Big data in consumer marine environment - real-time streaming and long-term storage

Robert Kantero
Master's thesis in Computer Engineering
Faculty of Science and Engineering
Åbo Akademi University
Supervisor: Jerker Björkqvist
2023

Abstract

The relevance of Internet of Things (IoT) and big data are ever growing as we continue to digitize the world we live in. The space of consumer leisure boating has not yet followed these trends as well as many other industries. In this thesis the architecture, design, and implementation of a platform enabling real-time data streaming and big data gathering from integrated plotters is presented. Research is made into viable technologies and design choices through various outlets and is discussed with senior staff of Nextfour. The implemented platform works as intended at a current relatively small scale, but will need further development in order to reach its full potential. The platform functions as an initial step in bridging the gap between IoT and big data applications in the consumer leisure boating and other more mature industries, such as the automotive industry.

Keywords: Internet of Things, IoT, big data, MQTT, time-series data, marine, boating

Acknowledgements

First and foremost I'd like to thank Nextfour for giving me the opportunity to do my Master's thesis for the company. All staff were incredibly helpful in answering questions and providing me with guidance, discussions, and resources. I especially want to thank my team-leader and supervisor in spirit, Marko Pekkala, who not only thought of the thesis project to begin with, but also allowed me extensive freedom in designing and implementing the project, and was always eager to engage in conversations about my ideas. Lastly, I want to extend my gratitude to my actual supervisors, Jerker Björkqvist from Åbo Akademi University and Johan Wessberg from Nextfour.

Contents

Chapter 1

Introduction

The value of data is ever-increasing in today's society. Companies use all kinds of data to better tailor their business models and increase profits and efficiency. On the rise for some time now, is the concept of big data. The aim is not to have a predefined metric in mind and collect the relevant data as traditionally, but to gather vast amounts of data for some machine learning models or other advanced analytical applications to use. The results of these analyses may vary from completely expected metrics, to new insights not thought of before, depending on the analytical applications used.

The space of consumer boating has not yet reached the same level of digitization as many other markets. In the automotive industry, connected On-Board Units (OBU) have been standard for many years already, and a subsection of Internet of Things (IoT) - Internet of Vehicles (IoV)[1] - can be said to have emerged. Nextfour's product Q Display is a step in this direction. Q Display is an integrated plotter for consumer boats, with all functionalities of traditional analog gauges, paper charts, and much more. Q Display is connected to the boat's engine/motor ECU, battery, fuel sensors, and other possible equipment such as VHF radio, sonar, radar, etc. It is also connected to the internet to enable weather services and software updates, amongst other things. All these connections mean large amounts of different data, which all go to waste if not logged and stored somewhere.

The project at hand is to create a platform enabling big data collection from Q Displays, as well as real-time data streaming. Other internal projects will then be able to leverage these services in order to create meaningful analyses of the gathered data. Customers - the boat manufacturers Nextfour is in business with - will be provided access to their Q Displays for real-time streaming. The decision on how, or if, to provide customers with the raw big data collected from their Q Displays is yet to be decided.

This thesis presents the challenges of designing and implementing the project, the solutions to said challenges and the technologies which the project utilizes. After this brief introduction, the overarching architecture will be presented. This is followed by

several chapters of showing how the different technologies have been leveraged in order to achieve the desired outcomes. After the technical chapters, the methodology and results are discussed. The thesis then ends with the conclusion, and a summary in Swedish.

Chapter 2

Project architecture

2.1 Overview

Initially, there were only the Q Displays in the field, and their corresponding back-end server. For simplicity's sake, in this thesis it is referred to as Q-server. The Q Displays and Q-server handled all communication amongst themselves. Now, three more entities are introduced: one service for databases, one for message brokers, and the clients who may use these new services. The database service is called ThunderQloud, while the message broker service is called HailstQrm.

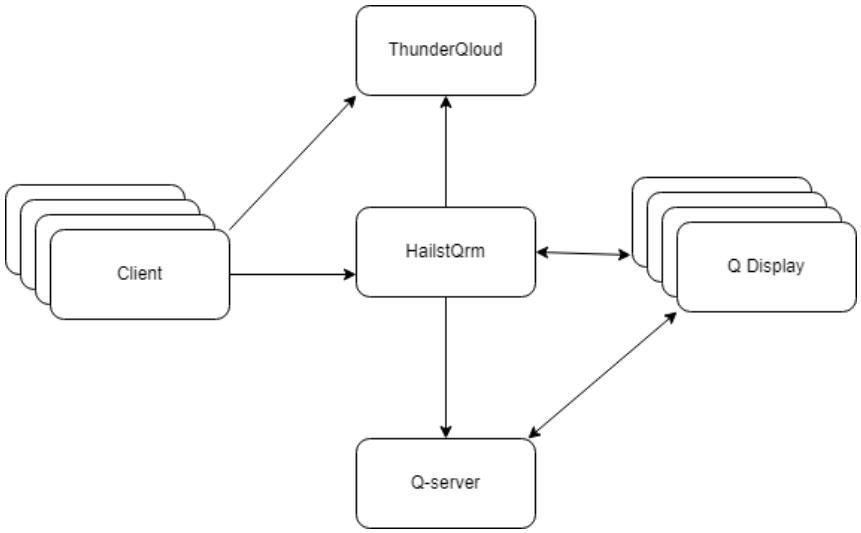

Figure 2.1: Architecture overview

2.1.1 Clients

Clients can be whatever end user leveraging HailstQrm and/or ThunderQloud. A Client can be a human sitting at their computer, a script, or another service completely.

2.1.2 Message brokering - HailstQrm

HailstQrm is a virtual machine hosted in the cloud, running two services. The services are two instances of MQTT message brokers.

2.1.3 Databases and long-term storage - ThunderQloud

ThunderQloud is similar to HailstQrm, but the services it runs differ slightly. There are two database instances along with their own web servers, an Nginx server, and a handful of Python scripts.

2.1.4 Back-end server - Q-server

Q-server is the backbone for the Q Displays' operations, which are outside of the scope of this thesis. Some additional implementation to Q-server which allows for HailstQrm and ThunderQloud functionality will be presented in chapter 5.

2.1.5 Q Display

Q Displays are the integrated plotters in consumer boats, which handle large amounts of data. Some critical pieces of data are already extracted, but this project aims to further extend the availability of said data.

2.2 New services

As no architectural changes are needed for Q-server or the Q Displays, deeper dives into their architecture is outside the scope of this thesis. HailstQrm's and ThunderQloud's architectures will be explained further.

They are both running on their own virtual machine instances, under the same cloud project. This greatly increases ease in networking configuration, as they reside in the same Virtual Private Cloud. The VPC has its own network, meaning the VM instances share this network and can communicate with each other. It is also easy to configure firewall rules to the outside internet via this VPC. This allows fine-grained control of which protocols and from which sources clients can connect to the VMs and the services running inside them.

The VMs run a generic Linux operating system, together with a custom startup script. This script pulls the latest version of the project branch's source code and executes another service-specific startup script found there. This way the VMs' startup scripts remain untouched, and changes to the actual source code's startup scripts will not be reflected in

the VM instances' startup script. Both service-specific startup scripts ensure the presence of all necessary dependencies, namely Docker Compose[2], export environment variables, and run commands to start the services specified in their compose.yaml files.

2.2.1 HailstQrm

HailstQrm consists of two services specified in its compose.yaml file. There are two instances of MQTT brokers: Broker and Bridge. The Broker and Bridge services are official Docker images of eclipse-mosquitto[3], with custom configurations.

2.2.2 ThunderQloud

ThunderQloud's list of services is rather robust compared to that of HailstQrm. There are two instances of QuestDB database Docker images[4] - QDB and QDB2 - a service running the official Nginx[5] Docker image, as well as three instances of Python services running their scripts. The Python scripts are run in Python's official alpine Docker images[6]. Additionally, there are two persistent disks mounted onto this VM instance. One disk is used by the databases, as their data needs to be persisted after reboots. The other is used for long-term storage.

Chapter 3

Message brokering with MQTT

This chapter discusses HailstQrm's archiutecture and the services it runs in more detail. Message Queuing Telemetry Transport (MQTT) and its implementation is also presented.

3.1 MQTT Technical specifications

Two types of entities are present in MQTT v3.1.1, Clients and Servers. It is always Clients who establish the network connection to the Server. A Client is a program or device that uses MQTT, and it can do four things:

- Publish Application Messages to a certain Topic
- Subscribe to a Topic in order to receive Application Messages published to that Topic
- Unsubscribe from a Topic
- Disconnect from the Server.

A Server is also a program or device, acting as an intermediary between Clients. The Server also does four things. It:

- Accepts Network Connections from Clients
- Accepts Application Messages published by Clients
- Processes Subscribe and Unsubscribe requests from Clients
- Forwards Application Messages that match Client Subscriptions.

Control Packets are the packets carried across the network by the MQTT protocol. Different types of Control Packets have different structures, but for this application the most interesting types are the PUBLISH, SUBSCRIBE, and UNSUBSCRIBE Control Packets. The PUBLISH type Control Packet is used to convey Application Messages, while the SUBSCRIBE and UNSUBSCRIBE type Control Packets are used to un/subscribe to Topics.

PUBLISH type Control Packets have header fields for Quality of Service and a Topic Name. Quality of Service determines how many times the packet is to be sent. It can be sent at most once, at least once, or exactly once. This enables reliability in situations with unstable network connections. The Topic Name must also be specified for each packet, as the basic concept of MQTT is to publish to and to subscribe to certain Topics. The Application Message itself is located in the payload of the packet.

SUBSCRIBE type Control Packets' payload need to include a maximum Quality of Service/Topic Filter pair. This means the subscribing Client is required to choose their own Quality of Service, regardless of what the publishing Client chose. The Topic Filter is an expression contained in a Subscription, to indicate an interest in one or more Topics. A Topic Filter can include wildcard characters.

The UNSUBSCRIBE type Control Packets' payload contain at least one Topic Filter to unsubscribe from. [7]

3.2 Implementation

There are multiple aspects that pertain to the implementation of the MQTT stack in this project.

- Communications need to be secure
- MQTT Servers are needed
- A topic structure is to be defined, so that communications can be organized and different services can utilize them efficiently.

In the following subsections, these points will be detailed further.

3.2.1 MQTTs - secure MQTT

This section will cover the authentication aspect of secure communications, while the server configuration section will cover authorization.

