Temperature Tracking in Cold Storage

Group 8

Frederik V. Helth
University of Southern Denmark
Odense, Denmark
frhel18@student.sdu.dk

Jonas Lund
University of Southern Denmark
Odense, Denmark
jolun18@student.sdu.dk

Mads E. Falkenstrøm
University of Southern Denmark
Odense, Denmark
mafal17@student.sdu.dk

Mathias B. Kristiansen
University of Southern Denmark
Odense, Denmark
matkr18@student.sdu.dk

Patrick Nielsen
University of Southern Denmark
Odense, Denmark
panie18@student.sdu.dk

1 GROUP REPORT

Extended Summary

• Verification and Validation(V&V) was conducted in UPPAAL to verify the system is working as intended by using formal modelling. The system uses multiple automatas to represent different states and related edges from which the states are connected.

• Derived from Model-Driven Software Development (MDSD) a Domain Specific Language (DSL) was developed to build a rule engine, which is a critical component of the system since it enables one of the most significant features: determining whether a temperature is too low or too high. The Meta Model is defined by the grammar, and generated by Xtext. The code is then generated, from the Meta Model instance, through the generator that will generate our rules engine to check temperatures. An validation for the grammar is made to make sure that the rule have a min and max, to specific the desired range you want the goods to be stored.

• The IoT hardware makes use of an Edge device and a Fog device, using internal communication through BLE to transmit data from a temperature sensor with a dynamic sampling regime. The Fog device then transmits the data through message queue telemetry transport (MQTT) to a cloud.

1.1 Problem and Objective

When food is delivered, specific temperature criteria must be met. As a result, it is critical that some mechanism exists to ensure that actions can be taken if food is delivered at temperatures that do not meet the requirements. Failure to meet the requirements will result in unhealthy food being thrown away or even sold as unhealthy food.

How can a system be developed, that guarantees that when food is transported, temperature measurements are taken and if the temperature falls below the standards, you are notified and can take actions?

1.2 Problem Description

According to the UN, food waste accounts for 8 to 10% of the overall worlds CO2 emissions [3]. Unused foods are responsible for 3.3 billion tonnes of CO2 emissions each year. The 814,000 tonnes of edible food that is wasted annually in Denmark corresponds to 3.8% of the total CO2 emissions emitted from Denmark each year, the Ministry of the Environment and Food estimates [5]. Less food waste has a direct environmental effect, as it reduces the need for resources and nature seizure.

1.3 Solution Approach

As a requirement for the project, the group had to incorporate the different courses throughout the semester. The group brainstormed different ideas, but chose to go with Temperature tracking in cold storage, as the system would have trust worthy elements. The system would require an IoT device to track temperatures in a cold storage room; that could be inside a truck or a warehouse. To ensure that goods are stored in the right conditions, a rule engine would be implemented, from a rule set generated by a DSL. A critical part of the system, is how temperatures are transfered from the IoT device to the back-end system for processing. The group verified and validated the MQTT protocol. The group devised these project requirements for how to incorporate elements from all of the courses.

Functional requirements	
ID	Description
FR1	• The system should provide a way to define rules for stored goods as a DSL
FR2	• The system must update an MQTT or alike
FR3	• The system should use Neo4j as graph storage
FR4	• At least 5 automatas of the system must be verified and validated

Table 1: Functional Requirements.

Non-functional requirements	
ID	Description
NFR1	• The system should be able to send sensor data every minute
NFR2	• The system should handle connection loss

Table 2: Non-Functional Requirements.

1.3.1 *Architecture Decisions.*

When designing the system, different architectural decisions had to be made. Publish-subscribe is a common pattern to use for IoT devices, because it is a lightweight machine to machine network protocol intended for connections using devices with limited network capacity [4]. To achieve this method, the MQTT broker is configured as a smart broker, which means it can route messages based on certain conditions. Contrary to MQTT, Apache Kafka is an event streaming platform, built for high scale, utilizing a dumb broker to get a high throughput of messages. As a result, MQTT and Apache Kafka are an ideal pairing for end-to-end IoT integration.

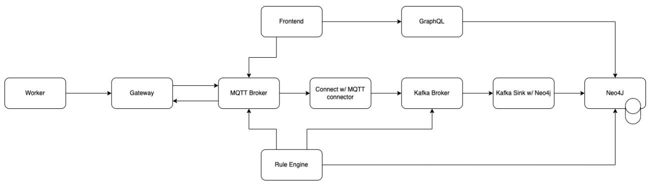


Figure 1: Overview of data pipeline.

Figure 1 shows the overall architecture for the system. The temperature sensor on the Worker IoT-Device collects the first data, which is sent over Bluetooth to the Gateway IoT-Device, which establishes a connection with MQTT and migrates data from the TEMPERATURE topic to the Kafka topic temperature via the Apache Kafka connector. A sink connection for Neo4j is constructed, which will export data from temperature topics into Neo4j. The front-end will be able to retrieve data from Neo4j via the GraphQL server, which unifies data representation. The Rules Engine will also check temperature topics on Apache Kafka to ensure that the temperature is within the appropriate range, failing which it will warn the IoT-device via the MQTT topic rules/alert.

