, it is possible to insert a middleman who would be capable of inspecting the packets flowing through, dissecting them for relevant information and finally make a decision about their future journey - all without the client ever knowing that someone has intercepted the connection.

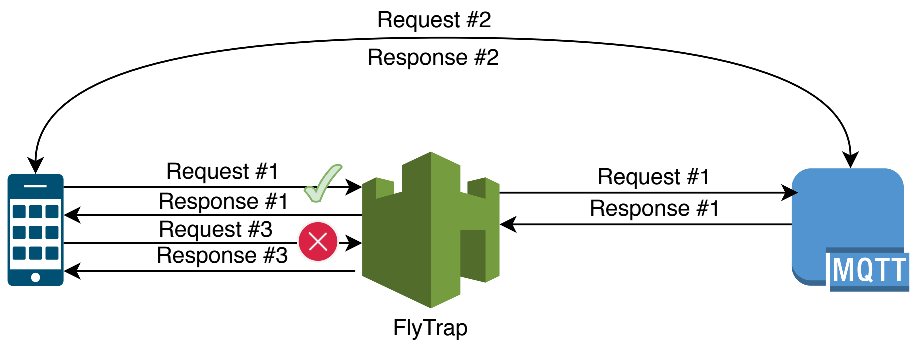


Figure 4.6: FlyTrap acting as a proxy.

Figure 4.6 demonstrates all 3 possibilities when a client attempts connection to a broker. In Request #1, FlyTrap dissects the packet and confirm that the phone indeed can be allowed to access a specific topic and then start a bidirectional proxy with the MQTT broker, passing the TCP packets between them. Request #2 shows that the same packet can be used with a vanilla MQTT Broker without FlyTrap, thus decoupling the client and secure proxy, as the former can be used without the need to change the latter. Finally, for the third request, it is found that the client cannot access the requested resource and is presented with a CONACK response, with the access denied flag set, terminating the connection. Though, in order to make such decisions, the proxy needs to inspect the contents of the packets.

Although TCP proxy is not be sufficient, as quickly as FlyTrap can tap into the connection, the same can be assumed for potential malicious actors, which could be listening on the flowing through packets. FlyTrap supports an extension to standard TCP - Transport Layer Security, or TLS for short, responsible for encrypting the TCP packets, significantly reducing the threat of man-in-the-middle attacks.

TLS sessions can be summarised in the following steps:

1. Initiate standard TCP session
2. ClientHello with client's cypher capabilities
3. ServerHello and exchange of the cypher suite, along with the server's certificate
4. Key exchange and change of cypher spec
5. Encrypted session starts

It is vital to point out, that due to step 3 requiring the server's certificate, FlyTrap needs to either obtain a copy of the broker's certificates or generate a new pair, ensuring that the connecting clients can trust it. Secure TLS connection remains optional, as it is understandable that sometimes enhanced security might cause undesirable performance losses (as found during the evaluation - see Section 6.2.1) or the MQTT broker simply does not support TLS connections. As a result TLS usage with FlyTrap is configurable via command-line arguments. It is also possible to set TLS only on half of the journey, i.e. Client <-> FlyTrap is TLS-encrypted, but FlyTrap <-> MQTT Broker is not - and vice versa.

Moreover, for every new connection, a new thread (or, goroutine[6]) is spawned which has its own context and is separated from others, meaning that isolation between connections is maintained.

4.2.1.4 Protecting from Brute-force Attacks

Some of the failed attempts[7] can result in a persistent log to the blockchain, which often resulusts in a cost - especially if FlyTrap is operating on a public blockchain. This introduces another threat, as malicious actors may attempt to drain the Framework of available funds by logging many operations in a short period. Distributed Denial of Service (DDoS) attacks can also occur and overload the server, which - depending on hardware - could only handle a limited number of simultaneous connections.

To combat those problems, the Framework tracks any failed authentication attempts in an internal dictionary, mapping IP address to the number of failed attempts. This dictionary then is looked up whenever a new connection is initiated. If it is, then the TCP link shuts down, and the client informed that it is currently blacklisted. Though to give the benefit of the doubt, there is a grace period of 2 prior failed attempts before a timed ban is applied. Whenever failed authentication occurs, the counter is increased by one - and if that count increases 3, the connection is terminated and dictionary updated with time 5 minutes in the future - that is the earliest time at which the given IP can attempt the connection again.

4.2.2 Broker Layer

This layer is where the communication with the actual MQTT Broker occurs. Typically, it would be desirable to place both FlyTrap and the broker (e.g. Mosquitto) on the same machine (or at least the same network) to minimise latency - though it is not necessary. Depending on the outcome of the authorisation from the Client Layer, every packet from the client is forwarded to the broker, and every packet from the broker sent back to the client. Since each connection is an individual thread (running on a separate port), it also maintains information such as originating (& destination) address and port - and this information persists as long as the original connection Client <=> FlyTrap remains open - which is only terminated if the client times out, requests disconnection or for any reason authentication to FlyTrap fails.

Similar to the section discussing encrypted connection, FlyTrap is capable of either connecting with TLS-capable brokers or via plain TCP if desired (though keeping the security implications in mind, as everyone intercepting the connection would be able to read the payload).

[6]Concept of threading defined by Golang. Goroutines can either be executed in the same address space or on a separate core, but it's all determined during runtime by Go runtime

[7]Such as providing forged signature or connecting from prohibited country.

4.2.3 Blockchain Layer

In this layer, communication with the Ethereum node occurs. As described in the architecture section, FlyTrap can either read or write new data onto the chain. FlyTrap should be allocated its own smart contract containing chain code capable of verifying connecting clients and relevant data structures.

4.2.3.1 Data Model

The root of all communication with blockchain is a smart contract that contains all chain code responsible for retrieving and storing information required by FlyTrap. Each organisation or entity willing to use FlyTrap should configure and deploy a new contract, which would be tied to a singular owner (i.e., a private key). Ethereum's chain code execution can be limited to only a particular set of addresses, here most of the sensitive operations (such as adding new publishers or subscribers) are restricted to either an owner or payable ETH[8] (set by the owner) - if the requestor is not an owner.

Then, each contract contains a single variable called "topics" which is a mapping (dictionary structure in Solidity, a programming language for Ethereum) from a string (name of the topic) to structure "Topic" where the metadata can be located, such as control lists. Every person can create their own topic on the contract, of which they would become the owner. The contract's owner can also request ETH payment (in addition to the network fees) from any people looking to create a new topic. Figure 4.7 shows all fields used in FlyTrap's chain code placed on the blockchain, to explain further the usage and purpose of each of the fields - with Figure 4.8 showing example values:

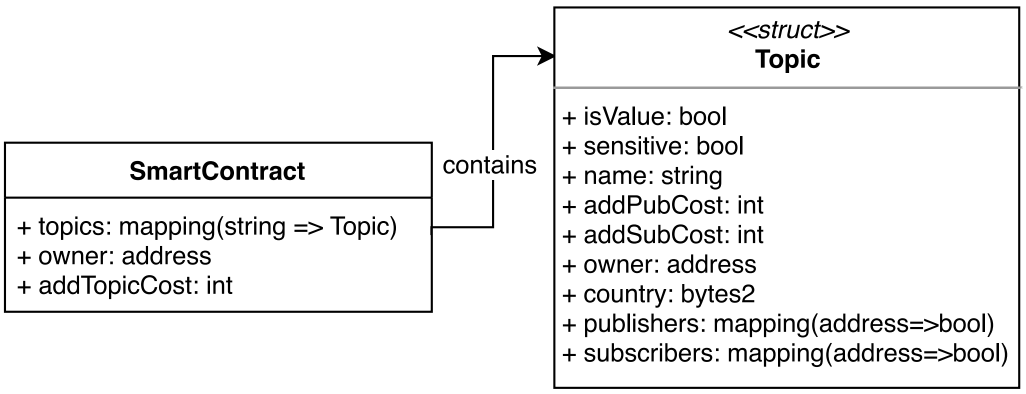


Figure 4.7: Topic structure stored on blockchain.

isValue - in Solidity, it is not possible to determine whether the given key exists in the mapping, as every value points to some arbitrary address space. An extra field helps to mitigate this issue, e.g. to avoid overwriting existing topics with new data.

sensitive - a bool flag the marking given topic as sensitive which enhances it with extra data reporting functionality outlined in Section 4.2.3.2.

name - a string storing the topic's name, as used by the clients on the broker. It is also used as the key in the topic's mapping.

[8]Cryptocurrency used on Ethereum blockchains

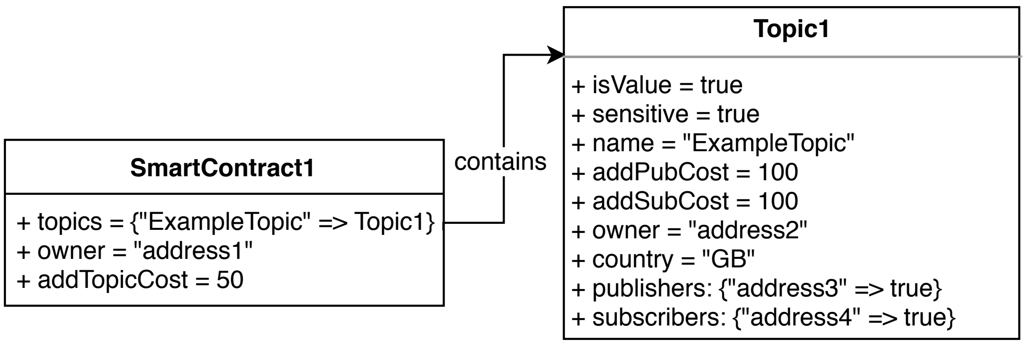


Figure 4.8: Example values for a topic structure stored on blockchain. Note, "address(1|2|3|4)" refers to a public key, which is its own variable type in Solidity.

addPubCost - integer, fee (expressed in ETH) for adding a new publisher to the topic. See Section 4.2.3.5 for details.

addSubCost - integer, similar as above, but for publishing.

owner - ETH public address of the person that created this topic.

country - 2-letter encoding of country to which access should be limited for all subscribers/publishers.

publishers - mapping from the address of a person to a true/false bool value to determine whether the given public key can publish to this topic. It is a way of creating dynamic lists in Solidity, while maintaining $O(1)$ lookup time.

subscribers - same as above, but for subscribers.

Apart from the structure show in Figure4.7 holding information inside the contract, events are also utilised. Event is a special structure used in Solidity, which attaches itself to the transaction log. Meaning that it is possible to append information such as a user-specified reason or timestamps to all operations. This is then encoded in Application Binary Interface (ABI)[9] JSON and sent along with the transaction. Though, it is also possible to emit empty events solely for logging purposes [**?**]. Figure 4.9 outlines what each event consists of (with Figure 4.10 showing example usage):

source - ETH address of the entity that caused the log to emit. For example, for adding new topics or modifying a topic's properties (such as subscribers) it is the initiator's request. For system-caused logs (e.g. report summaries) it would be the contract's address.

target - ETH address of the entity affected by the logged action. For example, for revoking publishers/subscribers, it would be the person that's being removed.

action - enumeration, which defines the exact reason for emitting the log, this can be one of the following values:

AddTopic - when a new topic is created on the contract.

[9]Encoding type used in Solidity: https://solidity.readthedocs.io/en/latest/abi-spec.html

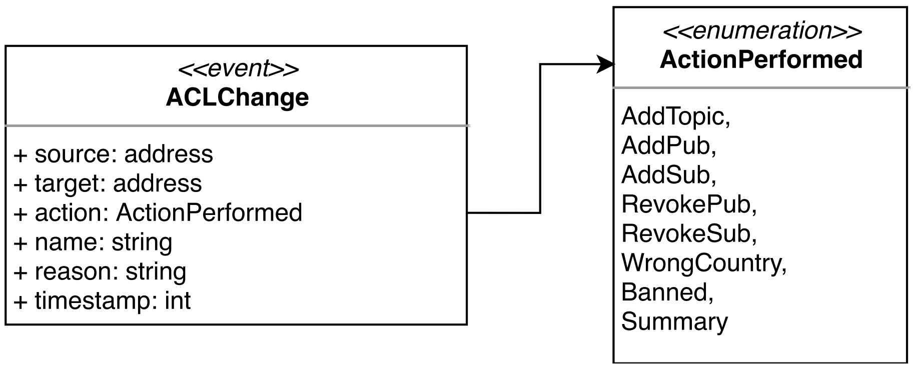


Figure 4.9: Event structure as stored in transaction log.

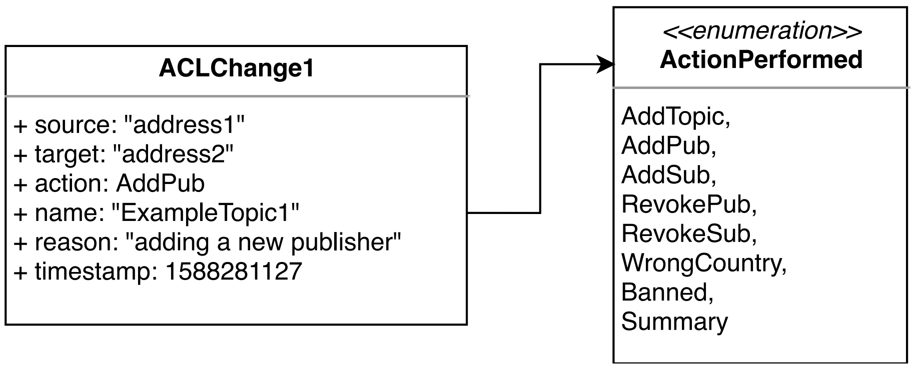


Figure 4.10: Example event log stored on Blockchain. It can be interpreted as follows: "address1" has added "address2" as a publisher to topic "ExampleTopic1" providing reason "adding a new publisher". This action modifies "publishers" mapping of "ExampleTopic1", by setting "address2" to true.

AddPub - when a new person is added as a publisher to the topic.

AddSub - when a new person is added as a subscriber to the topic.

RevokePub - when a person is removed from authorised publishers in the topic.

RevokeSub - when a person is removed from authorised subscribers in the topic.

WrongCountry - when a connection was attempted from an IP located in a country that is not permitted.

Banned - when a person failed to authorise when connecting for three times in a row and was placed on a blacklist.