Eclipse Mosquitto is an open source message broker implementing MQTT[8]. It uses Transmission Control Protocol (TCP) as its underlying transport protocol and is thus able

to leverage Transport Layer Security (TLS)[9]. To enable the use of TLS the servers will need their individual server certificates and the clients will need client certificates. The company has an in-house Certificate Authority (CA), which is used to issue these certificates. The certificates used are RSA encrypted SHA-256 X.509 certificates.

3.2.2 MQTT Servers

The two servers - Broker and Bridge - need to be configured in order to use TLS for authentication. They also need some sort of authorization, in order to make communications secure. In this section, the server configurations will be presented.

3.2.2.1 Broker

The configuration file of Broker looks as follows:

```
listener XXXX

require_certificate true
use_identity_as_username true

cafile /mosquitto/data/cacerts/ca.crt
certfile /mosquitto/data/certs/cert.crt
keyfile /mosquitto/data/certs/cert.key

plugin /usr/lib/mosquitto_dynamic_security.so
plugin_opt_config_file /mosquitto/data/dynsec/dynamic_security.json

#log_type error, warning, notice
log_type information
```

Figure 3.1: MQTT Server configuration file broker.conf

Mosquitto servers can define custom ports which they listen to. Any settings below the `listener` line apply only to that specific port, until the optional next `listener` is specified. Specifying a port together with the configuration options `require_certificate` and `use_identity_as_username` set to `true`, connecting Clients are required to authenticate using TLS certificates. The field Common Name (CN) in the Client's certificate Subject field is used as the connecting Client's username. In order to make authentication

possible, the server needs a CA certificate, server certificate, and its corresponding private key. These are mounted into the Docker Compose service into the paths specified in `cafile`, `certfile`, and `keyfile`[9].

In order to add authorization together with authentication, Broker uses Mosquitto's Dynamic Security plugin[10]. The plugin allows the creation of three main objects, `clients`, `groups`, and `roles`. In this context, `clients` are the objects created in the plugin, and the actual MQTT Clients connecting to the server are called `users`.

`Clients` have many attributes, but in this application the key attributes are `username`, `groups`, and `roles`. `Username` maps to the username provided in the CONNECT Control Packet when a `user` connects. This is a key feature, as previously mentioned in the Server configuration file, it is specified that the connecting `user`'s certificate CN is used as username. This way `users` can be mapped to specific Client certificates. A `client` can be member of any number of `groups` and be assigned any number of `roles`.

`Groups` can host multiple `clients` and have `roles` assigned to them. This allows for multiple different `clients` to share specific `roles` via one `group`.

`Roles` contain Access Control Lists (ACLs), and can be assigned to `clients` and/or `groups`. ACLs are constructed by listing the `acltype`, `topic`, `priority`, and `allow` values.

At the time of writing this thesis, the Dynamic Security plugin does not support variables in the ACLs, which forces the creation of multiple `roles`. The author of Mosquitto, Roger Light, promises this feature in an upcoming release[11]. This would greatly reduce the bloat of the `dynamic_security.json` file, as most `clients` could be lumped under the same `role`. Example `dynamic_security.json` files with and without bloat are presented in figures 3.2 and 3.3

3.2.2.2 Bridge

The configuration file for Bridge looks as follows:

```
listener YYYY

require_certificate true
use_identity_as_username true

connection bridge-to-broker
address ${BROKER_ADDRESS}:8883

topic +/+/sys/cmd in 0

topic +/+/sys/# out 0
topic +/+/res/# out 0

local_clientid incoming
remote_clientid outgoing

cafile /mosquitto/data/cacerts/ca.crt
certfile /mosquitto/data/certs/cert.crt
keyfile /mosquitto/data/certs/cert.key

bridge_cafile /mosquitto/data/cacerts/ca.crt
bridge_certfile /mosquitto/data/certs/bridge.crt
bridge_keyfile /mosquitto/data/certs/bridge.key

acl_file /mosquitto/config/acl.conf

#log_type error, warning, notice
log_type information
```

Figure 3.4: MQTT Server configuration file bridge.conf

As both Broker and Bridge share the same host VM, they need different ports to listen to. Although it is possible to use the same port in the configuration file for the Bridge service and have Docker Compose map the service port to a different host port, for clarity it is easier to just define another port in the configuration file. Enabling TLS is the same as for Broker, but using Bridge as an MQTT bridge requires additional certificates. These files can be seen in the configuration file under `bridge_cafile`, `bridge_certfile`, and

`bridge_keyfile`. The combination of both sets of certificates enables Bridge to both act as its own standalone MQTT server and as an MQTT client to Broker[9].

A connection to Broker is defined under `connection` and `address`. The `topic` keyword specifies which topics are to be bridged. The syntax is `topic topic pattern direction QoS`. The `direction` can have the following values;

- out = publish from the broker
- in = receive from remote broker
- both = publish and receive.

In the above context, broker refers to the broker being configured, while remote broker refers to its paired broker[12]. In this case Bridge is broker and Broker is remote broker. This configuration ensures that messages sent to Bridge on topics `+/+/sys/cmd` are not forwarded to Broker, rather it subscribes to this topic from Broker. Bridge will forward messages to Broker sent to it on topics `+/+/sys/#` and `+/+/res/#`.

Authorization is simpler for Bridge, as there are but two predefined types of connecting `users`. There are scripts, which use a specific TLS certificate, as well as Q Displays, which all have a certain type of certificate. The scripts are granted `readwrite` access to all topics, while the Q Displays are granted `readwrite` access to topic filter `+/%u/#`, meaning wildcard/username/wildcard. Other `users` are denied[9].

3.2.3 Topic structure

3.2.3.1 Group ID

Group IDs (GIDs) are the base of the topic structure. Each Q Display belongs to one or more GIDs. By creating GIDs for each of our customers, ourselves, and possible future needs, access to different sets of Q Displays via MQTT is easy. This Q Display - GID relationship is not trivial to manage and will be further explored in chapter 5.

3.2.3.2 Serial Number

Each Q Display has its own unique Serial Number (SN), which matches its TLS certificate CN. This means the Q Displays may share different topic roots with others, but branch out into their own topic leafs, e.g:

- `{GID 1}/{SN 1}/#`
- `{GID 1}/{SN 2}/#`

- {GID 1}/{SN 3}/#
- {GID 2}/{SN 1}/#
- {GID 2}/{SN 3}/#
- {GID 2}/{SN 4}/#

can be seen as:

- {GID 1}/ <- shared topic
  - {SN 1}/# <- private topic
  - {SN 2}/# <- private topic
  - {SN 3}/# <- private topic
- {GID 2}/ <- shared topic
  - {SN 1}/# <- private topic
  - {SN 3}/# <- private topic
  - {SN 4}/# <- private topic.

3.2.3.3 Value set

Value set is Q Display's proprietary method for inter-application communication. Values - such as RPM, GPS position, etc. - are addressed by specified combinations of SetId (SID), ValueId (VID), and InstanceId (IID). These are used extensively in the topic structure.

3.3 Usage

3.3.1 Q Display

Q Displays have a list of GIDs they belong to and instantiate a number n amount of Clients subscribing to each available combination of {GID}/{SN}/sys/cmd topic on startup. They publish a message containing a session ID in the form of a UUID to corresponding {GID}/{SN}/sys/valueset/{SID}/{VID}/{IID} topic. Then they wait for incoming commands from Clients. Messages received on topic {GID}/{SN}/sys/cmd are handled appropriately. All message payloads are JSON[13] formatted. If the command received is valid, a response indicating so will be sent to topic {GID}/{SN}/res/cmd/{cmd}. If the command cannot be resolved, a message indicating so will be published to topic {GID}/{SN}/res/cmd.

3.3.2 Clients

For this application, the interesting topics are:

- `{GID}/{SN}/sys/cmd`
- `{GID}/{SN}/res/cmd`
- `{GID}/{SN}/res/cmd/{cmd}`
- `{GID}/{SN}/sys/valueset/{SID}/{VID}/{IID}`

and their usages will be explored further. As can be seen in figures 3.2 and 3.3, Clients are restricted to topics starting with their own GID. This ensures access to only the Q Displays they are authorized to access.

3.3.2.1 Publishing

As Q Displays only subscribe to topic(s) `{GID}/{SN}/sys/cmd`, this is the topic Clients use to issue commands to Q Displays. The payload of messages published on this topic contain the commands the Client wishes the Q Display to execute. The commands `enable-value` and `set-valueset-update-interval` are the two commands in scope. As Q Display's value set concept is proprietary, the payloads for these commands will not be shared publicly.

Enabling a value set value for streaming is done with `enable-value`. Multiple ValueIds and InstanceIds can be enabled simultaneously, but disabling value set value streaming must be done using an entire SetId, ValueId, InstanceId combination.

The other command in scope, `set-valueset-update-interval`, is used to set the minimum interval for receiving value set value changes in milliseconds. Changes are reported if the time difference between the last and current report exceeds the interval. Interval value 0 will report changes immediately.

Publishing has to be done to one specific topic, so wildcard topics are not available. Commands can thus be sent to one individual Q Display per message.

3.3.2.2 Subscribing

By subscribing to the topic `{GID}/{SN}/sys/valueset/{SID}/{VID}/{IID}` corresponding to Q Display session ID, the Client will be notified when a Q Display matching the `{GID}/{SN}` comes online. This is useful for automation, e.g sending a predefined `enable-value` command to the Q Display. By then subscribing to either each individual `{GID}/{SN}/sys/valueset/{SID}/{VID}/{IID}` topic,

`{GID}/{SN}/sys/valueset/#`, or any other combination, the Client can receive the intended data. For information about issued commands, the Client can also subscribe to topics `{GID}/{SN}/res/#`. Receiving all communication the Q Display sends out can of course be received by subscribing to topic `{GID}/{SN}/#`.

Subscriptions support wildcards in the topic filter, meaning one Client can subscribe to their entire GID at once, via a subscription to topic `{GID}/#`. Other combinations are of course allowed, such as `{GID}/+/sys/valueset/#`, if all value set value data from all Q Displays in this GID is the intended target for subscription.

3.3.2.3 Overview

In figure 3.5, the MQTT architecture is displayed. Clients communicate with Broker, unknowing of Bridge's existence. Q Displays communicate with Bridge, unknowing of Broker's existence. As Broker has strict authorization, the multiple Q Displays - who are all equal Clients - are funneled into one Client (Bridge) from Broker's point of view.