Front-end
To illustrate the solution, the group will also provide a front-end platform that can be used to monitor incoming data. The platform serves simply as a demonstration tool and is not part of the project's scope. As a result, no technical demonstration of how the front-end works will be provided.

1.3.2 *IoT Decisions.*

The hardware is split up into a gateway device and a worker device, to make use of the "Edge" - "Fog" - "Cloud" architecture. The gateway device operates in the "fog" and is placed in the driver's cabin of a truck and connected to either an integrated access point of the truck or a mobile hotspot, thus being able to connect to the internet. Being placed in the cabin, gives the driver the ability to keep an eye on the temperature of the transported products, through the OLED display connected to the gateway. Additionally an LED will indicate the current status of the temperature compared to the rule set up by the customer. If a rule is broken it can be set up to trigger the LED to flash different colours based on if the temperature is too high or too low. The worker device on the other hand operates on the "edge", is placed in the cooling trailer and will be collecting temperature data as soon as it is turned on.
When the two devices are in range of each other, they will automatically pair via Bluetooth Low Energy (BLE) and the temperatures will be transmitted from the worker device to the gateway device and lastly uploaded to the server. This separation allows the worker device to collect data from inside the trailer even before it is coupled to the truck, provided that it is connected to a battery or the power supply of the truck.

For the communication between the two devices there are several technologies to choose from. As previously mentioned the solution uses BLE to transfer data. BLE was chosen due to the fact that its performance space fits well with the solution. [8] BLE has a theoretical range of up to several hundred meters with significant pathloss starting at around 33 meters. However, this still allows the worker to sit anywhere in the trailer and still connect to the gateway, but also limits the range it can interfere with other devices on busy truck stops or distribution centers. BLE also has low power consumption, which can be crucial if the worker has to run on a battery. Even though its throughput is fairly low at roughly 260 kbit/s, it is still more than needed, as it is only expected to transmit temperature data once every 60 seconds. As the group is unable to test it in the real environment, it is assumed that the connection is still stable enough, given the possible obstacles in between the two devices such as the trailer and cabin walls, which in a real scenario could have significant effect on signal strength.

Other possible technologies that could be used include ZigBee, which with its operating frequency of 2.4 GHz could prove to be more usable in the real environment as its signal could penetrate obstacles more easily that BLE, but its increased cost and power consumption makes it less desirable in the field. Another potentially better technology could be LoRa(Wan) [7]. LoRa can provide a higher range of several kilometers, while still being energy efficient. Its low bandwidth of up to 21kb/s is still more than enough for this use case. If however every Truck on the road would use LoRa, it could, due to its high range, create a lot of interference and thus negating its strengths.

Another decision had to be made on the software side of the IoT devices. While traditionally microcontrollers used to be almost exclusively programmed using lower level languages such as C, C++ or Rust, the introduction of Micropython made it possible to control hardware using a higher level language in addition to an already feature rich community. So while C++ and co. Offer more detailed control over memory allocation, higher speeds and less space overhead, the learning curve and error susceptibility make it less approachable. In a scenario where these properties are critical, it might be necessary to use a low level language, but as the system is not timing critical or needs to be space efficient, the benefit of familiarity using python outweighs the downsides of it.

1.3.3 *Domain Specific Language Decisions.*

As previously stated, a rule engine was to be implemented into the system, where the temperature data is held up against the rules specified by the user. In order to create this rule engine, it was decided to develop an external DSL code generator, using Xtext. The decision to use Xtext on this part of the system, was based on how trucks may have different temperature limitations depending on the goods they transport, and therefore having a way for the user to easily define the limitations of their transport truck, would be needed. By using Xtext, it is possible to create a custom, easy to use language, with the domain specific grammar, which makes Xtext a viable choice.[6]

The requirements for the DSL are simple. The user should be able to define a new rule, where either minimum- or maximum temperature can be defined, or both if a range of temperatures are needed. Furthermore each minimum and maximum limitation should be able to have different actions attached to them, e.g. if the temperature goes too low a blue led might light up, and if the temperature goes too high a red led would light up.

The generated file should then be a .py (python) file. The rule engine should constantly check temperature data consumed from Kafka, and check if the data is breaking the rules, specified by the user. In case a rule is broken, the actions defined by the user should be carried out.

1.3.4 *V&V.*

The approach for doing analysis and verification and validation on the temperature sensor system, was by diving into an important part of the system. Thus, verification and validation was done on the MQTT protocol as the primary focus, but also how it integrates into the rest of the architecture. When looking at it from a high-level perspective, one of the major advantages of this approach is that, the system is build up around transferring temperature data from sensors further into another system, and this relies heavily on the MQTT protocol, thus it is vital to ensure that it works as it should. Looking primarily on the V&V it was however discussed as a drawback, that the group didn't have the necessary skills to conduct analysis on such a scale as MQTT protocol poses. However relevant it is, the scale of it was much larger than anticipated, and added a lot of complexity to the verification and validation. Appendix C displays the entire report for further details on how Verification and Validation was done.