Summary - system-generated log for sensitive topics (see 4.2.3.2).

name - string used to signify which topic was affected. If an action does not involve a topic, it contains the offending IP address used to make the request.

reason - string provided either by the user or by the system to provide further context on the action (in particular, to answer questions given by GDPR, "why was access provisioned?").

timestamp - UNIX timestamp (expressed as an integer) to determine when the transaction happened.

4.2.3.2 Report Generation

For the most sensitive topics, it is possible to enhance security with extra measures and reporting. For every topic marked as sensitive, FlyTrap would maintain an in-memory list of all publishers or subscribers that have recently accessed the topic. Then, every specified period of time, a log is generated and placed on the blockchain that would outline all Ethereum public keys that have either published or subscribed to the given topic.

To put it in perspective: if the reporting frequency is set at 30 minutes and the system is started at 12:00, then client A publishes to topic X at 12:05, and client B publishes to topic X at 12:15, finally, at 12:30 a report would be generated and placed on the blockchain which would signify that both client A and client B published to topic X between 12:00 and 12:30. The reports should be read as follows: In the past Y minutes[10], the following public keys accessed the following sensitive topics.

It is important to consider the implications of more and less frequent reporting - since every transaction placed on a blockchain with PoW[11] consensus algorithm has an embedded gas price[12], so the owner of the system would need to either accept the increased costs or decrease the reporting frequency. Of course, for PoA[13] algorithms, there is no gas cost, but still, the chain code's size would grow faster in case of more frequent reporting. However, with decreased frequency, the precision also falls, so if our reporting is set to 24hrs, now we can only determine the access history in daily windows (rather than 30min ones, as shown in the example above).

4.2.3.3 Caching Operations

Communicating with blockchain is a computationally expensive operation, and it is vital to ensure that it happens as rarely as possible in order to limit the latency caused by the authentication checks. Following the brute force approach, the Framework would issue a new request every time a connection starts - regardless whether it is part of a series of requests arriving in bulk. FlyTrap uses Ethereum as a de facto database layer, but at the same time aims to cache the operations in-memory, which can then be quickly accessed if repeated requests occur. For checking the permissions, whenever a new request is issued, the result is stored in a map with mutex lock (to avoid race conditions between different goroutines).

To avoid memory filling too quickly (and eventually reaching the capabilities of the server that is running the software), the mapping used for caching is erased every 24 hours or when the process is terminated/restarted - whichever happens first.

4.2.3.4 Interacting with Blockchain

As FlyTrap aims to be deployed on a publicly available blockchain, anyone can interact with the data (as long as they pay the requested fee or identify with the relevant owner's private key). For administrative operations, a simple CLI is also included, which can be called directly (not necessarily through FlyTrap) to perform administrative tasks, such as adding new topics or modifying topic restrictions.

[10]Where Y is the frequency

[11]Proof-of-Work, see Section 2.3.2 for details

[12]Gas is a unit of work performed on Ethereum. The more complex the operation, the more gas is required, and thus more ETH currency is needed

[13]Proof-of-Authority, see Section 2.3.4 for details

4.2.3.5 Monetisation

One of the biggest novelties of this project is the possibility to monetise the data published on MQTT brokers. When creating a new topic, a user interacting with the smart contract can specify a cost associated with adding publishers or subscribers to the blockchain. Once set, any arbitrary person can add their own public key to the ACL - granted they have enough funds to cover the fees. Then, those fees are transferred back to the topic creator - which can be spent or exchanged for fiat currency. The first revision of FlyTrap only allows fees to be set on changing ACLs[14] - for the future it might be interesting to also consider different tiers for different access frequencies.

4.2.4 Presentation Layer

This layer provides a front-end for the data hosted on blockchain allowing users to inspect and interact with the stored information. Similarly to section 4.2.3.4, this could also be read through any of the publicly available front-ends used to interact with Ethereum transactions, but to provide a complete package, the project ships with a simple website which aims to satisfy the requirements stated in Chapter 3.

4.2.4.1 Website

The website does not allow for writing data to the blockchain (thus, does not require Ethereum wallet) which implies that all reading operations are free. The website's backend is loosely coupled with the Blockchain Layer, meaning that it permorms all of FlyTrap's free operations.

Due to how blockchain is designed, it is not possible to lookup a transaction from a specific time. Since it follows the design of a linked-list, lookup of all elements is required to find the requested date, thus resulting in the worst case complexity of $O(n)$, where n is the amount of transactions in the smart contract. This might severely impact performance, especially as the chain grows, thus the web app keeps track of transaction hashes in 5-day intervals in-memory, updating as further requests are performed.

Front-ends operations are also restricted by a time-constraint, i.e. it would be possible to request reports generated between 1970-01-01 and 1971-01-01, warning the user that less restrictive constraints might result in longer lookup times.

[14]Disregarding how often the data is being accessed or how much data is published.

Chapter 5

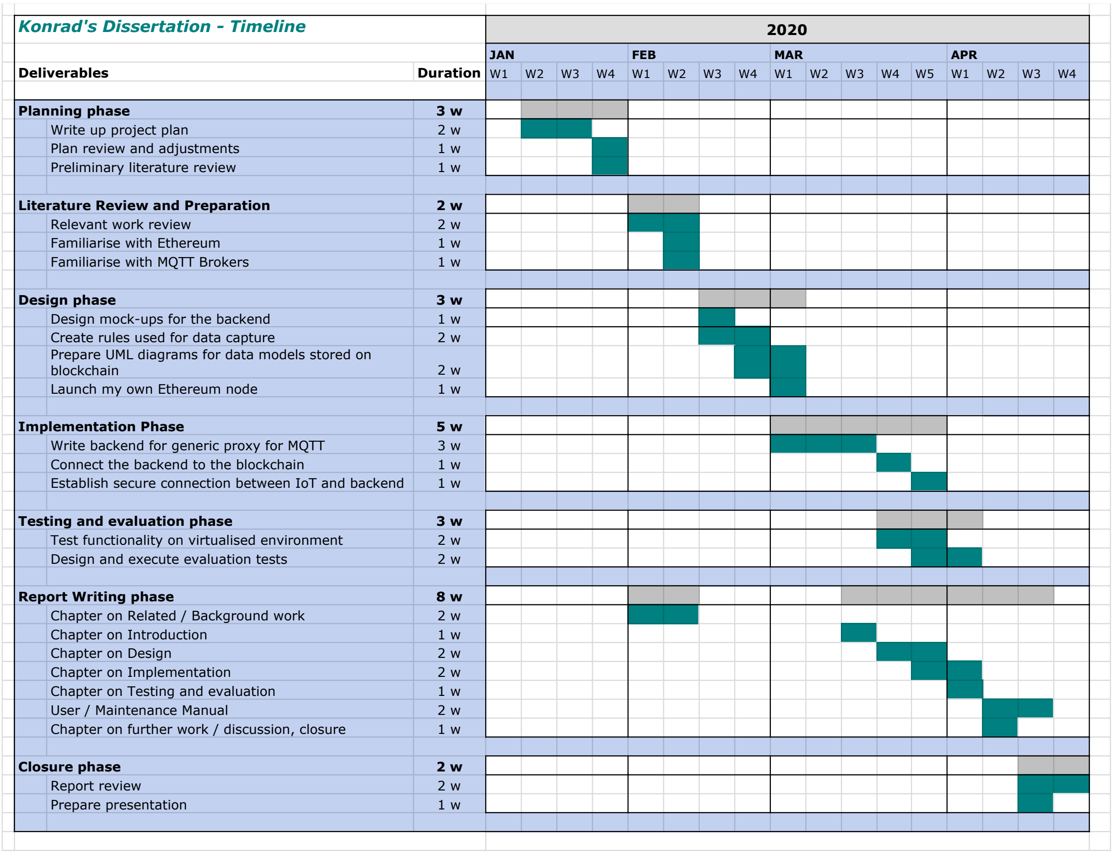
Implementation

This chapter is discussing the technical specifications of the project and discuss the decisions that affected the process. We also overview the workflow and iterative approach to the project. Last but not least, a credit a list of third-party libraries that this framework makes use of is also included.

5.1 Development Process

In this section, we outline how the work on my dissertation was conducted along with pointing out the pitfalls and making sure that the produced code remains fully functional.

5.1.1 Project plan



Konrad's Dissertation - Timeline		2020																
		JAN				FEB				MAR					APR			
Deliverables	**Duration**	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W5	W1	W2	W3	W4
Planning phase	**3 w**		▨	▨	▨													
Write up project plan	2 w		■	■														
Plan review and adjustments	1 w				■													
Preliminary literature review	1 w				■													
Literature Review and Preparation	**2 w**					▨	▨											
Relevant work review	2 w					■	■											
Familiarise with Ethereum	1 w						■											
Familiarise with MQTT Brokers	1 w						■											
Design phase	**3 w**							▨	▨	▨								
Design mock-ups for the backend	1 w							■										
Create rules used for data capture	2 w							■	■									
Prepare UML diagrams for data models stored on blockchain	2 w								■	■								
Launch my own Ethereum node	1 w									■								
Implementation Phase	**5 w**									▨	▨	▨	▨	▨				
Write backend for generic proxy for MQTT	3 w									■	■	■						
Connect the backend to the blockchain	1 w												■					
Establish secure connection between IoT and backend	1 w													■				
Testing and evaluation phase	**3 w**												▨	▨	▨			
Test functionality on virtualised environment	2 w												■	■				
Design and execute evaluation tests	2 w													■	■			
Report Writing phase	**8 w**					▨	▨					▨	▨	▨	▨	▨	▨	
Chapter on Related / Background work	2 w					■	■											
Chapter on Introduction	1 w											■						
Chapter on Design	2 w												■	■				
Chapter on Implementation	2 w													■	■			
Chapter on Testing and evaluation	1 w														■			
User / Maintenance Manual	2 w															■	■	
Chapter on further work / discussion, closure	1 w															■		
Closure phase	**2 w**																▨	▨
Report review	2 w																■	■
Prepare presentation	1 w																■	

Figure 5.1: Project timeline.

Figure 5.1 illustrates the timeline of the project, and all included phases. Each of those tasks was divided onto smaller subsections (not shown on the diagram), which were worked on using an agile approach, as described in the next paragraph.

5.1.2 Iterative Approach

When designing my project plan, I distributed the workload into small work chunks and features which would be implemented in an iterative manner. I was following agile methodologies, dedicating each week to a different feature, where I would go through the entire development cycle for each unit. For example, generating the reports (section 4.4.2) was first introduced during a meeting with my supervisor, where I would establish the requirements and success criteria for this particular story (following definition of stories in agile approach). Then I would spend up to two days designing the flow and functionality, followed by an extra two days implementing the designed features. At the very end, I would conduct testing and regression testing to make sure that the rest of the project still remains operational. This would be concluded with a meeting with my supervisor to reflect on the sprint and determine whether success criteria were met.

5.1.3 Regression Testing

Since I was following an agile workflow when working on the project, it was essential to ensure that none of the stories (or, tasks) affected each other when completed. Otherwise, I could have found myself in a situation where working on the reporting of the events to the blockchain might have broken another, unrelated feature, e.g. verifying the authenticity of connecting clients. That is the reason behind keeping regression testing as a vital phase of development. I was able to achieve it through continuous, automated testing - which includes unit, integration, regression and manual testing.

5.2 Technologies

In this section, we describe the technologies used in this project, which is, languages, frameworks, third-party libraries and different approaches when writing the code.

5.2.1 Languages Used

The following programming languages have been used in this project:

- **Golang**[1] (v1.14) - main driver behind the proxy, MQTT client and for communicating with the blockchain. It has also be used in the backend for the web application.
- **Solidity**[2] (v0.6.6) - language used to develop smart contracts on Ethereum.
- **HTML5 + CSS3 + JavaScript** - frontend stack used to create a simple website to display the contents of blockchain.

5.2.1.1 Golang

Go (short for Golang) has been introduced in 2009 by Google [**?**]. It is a language which strongly follows parallel programming paradigms, allowing the developer to make use out of all available threads and cores to maximise the performance. Multi-thread performance was exceptionally sought after in this project, as FlyTrap is expected to handle many simultaneous proxy connections. In Golang, the programmer can make use of so-called goroutines, which the runtime can

[1]https://golang.org/
[2]https://solidity.readthedocs.io/en/latest/

dynamically either place on separate cores or run them concurrently on the same core - depending on the task.

All Ethereum frameworks (such as go-ethereum or geth, explained further in section 5.2.2) were also designed first in Golang. By deciding to write my project in Go as well, I ensured maximum compatibility, as I was able to use the official SDK's, rather than having to rely on unofficial solutions. I also was already experienced with Golang, as I spent a year in industry at Google working on several projects in that language, which decreased the learning overhead for my dissertation.

Go in the project was used to write the backend of the webserver, CLI to communicate with the blockchain and the proxy itself which handles all incoming connections.

5.2.1.2 Solidity

Solidity [**?**] is the language used to write smart-contracts in Ethereum, so it was a necessity. Its syntax is similar to Javascript, though it is statically, strongly typed. Go-ethereum includes a utility which translates Solidity code into Golang methods, which then can be used to make calls against the blockchain. There is no API which can be used to query/create transactions and rather developers need to use official SDKs published by the Ethereum team.

5.2.1.3 HTLM5 + CSS3 + JavaScript

Since the web application serves only as a presentation and demonstration, I decided against any complex JavaScript or TypeScript frameworks and rather opted for a simple solution consisting of plain HTML + CSS and JavaScript. The web app also includes a small backend; thus, asynchronous calls are performed from the client-side towards the server to fill the content tables dynamically.

5.2.1.4 Considered Alternatives

Apart from Golang, I was also considering Python as a main driver for the project. The majority of my projects in the past included Python, and my familiarity was sufficient to avoid any roadblocks. Though, compared to Golang, there are several problems, which would be especially prominent in this project, one of them being Global Interpreter Lock[**?**], which effectively restricts the scope of possible multithreading.

The choice was also dictated by a personal preference, as I wanted to apply the knowledge I gained over my year in industry during my final year at University. I believed that having a comparison on how a language performs in both industrial and academic environment might provide invaluable experience.

5.2.2 Third Party Libraries & Resources

Following third party resources were utilised in the project. All of them include an open-source license, allowing unrestricted, free use (fulfilling **NFR4**):

ethereum/go-ethereum v1.9.10 [**?**] - official Go implementation of Ethereum blockchain used as a bridge between ETH nodes and any Golang programs, allowing to perform CRUD[3] operations on the blockchain.