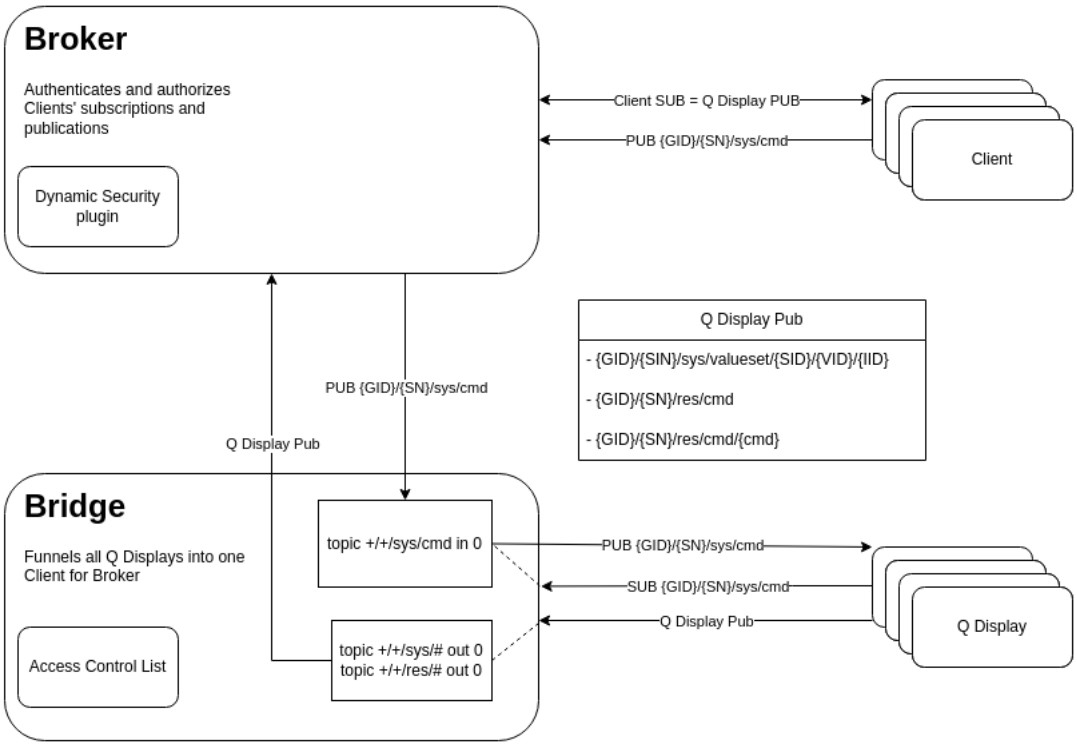

Figure 3.5: MQTT communications diagram

```
{
    "defaultACLAccess": {...},
    "clients":  [{
            "username": "master_client",
            "roles":    [{
                    "rolename": "master"
                }],
        }, {
            "username": "client_0001",
            "roles":    [{
                    "rolename": "role0001"
                }]
        }, {
            "username": "client_0002",
            "roles":    [{
                    "rolename": "role0002"
                }]
        }, ...
            ],
    "groups":   [],
    "roles":    [{
            "rolename": "master",
            "acls":     [{
                    "acltype":  "publishClientSend",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "publishClientReceive",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "subscribePattern",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "unsubscribePattern",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }]
        }, {
            "rolename": "role0001",
            "acls":     [{
                    "acltype":  "publishClientSend",
                    "topic":    "0001/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "publishClientReceive",
                    "topic":    "0001/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "subscribePattern",
                    "topic":    "0001/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "unsubscribePattern",
                    "topic":    "0001/#",
                    "priority": 0,
                    "allow":    true
                }]
        }, {
            "rolename": "role0002",
            "acls":     [{
                    "acltype":  "publishClientSend",
                    "topic":    "0002/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "publishClientReceive",
                    "topic":    "0002/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "subscribePattern",
                    "topic":    "0002/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "unsubscribePattern",
                    "topic":    "0002/#",
                    "priority": 0,
                    "allow":    true
                }]
        }, ...
        ]
}
```


Figure 3.2: Example Dynamic Security configuration file with bloat, due to no variable support in ACLs.

```
{
    "defaultACLAccess": {...},
    "clients":  [{
            "username": "master_client",
            "roles":    [{
                    "rolename": "master"
                }],
        }, {
            "username": "client_0001",
                "groups":   [{
                        "groupname": "clients"
                    }]
        }, {
            "username": "client_0002",
                "groups":   [{
                        "groupname": "clients"
                    }]
        }, ...
            ],
    "groups":   [{
                "groupname": "clients"
                "rolename": "clients_role"
            }],
    "roles":    [{
            "rolename": "master",
            "acls":     [{
                    "acltype":  "publishClientSend",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "publishClientReceive",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "subscribePattern",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "unsubscribePattern",
                    "topic":    "#",
                    "priority": 0,
                    "allow":    true
                }]
        }, {
            "rolename": "clients_role",
            "acls":     [{
                    "acltype":  "publishClientSend",
                    "topic":    "{client:username}/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "publishClientReceive",
                    "topic":    "{client:username}/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "subscribePattern",
                    "topic":    "{client:username}/#",
                    "priority": 0,
                    "allow":    true
                }, {
                    "acltype":  "unsubscribePattern",
                    "topic":    "{client:username}/#",
                    "priority": 0,
                    "allow":    true
                }]
        }]
}
```

Figure 3.3: Example Dynamic Security configuration file without bloat, due to variable support in ACLs.

Chapter 4

Databases and long-term storage

This chapter presents the architecture and implemented services of ThunderQloud, as well as what time-series databases and QuestDB are.

4.1 Time-series databases

Time-series databases are designed to efficiently store and process time-series data, i.e. data points associated with timestamps. Examples of typical time-series data include financial market data and sensor readings. Time-series data have key characteristics due to their temporal nature.

- The order of data is important.
- The volume of data is typically very large.
- Data flow is uninterrupted within a time window, continuous or cyclical.
- Over time, the relevance of each individual data point diminishes.
- Aggregation or down-sampling is leveraged for analysis over time intervals.

These characteristics lend themselves to identifying trends over time. This can be used to create models for forecasting or anomaly detection, for example.

Time-series databases focus on ingestion speed rather than transactional guarantees, offered in SQL databases. Data is usually written by appending, rather than updating records. Streaming protocols, such as InfluxDB line protocol (ILP)[14], further speed up ingestion. ILP is a text-based protocol that can compactly represent data points and ingest lines - data points - schema-lessly. Once ingested, data is indexed and partitioned by time, allowing for fast retrieval for time-based queries. Built-in interpolation, down-sampling, and aggregation functions also allow for fast queries. Older data can also efficiently be stored and archived using compression and retention techniques[15].

4.1.1 QuestDB

QuestDB is a time-series database with an engine built from the ground up to be both as efficient as possible and easy to use. Data can be inserted and queried via multiple methods, but for this application ILP is used for ingestion, while Postgres wire protocol and the inbuilt HTTP server are used for querying. More about the decision to use QuestDB in chapter 7.

4.2 Implementation

As mentioned in section 2.2.2, there are two QuestDB instances. A mounted persistent disk hosts the data for both QuestDB instances. Each instance uses a slightly modified configuration file. ILP is the preferred ingestion method and as such enabled by default in the configuration. Postgres wire protocol needs user configuration, and the in-built HTTP server is not set to read only mode by default. The symbol capacity is set to 256 by default. The full configuration options can be found at QuestDB's web page[16].

Symbols are a data type unique to QuestDB. It is an optimized way to store repetitive strings[17]. They are used in QDB as identifiers for Q Displays. The QuestDB configuration documentation states the symbol cache should be roughly equal to the number of unique symbols in the database. The capacity must be a power of two. Having many more symbols than the cache will negatively impact performance. As there are many more Q Displays than 256, this value needs to be modified. Figure 4.1 lists the modifications to the configuration files.

```
### Cairo engine ###
cairo.default.symbol.capacity = 8192

### REST API ###
#http.security.readonly = true
# defaults to false, but ENV VARs always override conf file

### Postgres wire protocol ###

pg.readonly.user.enabled = true
pg.readonly.user = XXXX
pg.readonly.password = XXXX

pg.user = XXXX
pg.password = XXXX
```

Figure 4.1: QuestDB server.conf

4.2.1 QDB

QDB is used for big data purposes. A large fraction all available data produced by Q Displays is stored here. The aim is to have the largest possible amount of data - within reason - to be able to create useful insights later on. There is one big database table, to which all Q Displays populate data. As seen in section 3.3, the Q Displays stream data according to the value set concept. This means the database table needs to reflect this. The table has six columns:

- `SN (symbol)` Q Display Serial Number
- `SID (int)` SetId
- `VID (int)` ValueId
- `IID (int)` InstanceId
- `val (double)` value
- `ts (timestamp)` timestamp.

The table is partitioned daily by the timestamp column. Value sets are enumerated, which gives easy access to specific value set values via SQL query. Complicated queries to this

table are cumbersome due to its structure. Fortunately, queries to this table are rare to non-existent. This is due to the facts that resources are somewhat limited, use cases are not yet clearly defined, and the data are only retained for a certain time. The data are exported to long-term storage, in order to save costs. Data ingestion, retention, and ejection will be presented in sections 4.2.5.2 and 4.2.5.4.

4.2.2 QDB2

QDB2 is used for statistical analysis on predefined metrics. The table structure is different from QDB, as each Q Display has its own table. All tables are similar in this database, and similar to the one table in QDB, except missing the `SN` column, as that information is carried in the table name instead. This allows for fast queries on data from a specific Q Display. As each table hosts less data by many orders of magnitude than the table in QDB, data retention is much more relaxed. At the time of writing, QuestDB's own partitioning system handles the volume of data without issue, meaning no custom data retention or ejection strategies are yet required. Data ingestion will be presented in section 4.2.5.3.

4.2.3 Long-term storage

The other persistent disk mounted on ThunderQloud is the long-term storage disk. Data from QDB is compressed and moved to this disk. The directory structure is much more user-friendly than the table in QDB would suggest. The structure is designed to allow for easy pin-point precision in data extraction for future analysis. Each Q Display belongs to one or more GIDs as previously established. There might not be a "primary" GID it belongs to, or it may change GIDs over time. As such, all Q Displays have their own directory structure under one shared directory. The shared directory is called `data`. Each Q Display has its own sub-directory `data/{SN}`. Furthermore, as section 4.2.5.4 will cover, data is added on a daily basis, so each `data/{SN}` directory has its own `data/{SN}/{DT}` directories. In these daily directories, each value set has its own zip file, `{SN}_{DT}_{SID}_{VID}.zip`. Note, that InstanceId is not reasonable to split into its own file, as InstanceId only separates different instances producing the same type of data.