1.4 Solution Description and Results

1.4.1 *Architecture.*

To accomplish the different requirements described. Below is the architecture that the group decided upon:

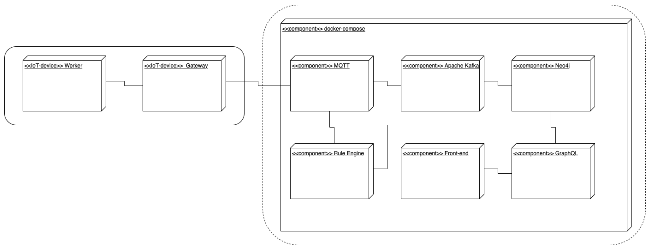


Figure 2: A high-level overview of the architecture.

This architecture achieves a distributed publish-subscribe pattern architectural design. That delivers the benefit of the design having insignificant or no influence inherent in networked interactions, such as delay or poor performance. The architecture enables the services to be scaled up in the event of increased load. The services are loosely coupled, which means that the client has little to no impact on temporary unavailability. The services are also inheriting resilience features.

Scaleability

As the primary functionality for this system is to monitor the temperature in cold storage units, ensuring scaleability is vital as the amount of cold storage units with multiple sensors in each could lead to vast amount of data being transferred. For this reason, Apache Kafka was chosen as the primary hub for distributing data. Kafka's high throughput, uptime and scaleability makes it a natural choice for handling large amounts of data. With the ability to construct multiple brokers to ensure that it can handle the large throughput and the concept of partitions to ensure no data is lost, Apache Kafka makes it so that future solutions can in most cases just plugin in an listen to the given topic and begin whatever job is required. There is however some important aspect to consider, and that is the power consumption of a Apache Kafka producer. For the IoT device, it could be crucial that power consumption is low, so no data is lost because the devices drain their power supply. The help this problem, MQTT is utilized. It is a lightweight protocol that is widely used for specifically IoT devices and works great with Apache Kafka as well.

Front-end

The development of the frontend, is out of scope of the project, but the web platform demonstrates the concept of monitoring data from the backend, which receives data from electronic devices. This allows any operator to monitor incoming temperature changes via the frontend platform. A smartphone application would also be beneficial for any method of transport to react to temperature changes on set, however for this project a web platform was sufficient for proof of concept.

1.4.2 *IoT.*

As mentioned in the Solution Approach section, the two IoT devices communicate using BLE. The way this is accomplished is using the Generic Access Profile (GAP) to assign individual roles to the devices important for inital connection, and the Generic Attribute Profile (GATT) to define how the data to be sent is handled by the two devices. [2] In this system the Gateway device is defined as a "Peripheral" by GAP, which acts as a beacon, continually broadcasting its signal until it connects to another device. The worker on the other hand acts as a "Central", which can scan for "Peripherals" and establish a connection between the two. Once the connection has been made, one of the devices will take the role of the Client, which in this case is the worker device, and the other is given the role of Server. In this configuration, the gateway device manages attributes, which the worker can send request to read from or write to. The worker device in this case requests a write action each time it transmits temperatures, to change the value of the current temperature attribute stored on the gateway device. When the gateway recieves this request it changes the value of the attribute according to the request and is then able to read the value and transmit it to the cloud via MQTT.

To guarantee a good foundation of temperature readings it is necessary to compose a sampling regime that ensures proper handling of the hardware and can handle smaller fluctuations in the accuracy of the readings. For this system, the focus lies not in the detection of quick environmental changes, but instead requires reliable and continuous measurements over a time period of several hours. The temperature sensor that was provided, is the DHT11 [1], which is fairly inaccurate with ±2°C and a resolution of 1. To still ensure reliable measurements, the sensor is sampled 60 times pr. minute and then calculate the average of the values which then gets sent to the gateway device. Even if the response time of the sensor is between 6 - 30 seconds, the solution is ensured to make unique readings and minimize any errors in the sensors responses. The sampling values can be dynamically adjusted to sample faster or slower and longer or shorter before averaging the values if wished to be.

The gateway device, while not having any direct sensors, needs to ensure timeliness when reading the values sent by the worker to ensure that no readings are lost due to being overwritten by the worker before the gateway read them. Therefore it checks for new messages twice a second and in the event of a new reading being present, updates its display, publishes the new data to the cloud and looks for a new message on a MQTT topic about any necessary actions it would need to take if a rule was broken. For this project the action it is able to take is to make an LED indicate the current state of the temperatures in relation to the rules set by the customer. As an example the LED could flash red in the case of the temperatures being too high, blue if they are too low and green if the temperatures are complying to the rules. The timing on the devices is managed in a way that the thread is not being blocked by a sleep operation, to make sure that other operations are still possible to be performed. Both the LED flash frequency as well as the frequency of which the gateway checks for new message can be adjusted dynamically.