[3]Create, Read, Update, Delete

eclipse/paho.golang v0.9.1 [?] - library used to unpack MQTT packets in Golang, without having to inspect each individual byte manually. It outputs a struct containing all fields as defined by MQTT v5.0 specification.

oschwald/geoip2-golang v1.4.0 [?] - coupled together with GeoLite2 IP Geolocation dataset from MaxMind [?] is used to determine the country in which the IP is located in, without having to query external APIs.

tabulator v4.4.3 [?] - JavaScript library used to create dynamic data tables, which can be asynchronously populated and filtered on the client-side.

5.2.3 Working With MQTT

As the project heavily relies on MQTT brokers, I had to look for a solution to simulate this environment on my machine. I had two options, either run the broker locally or use one of the publicly available test brokers.

5.2.3.1 Online Broker

Both HiveMQ[4] and Mosquitto[5] provide free, publicly available brokers capable of both TLS and plain TCP connections. Connecting FlyTrap to a live, online broker helped me test how the system would behave in a situation where FlyTrap and the broker are not located on the same machine and whether an inter-network proxy works fine.

5.2.3.2 Local Broker

For maximised performance, local broker should be used, to minimise the latency of TCP/TLS connections. For that, I have used Mosquitto 1.6.9[6]. It is also possible to generate a new set of certificates and RSA public/private key pair for TLS encrypted MQTT connections[7].

5.2.4 Working With Blockchain

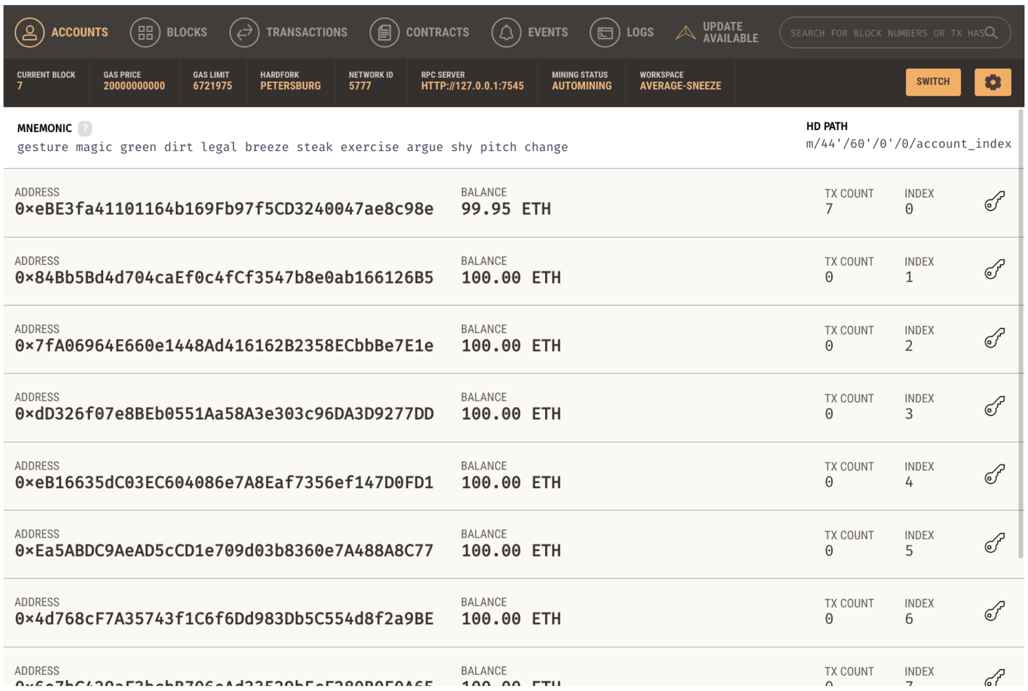
Ethereum was to be used as a data layer for the application. Of course, testing on the public chain was unfeasible, due to the tremendous costs involved. Fortunately, Ethereum provides an easy way to start your own network, which would behave identically as the real one (including fake credits to use) - in the end, it would be indistinguishable from the real node for the applications attempting communication. When working on the project, I considered two ways of replicating blockchain and in the end, made sure to test FlyTrap in both approaches:

5.2.4.1 Ganache

Ganache[?] is a framework designed for emulating Ethereum environment. It can be thought as a sandbox environment, where you can generate any required number of dummy accounts, preloaded with currency, which then can be used to transact with each other - all running on localhost, without the need of configuring a proper blockchain network. Extra blocks are mined automatically - as needed - without the computationally expensive hash cracking. It is also possible to dump the workspace into a file and check into version control.

Figure 5.2 shows how the user interface looks like. It is possible to generate as many wallets as required, each with a public/private key pair and preloaded with 100 ETH.

[4]https://www.hivemq.com/
[5]https://test.mosquitto.org/
[6]https://mosquitto.org/blog/2020/02/version-1-6-9-released/
[7]through 'openssl' utility



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
7	20000000000	6721975	PETERSBURG	5777	HTTP://127.0.0.1:7545	AUTOMINING	AVERAGE-SNEEZE

MNEMONIC
gesture magic green dirt legal breeze steak exercise argue shy pitch change

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0×eBE3fa41101164b169Fb97f5CD3240047ae8c98e	99.95 ETH	7	0
0×84Bb5Bd4d704caEf0c4fCf3547b8e0ab166126B5	100.00 ETH	0	1
0×7fA06964E660e1448Ad416162B2358ECbbBe7E1e	100.00 ETH	0	2
0×dD326f07e8BEb0551Aa58A3e303c96DA3D9277DD	100.00 ETH	0	3
0×eB16635dC03EC604086e7A8Eaf7356ef147D0FD1	100.00 ETH	0	4
0×Ea5ABDC9AeAD5cCD1e709d03b8360e7A488A8C77	100.00 ETH	0	5
0×4d768cF7A35743f1C6f6Dd983Db5C554d8f2a9BE	100.00 ETH	0	6
[illegible]	[illegible]	[illegible]	[illegible]

Figure 5.2: Ganache Interface.

5.2.4.2 Geth

Geth is an environment much closer to the real Ethereum node. In fact, it is used in many production environments to connect to Ethereum networks. Geth is also used in production environments to set up new networks or nodes. When testing my framework, I have set up a new Proof-of-Authority network, with only one node. Since we are in control of the network, we can configure all parameters, such as cost of gas (decreasing it down to 0), allowed accounts or speed of mining new blocks. This network is also indistinguishable from a public one, similar to Ganache. Geth can also be used to connect to live networks and start mining cryptocurrency on your workstation.

Similar as with Ganache, Proof-of-Authority network do not involve hash cracking, thus computational cost is minimal.

5.2.5 Configuration

Application involves a lot of configurable values, to ensure that the end-user finds their preferred combination of settings (such as TLS certs, ports - see User Manual for an overview of all possibilities). There are several ways to state constants and parameters when running any of the components:

- Command line parameters - to specify the environment when running the binaries.
- Environmental Variables - alternative to command line parameters. FlyTrap first inspects command line parameters and if not, looks for environmental variables defined by the host OS.
- Constants in the code - for some immutable values, that do not require intervention (though the option to change them remains documented).

- Website requests - for the web application that ships alongside the project, user can also specify fields such as contract address or which port Ethereum node is running under.
- Static files at projects root - since FlyTrap requires some user-provided files (such as private keys or certificates), running the proxy requiries providing a path to those files via one of the methods specified above.

5.2.6 Logging

All significant operations on FlyTrap, Blockchain CLI and Web Backend are logged to standard error through log Golang module[8]. These logs are only produced for primary operations such as new connection or any actions resulting in writing to the blockchain.

They are not persistent and disappear upon restart. Users are free to implement their own redirection to local files if desired.

[8]https://golang.org/pkg/log/

Chapter 6

Evaluation

FlyTrap imposes an overhead to vanilla MQTT brokers. It is vital to ensure that the added layer of authentication does not severely impact the operation of the broker, as MQTT is a time-sensitive protocol, requiring frequent and rapid responses. In this chapter, we present experiments to measure latency caused by FlyTrap, the operational cost on a public blockchain and also reflect back to user stories to ensure that FlyTrap fulfils the specified needs. In the end, we reflect on the findings and answer the stated research questions, determining the feasibility of the solution in the real world.

6.1 Experimental Design

This section includes details on how the experiments were designed and what the main motivations behind each of them were, along with a description of the hardware and the software used, to ensure reproducibility.

6.1.1 Research Question(s)

Four separate aspects of the implementation were investigated and these can be summarised with the following research questions, which each of the subsequent section attempts to answer:

1. How significant is the performance hit by proxying the connection through FlyTrap, as opposed to directly connecting to the broker (further referred to as state-of-the-art)?

2. How does FlyTrap scale as the number of concurrent requests goes up?

3. How expensive would the operation of FlyTrap be on a Proof-of-Work type of blockchain network?

4. How does FlyTrap address the problems highlighted by the use-case scenarios in Section 3.1.4?

6.1.2 Experiments

Each of the questions specified above has its own designated test.

6.1.2.1 Latency Evaluation

Performance impact was assessed using the same MQTT client that was written for the purposes of this project. For the first test making the requests against the MQTT broker and in the other against FlyTrap, which would then proxy the connection (granted access is allowed) to the broker. To capture exact response time, the experiment measured the time taken between sending the initial CONNECT/SUBSCRIBE/PUBLISH packet until a relevant *ACK response is received. Every time a new measurement is taken, FlyTrap was restarted thus erasing all in-memory cache. In the last measurement, FlyTrap is not restarted (referred to as "Flytrap-Cached") to take into account performance gains caused by caching. Everything is also repeated with TLS both enabled and disabled to measure the impact of having to encrypt/decrypt TLS packets twice.

6.1.2.2 Scalability Evaluation

A latency evaluation alone would not be sufficient to confirm FlyTrap's usability. It could be the case that a single client publishing messages is handled almost with no time loss, but as the number of clients increases (it is common for MQTT brokers to handle lot of clients at once), the response time might be growing exponentially and thus the software might be unusable for commercial purposes.

In addition to the latency evaluation, it is important to examine how FlyTrap performs with many simultaneous connections. 10/100/1000/10000 clients (where the number of concurrent clients is the experiment's independent variable and average response time is the dependent variable) sending a single PUBLISH packet (with the same size for every client) are initiated and their response times captured. Then response time for each test is averaged and placed on a chart to determine the growth speed.

Note: As tests are executed on a single machine, we cannot talk about *true* concurrency, as the CPU scheduler might allocate CPU time for different clients at different intervals. In order to fully simulate such environment, we would require a CPU with X cores (where X is the number of clients), or X machines on a local network - which was not feasible for this project. Having that said, subsequent tests assume that clients executed on the same CPU *are* concurrent, which should provide enough insight into growth rate of response times with regards to scalability.

6.1.2.3 Cost Evaluation

To test the cost, the Ganache testing network is used, as it accurately simulates real Ethereum networks. Different operations performed by FlyTrap are considered as independent variables:

1. Creating a new contract
2. Creating a new topic
3. Adding a new publisher/subscriber
4. Revoking a publisher/subscriber

As the gas cost can vary, each of those operations was executed 100 times saving their cost in Gas (as reported by Ganache), every time using a fresh blockchain instance. Then, those prices (which would be the dependent variable) were averaged and translated into USD for a clearer overview.

6.1.2.4 Scenarios Evaluation

Finally, to test the satisfaction of the requirements, scenarios described in Chapter 3 of this dissertation were used. A walk-through of the FlyTrap's operation was included to pinpoint the benefits provided by the produced software and to illustrate how the problem presented by the use-case has been addressed.

6.1.3 Testing Environment

Tests were performed on the University-provided hardware, with the specs as below:

6.1.3.1 Hardware

A virtual machine provided by the University has been used. Throughout the tests, the current activity on the machine was monitored through the 'top' utility to ensure that no other program might influence the response times. Additionally, to avoid potential slowdowns with caching, each test had 5 warm-up requests, which were discarded from the evaluation, followed by 100 experimental measurements.

Table 6.1 shows full specification of the hardware used for running the tests:

Component	**Value**
OS	Ubuntu 16.04.6 LTS
Kernel	4.4.0-166
CPU	Intel Xeon CPU E5-2630 @ 2.30GHz
Cores	4
L3 Cache	15 MB
RAM	70 GB

Table 6.1: Hardware of the testing environment.

6.1.3.2 Software

As network latency can be unpredictable and varied, all required software and frameworks were installed locally on the machine, with FlyTrap communicating via local TCP ports. This also included local instances for both MQTT broker and Ethereum node. For TLS encryption, a self-signed certificate was also created and provided for FlyTrap. Table 6.2 presents a list of specific software used for tests:

Software	**Version / Implementation**
Blockchain	Local instance of Geth 1.9.13 running blockchain with Proof-of-Authority
Golang	Golang 1.13
MQTT Broker	Local instance of Mosquitto 1.6.9

Table 6.2: Software of the testing environment.

All source code from the project was compiled into binaries before execution to reduce potential noise further.

6.2 Latency Evaluation

This test, compared FlyTrap's performance against the state-of-the-art, which in this situation was a be plain MQTT Broker offering authentication through username/password. Each test was executed on a connection with either TLS enabled or disabled.

Furthermore, as mentioned above, each packet was submitted 105 times, discarding the first 5 results. Then for each scenario, the average was calculated and placed on a bar chart. Additionally, to ensure statistical significance, a Two-Tailed T-Test was performed on CONNECT tests to extract the p-value. For SUBSCRIBE/PUBLISH - as we are dealing with three data groups - Kruskal-Wallis Test (since data is not of a Gaussian distribution, thus cannot use One-Way ANOVA Test) was used to determine whether the null hypothesis could be rejected.

Note: As no caching occurs during the CONNECT stage, "FlyTrap - Cached" has been omitted, as it produces the same results as regular 'FlyTrap'

6.2.1 With TLS

Figure 6.1 shows the average response time for each of the packets. Standard error has been omitted from the representation, as for each of the scenarios, it was found to be less than 0.1%. When sending the request through FlyTrap, an increase in response time is observed of 53.73%/61.92%/62.67% respectively for CONNECT/SUBSCRIBE/PUBLISH packets.

For FlyTrap - Cached, when compared with the vanilla broker, this increase is significantly less prominent, showing an increase of 18.12%/22.19% respectively for SUBSCRIBE/PUBLISH packets.