In order to be able to group Q Displays together with their peers based on GID, symbolic links are used. This way, the symbolic link `org_data/{GID}/{SN}/{DT}/{SN}_{DT}_{SID}_{VID}.zip` points to the same file as corresponding hard link `data/{SN}/{DT}/{SN}_{DT}_{SID}_{VID}.zip`. If a Q Display changes GIDs, no hard links need to be tampered with. This also eliminates the need of duplicating data because of multiple GIDs for one Q Display. More on how this is

achieved in section 4.2.5.4.

4.2.4 Nginx

The use of an nginx proxy server is needed, as the in-built QuestDB web server only supports HTTP - not HTTPS - and the port number it listens to is fixed at a certain port number. Without an nginx proxy, the QuestDB Docker Compose services - QDB and QDB2 - would be required to expose their ports to the host VM, and be mapped to ports different from each other. Connections to these ports would be unauthenticated plain-text HTTP connections. For ease of use, as well as security reasons, access via subdomain names combined with SSL encryption, authentication, and authorization is preferred over unauthenticated, unauthorized, and unencrypted connections to specific port numbers. This way, the services don't need to expose ports to the VM, since Docker Compose creates its own network, where the services can communicate with each other by name. The nginx service is thus the only service out of these three that needs to expose a port.

The nginx service has two servers configured. Both listen for SSL connections on the same port, but under different server names. As such, different subdomain names route to different QuestDB instances. The `server` block for QDB in the nginx configuration file is presented in figure 4.2. Another `server` block for QDB2 is present, with its unique characteristics. Nginx authenticates and authorizes connecting clients based on provided certificates.

```
...
server {
    listen XXX ssl;
    server_name sub.domain.com;

    ssl_certificate /etc/nginx/certs/server.crt;
    ssl_certificate_key /etc/nginx/certs/server.key;

    ssl_client_certificate /etc/nginx/certs/ca.crt;
    ssl_verify_client on;

    set $client_cn $ssl_client_s_dn;
    if ($client_cn !~* "CN=XXXX") {
        return 403;
    }

    location / {
        proxy_pass http://QDB:XXXX/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
...
```

Figure 4.2: Snippet from nginx.conf

4.2.5 Python scripts

The following Python scripts running in their respective Docker Compose services handle their own specific tasks. All tasks are essential for ThunderQloud's operations.

4.2.5.1 Enabler

The enabler script enables database population for Q Displays. It uses the paho-mqtt library[18] to instantiate an MQTT client. This client subscribes to a database-specific Q Display session ID topic on HailstQrm. When a Q Display sends a message indicating it is online, a callback function parses the necessary information and runs an initialization routine function. This function generates a command that is sent back to the specific Q

Display. The command tells the Q Display to start populating the database. Pseudo-code for this script can be seen in figure 4.3.

The Q Display does not start sending multitudes of individual messages with one value and a timestamp in JSON format like it does when streaming data in real time. Rather it collects data into organized JSON arrays in larger chunks first, then encodes these chunks of JSON into CBOR[19] and only then sends the message to `db/{SN}`. CBOR is short for Concise Binary Object Representation and is based on the JSON data model. As JSON is a text-based and CBOR is a binary data format, the overall footprint in data usage is smaller this way. There are fewer messages sent, and the larger amount of data per message - not necessarily by size, but by amount of information - is not as compromising due to the concise packing achieved by CBOR.

```
imports

HOST= os.getenv("BROKER_ADDRESS")
PORT= 8883

CA_PATH =   some path
CERT_PATH = some path
KEY_PATH =  some path

LISTEN_GID = some GID

ENABLED_VALUES = [
    some values
]

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe(database specific session ID topic)

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    handle_payload(msg.topic, msg.payload)

def handle_payload(topic, payload):
    parse topic and payload

    if (suitable condition):
        device_initialization_routine(GID, SN)

def device_initialization_routine(gid, sn):

    for value in ENABLED_VALUES:
        payload = {
            command
        }
        client.publish(topic=Q Display specific topic, payload=json.dumps(payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.tls_set(CA_PATH, CERT_PATH, KEY_PATH)
client.connect(HOST, PORT, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
client.loop_forever()
```

Figure 4.3: Enabler script pseudo-code

4.2.5.2 QDB Populator

This script populates QDB's database with data from Q Displays. It uses the paho-mqtt, cbor2[20], and questdb[21] libraries, as well as some Python native ones.

The paho-mqtt library is again used to instantiate an MQTT client, which subscribes to the topic filter `db/+` on Bridge in HailstQrm. This script, as well as QDB2's populator script, are the only Clients other than Q Displays allowed to contact Bridge directly. The reason is to not unnecessarily burden the MQTT servers with additional traffic, as messages on this topic are strictly meant for database population. Thus other Clients never have any need to see them.

When a message is received, the callback function runs a function handling the payload. This function:

1. Validates Q Display serial number
2. Decodes the CBOR encoded payload
3. Connects to the database
4. Validates an entry in the decoded payload
5. Writes valid entry as a row and places it in buffer
6. Repeats steps 4 and 5 until there are no more entries
7. Flushes buffer.

Pseudo-code for this script can be seen in figure 4.4.

```
imports

HOST= os.getenv("BROKER_ADDRESS")
PORT= 8882

CA_PATH =   some path
CERT_PATH = some path
KEY_PATH =  some path

P_IDX_TS = 0
P_IDX_SID = 1
P_IDX_VID = 2
P_IDX_IID = 3
P_IDX_VAL = 4

def validate_sn(SN):
    some logic
    return bool

def validate_payload(payload):
    some logic
    return bool

def validate_timestamp(ts):
    some logic
    return bool

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe("db/+", qos=1)

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    handle_payload(msg.topic, msg.payload)

def handle_payload(topic, payload):
    t = topic.split("/")
    sn = t[1]

    if validate_sn(sn):
        data = cbor2.loads(payload)
        try:
            with Sender("QDB", 9009) as sender:
                for e in data:
                    uint_value =e[P_IDX_TS]
                    ts = int(uint_value)/1000
                    dt = datetime.fromtimestamp(ts)
                    nano = TimestampNanos.from_datetime(dt)

                    if validate_payload(e) and validate_timestamp(dt):
                        sender.row(
                            'tablename',
                            symbols={
                                'SN': sn
                            },
                            columns={
                                'SID' : e[P_IDX_SID],
                                'VID' : e[P_IDX_VID],
                                'IID' : e[P_IDX_IID],
                                'val' : e[P_IDX_VAL],
                            },
                            at=nano
                        )
                sender.flush()
        except IngressError as e:
            sys.stderr.write(f'Got error: {e}\n')

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.tls_set(CA_PATH, CERT_PATH, KEY_PATH)
client.connect(HOST, PORT, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
client.loop_forever()
```


Figure 4.4: QDB Populator script pseudo-code

4.2.5.3 QDB2 Populator

This script works similarly to QDB's populator script, but has some added functionality. It uses the same libraries and the MQTT client works the same way. There are only a handful of selected value sets allowed in this database, and the database schema differs slightly as well. As it would cost Q Displays additional network load to send only the selected values to a different topic for this MQTT client to receive, this client subscribes to the same topic as the client in QDB's populator script. The same callback for received messages is implemented here as well. The function handling the payload needs some additional parsing logic, however. Initially, the same instructions apply: extract the serial number from the topic and decode the payload. This is where the two scripts branch off. The control flow of the function can be summarized as below:

1. Validate Q Display serial number
2. Decode the CBOR encoded payload
3. Validate an entry in the decoded payload
4. Add valid entry to a list
5. Repeat steps 3 and 4 until no more entries
6. If the list is populated, connect to database and input list.

The validation process now has an additional validity check, `validate_update_interval()`. This function leverages the global variables `ALLOWED_COMBINATIONS` and `last_updated_dict`. These variables keep a record of which value sets for which Q Displays have been inserted into the database at which timestamps. The `validate_update_interval()` function only validates entries for which the desired time interval has passed since last insertion as true.

The database connection is established in the same manner, a `Sender` object is used to create `rows`, which are then flushed. The mechanism for populating these `rows` is slightly different, which can be seen in in figure 4.5. The figure displays the difference in implementation from QDB's Populator script .

```
INTERVAL_10 = timedelta(seconds=10)
INTERVAL_60 = timedelta(seconds=60)

ALLOWED_COMBINATIONS = {
    some value set combinations
    }

# Dictionary to store last updated timestamp for each key
last_updated_dict = {}

def validate_update_interval(e, sn, dt):
    some logic
    return bool


def handle_payload(topic, payload):
    t = topic.split("/")
    sn = t[1]

    if validate_sn(sn):
        data = cbor2.loads(payload)
        valid_data = []

        for e in data:
            uint_value =e[P_IDX_TS]
            ts = int(uint_value)/1000
            dt = datetime.fromtimestamp(ts)

            if validate_payload(e) and validate_timestamp(dt):
                if validate_update_interval(e, sn, dt):
                    nano = TimestampNanos.from_datetime(dt)
                    valid_data.append(
                        {
                            'table': '{}'.format(sn),
                            'columns': {
                                'SID' : e[P_IDX_SID],
                                'VID' : e[P_IDX_VID],
                                'IID' : e[P_IDX_IID],
                                'val' : e[P_IDX_VAL],
                            },
                            'at': nano
                        }
                    )

        if valid_data:  # if there is at least one valid payload item
            try:
                with Sender("QDB2", 9009) as sender:
                    for row in valid_data:
                        sender.row(row['table'], columns=row['columns'], at=row['at'])
                    sender.flush()
            except IngressError as e:
                sys.stderr.write(f'Got error: {e}\n')
```

Figure 4.5: QDB2 Populator script pseudo-code snippet

4.2.5.4 QDB Ejector

This script runs a defined job - `job()` - based on a schedule. This job has two parts. The first part is designed to capture all data received from the previous day, write it into individual csv files based on the structure `data/{SN}/{DT}/{SN}_{DT}_{SID}_{VID}.csv` specified in section 4.2.5, compress these csv files into zip files, remove the temporary csv files, and create symbolic links to the zip files based on Q Display GIDs. The second part drops partitions older than a specified number of days from the database. The second part will not run if the first part fails for any reason. This ensures data is not removed before it

is safely stored. The control flow of the script can be summarized as below.

1. Start `job()` at 01:00
2. `job()` runs `collect()`
3. `collect()` runs `extract()`
4. `extract()` runs `getDeviceGidMapping()`
5. `getDeviceGidMapping()` runs `getGidSerialMapping()`
6. `getGidSerialMapping()` runs `getGids()` and `serialsOfGid()`
7. `extract()` runs `getDeviceGids()`
8. `extract()` returns a Boolean `res` to `collect`, which returns it to`job()`
9. If `res = True`, carry on. If not, stop.
10. `job()` runs `drop()`
11. `drop()` returns a Boolean `res` to `job()`
12. If `res = True`, `job()` ran successfully. If not, log error.