When the worker device detects it has lost the BLE connection to the gateway, it starts storing the temperature data into memory instead of sending it to the gateway device. When the connection gets reestablished, the worker begins to transmit the stored readings back to the gateway. This makes sure that in the event of a decoupling of the trailer or before coupling in the first place, the sensor still keeps track of the temperature and can notify the gateway once it is in range again. Storing the data in memory however, comes with difficulties on its own, as the memory on microcontrollers is rather limited. To justify this decision a routine has been designed to still allow for reliability. Each time the worker has to store a temperature reading, the available memory is checked against a threshold. If this threshold is surpassed, the worker deletes every second entry in the stored temperatures to make room for new ones. This solution provides plenty of space for storing temperature readings at the cost of reducing the overall resolution of the readings. Internal tests have shown that the worker is able to store temperature data for 5 hours at the before mentioned sampling of 60 times a minute averaged, before surpassing a threshold of 70% allocated memory. This has been determined to be reasonable, as the time frame in which the BLE connection might be severed, likely will be less than 5 hours at a time.

1.4.3 *Domain Specific Language.*

Grammar

The starting rule of the grammar, "RulesModel" requires a prefix keyword, "Rule" followed by "temperature" and some ID for a name. The model then holds a feature list where the individual rules are stored. The feature takes 0 to many rules, and the grammar is ended by the keyword "end".

The feature list mentioned before, holds all the rules. A rule is defined by a name, which is limited to a predefined list of names, to determine what rule is being defined, e.g. maxTemperature or minTemperature, followed by an integer defining the value of which the rules is broken if it exceeded. Lastly a feature list is defined, taking zero to many actions, which is to be made if that specific rule is broken.

An action is defined by the keyword "action" followed by a predefined keyword, eg. "LEDblink" or "log", indicating the type of action to be carried out. Lastly a String is defined with the e.g. the color of the LED to light up, or the log message.

An example of the grammar can be seen in figure 3, and the grammar implementation can be seen in 4

```
Rule temperature apples
maxTemperature 21 action LEDblink "red"
minTemperature 1 action LEDblink "blue"
end
```

Figure 3: Example grammar

```
grammar org.healthydrone.dsl.expressions.Expressions with
    org.eclipse.xtext.common.Terminals

generate expressions
    "http://www.healthydrone.org/dsl/expressions/Expressions"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

RulesModel:
  'Rule' 'temperature' name=ID
  rules += Rule*
  'end'
;

Rule: name=AllowedRules value=SignedInt (actions+=Action*);

SignedInt returns ecore::EInt: '-'? INT;

enum Color: RED = 'red' | GREEN = 'green' | BLUE = 'blue';

Action: 'action' name=AllowedActions value=STRING;

AllowedActions: 'LEDblink' | 'log';

AllowedRules: 'minTemperature' | 'maxTemperature';
```

Figure 4: Grammar implementation

Generator

The generator generates a .py (python) script, which serves as the rule engine of the particular rules defined in the grammar.

The generator connects to MQTT and Apache Kafka, then instantiates any variables that are required. A while loop is developed that receives the current temperature in the cold storage from Apache Kafka and compares it to the user-defined rules. If any temperature is beyond the stated range in the rule set, the action defined can publish a topic to the MQTT, causing an alert to be sent to the IoT-Device and a notification to be displayed on the front-end.

Validation
Three main validations are made, within the Xtext grammar.

Firstly a validation is run to see if a rule is already defined, to prevent having e.g. minTemperature defined twice, to prevent unexpected behavior.

Secondly, a validation is made to check if the action "LEDblink" is defined, that a valid color is provided, and otherwise throw an error.

Third, a validation is made to check if the minimum temperature value is higher than the maximum temperature value.

1.5 Conclusion

This project encapsulates the three courses "Model-driven software development" (MDSD), "Software technology for internet of things"(IoT) and "Software system analysis and verification"(V&V) and utilizes various techniques from these to create this system for tracking temperature in cold storage.

When looking at the system from the perspective of the individual courses, it becomes clear that the IoT aspect is greatly utilized when working with the temperature sensors and the distribution of it using MQTT for lightweight and efficient transfer of data to kafka on a remote server, from which the data can be further distributed throughout the system. This solution ensures scale-ability and easy access to the data, should a new tool be brought in use.

From MDSD a DSL was created to construct a rule engine which is a core part of the system as it enables one of the most important features: checking if a temperature is too low or too high and thus spoils the cargo. In conjunction with the IoT solution, this enables users to see when a rule is broken and act on that information accordingly.

Finally, the data is transferred using MQTT and onward, so to ensure this functionality, V&V was conducted on this protocol and the immediate surrounding components.

The achieved solution is a composition of these tools and courses, that enables users to conduct surveillance on cold storage temperature and monitor if established rules are broken, allowing for immediate action if this should be necessary.

2 INDIVIDUAL REPORT

Extended Summary

• Verification and Validation(V&V) was described to accommodate the extension of the project, however was not implemented.

• Domain Specific Language (DSL) was updated, to allow multiple sensors, herein the grammar, generator and validator, to generate a more versatile rule engine.

• The Internet-of-Things (IoT) part was simulated through python to publish different temperatures. Actual implementation was described.