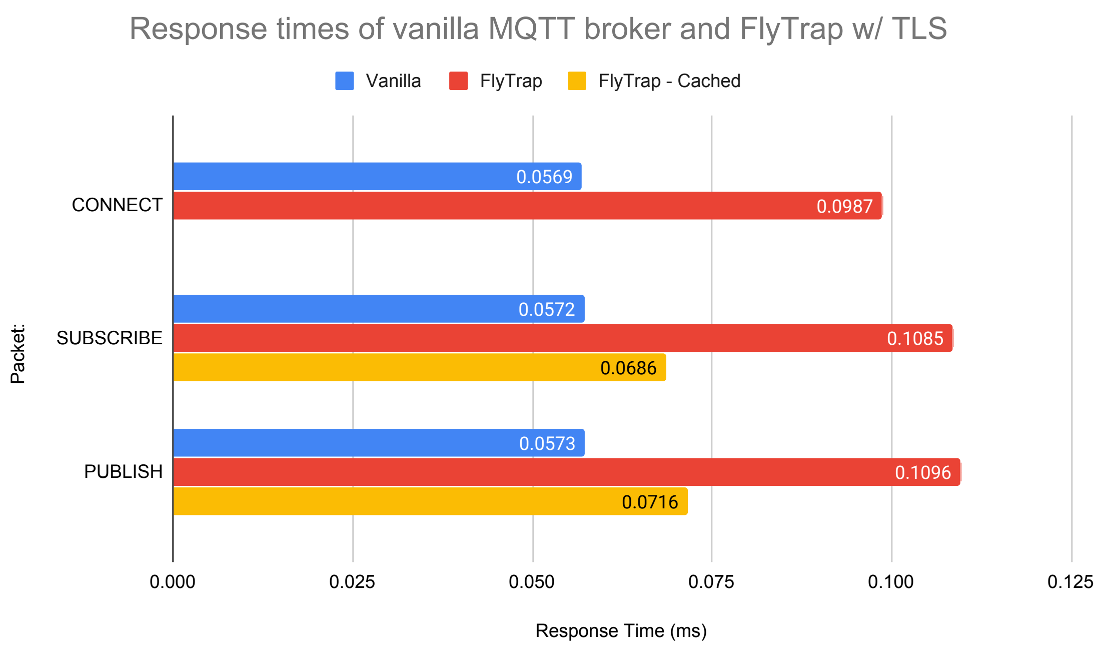


Figure 6.1: Response times of most common MQTT requests via a connection with SSL/TLS encryption.

Statistical tests have been performed on all three datasets. Each of the 100 results for every packet has been compared with the counterpart, for example, 100 CONNECT response times via Vanilla Broker vs 100 CONNECT response times via FlyTrap. This allows us to reject the null hypothesis, as for the first test, the *p-value* is < 0.05. Tests number 2 and 3 resulted with *p-value* $= 0$, which can be interpreted as significantly lower than 0.001.

#	Test Data	Performed Test	p-value
1	CONNECT: Vanilla vs FlyTrap	Two Tailed T-Test	$2.16 * 10^{-69}$
2	SUBSCRIBE: Vanilla vs FlyTrap vs Cached	Kruskal-Wallis	$<<< 0.001$
3	PUBLISH: Vanilla vs FlyTrap vs Cached	Kruskal-Wallis	$<<< 0.001$

Table 6.3: Outcome of statistical tests on each of the MQTT packets considered for TLS-encrypted connection.

6.2.2 Plain TCP

The test above was repeated, this time disabling TLS encryption for both ends of the connection. The first thing that we can notice is the overall decrease in response times when compared with the TLS connection. This is a clear indicator that TLS does indeed add an overhead to standard TCP connections. Experiment results were placed on a bar chart similarly as before. Again, standard error bars were omitted, as they were found to be less than 0.1%.

Figure 6.2 shows the average response time for each of the packets. When sending the request through FlyTrap, we notice an increase in response time of 18.13%/43.30%/31.58% respectively for CONNECT/SUBSCRIBE/PUBLISH packets. For FlyTrap - Cached, when compared with the vanilla broker, this increase was less prominent, showing an increase of 23.41%/15.55% respectively for SUBSCRIBE/PUBLISH packets.

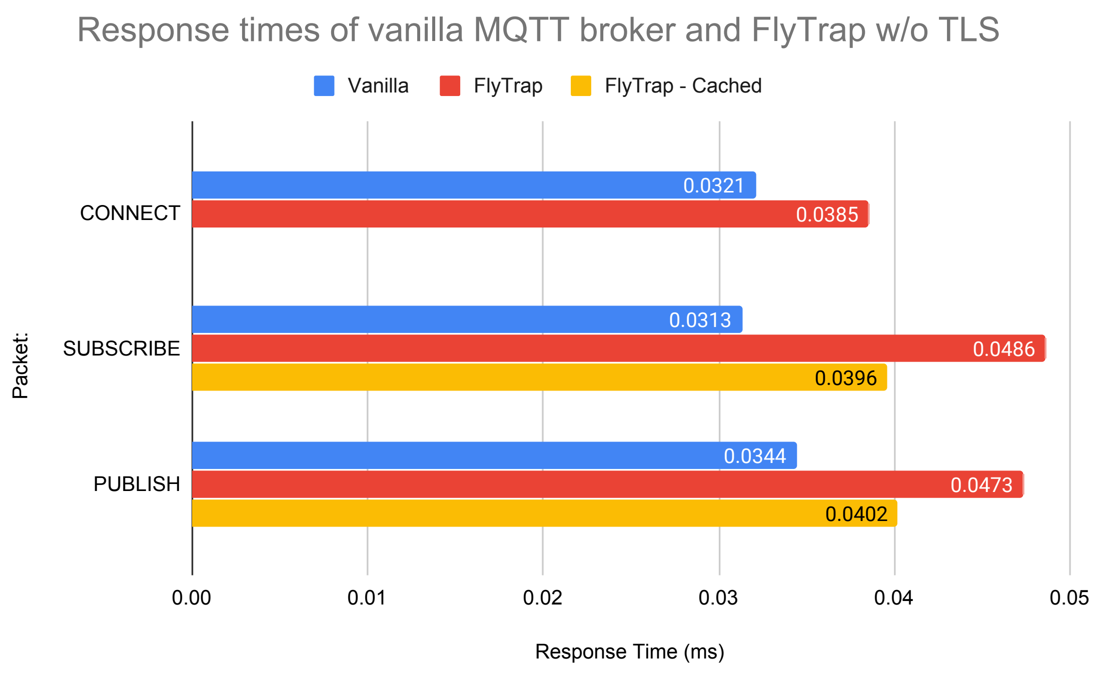


Figure 6.2: Response times of most common MQTT requests via a plain TCP connection.

For each of the tests, the *p-value* was found to be < 0.05, allowing the null hypotheses to be rejected. Table 6.4 shows the values of the performed statistical tests.

#	Test Data	Performed Test	p-value
1	CONNECT: Vanilla vs FlyTrap	Two Tailed T-Test	$8.60 * 10^{-20}$
2	SUBSCRIBE: Vanilla vs FlyTrap vs Cached	Kruskal-Wallis	$<<< 0.001$
3	PUBLISH: Vanilla vs FlyTrap vs Cached	Kruskal-Wallis	$<<< 0.001$

Table 6.4: Outcome of statistical tests on each the MQTT packets considered via a plain TCP connection.

6.3 Scalability Evaluation

In this test, 10/100/1000/10000 concurrent clients were dispatched to PUBLISH a single message, with identical size across all tests, through FlyTrap with TLS enabled. Every client was dispatched at the same time, as a separate process, thus ensuring that they did not interfere with each other. Each identified with a different Public Key, forcing FlyTrap to perform authentication flow. Then the time when PUBACK was received was captured and saved.

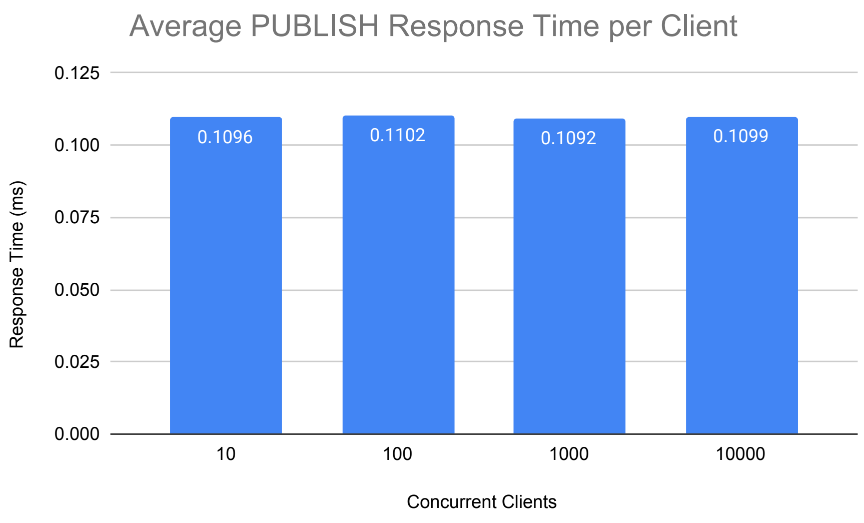


Figure 6.3: Average response times with regards to number of concurrent clients.

See Figure 6.3 with the outcome. As with previous tests, standard error has been omitted, as it was less than 0.1%. We can see that the response time remains constant across a different number of concurrent clients, with less than 1% change between tests.

6.4 Cost Evaluation

As discussed in Section 2.3.4, Proof-of-Authority networks do not have any currency involved, and thus there is no reward for mining. However, this removes the benefits of transparency and publicity of the data. On the other hand, running it on a public network (a.k.a. Proof-of-Stake), where miners compete to add new blocks to the network, involves fees and payments and it is essential to keep this cost to the minimum.

Six operations performed by FlyTrap were executed a hundred times to extract the average costs in gas. To put those numbers into perspective, the equivalent value in USD was also included

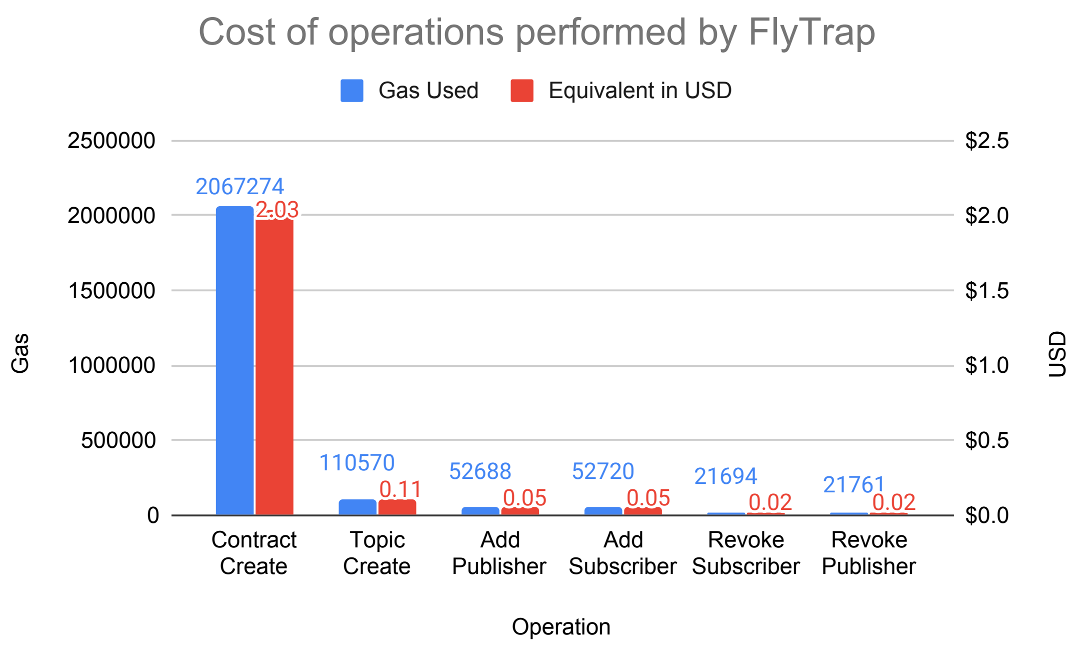


Figure 6.4: Cost comparison of various operations on blockchain.

- assuming relations of 1 ETH = $182.29 and 1 Gas = 0.0000000054 ETH[1].

Figure 6.4 shows the cost in Gas and in USD for each of the six operations. A maximum charge of $2.03 and a minimum of $0.02 is noticed. However, it is important to point out that contract creation occurs only once, when setting up the system. As it allocates memory space on the network for FlyTrap, it is the most expensive operation. All subsequent actions that are modifying its state are priced at below $0.10.

6.5 Scenarios Evaluation

The last part of the evaluation, examines whether the produced software addresses the problems stated by the Use-case Scenarios in Chapter 3. This focuses mostly on a walk-through of each of the scenarios, demonstrating how the implemented features satisfy the business need defined in the relevant scenario. Scenario #5 has been omitted, as the security aspect is be apparent throughout the remaining tests.

6.5.1 Setup

Each scenario shares same number of common setup steps, which sets up the initial network and proxy. Those steps will be summarised here - and can be assumed to be executed prior to any of the subsequent scenarios.

Step 1 - Configure the blockchain. For the sake of this experiment, we use Ganache, which starts a test network, with accounts that have 100 ETH pre-filled currency. Once started, a window similar to Figure 5.2 appears.

Step 2 - Set endpoint for the blockchain node (Ganache default) and create a new contract for FlyTrap.

[1]Data from https://ethgasstation.info/ and https://cex.io/eth-usd as of 2020-04-18

```
$ export BLOCKCHAIN_ADDRESS="http://localhost:7545"
$ go run cmd/blockchain/main.go -new -contract=""
2020/04/18 20:28:44 Generated new contract, address is:
0xD0BaD0f1fC6D627E1e6fcE6020De9BbA2507498f
```

Step 3 - Set contract as environmental variable, add new topic and add publishers/subscribers.

```
$ export BLOCKCHAIN_CLIENT="0
   xD0BaD0f1fC6D627E1e6fcE6020De9BbA2507498f"
$ go run cmd/blockchain/main.go -new_topic "creating test
   topic" -topic "TopicName"
$ go run cmd/blockchain/main.go -topic "TopicName" -client "<
   client_pubkey>" -pub "adding test publisher"
$ go run cmd/blockchain/main.go -topic "TopicName" -client "<
   client_pubkey>" -sub "adding test subscriber"
```

Step 4 - Start web server. The can now be accessed website under localhost:8081.

```
$ go run webapp/main.go
```

Step 5 - Finally, ready to start the FlyTrap proxy. The command assumes TLS as default, with summary reports generated every hour, connection to test mosquitto broker and starts accepting connections under port 8888.

```
$ go run cmd/flytrap/main.go -b-freq=1h
2020/04/18 21:04:15 Now accepting connections under :8888
```

6.5.2 Scenario #1

To recap, the first scenario outlined the need for guarding the access to the data from people outside given jurisdiction, for example, only people within the UK should have access to some designated data, so the access can be governed and regulated to ensure compliance with the law.