If at any point something fails between 3 and 9, `job()` stops and logs the error. Data will not be lost, it will just be present in the database and can be inspected and recovered manually. If something goes wrong after 9, there is an issue with dropping old partitions - although it should always be just one partition to be dropped, as the job runs daily - which also means the partition stays in the database and can be inspected manually.

The `extract()` and `drop()` functions use the PostgreSQL database adapter library psycopg2 [22] to connect to the database. The combination of GID- and Q Display related functions are used in order to minimize the amount of necessary requests sent to Q-server REST API endpoints, while still acquiring relevant up-to-date mapping of relationships. Pseudo-code for this script shown in figure 4.6.

```
imports
# Schedules job() to be run at 01:00 every day
schedule.every().day.at("01:00").do(job)

def job():
    res = collect()
    if not res:
        return

    res = drop()
    if not res:
        return

# Collects data from yesterday
def collect():
    connect to dabase
        res = extract(curs)
    close connection

    return res

# Drops n day old partition
def drop():
    res = False
    try:
        connect to database, drop old partition, close connection
        res = True
    except: error handling logic

    return res

# Used to extract yesterday's data, zips it, moves to long term storage
def extract(cur):
    res = False
    try:
        device_gid_mapping = getDeviceGidMapping()
        cur.execute(select data from yesterday)
        SNs = cur.fetchall()
        for SN in SNs:
            GIDs = getDeviceGids(str(SN[0]), device_gid_mapping)
            ...
            for SID in SIDs:
                ...
                for VID in VIDs:
                    logic for generating zip files into long term storage out of queried data
        res = True
    except: error handling logic

    return res

# Helper function for getGidSerialMapping()
# Gets device serial numbers of a certain GID via a Q-server API endpoint
def serialsOfGid(GID):
    some logic
    return serials

# Helper function for getGidSerialMapping()
# Generate strings for API request
def getGids():
    some logic
    return GIDS

# Helper function for getDeviceGidMapping()
# Returns a dictionary mapping each GID to its corresponding serials
def getGidSerialMapping():
    some logic
    return gid_serial_mapping

# Helper function for getDeviceGids()
# Returns a dictionary mapping each device SN to its corresponding GIDs
def getDeviceGidMapping():
    some logic
    return device_gid_mapping

# Returns a list of GIDs for a specific Q Display
def getDeviceGids(SN, map):
    some logic
    return map[SN]
```


Figure 4.6: QDB Ejector script pseudo-code

4.3 Usage

4.3.1 QDB

The primary use case for QDB - gathering big data - is automated by its aforementioned scripts. While the database can be queried manually, there is little to no use by it. Data is only retained a for a certain time, so no meaningful analysis is likely to be done directly via QDB. Future company projects will be responsible for creating insights based on the data gathered by QDB. Customer access to the data is off limits directly via QDB, as data from all Q Displays is present in the same table.

4.3.2 QDB2

The use cases for QDB2 are also mostly outside of the scope of this project. Other projects will heavily leverage QDB2, though, as a fleet management project and the Q Display mobile application are gearing up for integrating with QDB2. The built-in QuestDB web server provides REST API endpoints[23] for external applications to use. Of course, the web server is accessed securely via nginx as per section 4.2.5. As this database houses all Q Displays as separate tables in plain sight, this database is also meant only for internal use. Customers will be granted access to their data via other projects in the company.

4.3.3 Long-term storage

All historical data from QDB is stored here. For now, access to these files is restricted to VM access. Thus, only company personnel with appropriate VM access can access the data. Files can be downloaded to local computers via `scp`[24]. Future development plans for customer access will be discussed in chapter **??**.

Chapter 5

Back-end server - Q-server

As the concept of GroupIds (GIDs) is completely new, and a requirement for the topic structure used by HailstQrm, additional implementation is needed on Q-server. Q-server has its own database, with existing schema. Adding GIDs will not tamper with existing implementation, rather only build new functionality on top of the old base. A new database table is created for GIDs, as well as a couple of linking tables between organizations and GIDs, and Q Displays and GIDs. Database migrations and a one-time migration script is run to create correct links to aforementioned linking tables. REST API endpoints with authentication and authorization return either list of GIDs a specific Q Display belongs to, or a list of Q Display serial numbers (SNs) belonging to a specific GID. The Q-server admin panel allows for managing GIDs, organizations, and Q Displays, as well as the links between them. Q Displays and other services - such as QDB Ejector script - utilize the endpoints for up-to-date information about Q Display - GID relationships.

Chapter 6

Data sampling

A fundamental part of gathering data into the databases is sampling. Choosing a sampling rate for QDB is not trivial, as the use case for gathered data is yet undefined. In future, some other project will analyze the data, but the intentions of what kind of information is to be extracted is not clear. While gathering every single change in value maintains complete integrity of data, this, however, accumulates to more load on the Q Displays, their network usage, and the database. Thus, some sort of sampling strategy is needed.

6.1 Enabler

The enabler script, which tells the Q Displays to start sending data to populate the databases, has instructions on sampling rates for each value set value. Most values are set to be sampled at 1Hz, while some, less time-sensitive values, are set to be sampled at 0.2Hz. This creates a satisfactory compromise between data integrity, costs, and performance. This can easily be reconfigured in future, if the need arises.

6.2 Q Display

Some additional sampling logic is also implemented in the Q Displays. Given a sampling rate of 1Hz for a value, the Q Display will perform in the following way. Sample value at `t`, wait until `t+1000ms`, evaluate if value has changed since last sample, if changed, sample value, else, continue until `t+2000ms`, and so forth. This way, unchanged values are not reported multiple times redundantly.

6.3 QDB2 Populator

As stated in section 4.2.5.3, QDB2 is populated with a custom sampling rate for a subset of metrics. The sampling is done in this script on the raw data published by Q Displays, eliminating the need for multiple streams of similar data.

Chapter 7

Methodology

7.1 System design methodology

As this project was a one person job, no strict software development model was applied. The development process did, however, have similarities with the Agile model. Instead of working with a team in defined sprints and discussing progress in daily stand-ups, research, development, and progress was continually reported to the entire Q Display development team in daily stand-ups.

Even though not specified in a project plan in detail, the order of developing services was organically clear from the beginning. In order for Q Display data to be stored somewhere, there was a need for a database. In order to populate the database to be, there was a need for a way to deliver the data to it. Naturally, the project then started with developing the message brokering services. This could be considered one rather long sprint. Then, when data could be transferred, it needed to be stored. That initiated another so-called sprint, developing the database and its related services. Finally, with the basics working, the GID structure needed implementing in Q-server, which could be considered yet another sprint.

As opposed to a team iterating over built software from previous sprints due to newly emerged requirements or dependencies in current sprints, this iterative compatibility development was handled concurrently. Since the entire team working on a sprint was just one person, the need for defined sprints to keep the team coordinated was rather moot.

7.2 Requirement analysis

Once more leaning towards an Agile model, the requirements for the complete system were not laid out before starting the project. In the beginning, there were only a handful of requirements. The system should:

- enable real-time streaming of data from Q Displays
- gather big data from Q Displays
- be flexible to allow for creation of arbitrary groups and grant group-specific API access
- be secure.

Over the course of designing and implementing the system, new requirements were elicited via discussions with staff, technical restrictions, and logical conclusions. The existence of the QDB2 service is an example of this, as it was not initially planned, rather the need for it arose due to requirements from another company project.

7.2.1 Decision to use MQTT for message brokering

The first of the ad hoc sprints called for a technology to relay information between Q Displays and clients. The Q Display already implemented MQTT for communication between other devices in the same local network, as well as an API for other services. From a Q Display developer standpoint, there would be minimal investment in researching and implementing slightly different functionality, as opposed to an entirely new protocol.

Low overhead played a key factor in the decision-making process, as the Q Displays may spend significant amounts of time in areas with low network reception. Coverage charts provided by Telia below illustrate this (Figures 7.1, 7.2, 7.3, 7.4).

As Laaroussi et al. note, Constrained Application Protocol (CoAP) performs better than MQTT in terms of latency and throughput[25]. When researching how to secure communications, since both MQTT and CoAP are of course not inherently secure, a publication by El Aidi et al. pointed out that CoAP uses a 4-byte header as opposed to the 2-byte header of MQTT[26]. Since real time streaming of data was one of the requirements for this application, a large amount of messages containing only one value and corresponding timestamp may be sent. In such a setting, duplicating the overhead comes with its implications. Performance in throughput and latency pale in comparison to network overhead in this marine environment. This, combined with the aforementioned already existing MQTT implementation, made the decision to use MQTT easy.