Individual Editorial for Group Report

Below are listed the areas I have contributed to this project

- Project idea generation, and finalisation.
- Product/prototype:
  - Verification and Validation of MQTT, primarily Uppaal implementation and verification.
  - Domain Specific Language, including grammar, code generation and validation rules for the grammar.
- Group Report
  - General work of the report, (spelling errors etc.)
  - Section Extended Summary, Section 1.1 and Section 1.2
  - Section 1.3.3: Domain Specific Language Decisions
  - Section 1.4.3: Domain Specific Language

2.1 Problem Description

The problem with the current project, is that it only allows for the use of a single sensor, for the IoT device. Considering a truck can be fairly large, and each corner of the truck might have different temperatures, it would make sense for the system to support multiple sensors for one truck.

2.2 Solution Approach

The aim of this extension is to provide functionality for multiple sensors on the back-end, by extending the Neo4j database, and by extending the DSL implementation, making rules specific for each sensor.

Functional requirements	
ID	Description
FR5	• The system should allow the use of multiple sensors
FR6	• The DSL should be able to specify rules for each sensor, including average sensor value

Table 3: Functional Requirements.

There's two ways of expanding the project, attaching a temperature sensor to each pallet of goods in the truck, or by having a sensor in each corner around the truck. The former would allow much more precise measurements of the temperature for a certain type of food. However it would also add a lot of complexity both to the IoT device, but also the rule engine implementation. You'd need to know what food each sensor is attached to, and that would take time and administration.

Because of this added complexity, the more sensible approach is to have x amount of sensors per truck. That way you can measure the general temperature in a section of the truck, or get the average temperature across all sensors. Therefore the chosen approach of extension is the latter mentioned.

As some elements of the extension aren't possible to implement due to limited hardware, and time, then IoT and V&V will be described theoretically, or mocked.

2.2.1 *IoT Decisions.*

Not many changes are necessary for the extension, when it comes to the IoT side. Using BLE, up to 32 connections can be used, which is more than enough for an extension with 4 or 6 sensors. In order to keep track of each sensor and the data they send, then they should each send their unique ID. This way the data can be differentiated between each sensor in the rule engine, as well as the front end for data visualisation.

2.2.2 *DSL Decisions.*

The rule engine would need to be changed in order to allow a minimum and maximum temperature limit for each sensor. Additionally the grammar validation needs to be updated to correspond with this change, by allowing multiple rules to be created with the same name, however to only allow one rule per sensor.

2.2.3 *V&V Decisions.*

With the addtional sensors, more automata should be implemented in the Uppaal program, and the implementation of timing throughout all automatas would be good to have, in order to have a better representation of the real life system.

2.3 Solution Description and Results

2.3.1 *IoT.*

The IoT aspect of the extension was mocked due to the lack of hardware, however could have been implemented otherwise. The mocking simply consists of sending a message for each "sensor" across the MQTT. The message being sent is a JSON object, which holds an id, value of the temperature and a timestamp. The ID represents the sensor from which the message originated.

The real hardware solution would require an update on the IoT device. The bluetooth module can handle multiple connections, however documentation is a bit vague but generally seems to agree to 8+ modules at the same time. The IoT device works as a Peripheral module, which waits for a connection, while the workers are central devices that scan for other devices, more specifically the IoT device in this case, and initiates the connection. Theoretically this could lead to hot-swapable sensors, however in that case the IoT device should manage ID's for each sensor.

The IoT device publishes any messages it retrieves from the workers immediately as it's received.

2.3.2 *Domain Specific Language .*

The DSL was updated so that each rule could define a set of sensors for which that rule would be valid for. This was done by adding a feature list with the keyword "sensors" to the Rule feature, and allowing 0 to many integers, each representing the id of the sensor.

By specifying the exact sensors the rule would apply for, it is also possible to define separate values for each sensor, e.g. sensor 1 and 2 have a minimum temperature of 5 while sensor 3 and 4 may have a minimum temperature of 2. Example of the implementation can be seen in Figure 5 and figure 6.

```
Rule temperature Betty
maxTemperature 15 sensors 1, 2, 3 action LEDblink "red"
minTemperature 5 sensors 2 action LEDblink "blue"
minTemperature 10 sensors 3 action LEDblink "red"
end
```

Figure 5: Example Extension Grammar

```
RulesModel:
  'Rule' 'temperature' name=ID
  rules += Rule*
  'end'
;

Rule: name=AllowedRules value=SignedInt ('sensors'
    (sensors+=INT (',' sensors += INT)*)*)?
    (actions+=Action*);

SignedInt returns ecore::EInt: '-'? INT;

enum Color: RED = 'red' | GREEN = 'green' | BLUE = 'blue';

Action: 'action' name=AllowedActions value=STRING;

AllowedActions: 'LEDblink' | 'log';

AllowedRules: 'minTemperature' | 'maxTemperature';
```

Figure 6: Grammar Extension Implementation

When the system only allowed one sensor, a validation rule was implemented to make sure that each rule was only specified once to avoid issues. This had to be changed, and it now instead checks if a sensor is already specified by another rule of the same name.

2.3.3 *Verification & Validation .*

The V&V aspect of the project is a bit tricky, as it would need a larger reiteration to be reflective of the new system. One of the main issues that arises is the lack of time representation in the system, which would need to be investigated how it affects real life, and then implemented into the Uppaal model.