In this scenario, we are looking to restrict access to a specific topic, only to people connecting from a country which has been permitted. To do so, we need to slightly modify step 2 from the setup process adding an extra ‘country’ flag to the command, as follows:

```
$ export BLOCKCHAIN_CLIENT="0
   xD0BaD0f1fC6D627E1e6fcE6020De9BbA2507498f"
$ go run cmd/blockchain/main.go -new_topic "creating test
   topic" -topic "RestrictedTopic" -country="GB"
```

From now on, only people connecting from British IPs are allowed, everyone else is presented with the following message:

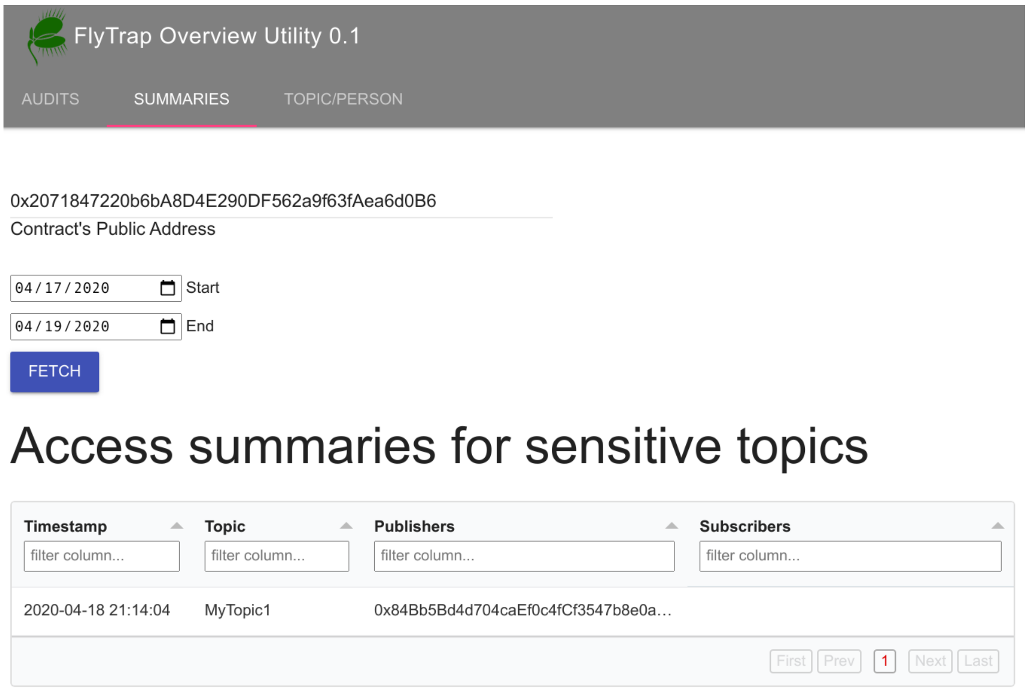
```
$ go run client.go -pub=20 -tls=false -priv="privkey1.asc"
2020/04/18 21:04:17 Using cached signature & public key
2020/04/18 21:04:17 Publishing message: Here Be Dragons #0
```

```
panic: error publishing: The PUBLISH is not authorized.
```

This shows how FlyTrap satisfies the need specified by the Scenario #1 and meets requirement FR8. Normally, the access the broker would have been allowed, but in this case, FlyTrap captures request coming from outside the UK and blocks the access.

6.5.3 Scenario #2

This scenario requires us to determine who accessed the requested resource on the specific day, so we can determine the scope of the leak. Once the steps in the setup have been completed (in particular, the 'freq' option) and the leak happens, we are ready to inspect the website.



0x2071847220b6bA8D4E290DF562a9f63fAea6d0B6
Contract's Public Address

04/17/2020 Start
04/19/2020 End

Access summaries for sensitive topics

Timestamp	Topic	Publishers	Subscribers
filter column...	filter column...	filter column...	filter column...
2020-04-18 21:14:04	MyTopic1	0x84Bb5Bd4d704caEf0c4fCf3547b8e0a...	

Figure 6.5: FlyTrap sensitive topics report.

Figure 6.5 shows how the website would look upon inspecting the access summaries. We can see that on April 18th in the evening, the topic 'MyTopic1' was accessed by a single person with the shown public key. Now, we can trace the leakage down to a single person and verify whether their credentials were compromised to assess the damage further.

This then would allow the business affected by the leakage to provide authorities with the list of users that might have been responsible for the unauthorised data access, satisfing requirements stated by General Data Protection Regulation and FR7 specified in Section 3.1.1.

6.5.4 Scenario #3

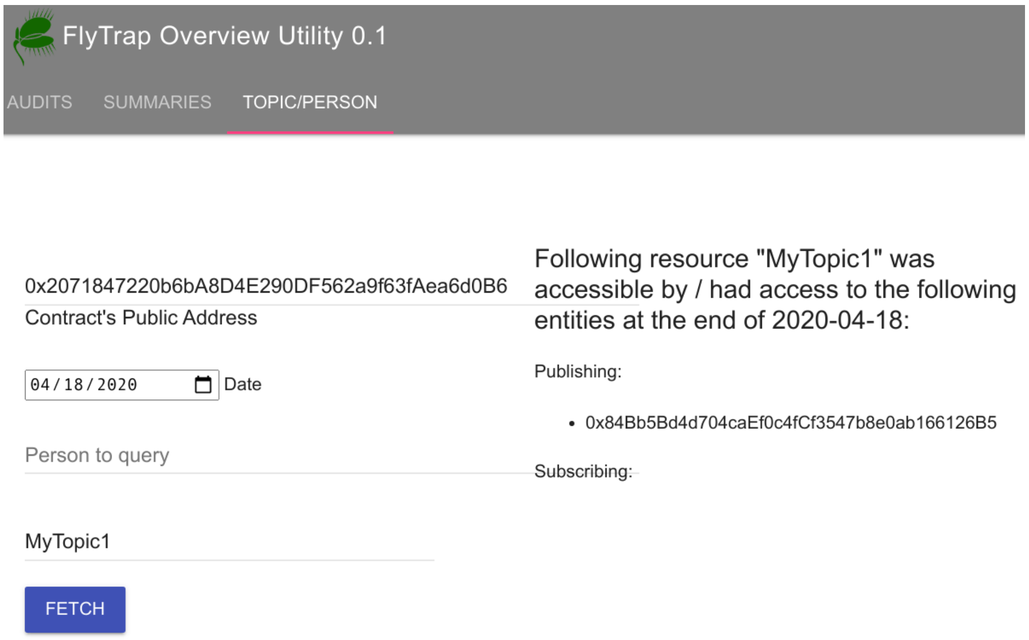
Here we are looking to verify who can access a particular resource, the difference from Scenario #2 is that we are looking at resources that are capable of accessing the data (that not necessariliy made use of those capabilities). This can be then used to track the particular servers[2] and ensure that all existing data (now requested to be erased) is actually erased.

Again, after following setup steps, we can go ahead and inspect the "Topic/Person" part of the web app, which would allow us to see a list of public keys that can access given the topic, as

[2]Which each would have their own public/private key pair allocated.

per Figure 6.6. Then, the system administrator would be able to correlate the public key with the piece of infrastructure accessing the information and ensure that the access is revoked followed-up with data cleanup (which is covered by FlyTrap, as it is possible that some other unrelated services might have accessed the information).

In the end, this helps us to ensure compliance with GDPR, to be exact, “Right to be forgotten”. Normally, those servers might be distributed across the network and since MQTT Brokers do not save access logs, finding them might be challenging. This also proves how FR7 is satisfied.



0x2071847220b6bA8D4E290DF562a9f63fAea6d0B6
Contract's Public Address

04/18/2020 Date

Person to query

MyTopic1

Following resource "MyTopic1" was accessible by / had access to the following entities at the end of 2020-04-18:

Publishing:

- 0x84Bb5Bd4d704caEf0c4fCf3547b8e0ab166126B5

Subscribing:

Figure 6.6: Current Access Control List Summary.

6.6 Results

6.6.1 Latency Evaluation

The overhead added by FlyTrap is apparent, reaching an extra 0.05ms per request. Though, it is interesting to see - looking at the “FlyTrap - Cached” bar in Figure 6.1 & 6.2 - that caching remediates this problem almost completely, reducing the average response time by up to 37%, negating the overhead introduced by the first request. We can conclude that FlyTrap would be the most efficient for situations where it is not restarted very often, allowing the cache to grow and in order to minimise the impact for clients often connecting to the broker. As expected, we also note longer response times for FlyTrap with TLS enabled. This is most probably caused by the fact that there needs to be two separate TLS sessions established, encrypting/decrypting the data twice.

For FlyTrap with enabled caching functionality, we notice an average increase of 0.009ms across both plain TCP and TLS when compared with a vanilla broker. This is significantly less than the specified 500ms in **NFR1** and thus meets this non-functional requirement and concludes the first research question: FlyTrap imposes an avarege of extra 20% overhead (0.03ms) per rerquest, when compared with the vanilla broker.

It is also interesting to see that the increased latency is much larger in TLS connections when compared to TCP. With TLS, not only do we have to deal with an extra overhead of having to contact blockchain to read authentication data, but also perform TLS encryption/decryption twice - the first time when communicating with the client and the second time when communicating with the broker.

6.6.2 Scalability Evaluation

On the machine used for testing the response times remained constant, indicating that the proxy is capable of handling many concurrent clients without significant impact on performance. If desired, multiple FlyTrap instances can also be deployed across many machines - each pointing to the same broker, further enhancing the response times. This concludes the second research question: FlyTrap scales in a constant time[3], up to 10000 concurrent clients, with no impact on response times.

It is important to note that FlyTrap has an overall limit of the maximum allowed concurrent connections of 65535 (equal to 2^{16}). Two limitations enforce this: first, MQTT Protocol can only accept client IDs of at most 2^{16} characters long, so naturally, it is not able to proxy more connections than that, since the broker would refuse them; second, FlyTrap operates on Linux TCP ports, with an allowed port range up to 65535. Where the latter problem could be fixed with WebSockets (discussed in the next chapter), the former would remain a limitation.

6.6.3 Cost Evaluation

For the cost evaluation, the most expensive operation would be the creation of the new contract, coming down to \$2.03. The contract contains the bare-bones definition of structures used by FlyTrap, so high price was to be expected. Though, this only needs to be performed once - the contract can be reused for as many topics as it is necessary. The remaining operations (which would also be performed more often), have much lower prices. A company looking to create 10 topics with 100 publishers would be looking at an investment of $\$2.03 + 10 * \$0.11 + 100 * \$0.05 = \$8.13$. This scales linearly, and cost remains the same, regardless of the current size of the contract and amount of pre-existing topics/publishers/subscribers.

Blockchain offers a solution with 100% uptime - and at the same time allowing for potential monetisation data access. The average cost is significantly higher, when compared to centralised solutions, such as SQL server hosted in the cloud - but it lacks the benefits of a distributed ledger. Ultimately, all pros and cons would need to be considered before deciding to utilise Ethereum as a data layer. This concludes the third research question: FlyTrap imposes a one-off network fee of \$2.03 with the subsequent operations costing between \$0.02 and \$0.11, which need to be performed ten times a day maximum.

6.6.4 Scenario Evaluation

Each of the scenario walkthroughs demonstrated how the software fulfils the business needs specified in Section 3.1.4, answering the fourth research question. The use-case of Scenarios #4 illustrated a need for stronger authentication and was omitted, since the setup phase and Scenarios #1, #2 and #3 already made use of authenticating connecting clients using public/private keypairs, thus fulfilling the requirement. For Scenario #5, please refer to the User Manual (Appendix A) for an explanation on how to set a cost on adding new publishers or subscribers.

[3]Implying $O(1)$, where n is the number of concurrent clients

Furthermore, by demonstrating that FlyTrap satisfies Scenarios #1, #2 and #3, we can also deduct that following requirements have been directly satisfied: FR1, FR2, FR6, FR7, FR8 and the following requirements have been indirectly satisfied: FR3, FR4, FR5.

Chapter 7

Conclusion, Discussion and Future Work

In this chapter, I conclude the work conducted in this paper, reflect on the results and suggest the next steps for the designed framework. I also overview my achievements, slowdowns and difficulties found.

7.1 Summary

In this dissertation, I have introduced a novel method of authenticating clients connecting to arbitrary MQTT Brokers through a middleware framework storing persistent information on Ethereum blockchain, allowing for higher transparency and availability. I have started by providing a motivation for the project, bringing up that standard MQTT protocol offers only limited authentication methods, with no option to log audit trails. Recent legislature requiring data administrators to adhere to a specific set of rules dictating how the data should be stored, maintained and erased if requested was also discussed.

As the project involved a lot of sophisticated software such as MQTT or Blockchain, a thorough background chapter was included in order to allow the reader to better understand the concepts and decisions taken throughout this thesis. An overview of related papers has also been included to make sure that my efforts were not duplicating any similar research.

Before discussing the design of the framework, I explored and listed requirements for the project formulated as user stories, scenarios and list of functional and nonfunctional requirements. Next, I proceeded with presenting a proposed architecture for FlyTrap, splitting it into four, independent modules: proxy, client, web app and blockchain - each capable of working on its own. I've discussed the data model stored on the blockchain and how access to sensitive topics is continuously logged to allow tracking back entities reading the data.

Then I went on to talk about the implementation details, discussing the workflows that I have been following, my testing strategy and how my development environment looked like. In particular, simulating both MQTT Broker and Blockchain posed a challenge, as both are somewhat sophisticated frameworks often requiring many modules to function correctly.