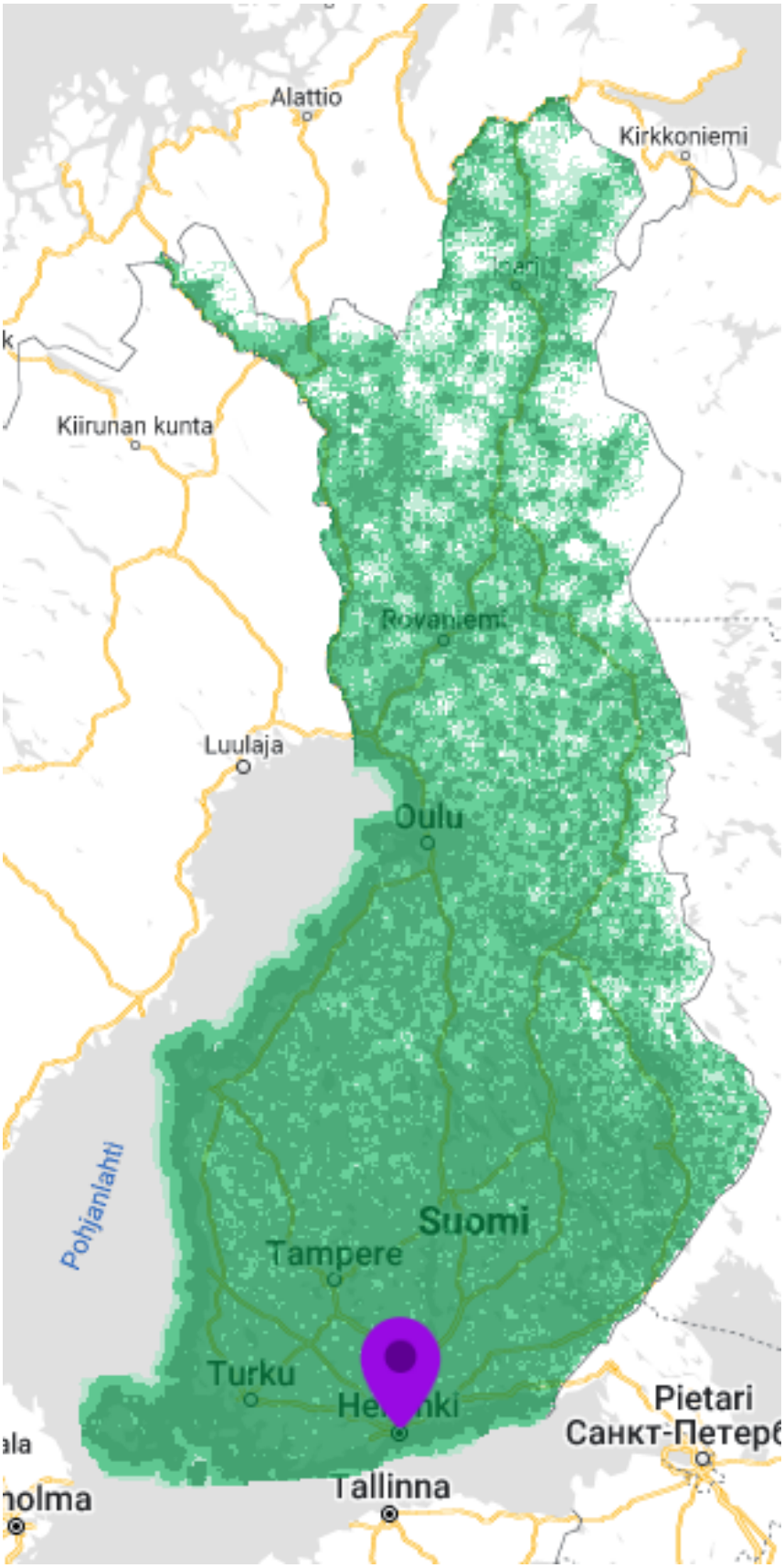

Figure 7.1: Telia IoT network coverage FIN

[27]

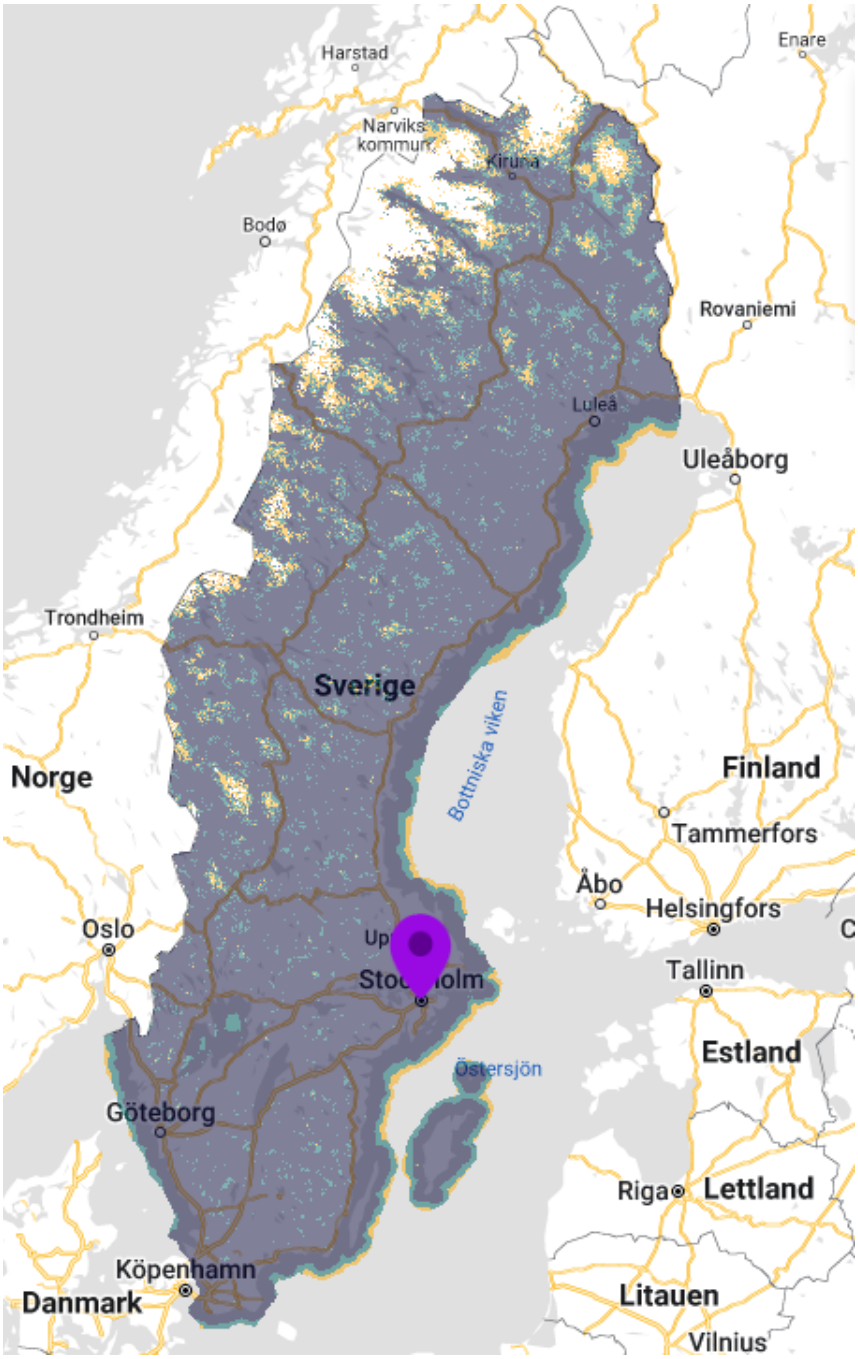

Figure 7.2: Telia IoT network coverage SWE
[28]

Figure 7.3: Telia IoT network coverage NOR
[29]

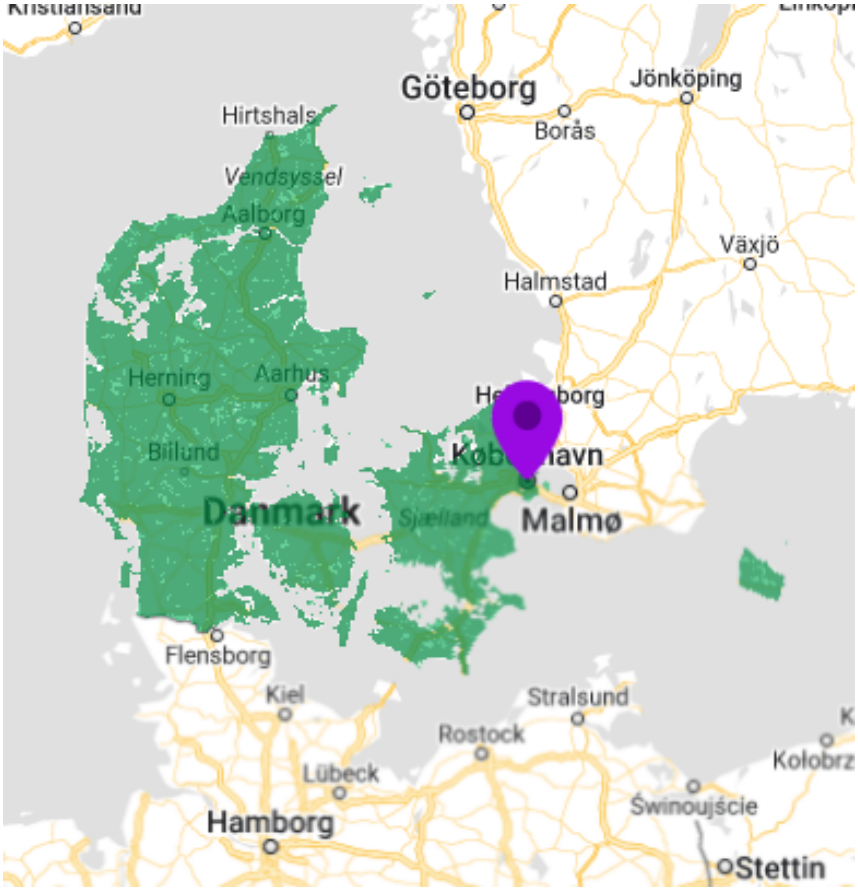

Figure 7.4: Telia IoT network coverage DK
[30]

7.2.2 Decision to use QuestDB

The purpose of ThunderQloud is to store data produced by Q Displays. As seen in chapter 3, the data they produce are in the form of individual value set values, accompanied by a timestamp. A parallel to sensor readings can easily be drawn, which heavily supports the notion to use a time-series database. Add to that the aim to produce information out of the data, built-in support for sampling and aggregation of time-series data in the database also became key factors.

For this application the final schema of the database was still uncertain during the decision making process of technology stack. Deciding which database management system to use influenced possible schema structures, so flexibility was a factor to consider. DB-Engines rank database management systems monthly by popularity. In this ranking, the top two contenders were InfluxDB and TimescaleDB. InfluxDB was at the top of the rankings, TimescaleDB somewhere in the top five.[31]. While researching which of these two would better suit the application at hand, an article by Yitaek Hwang[32] mentioned another contender, QuestDB. In his article, he presents each DBMS and concludes each

section with a pros and cons list.

In short, InfluxDB has a huge community, schema-less ingestion, and support for popular tools, but suffer in performance with high cardinality datasets, have multiple different conflicting open source versions, and a custom query language. As high cardinality was likely to be present due to the Q Display value set structure, InfluxDB did not look like an optimal solution.

TimescaleDB is an extension of PostgreSQL and as such has PostgreSQL-compatibility, scales better than InfluxDB with data cardinality, and has various available deployment models. However, TimescaleDB also enforces schema configuration limited by PostgreSQL, needs extra storage for continuous data aggregation, does not support any streaming ingestion protocols, and is partly under a proprietary license. As there is no flexibility in the schema configuration and no support for input via streaming protocol, TimescaleDB might not have been the right fit for this application, either.

QuestDB offers flexibility in the form of supporting various input protocols. The database engine is built from the ground up, focusing on performance, compatibility, and querying. The performance aspect covers high throughput ingestion and addresses InfluxDB's issue with high cardinality by using SIMD instructions and a just-in-time (JIT) compiler for query execution. The focus on compatibility refers to the fact that QuestDB supports ILP, Postgres wire protocol, as well as a REST API for both ingestion and querying. Client libraries for ILP are also available. Querying is made easy, as QuestDB uses SQL as its query language.