A more in-depth representation of the IoT device and the temperature sensors would also be a plus in the V&V Model, so that it's less abstract and closer to the actual implementation. That way you could maybe even extend the Uppaal model to verify Average temperature checking and making sure everything is in sync and nothing deadlocks if a sensor fails.

2.3.4 *Challenges.*

There are some main challenges with the implementation described, mainly how the data is represented. Since the data is published the moment the IoT device receives it, it's hard to get accurate average temperatures for the whole truck.

Some ways of combating this is by sending data from all sensors in the same message, however this opens up the issue if a sensor dies, where you then need to handle synchronisation of the sensors, e.g. by sending incomplete data from just 3 of the 4 sensors.

2.4 Conclusion

The extension of the project contains an implementation the "Model-Driven Software Development" aspect of the project, while the courses "Software Technology for Internet of Things" and "Software System Analysis and Verification" is described theoretically due lack of hardware and time to develop a more in depth V&V Model in Uppaal.

The goal of the extension was to implement multiple sensors for the system. While not having hardware to implement this change physically, the IoT aspect was instead simulated by running a python script that publishes random temperature values across the MQTT connection.

The DSL was implemented to adapt to this change, so the user can define rules for each sensor, which is reflected in the Rule Engine that is generated. The validation of the grammar for the DSL was also updated to allow this change.

The V&V aspect of the project was briefly described, specifically how the Uppaal model should be changed to accommodate the extension. This included the implementation of time and more in-depth sensor and IoT device representation.

REFERENCES

[1] D-Robotics UK 2010. *DHT11 Humidity & Temperature Sensor.* D-Robotics UK.

[2] Mayur R. Deogade, Harshal Belan, and Milind Rane. 2019. Range Performance Validation Trials by changing RF Data Rate. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*. 1–5. https://doi.org/10.1109/I2CT45611.2019.9033794

[3] Clementine O'Connor Hamish Forbes, Tom Quested. 2021. UNEP Food Waste Index Report 2021. (2021). https://www.unep.org/resources/report/unep-food-waste-index-report-2021

[4] OASIS Standard Incorporating. [n.d.]. MQTT Version 3.1.1 Plus Errata 01. ([n. d.]). http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/

[5] Simon S. Wittus Mads Werge, David McKinnon. 2019. Madaffald fra detailhandel og anden fødevaredistribution 2019. (2019). https://www2.mst.dk/Udgiv/publikationer/2021/05/978-87-7038-320-2.pdf

[6] Sven Efftinge Sebastian Zarnekow. 2014. Developing Domain-Specific Languages wth Xtext. (2014). https://www.eclipse.org/community/eclipse_newsletter/2014/august/article1.php

[7] Kamil Staniec and Michał Kowal. 2018. LoRa Performance under Variable Interference and Heavy-Multipath Conditions. *Wireless Communications & Mobile Computing (Online)* 2018 (2018), 9. https://www-proquest-com.proxy3-bib.sdu.dk/scholarly-journals/lora-performance-under-variable-interference/docview/2407627571/se-2 Name - SigFox; Copyright - Copyright © 2018 Kamil Staniec and Michał Kowal. This work is licensed under http://creativecommons.org/licenses/by/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2020-05-29.

[8] Jacopo Tosi, Fabrizio Taffoni, Marco Santacatterina, Roberto Sannino, and Domenico Formica. 2017. Performance evaluation of Bluetooth Low Energy: A systematic review. *Sensors (Basel)* 17, 12 (Dec. 2017).