Finally, I designed experiments aiming to evaluate FlyTrap's feasibility, how severely the performance hit was, whether the software is capable of scaling up as the number of concurrent clients might go up, cost of operating on Proof-of-Stake blockchain network and reflected to scenarios to determine whether specified requirements have been met. The increased response time through FlyTrap ranged from 18.12% to 62.67% when compared to vanilla broker. FlyTrap was also found to scale effortlessly, with almost no loss as the number of clients increased. Operational cost included a one-off payment of circa \$2.03, with general costs of circa \$0.11 per operation.

Detailed User Manual and Maintenance Manual have also been included as appendices, to outline how to set up your own proxy and how to read the source code created in this project.

7.2 Discussion

7.2.1 Proof-of-Stake vs Proof-of-Authority

When I first introduced the concept of blockchain in this paper, I discussed the differences between proof-of-stake and proof-of-authority consensus algorithms. As a reminder, Proof-of-Stake involves mining rewards for adding new blocks to the chain (thus involves currency) and is currently used for the public Ethereum network, whereas Proof-of-Authority does not involve any rewards and all blocks are verified by a set of "sealers" whose job is to verify whether they can be processed or not.

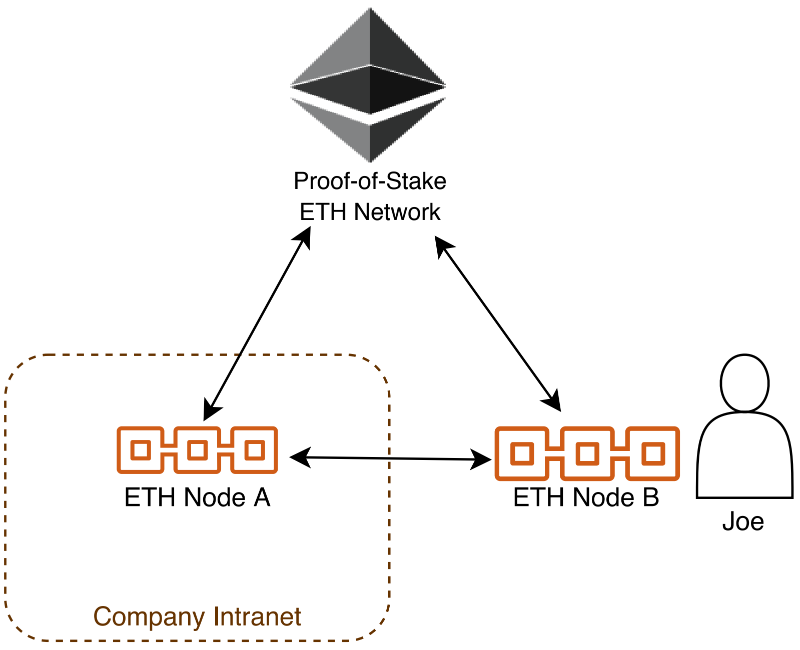


Figure 7.1: Public Proof-of-Stake network

Figures 7.1 & 7.2 illustrate those differences. In the first example, the company has complete control over its blockchain network, and nobody from the outside can read it or add new blocks. Even though Joe has configured his own node, he will not be able to connect to it. In the second example, the company is connecting to the public node, to which Joe - and anyone else - can also connect to read or submit new blocks.

Let us first look at Proof-of-Stake. This includes all benefits coming from blockchain architecture. That is, the entire layer is decentralised, meaning that hardware failure will not cause data loss, as every participant will be holding a copy. This ensures a pure 100% uptime of the data layer. Moreover, all transactions are transparent and immutable. Nobody can deny submitting it or alter the past by removing some blocks from the chain (which is not even possible, to begin with) - perhaps this would be the most beneficial for the authorities and law enforcement organisations, which no longer have to trust data provided by the company, as they can host their own node and read stored data.

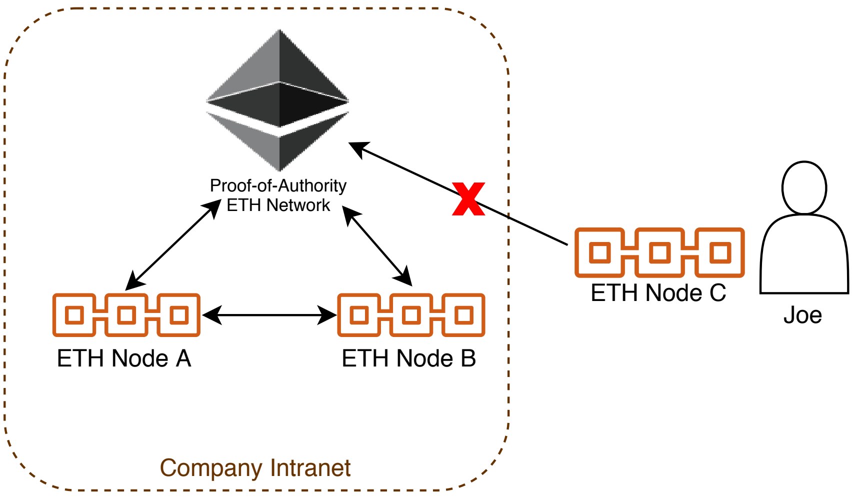


Figure 7.2: Private Proof-of-Authority network

7.2.2 Comparison with State-of-the-Art

Both IoT and Blockchain are still developing technologies, which only recently gained traction in the industry. The lack of a similar solution forced me to look beyond the original scope of the project and investigate papers which were also combining those two concepts. Because of that, state-of-the-art comparison involves only the un-changed MQTT Broker. I was satisfied to see only minor increase, while providing the extra security capabilities.

7.3 Reflection

This section will reflect back on the challenges that I faced when working on this project. I will also outline things that went exceptionally well and things that did not go as planned.

7.3.1 Achievements

Before deciding to work on this particular topic, I did not know MQTT or blockchain. I only briefly heard about such technologies, but never had a chance to apply them in any of my projects. The majority of time working on this dissertation was spent on background research and learning how to bring up my own Ethereum network, research industry trends and conducting a thorough evaluation to assess my solution.

I am also happy to say that I successfully followed my original project plan, only with minor deviations. At no point I was facing pressure or closing on deadlines, always leaving an extra week or two as a buffer in case things did not go as expected.

Writing such lengthy documentation was also always a challenge for me, so I am very proud that I was able to stay persistent and finalise this project, gaining valuable research experience.

7.3.2 Difficulties & Lessons Learnt

When starting to plan this project at the end of the last year, I never anticipated that I would have to conduct it in circumstances of a worldwide lockdown. I have to admit, working on my project entirely remotely has been a big challenge which made me appreciate face-to-face teaching & guidance even more. As much as my online meetings were going smoothly and without problems, I started to miss the whiteboarding in my supervisor's office - which was invaluable during brainstorming.

As I planned my risk management when writing the project plan, I didn't take global pandemic into account - and I don't think that anyone did. With remote work, I lost the benefit of networking with my peers and discussing our ideas, which provided an immense help before. We were able to overcome this challenge by organising a weekly remote meetup through videoconferencing to carry on, as we knew that the help that we can offer each other is invaluable.

I also learned to always ask for help if faced with a roadblock for an extended time. For example, at one point, I encountered a problem where my software was no longer capable of handling concurrent clients. I spent roughly 2 days dissecting my code line by line, to determine the root cause and eventually pinpointed it to the fact that the Client ID field needs to be unique for the broker to respond. When discussed it with one of my co-supervisors, I learned that he also faced a similar problem in the past and also got stuck for a couple of days. If I approached him earlier, I could have saved those two days.

Several technical challenges also came up, that I either had to omit from the final implementation or think about a workaround. One of the examples being the sheer cost of operating applications on a distributed platform. It is incredibly easy to start exhausting the resources (and thus bumping the price) exponentially, so I had to think about limiting potential writes to a minimum. That's how the caching and crafting data models came to life, as I started to become concious of available resources. Another example is TLS, which almost doubled the response times. I learned that sometimes it might be more desirable to decide against TLS in favour of plain TCP in order to have a more responsive solution.

7.4 Future Work

This section will outline the possible future for FlyTrap and how it could be brought forward with further enhancements to the provided security.

7.4.1 Testing TLS

Following up from the evaluation chapter, we observed a significantly increased performance loss when proxying TLS connections from Client towards FlyTrap and then from FlyTrap to MQTT Broker. Further testing could be needed to determine whether the impact is more significant on Client<->FlyTrap or FlyTrap<->Broker side and provide appropriate guidance on how to provide the best possible performance.

For example, for situations where both FlyTrap and MQTT Broker are located on the same network, it would be wasteful to encrypt packets flowing between FlyTrap and Broker, as man-in-the-middle attacks would not be possible, since all traffic occurs on loopback network device[1]. It would be interesting to see whether enabling TLS only on the first part of the journey provides any significant gains.

Furthermore, alternative methods of encrypting TCP packets should be explored. This project uses the standard TLS session with all settings left at default, so tests involving different cypher suites and different key-exchange algorithms would be beneficial.

7.4.2 WebSockets

The most recent versions of popular MQTT Brokers (such as Mosquitto) introduced support for WebSockets - with many of the testing online brokers already offering support, e.g. HiveMQ

[1]Also known as localhost, i.e. inter-process communication through TCP.

WebSocket-based broker can be accessed under broker.hivemq.com:8000. WebSocket [?] is built on top of HTTP layer, meaning that all traffic flows through a single port. It is no longer necessary to utilise multiple ports to send concurrent messages.

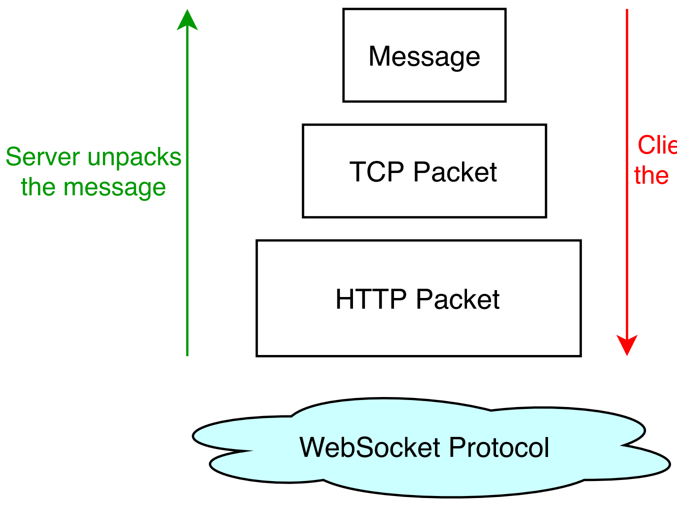


Figure 7.3: Overview of WebSocket protocol

Figure 7.3 demonstrates how messages are encapsulated by the client and then unpacked by the server. The most apparent benefit is the increased number of possible proxy connections. FlyTrap currently is tied to the maximum of 65535, which is the maximum number of ports on Linux. Additionally, many firewalls forbid outgoing and incoming connections from non-standard ports. Whenever a new proxy connection is initiated, FlyTrap opens a new ephemeral port, which could otherwise be blocked.

At the moment, the framework operates solely on the TCP layer, sending raw TCP/TLS packets. By adding support for the WebSocket protocol, the limit of maximum concurrent connections could be increased, and by limiting it to a single port (for example, 80), it could become more accessible to devices behind strict firewalls.

7.4.3 Device Authenticity

FlyTrap currently relies only on authentication through a signature containing the public key, signed by the corresponding private key. While it works perfectly fine for the purposes of this project and provides a reasonable protection barrier from attackers, it does not take into consideration situations where the IoT device containing the private key/signature gets stolen, and its data becomes compromised. At this point, the attacker could calculate their own signatures and start making requests against FlyTrap. IoT devices are often neglected when it comes to physical security - they are often placed in public spaces, where anyone could potentially remove it. Normally, it is not a concern, as they are not targeted due to their low cost, but by containing secret data (and signature / private key could be considered secret), it might cause them to become a target.

This problem fell out of scope for this project, and thus it was not considered when designing the system. To tackle this, one might want to introduce methods of verifying the authenticity of connecting devices. In 2012 Apple filed a patent with the United States Patent and Trademark Office [?] offering a solution for this problem. In the future, ideally, FlyTrap should be enhanced by a similar solution, such that it would be able to identify whether the connecting device is a genuine IoT sensor or an attacker connecting from a laptop.

7.5 Conclusion

Through my work in this project, I have shown how Ethereum and MQTT can be combined to provide an efficient Authentication and Accountability framework for MQTT brokers by utilising decentralised blockchain as a data layer. This solution provides a novel set of features such as brute force attack protection, data access monetisation (by collecting payments in a form of cryptocurrency) or a user-friendly front-end to browse the past transactions in order to answer common GDPR questions. The evaluation also proved that the performance impact is acceptable with regards to the gained features. Due to the complex set of dependencies, it is currently aimed at business that are already using blockchain and MQTT, since for them the setup overhead would be minimal. But with the increasing popularity with decentralisied solutions and IoT, FlyTrap could also help business looking to switch, as all needed components (such as MQTT client, front-end or Blockchain CLI) are provided as part of the solution. FlyTrap also has an exciting collection of future work suggestions, which might make it even more accessible and attractive to the potential customers..

Appendix A

User Manual

This appendix provides instructions on how to bring the proxy up and how to operate each of the project's modules.

A.1 Blockchain

For simplicity, this guide will be using Blockchain network created through Ganache. It is not essential and any other Ethereum node can be used, which should be accessible via TCP connection.

To install Ganache, please follow instructions on https://www.trufflesuite.com/ganache to download AppImage and execute it from command line, like shown in listing A.1. This should bring up interface similar as shown on figure 5.2. Take note of the top two accounts, we will be using them in further examples.

Listing A.1: Launching Ganache

```
$ chmod +x ganache-2.3.1-linux-x86_64.AppImage
$ ./ganache-2.3.1-linux-x86_64.AppImage
```

You can click on the key icon on the far right in order to retrieve account's private key. I recommend saving the first one as 'privkey.asc' and the second one as 'clientPriv.asc'. Those will be our private keys used in further examples.

A.2 MQTT Broker

We will be using HiveMQ test broker located under broker.hivemq.com:1883, so you don't need a local instance on your machine. HiveMQ doesn't support TLS connections, so all examples will have tls flag set to false, though if desired, it can be set to true, as long as the target support TLS and provides a certificate.