The pros for QuestDB include fast ingestion, high performance with low resources, support for multiple protocols, standard SQL queries, SIMD optimization, and open source. The cons include a smaller community and fewer available integrations than InfluxDB.

Based on these findings - QuestDB supporting schema-less ingestion and many ingestion protocols, high performance queries with a familiar query language, and most flexibility of all - the decision to use QuestDB was made.

7.3 Architecture

As presented in chapter 2, there were only Q Displays and the Q-server in the Q ecosystem. The development methodology supported the plan to divide the new platform into separate services, one for message brokering, one for databases, and to keep the bookkeeping of Q Display - GID relationships separate in the Q-server. It would have been possible to build everything as microservices alongside Q-server, but with the limited knowledge of requirements in the beginning of the project, such an architecture would

have been cumbersome to design. This approach allowed for isolating the services and developing, failing, and iterating fast, without causing outages in established services.

7.4 Testing and validation

7.4.1 HailstQrm

Using scripts to simulate up to tens of thousands of MQTT clients, and tailored test sets for testing authentication and authorization, the design for HailstQrm was validated. Test sets are executed upon changes in the system, in order to keep the system secure.

7.4.2 ThunderQloud

Running a local QuestDB instance and compressing ejected data allowed for calculations and projections validating the use of ThunderQloud's current setup, as well as the required attached persistent disks' sizes. The populator and ejector scripts are manually observed and tested in the staging environment before deployment to production.

7.4.3 Q-server

Q-server uses unit tests in order to validate the current build before deploying it in its pipeline. Unit tests covering all GID related implementation were written.

7.5 Deployment and maintenance

Currently, both HailstQrm and ThunderQloud are deployed quite manually. There are four VM instances in the cloud project; one production environment for both services, as well as one staging environment for both services. By pushing source code changes to the remote repository, and then manually restarting a VM instance, changes are applied. The staging VM instances pull source code from the remote repository's staging branch, while the production instances pull from master branch.

Computing resources for both VM instances are constantly monitored by the cloud provider's monitoring agent, and can be adjusted on the fly. If anything goes wrong with the services, they produce logs via Docker to the host VM, which can be inspected and dealt with accordingly.

7.6 Limitations and challenges

A clear project structure and a thorough requirement elicitation process in the beginning of the project would have streamlined the entire project. This was not the case, however, as the message brokering part needed to be designed and implemented quickly due to outside factors. This set the precedent for the entire project, where clear processes were outweighed by working results, and developing them quickly.

Chapter 8

Results

8.1 Quantitative data

ThunderQloud launched without QDB2 in the end of spring. Over the course of summer, nearly 4500 Q Displays were systematically populating QDB with their data. Now that the boating season is over in northern Europe, where most of the Q Displays are geographically, activity is very low. As older data is exported from QDB to LTS, the current volume of QDB is at an all-time low, only 6 Gigabytes worth of data from ca 200 Q Displays. LTS has accumulated over 80 Gigabytes worth of compressed data. With a compression rate of 14 to 1 discovered in early testing, which is most likely extremely consistent due to all database entries being in the same format, LTS has an estimated volume of 1.2 Terabytes of raw data.

QDB2 launched in fall, missing high season. Ca 500 Q Displays have populated 1.5 Gigabytes of raw data.

8.2 System performance and reliability

ThunderQloud ran through summer on a 6-core E2 instance, with 6GB RAM. During this time CPU usage averaged at ca 20% and memory usage averaged at ca 30%. QDB Ejector's nightly job demanded higher resources, though, as computing resource usage spiked considerably. During high season in late June and early July, nightly spikes could reach percentages ranging from 60-70% for CPU usage and 70-80% for memory usage. As the season started to slow down in fall, resources were scaled back to 4 cores and 4GB of RAM. Resource usage has lowered even still, with averages of 10-15% and 15-30% for CPU and memory, respectively.

Now that QDB is relatively empty, query speeds are rapid. Simple queries generally execute in less than a second, while more complex ones can take up to 5 seconds. During

high season, when there were 1.5-2 billion entries at any given time, complex queries could execute as slowly as 30 seconds. Computational resources were more abundant during summer, though. QDB2 is much faster due to its data being split into different tables. Even complex queries execute in less than a second, while simple queries are so fast that internet speeds are the limiting factor. Generally queries execute in 10-20 milliseconds.

HailstQrm needs very little resources in comparison to ThunderQloud, as it only runs two mosquitto MQTT servers. HailstQrm runs on a pre-set E2-small instance with 2GB RAM. During summer, both CPU and memory usage averaged at ca 40%.

HailstQrm has not had any issues with reliability in its services. ThunderQloud has had one critical fault, where QDB suffered from a known and supposedly fixed bug[33]. The bug did not allow for new transactions to be processed and required deleting a file keeping track of transactions and restarting QuestDB[34]. Fortunately this happened to QDB in late fall during low activity, and the workaround was applied quickly.

8.3 System usability and functionality

The initial requirements laid out in chapter 7, although few, are met. Groups with associated GIDs can be created by staff via the admin interface on Q-server, Q Devices can be added to these groups, and so clients are able to use the API via HailstQrm. The enabler, ejector, and populator services in ThunderQloud ensure big data gathering from the Q Displays is automated and works as required.

The only major drawback in the system is the accessibility of the gathered big data in ThunderQloud's file system. While the requirement of gathering big data may be satisfied, working with old historical data directly could prove cumbersome. The data is neatly organized, indexed, and zipped in the file system, but analyzing it requires transferring certain desired files from the VM, unzipping them, and only then can processing begin with whatever tool is chosen. There will probably be a need for developing yet another service just to retrieve and unpack desired data to prepare for analysis. Alternatively, another database and/or storage solution may be used in future, to which the already existing data would then be imported. Such drastic measures have not yet been set in motion, though.

8.4 Qualitative observations

HailstQrm and ThunderQloud are, as stated in section 7.5, deployed manually. As the staging and production environments for the services have been established and are oper-

ational, they have not yet been officially launched as marketable products, so this setup works for now. Future automated deployment pipelines are needed, however. As use of these services will grow, scaling might also become an issue.

Chapter 9

Conclusion

This thesis covers the implementation of a platform enabling the gathering of big data in a consumer marine environment, as well as streaming selected data in real time. Clients are able to leverage the MQTT protocol in order to communicate with Q Displays out in the seas and lakes, where they are used. Direct access to Q Displays can be granted based on their associated GroupIds, while summarized reports on insights gained from analyses may be generated in future. Other company services will allow for pre-defined statistical analysis generated from data gathered in this project.

To the best of the author's knowledge, this is where IoT in marine environments on a consumer scale has its beginning. While IoT has been leveraged on an industrial scale for large ships[35], this project specifically targets consumer leisure boats. As a result, a blueprint is now available for competitors to implement their own versions of such a platform.

As publications describing similar platforms to this one were not available during the research and implementation phases of this project, every decision taken was a novel one. This might not always lead to the optimal choice, which is brought up in chapter 8, with e.g. future plans of restructuring deployment strategies.

Future studies on what insights can be gained from consumer boating big data would shed light on what to do with the gathered data.

This project firmly takes a foothold in the domain of IoT for consumer boating. As the use of IoT in other industries, such as the automotive industry, are ahead by a large margin, this project might be the first step in the direction of closing the gap.

Chapter 10

Svensk sammanfattning

10.1 Introduktion

Värdet av data stiger konstant i dagens samhälle, i synnerhet värdet av stordata. Flera marknader har anpassat sig till denna trend, till exempel blir bilindustrin år för år mer digitaliserad. I marinindustrin har det tagit längre för digitaliseringen att nå båtmarknaden för fritidsänamål. Nextfours Q Display är en integrerad kartplotter för konsumentbåtar. Q Display är kopplad till båtens motor, batteri, sensorer, VHF radio, ekolod, radar och eventuella övriga tillbehör. Den är också uppkopplad till internet, bland annat för mjukvaruuppdateringar och väderleksrapporter. De flesta av dessa data är outnyttjade, det vill säga att de inte lagras någonstans. I denna avhandling behandlas projektet att bygga en plattform som möjliggör insamling av stordata och realtidsströmning av data från Q Displayer.

10.2 Implementation

Q Displayerna ute på fältet har alltid varit i kontakt med en backend server, Q-server, men i och med detta projekt utvidgas Q-ekosystemet med två nya tjänster: HailstQrm och ThunderQloud. HailstQrm är en virtuell värddator i molnet som kör två mosquitto [8] MQTT [7] servrar. ThunderQloud är en annan virtuell värddator i molnet, men den kör fler tjänster. ThunderQloud kör två stycken QuestDB-databaser [15], en nginx-server [5], såväl som fyra stycken Pythonskript.

Q-serverns funktionalitet utvidgas med ett nytt gruppkoncept, till vilka man kan lägga till Q Displayer. Grupperna har ett Id, ett så kallat GID. Detta GID-koncept är kritiskt viktigt för att kunna separera Q Displayer baserat på olika kriterier. Utöver det implementeras två nya REST-slutnoder, en som besvarar vilka Q Displayer som hör till ett visst GID, samt en som besvarar vilka GID en Q Display tillhör.

HailstQrm möjliggör kommunikation mellan Q Displayer och MQTT-klienter via MQTT-protokollet över internet. Utöver det sköter HailstQrm autentisering och behörighet över TLS med hjälp av mosquittos inbyggda hjälpapplikation. Behörigheten verifieras med hjälp av klientens TLS-certifikat, som bör motsvara MQTT-ämnet klienten försöker publicera eller prenumerera på. Eftersom nätverkstrafiken sker över TLS är den krypterad, till skillnad från MQTT utan TLS. MQTT-ämena i användning är strukturerade enligt `{GID}/{Q Display serienummer}/`

- `sys/cmd/{kommando}`
- `sys/valueset/{SID}/{VID}/{IID}`
- `res/cmd`
- `res/cmd/{cmd}`.

Exempelvis kunde en kund ha 100 stycken Q Displayer, vilka alla befinner sig i deras GID 0015. Då kunde kunden skicka ett kommando åt alla sina Q Displayer via en publikation till `0015/+/sys/cmd/enable-value`. Ifall de vill ha svar på vilka Q Displayer som tagit emot kommandot, kan de prenumerera på
`0015/+/res/cmd/enable-value`. Beroende på vilka värden de aktiverade strömningen av, vore det också ändamålsenligt att prenumerera på
`0015/+/sys/valueset/{SID}/{VID}/{IID}`, eller `0015/+/sys/valueset/#` för att motta strömningen av data.