APPENDIX

A FIRST APPENDIX

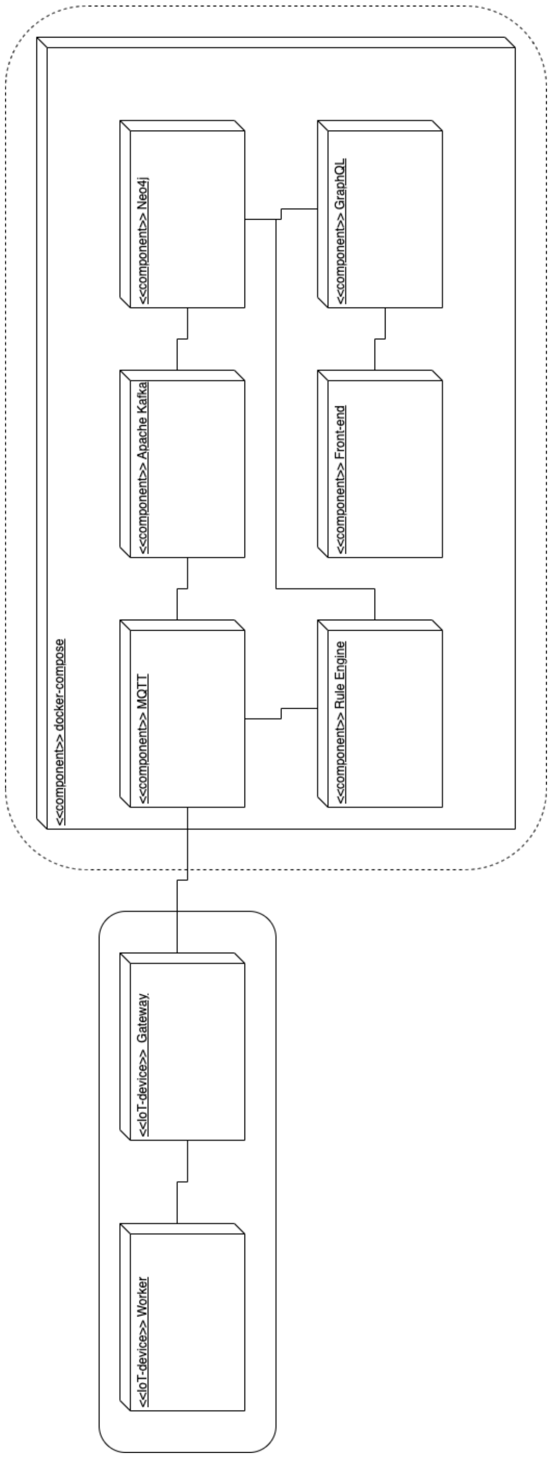


Figure 7: An enlarged image of the high-level overview of the architecture.

B SECOND APPENDIX

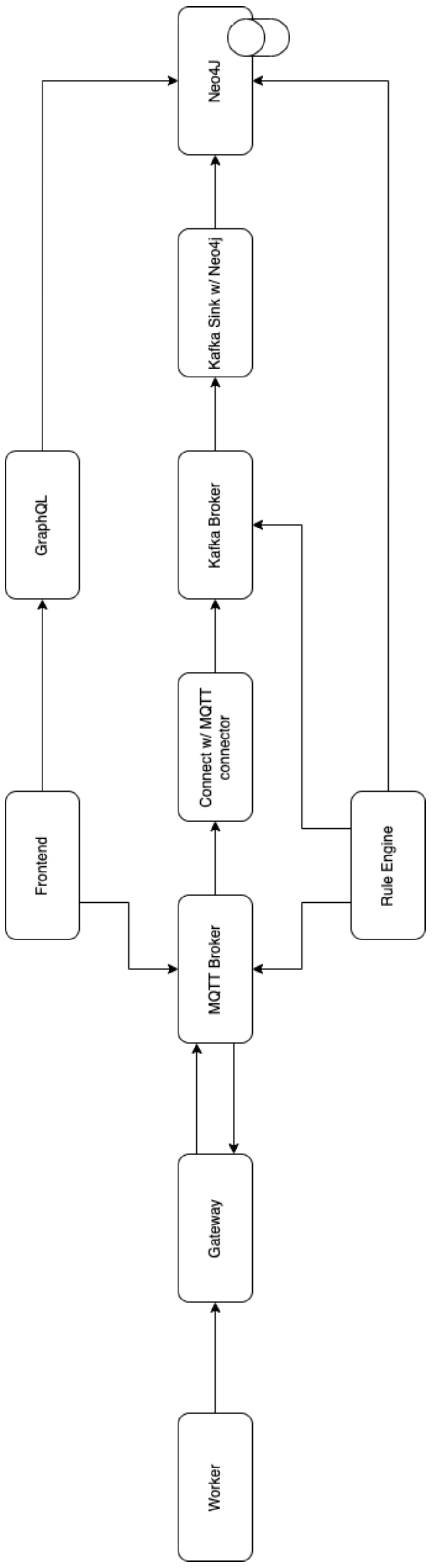


Figure 8: An enlarged image of the data pipeline overview.

C THIRD APPENDIX

Report on Verification and Validation of the MQTT Protocol, made in Software System Analysis and Verification, starting from next page.

SSAV Mini-Project on MQTT protocol, Group 8

1st Frederik V. Helth
Faculty of Engineering
University of Southern Denmark
Odense, Denmark
frhel18@student.sdu.dk

2nd Jonas Lund
Faculty of Engineering
University of Southern Denmark
Odense, Denmark
jolun18@student.sdu.dk

3rd Mads E. Falkenstrøm
Faculty of Engineering
University of Southern Denmark
Odense, Denmark
mafal17@student.sdu.dk

4th Mathias B. Kristiansen
Faculty of Engineering
University of Southern Denmark
Odense, Denmark
matkr18@student.sdu.dk

5th Patrick Nielsen
Faculty of Engineering
University of Southern Denmark
Odense, Denmark
panie18@student.sdu.dk

***Abstract*—This paper presents a formal modeling of the Message Queue Telemetry Transport (MQTT) protocol. Formal verification has been conducted using UPPAAL model checker to evaluate the performance and resiliency of the MQTT protocol in conjunction with a project using IoT devices. Verification queries were formulated to verify specific properties and modeling the MQTT protocol as a "beginner project" was discussed.**

***Index Terms*—Model Checking, Uppaal, Automata, Verification, Validation**

I. Introduction

During our semester project and the course in Verification and validation, we where tasked to make a simulation model of one of our key components in the project. The purpose of the project was to create an IoT device that would measure the temperature inside cold storage room and transfer the data to the cloud for analysis. The communication between the devices and the cloud utilized the MQTT protocol. MQTT is good at distributing information more efficiently, Reducing network bandwidth consumption, increase scalability and is well-suited for remote sensing and control [1]. We had to find a critical part of the system to validate and verify while working on this project. We discovered that the MQTT protocol was the most crucial component, therefore we chose to model it in order to check and verify its operation.