A.3 Parameters

A.3.1 Common env variables

FlyTrap and Blockchain CLI makes use of two environmental variables. It is very important to set them before performing any action:

FLYTRAP_CONTRACT - public address of FlyTrap's smart contract. You will not know it until you run blockchain CLI with -new flag. Once you obtain it, it's recommended to update this variable.

BLOCKCHAIN_ADRRESS - endpoint to access Ethereum node. For Ganache, default is http://localhost:7545.

I recommend to put the following two lines in your .bashrc file (or corresponding rc file, if your shell is different than bash):

```
export BLOCKCHAIN_ADDRESS="http://localhost:7545"
export FLYTRAP_CONTRACT="<contract address>"
```

A.3.2 Flytrap

```
$ go build -o flytrap cmd/flytrap/main.go
$ ./flytrap -help
Usage of ./flytrap:
  -b string
        Provide address of MQTT Broker. (default "test.
           mosquitto.org:8883")
  -bcrt string
        location of broker's cert for TLS connection (default
           "broker.crt")
  -crt string
        Location of your server's .crt used for TLS connection
            (default "server.crt")
  -freq duration
        How often usage summaries should be saved on
           blockchain (default 30s)
  -key string
        Location of your server's .key file used for TLS
           connection (default "server.key")
  -p string
        Port to use for proxy server (default ":8888")
  -tls
        Whether the proxy should be encrypted on both sides.
           You will need to provide .crt and .key files if so.
            (default true)
```

IMPORTANT You need to set BLOCKCHAIN_ADDRESS and FLYTRAP_CONTRACT environmental variables, as they are used to determine respectively address of blockchain node to connect to and address of FlyTrap's contract on blockchain where the ACL rules are located at.

A.3.3 Blockchain

```
$ go build -o blockchain cmd/blockchain/main.go
$ ./blockchain -help
Usage of ./blockchain:
  -client string
```

```
        Public key of client to manipulate
  -contract string
        Specify address of your contract
  -country string
        topic will only be accessible to people connecting
           from this country. 2 letter ISO code. (default "GB"
           )
  -date string
        Provide date that should be searched for. E.g.
           2020-02-29
  -logs
        Receive relevant event logs. Providing -client and -
           topic flags will restrict the search only to
           relevant fields.
  -new
        Whether to create new contract for flytrap
  -new_topic string
        Reason to create a new topic
  -pub string
        Reason to add as publisher
  -pub_cost int
        cost of adding as publisher in wei
  -rpub string
        Reason to revoke as publisher
  -rsub string
        Reason to revoke as subscriber
  -s    whether new topic is to be marked as sensitive (
     default true)
  -sub string
        Reason to add as subscriber
  -sub_cost int
        cost of adding as subscriber in wei
  -topic string
        name of the topic to modify (default "MyTopic1")
  -v    Enable verbose mode
```

IMPORTANT You need to set BLOCKCHAIN_ADDRESS and FLYTRAP_CONTRACT environmental variables, as they are used to determine respectively address of blockchain node to connect to and address of FlyTrap's contract on blockchain. Those values are needed to run any commands (except creating new contract, for that only the former is required).

A.3.4 MQTT Client

This binary is used to use MQTT requests towards the broker. Please note, you should place clientPriv.asc file that you obtained from Ganache and place it adjacent to the binary. During execution, the client will produce an extra pub_clientPriv.asc and sig_clientPriv.asc files which serve as cache to avoid repeated signature calculations.

```
$ go build mqttclient/client.go
$ ./client -help
Usage of ./client:
  -conn_test
        whether to only connect and then immediately
           disconnect
  -crt string
        location of cert for SSL/TLS connection (default "
           flytrap.crt")
  -f    Whether to force recomputation of signature
  -id string
        ID of connecting client
  -ip string
        location of MQTT broker (default "localhost:8888")
  -msg string
        message to be published (default "Here Be Dragons")
  -priv string
        Location of your private key file (default "privkey1.
           asc")
  -pub int
        How many messages to publish
  -sub int
        How many messages to receive via subscription
  -tls
        whether to use TLS (default true)
  -topic string
        Topic for use for pub/sub (default "MyTopic1")
```

A.3.5 WebApp

WebApp doesn't offer any command line parameters and by default will run on port :8081. If you would like to change this to something else, you can edit the constant in Line 14 of webapp/-main.go file.

FLYTRAP_CONTRACT is an optional env variable. If set, the contract field will be pre-populated when website is rendered. Otherwise, it will remain blank.

A.4 Usage Examples

This section will walk you through how to use each of the modules to perform all functionality offered by FlyTrap.

A.4.1 Blockchain

Operations described in this section will demonstrate how to manipulate and interact with FlyTrap's smart contract located on Ethereum. Before running any of those examples, make sure you have configured your env variables as per section A.3.1.

IMPORTANT Make sure privkey.asc is in the same directory from where you execute the go run commands.

A.4.1.1 Creating a new contract

This operation will create a new contract on Blockchain and output its public key. Make sure your account has enough funds, as it might cost some gas.

```
$ go run cmd/blockchain/main.go -new -contract=""
2020/04/20 18:02:37 Generated new contract, address is: 0
   x278AC0391ceA1E7664D4242C7398FCDe79539a93
```

I **strongly** recommend to save the obtained address in environmental variable as per section A.3.1.

A.4.1.2 Creating a new topic

This operation will create a new topic on FlyTrap's contract.

```
$ go run cmd/blockchain/main.go -new_topic="reason to create
   topic" -topic="MyTestingTopic" -pub_cost=0 -sub_cost=0
```

As a -new_topic param you need to provide a reason to create a new topic, along with -topic flag to specify its name. If pub cost and sub cost is set to 0, then only topic's owner can alter its state. Otherwise, anyone can do it, provided they pay required amount (which would be transferred to the original owner)

A.4.1.3 Adding subscribers/publishers

```
$ go run cmd/blockchain/main.go -topic="MyTestingTopic" -
   client="<client_pubkey>" -pub="Addining a new publisher"
```

Under -client flag, you can provide corresponding pubkey, as obtained from Ganache. This will allow it publish to MyTestingTopic. To add as a subscriber, change flag pub to sub.

A.4.1.4 Revoking subscribers/publishers

```
$ go run cmd/blockchain/main.go -topic="MyTestingTopic" -
   client="<client_pubkey>" -rpub="Revoking publisher"
```

Similar to adding, with the only exception that pub/sub flag is changed to rpub/rsub. This command will undo the addition and forbid specified client from further interaction with MyTesting-Topic.

A.4.2 FlyTrap

This section will explain how to bring up your own FlyTrap proxy and connect it to HiveMQ test broker.

A.4.2.1 Starting up FlyTrap

IMPORTANT Make sure privkey.asc is in the same directory as FlyTrap and BLOCKCHAIN_ADDRES & FLYTRPAP_CONTRACT environmental are set and active. You can verify it by running '$ echo $BLOCKCHAIN_ADDRESS $FLYTRAP_CONTRACT'. If you can see both values output to the terminal, you are ready to go.

```
$ echo $BLOCKCHAIN_ADDRESS $FLYTRAP_CONTRACT
http://localhost:7545 0
   x61c4603936627946aAb47a86e135b983171eFF44
$ go run cmd/flytrap/main.go -b="broker.hivemq.com:1883" -tls=
   false -freq=24h -p ":7777"
2020/04/20 18:26:12 Now accepting connections under :7777
```

This will initiate proxy under port :7777, generating reports for sensitive topics every 24hrs and connecting to MQTT broker located under broker.hivemq.com:1883. I recommend to minimize this window and move forward to the next section explaining how to submit messages through FlyTrap.

A.4.3 MQTT Client

This section will explain how to use provided MQTT client to subscribe and publish your own messages.

IMPORTANT For those tests, make sure that the private key clientPriv.asc that you obtained from Ganache is in your working directory.

NOTE This client can also be used for any other MQTT Broker, i.e. -ip parameter can either point to FlyTrap or directly to the Broker (e.g. broker.hivemq.com:1883)

A.4.3.1 Publishing

```
$ go run client.go -pub=3 -topic="MyTestingTopic" -priv="
   clientPriv.asc" -ip="localhost:7777" -tls=false -f
2020/04/20 18:42:43 Published message: Here Be Dragons #0
2020/04/20 18:42:43 Published message: Here Be Dragons #1
2020/04/20 18:42:43 Published message: Here Be Dragons #2
2020/04/20 18:42:43 Published 3 messages, as requested,
   disconnecting
```

Command above will publish three messages to the topic MyTestingTopic under a broker located on localhost:7777 (where FlyTrap is currently running). We also specified the exact location of our private key and disable TLS (as testing hivemq broker doesn't support it). -f flag is used to compute signature again, regardless whether it's cached or not.

We can also refer back to our FlyTrap terminal to inspect that the messages where indeed received and passed forward to the broker:

```
2020/04/20 18:42:42 Now accepting connections under :7777
2020/04/20 18:42:43 Starting new proxy connection ([::1]:54764
    >> broker.hivemq.com:1883)
2020/04/20 18:42:43 Client "0
   x4514A540Cf6c81B482Ac88d0627d4F71e9363885" was authorised
   to publish to topic "MyTestingTopic"
2020/04/20 18:42:43 Client "0
   x4514A540Cf6c81B482Ac88d0627d4F71e9363885" was already
   authorised to publish to topic "MyTestingTopic"
2020/04/20 18:42:43 Client "0
   x4514A540Cf6c81B482Ac88d0627d4F71e9363885" was already
   authorised to publish to topic "MyTestingTopic"
2020/04/20 18:42:43 Proxy Connection terminated gracefully.
   ([::1]:54764 >> broker.hivemq.com:1883)
```

A.4.3.2 Subscribing

```
$ go run client.go -sub=3 -topic="MyTestingTopic" -priv="
   privkey1.asc" -ip="localhost:7777" -tls=false -f
```

Subscribing works very similar to publishing, with the only exception that client will maintain connection until specified number of messages has been received (in this case, three) Again, we can see relevant runtime logs on FlyTrap's terminal window:

```
2020/04/20 18:46:35 Starting new proxy connection ([::1]:54824
    >> broker.hivemq.com:1883)
2020/04/20 18:46:35 Client "0
   x4514A540Cf6c81B482Ac88d0627d4F71e9363885" was authorised
   to subscribe to topic "MyTestingTopic"
2020/04/20 18:50:05 Proxy Connection terminated gracefully.
   ([::1]:54824 >> broker.hivemq.com:1883)
```

A.4.4 WebApp

IMPORTANT Make sure privkey.asc from Ganache is in your working directory.

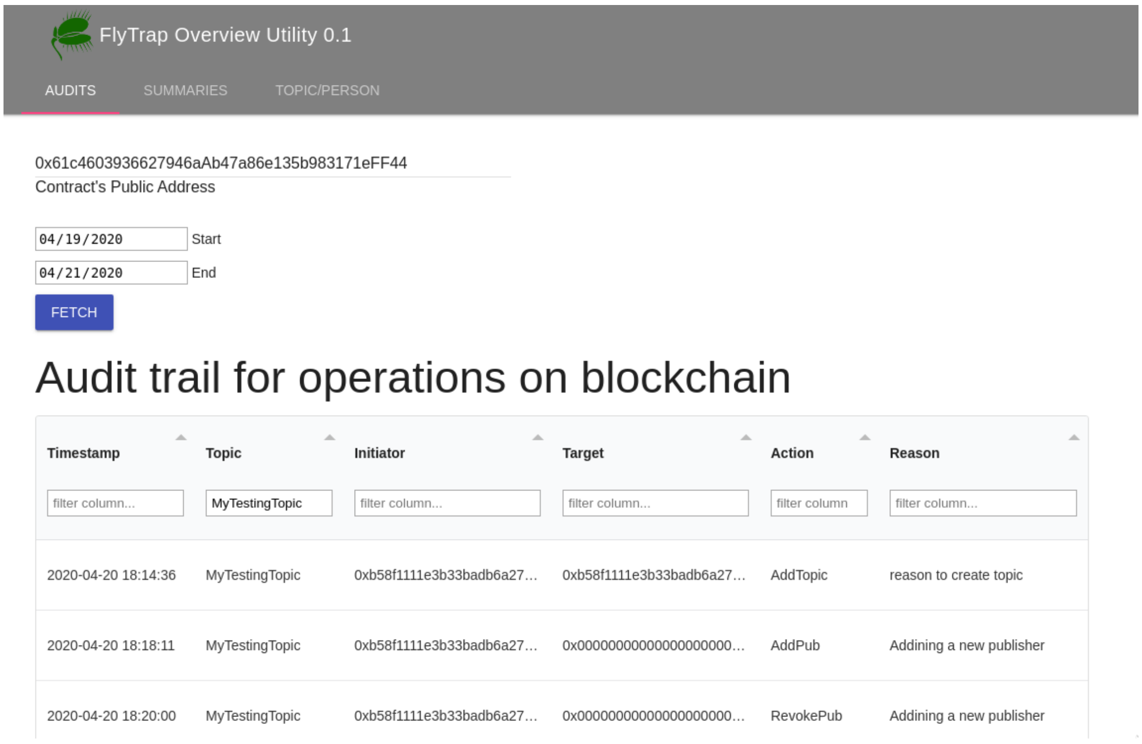
Web application provides an extension to the project providing an easy overview of the data stored on blockchain. It queries the smart contract directly and does not require any of the remaining components to function. To start it, run the following command in your terminal:

```
$ go run main.go
2020/04/20 18:53:06 Server is now running under :8081
```

You can now navigate to http://localhost:8081 in a browser of your choice to access the application.

A.4.4.1 Audits

First tab is called 'Audits' and contains information about all administrative actions taken on the given contract. The page will load empty (depending whether you set FLYTRAP_CONTRACT variable, 'Contracts Public Adress' might already by pre-populated). Enter the public address obtained when creating a new contract and time constraints. Please note, higher constraints might result in longer loadtime, so be sure to be as specific as possible.