Ovanstående är kortfattat vad Pythonskripten i ThunderQloud gör. Ett skript prenumererar på ett MQTT-ämne dit Q Displayer skickar ett unikt meddelande som indikerar att enheten kopplats online. Då skriptet får ett meddelande om en Q Display som kommit online, svarar det med en serie av kommandon, som sätter Q Displayen i stånd att samla data åt databaserna. Q Displayen börjar då samla, packa och skicka data till ett specifikt MQTT-ämne ämnat för databaserna. Båda QuestDB-databasinstanserna, QDB och QDB2, har egna Pythonskript som för in dessa data. De prenumererar på detta databas specifika MQTT-ämne, hanterar meddelandena och överför utvunna data till respektive databas. Det sista skriptet har som uppgift att exportera äldre data från QDB-databasen till värddatorns filsystem i komprimerad form. Dessutom frågar skriptet Q-servern vilka GID Q Displayer hör till, så det kan organisera filsystemet nätt.

Databaserna QDB och QDB2 är byggda med QuestDB. QuestDB är en tidsseriedatabas, vilket innebär att varje datapost är tidsstämplad och tabellerna sorteras automatiskt i rätt ordning. QDB har alla Q Displayer i samma tabell, med kolumner för serienummer, SID, VID, IID, värde och tidssämpel. QDB2 delar istället upp tabellerna per serienummer och lämnar bort den kolumnen. Dessutom matas endast vissa specifika data in i QDB2.

QDB fungerar som databas för stordata, medan QDB2 fungerar som databas för specifik statistik.

10.3 Metodologi

Detta projekt utfördes av en person, med en väldigt hektisk start. HailstQrm hade brottom att bli implementerad, på grund av orsaker indirekt relaterat till projektet. Det gjorde att hela projektet startade utan en klar projekt plan. Metoden för systemutveckling blev därför aldrig väl definierad, men processen följde nogårlunda en Agile metod. Någon utförlig kravhanteringsprocess utfördes ej, men det fanns fyra klara krav redan i början. Systemet ska:

- möjliggöra realtidsströmning av data från Q Displayer
- samla stordata från Q Displayer
- möjliggöra skapandet av grupper och bevilja grupp-baserade behörigheter
- vara säkert.

Nya krav uppenbarade sig under utvecklingsprocessen, via diskussioner med kollegor, tekniska begränsningar, såväl som logiska slutsatser. Till exempel hörde QDB2-databasen ursprungligen inte till planerna, men den blev ett krav på grund av krav inom andra projekt inom bolaget.

HailstQrm stresstestades med simulationer av tiostusentals klienter som skickade meddelanden. Dessutom körs en mängd test som verifierar säkerheten i kommunikationerna då systemet uppdateras. ThunderQloud testades lokalt för att bekräfta designen. Skripten observeras och testas manuellt i testomgivning före en uppdatering sker i produktionsomgivning. Q-server utnyttjar enhetstest och de nya funktionaliteterna är väl testade varje gång koden ändras.

10.4 Resultat

I sin helhet har hela systemet lyckats uppfylla kraven som ställdes i början. Q Displayer kan via kommandon genom HailstQrm beordras att strömma data både i realtid och samla data som Pythonskript samlar in i databaser och lagras i ThunderQloud. All nätverkstrafik är krypterad och kommunkation med alla tjänster kräver korrekta TLS certifikat.

Dock är tillgängligheten till äldre historiska stordata något dålig. Eftersom det är frågan om så stora mängder, är det inte lönsamt att hålla allting i QDB, så äldre data ex-

porteras och komprimeras till ThunderQlouds filsystem. Det gör att man måste exportera de data man vill bearbeta från filsystemet, istället för att bara fråga databasen direkt.

10.5 Slutsats

Avhandlingen behandlar designen, implementattionen, metodologin, samt resultaten av ett projekt, där målet var implementera ett system som tillåter insamling av stordata och strömning av realtidsdata från Q Displayer. Även om alla beslut fattade under projektets lopp inte var de mest optimala, kan slutresultatet ses som nöjaktigt, eftersom kraven på systemet har blivit uppfyllda. I och med detta projekt har ett stort steg tagits för att minska klyftan mellan IoT i konsdumentbåtande och IoT i andra ledande industrier.

References

[1] M. Gerla, E. Lee, G. Pau, et al. “Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds”. In: *2014 IEEE World Forum on Internet of Things (WF-IoT)* (2014). DOI: 10.1109/WF-IoT.2014.6803166.

[2] Docker. *Overview of Docker Compose | Docker documentation*. URL: https://docs.docker.com/compose/. (accessed: 19.8.2023).

[3] Eclipse. *eclipse-mosquitto - Official Image | Docker Hub*. URL: https://hub.docker.com/_/eclipse-mosquitto. (accessed: 19.8.2023).

[4] QuestDB. *questdb/questdb - Docker Image | Docker Hub*. URL: https://hub.docker.com/r/questdb/questdb. (accessed: 19.8.2023).

[5] nginx. *nginx - Official Image | Docker Hub*. URL: https://hub.docker.com/_/nginx. (accessed: 19.8.2023).

[6] Python. *python - Official Image | Docker Hub*. URL: https://hub.docker.com/_/python. (accessed: 19.8.2023).

[7] OASIS. *MQTT Version 3.1.1*. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. (accessed: 19.8.2023).

[8] Eclipse. *Eclipse Mosquitto*. URL: https://mosquitto.org. (accessed: 21.8.2023).

[9] R. Light. *mosquitto.conf man page | Eclipse Mosquitto*. URL: https://mosquitto.org/man/mosquitto-conf-5.html. (accessed: 21.8.2023).

[10] Eclipse. *Dynamic Security Plugin | Eclipse Mosquitto*. URL: https://mosquitto.org/documentation/dynamic-security/. (accessed: 21.8.2023).

[11] ralight. *Feature: Variables in ACL topic of the Dynamic Security Plugin #2784*. URL: https://github.com/eclipse/mosquitto/issues/2784. (accessed: 21.8.2023).

[12] S. Cope. *Mosquitto MQTT Bridge -Usage and Configuration*. URL: http://www.steves-internet-guide.com/mosquitto-bridge-configuration/. (accessed: 22.8.2023).

[13] json.org. *JSON*. URL: https://www.json.org/json-en.html. (accessed: 24.8.2023).

[14] InfluxData. *Line protocol | InflixDB OSS 2.7 Documentation*. URL: https://docs.influxdata.com/influxdb/v2.7/reference/syntax/line-protocol/. (accessed: 23.8.2023).

[15] QuestDB. *What is a Time-Series Database? | QuestDB*. URL: https://questdb.io/glossary/time-series-database/. (accessed: 23.8.2023).

[16] QuestDB. *Configuration | QuestDB*. URL: https://questdb.io/docs/reference/configuration/. (accessed: 23.8.2023).

[17] QuestDB. *Symbol | QuestDB*. URL: https://questdb.io/docs/concept/symbol/. (accessed: 23.8.2023).

[18] R. Light. *paho-mqtt · PyPI*. URL: https://pypi.org/project/paho-mqtt/. (accessed: 24.8.2023).

[19] C. Bormann. *CBOR — Concise Binary Object Representation | Overview*. URL: https://cbor.io/. (accessed: 24.8.2023).

[20] A. Grönholm. *cbor2 · PyPI*. URL: https://pypi.org/project/cbor2/. (accessed: 24.8.2023).

[21] A. Cimarosti. *questdb · PyPI*. URL: https://pypi.org/project/questdb/. (accessed: 24.8.2023).

[22] D. Di Gregorio. *psycopg2 · PyPI*. URL: https://pypi.org/project/psycopg2/. (accessed: 25.8.2023).

[23] QuestDB. *REST API | QuestDB*. URL: https://questdb.io/docs/reference/api/rest/. (accessed: 25.8.2023).

[24] T. Rinne and T. Ylönen. *scp(1): secure copy - Linux man page*. URL: https://linux.die.net/man/1/scp. (accessed: 25.8.2023).

[25] Z. Laaroussi, R. Morabito, and T. Taleb. "Service Provisioning in Vehicular Networks Through Edge and Cloud: An Empirical Analysis". In: *2018 IEEE Conference on Standards for Communications and Networking (CSCN)* (2018). DOI: 10.1109/CSCN.2018.8581855.

[26] S. El Aidi, A. Bajit, A. Barodi, et al. "An Optimized Security Vehicular Internet of Things -IoT-Application Layer Protocols MQTT and COAP Based on Cryptographic Elliptic-Curve". In: *2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)* (2020). DOI: 10.1109/ICECOCS50124.2020.9314579.

[27] Telia. *Mobiiliverkon kuuluvuus, häiriötilanteet ja parannustyöt*. URL: https://www.telia.fi/asiakastuki/kuuluvuuskartta. (accessed: 20.8.2023).

[28] Telia. *Täckningskartor*. URL: https://www.telia.se/privat/support/tackningskartor. (accessed: 20.8.2023).

[29] Telia. *Dekningskart*. URL: https://www.telia.no/nett/dekning/. (accessed: 20.8.2023).

[30] Telia. *IoT dækningskort*. URL: https://www.telia.dk/kundeservice/dakning-og-drift/iot-dakningskort/. (accessed: 20.8.2023).

[31] DB-Engines. *DB-Engines Ranking - popularity ranking of time Series DBMS*. URL: https://db-engines.com/en/ranking/time+series+dbms. (accessed: 23.8.2023).

[32] Yitaek Hwang. *Comparing InfluxDB, TimescaleDB, and QuestDB Time-Series Databases*. URL: https://questdb.io/blog/comparing-influxdb-timescaledb-questdb-time-series-databases/. (accessed: 23.8.2023).

[33] yjclsx. *An exception occurred while querying data*. URL: https://github.com/questdb/questdb/issues/1490. (accessed: 12.12.2023).

[34] Jon. *What does 'max txn-txn-inflight limit reached' in QuestDb, and how to I avoid it?* URL: https://stackoverflow.com/questions/67785629/what-does-max-txn-txn-inflight-limit-reached-in-questdb-and-how-to-i-avoid-it. (accessed: 12.12.2023).

[35] E. Wings, S. Reck, H. Boomgaarden, et al. "Implementing a Low-Cost Control Unit Network focusing on Data Collection and Flettner Rotor Control". In: *2022 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)* (2022). DOI: 10.1109/IAICT55358.2022.9887390.