0x61c4603936627946aAb47a86e135b983171eFF44
Contract's Public Address

04/19/2020 Start
04/21/2020 End

Audit trail for operations on blockchain

Timestamp	Topic	Initiator	Target	Action	Reason
filter column...	MyTestingTopic	filter column...	filter column...	filter column	filter column...
2020-04-20 18:14:36	MyTestingTopic	0xb58f1111e3b33badb6a27...	0xb58f1111e3b33badb6a27...	AddTopic	reason to create topic
2020-04-20 18:18:11	MyTestingTopic	0xb58f1111e3b33badb6a27...	0x00000000000000000000...	AddPub	Addining a new publisher
2020-04-20 18:20:00	MyTestingTopic	0xb58f1111e3b33badb6a27...	0x00000000000000000000...	RevokePub	Addining a new publisher

Figure A.1: Audits tab of the web application

When finished loading, "Audit trail for operations on blockchain" table will be populated with the audit trail. It consists of six columns:

Timestamp - when the operation was performed

Topic - affected topic

Initiator - public key of the person initiating the operation

Target - public key of the person affected by the change

Action - what action was performed

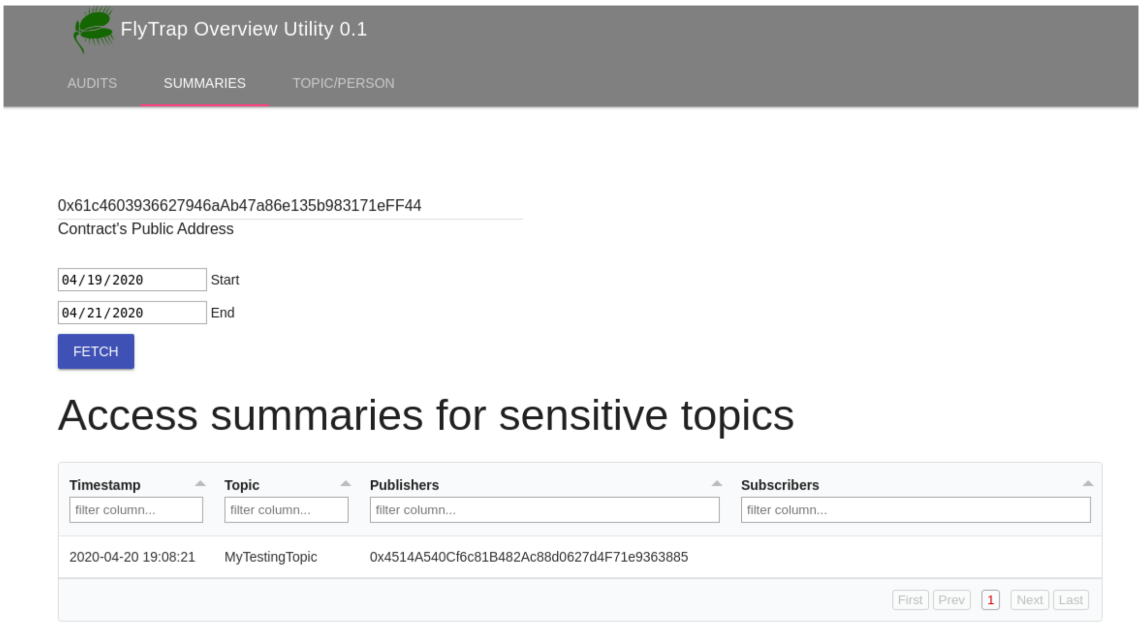
Reason - reason provided by the initiator

The table can be then further filtered by filling the textboxes at the top of every column. In the example shown in figure A.1, Topic column has been restricted only to actions containing MyTestingTopic. We can see the operations that were performed earlier in this manual.

The filters can be also further combined for even more precise search, e.g. to look up all AddPub events occurred on MyTestingTopic, we'd also fill the text box at Action column.

A.4.4.2 Summaries

This tab provides an overview of all logs registered on sensitive topics. As explained in section 4.2.3.2, their frequency relies on -freq field set when running FlyTrap.



0x61c4603936627946aAb47a86e135b983171eFF44
Contract's Public Address

04/19/2020 Start
04/21/2020 End

Access summaries for sensitive topics

Timestamp	Topic	Publishers	Subscribers
2020-04-20 19:08:21	MyTestingTopic	0x4514A540Cf6c81B482Ac88d0627d4F71e9363885	

Figure A.2: Summaries tab of the web application

Similar to Audits tab, we also need to fill the public address and time constraints. When ready, we can click "Fetch" which will populate the table, as shown in A.2. This time, the table consists of 4 columns.

Timestamp - when the report was generated

Topic - topic for which the report was generated

Publishers - list of public keys that published to the given topic in the past X minutes from the specified Timestamp, where X is the frequency set in FlyTrap's command line parameters.

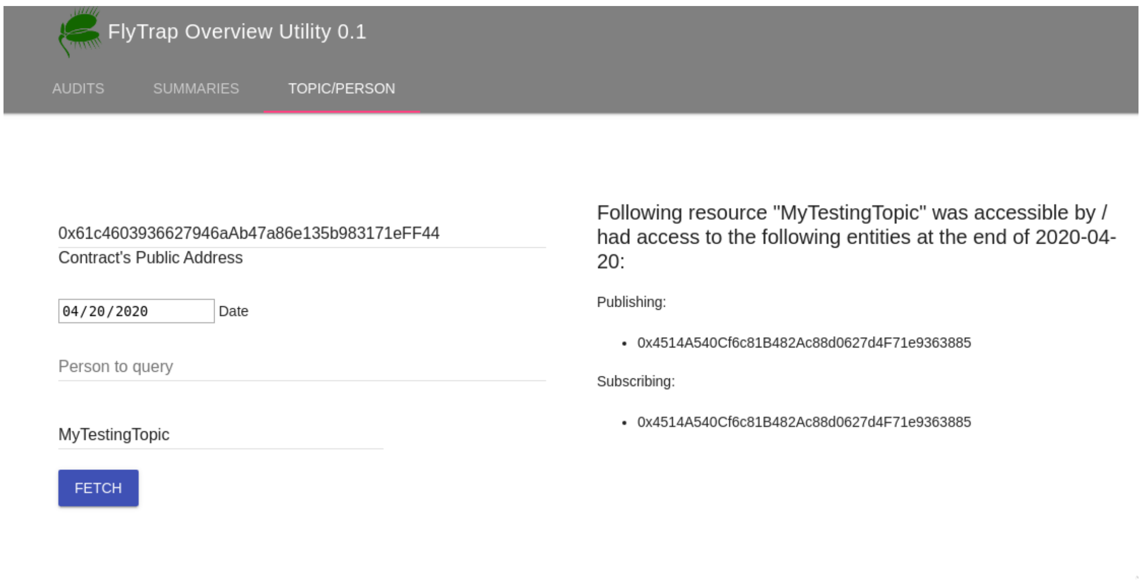
Subscribers - similar as publishers, but for subscribers.

This table can also be further filtered, by using text boxes at the top of every column, to provide more precise records.

A.4.4.3 Topic/Person

This tab allows to look into the past and verify who had publishing/subscribing access to the given topic - or which topic given public key could publish/subscribe to. This is helpful for situation when a leak happens and you need to quickly determine who can access given resource, in order to revoke it.

Regardless whether we want to find out the first or the latter, we need to again fill the contract's public address along with a date. The results then will show access summaries at the end of specified day.



0x61c4603936627946aAb47a86e135b983171eFF44
Contract's Public Address

04/20/2020 Date

Person to query

MyTestingTopic

Following resource "MyTestingTopic" was accessible by / had access to the following entities at the end of 2020-04-20:

Publishing:

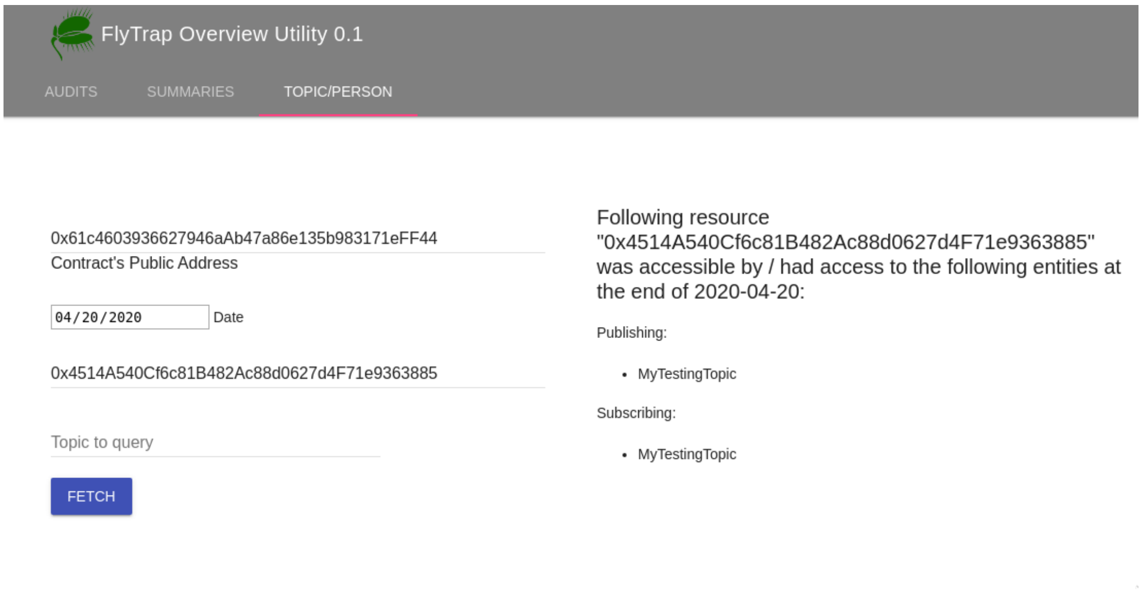
- 0x4514A540Cf6c81B482Ac88d0627d4F71e9363885

Subscribing:

- 0x4514A540Cf6c81B482Ac88d0627d4F71e9363885

Figure A.3: Topic access summary

As shown on figure A.3, if we fill "Topic to query" text field, we we will obtain a result telling us that public key "0x4514A540Cf6c81B482Ac88d0627d4F71e9363885" was allowed to both publish and subscribe at the end of 2020-04-20.



0x61c4603936627946aAb47a86e135b983171eFF44
Contract's Public Address

04/20/2020 Date

0x4514A540Cf6c81B482Ac88d0627d4F71e9363885

Topic to query

Following resource "0x4514A540Cf6c81B482Ac88d0627d4F71e9363885" was accessible by / had access to the following entities at the end of 2020-04-20:

Publishing:

- MyTestingTopic

Subscribing:

- MyTestingTopic

Figure A.4: Person access summary

And if we fill "Person to query" text box - as per figure A.4, we will find out that the given public key could both publish and subscribe to MyTestingTopic.

Appendix B

Maintenance Manual

B.1 Prerequisites

To bring the whole framework up, three elements will be needed:

- Linux - at the moment, the software stack is only compatible with Linux environments. It has been tested on ArchLinux and Ubutnu 16.04 - and for those distributions operation can be guaranteed. For other Linux distros, your mileage may vary.
- Golang - to compile (or run) all source files. All third-party Golang packages required by this project are managed through Go Modules[1], so there is no further action needed. Upon compilation, Go runtime will download all requirements and place them in your $PATH (provided you have successfully installed Go). If desired, they can be manually inspected in 'go.mod' file in project's root.
- Blockchain Node - either a dummy one provided by Ganache or a live node started through Geth. Remote nodes are also accepted.
- MQTT Broker - either a local broker (e.g. mosquitto on Ubuntu) or an online broker, reachable via TCP/TLS (e.g. test.mosquitto.org)

B.2 Compiling Source Code

Project consists of 4 binaries, which can be compiled as follows. Each 'go build foo.bar' operation will produce a binary named 'bar', which then can be executed by running $ ./bar. To specify exact name of the produced binary, an extra -o flag can be provided. Note, compilation is optional and all sourcecode can be also interpreted by the runtime, this can be achieved by replacing 'go build' with 'go run'.

See below on how to compile binaries for each of the modules:

```
# compile blockchain CLI
$ go build -o blockchain blockchain/cmd/blockchain/main.go
# compile flytrap
$ go build -o flytrap flytrap/cmd/flytrap/main.go
# compile mqttclient
$ go build -o mqttclient mqttclient/client.go
```

[1]https://blog.golang.org/using-go-modules

```
# compile webapp. note, 'templates' folder has to be adjacent
   to the produced binary when it is executed
$ go build -o webapp webapp/main.go
```

B.3 Project File Tree

```
root
├── blockchain
│   ├── cmd
│   │   └── blockchain
│   │       └── main.go
│   ├── blockchain.go
│   ├── flytrap.go
│   ├── flytrap.sol
│   └── logging.go
├── flytrap
│   ├── cmd
│   │   └── flytrap
│   │       └── main.go
│   ├── proxy.go
│   ├── summary.go
│   ├── verification.go
│   └── GeoLite2-Country.mmdb
├── mqttclient
│   └── client.go
├── webapp
│   ├── templates
│   │   ├── index.html
│   │   └── script.js
│   └── main.go
└── evaluation
```

B.4 Files Overview

Files that can be compiled and are considered main entry point for the given component are marked with an asterisk (*)

B.4.1 Blockchain

This module contains code responsible for communicating with blockchain.

blockchain.go contains all logic related to making calls to Blockchain to retrieve or write data. It's referenced by FlyTrap when checking access permissions.

cmd/blockchain/main.go * main executable and CLI when communicating with blockchain. Executing this file.

flytrap.sol smart contract definition written in Solidity.

flytrap.go Solidity code translated into Golang, which then can be used to make calls towards blockchain.

logging.go logic to query state of blockchain in order to pull logs to be used in web frontend.

B.4.2 Flytrap

This module contains the main logic behind FlyTrap proxy.

cmd/flytrap/main.go * main executable of FlyTrap proxy.

summary.go code containing logic to cache publishers/subscribers and dump them periodically as a transaction to blockchain.

verification.go logic to query the GeoLite2 database in order to translate IP into a country that it is located in.

GeoLite2-Country.mmdb MaxMind database file used to lookup IP addresses.

B.4.3 MQTT Client

client.go * simple MQTT client used to PUBLISH/SUBSCRIBE to MQTT Brokers, enhanced with extra functionality needed by FlyTrap calculating and attaching signature to CONNECT packets.

B.4.4 Webapp

This module is used to start up a frontend instance for FlyTrap allowing inspecting of data stored on blockchain.

templates HTML + JS files used to render the website.

main.go * backend used to handle incoming requests and query blockchain, returning information filling the website.

B.4.5 Evaluation

This folder contains dump of all tests results performed for the purposes of evaluation.