A. Architecture

The architecture accomplishes an architectural design of a distributed publish-subscribe pattern. This is achieved by utilising the MQTT protocol. The MQTT gives us a loose coupling and allows for one-to-many communications, that is beneficially for us to communicate with our IoT device. The architecture described in this paper is only a subpart of the entire system, as we have narrowed it down to present what is necessary to understand the system and to reduce complexity for modelling the system.

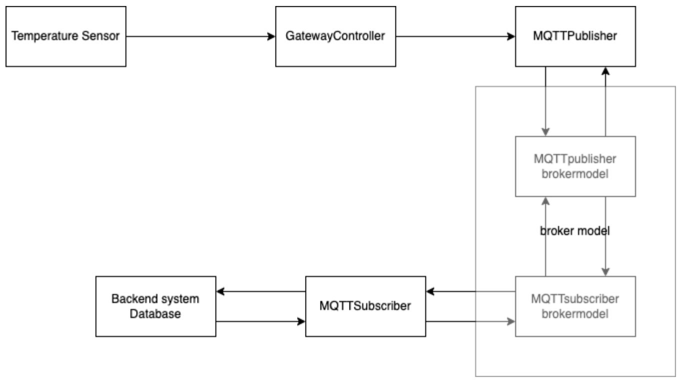


Fig. 1: Architecture.

In Figure 1 the GatewayController is collecting temperature data from temperature sensor in near-real-time, that is reported via MQTTPublisher to the MQTT-broker. The MQTT-broker is a middleware that manages topics. The Backend system can then listen in on the topics via the MQTTSubscriber, then do statically analysis on the topics and then store the results in a database.

II. Verification and Validation

During the course of V&V the group have been introduced to UPPAAL model-checker [2] that consists of three main parts: a description language, a simulator and a lastly the model-checker. Given a finite state model of a system and a formal property, model checking is an automated approach that verifies whether the property holds (a given state) in that model. UPPAAL provides us with a toolkit for real-time system verification that allows us to reproduce certain features of our system to confirm that it functions as planned. the UPPAAL model-checker is based on timed automata theory, with extra features like bounded integer variables and urgency in its modeling language. UPPAALS's query language is a subset of CTL (computation tree logic), and it's used to provide properties to verify

A. Requirements

Given that the MQTT protocol is a critical component of our system, we investigated its specifications and identified a set of both functional as well as non-functional requirements, it would need to fulfill in our system. Using UPPAAL we utilized its statistical model checking, to verify its behaviour against the specified requirements.

Functional requirements	
ID	Description
FR1	• A publisher must be able to publish data
FR2	• A subscriber can subscribe to such data
FR3	• A subscriber should be able to disconnect
FR4	• If a ping request is send, a ping response should be returned

TABLE I: Functional Requirements.

Non-functional requirements	
ID	Description
NFR1	• The system must handle connectivity issues
NFR2	• The system should not deadlock

TABLE II: Non-Functional Requirements.

III. Model

The model that is build is designed primarily around the MQTT protocol, and how it interacts with our system. It consists of 10 different automata, all representing a vital part of either the MQTT protocol or a backend part of our system in which it exists. To better understand how the system works, some abstractions have been made to illustrate different parts of the system, while the MQTT protocol is modeled more in depth. Figure 2, 3 and 4, all represents a part of the backend system.

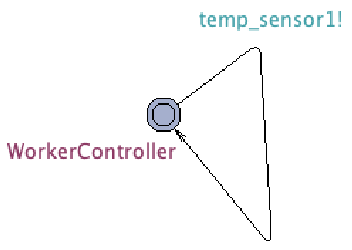


Fig. 2: Workercontroller.

The workercontroller has some abstractions, as it is assumed that it just transfers data. No further elaboration on how the workercontroller and the gatewaycontroller interacts will be described in this paper.

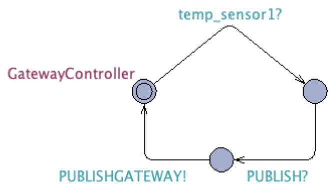


Fig. 3: Gatewaycontroller.

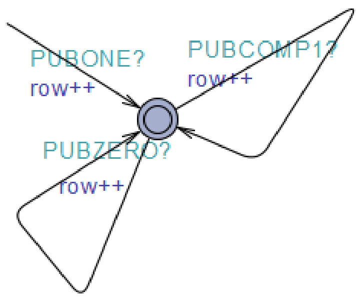


Fig. 4: Database model.

The database works by simply listening for synchronizations and then incrementing an integer, symbolizing a row in a database. This abstractions was made to simply illustrate what happens when a subscriber has received a message and everything related to the MQTT protocol is done. While the system has more than a database that is reliant on the data transferred from the sensor, a database was chosen as to not broaden the scope unnecessarily.

Looking at figure 2, it represents a sensor transmitting data to a gateway controller, illustrated in figure 3. It then acts as the first interaction with the MQTT protocol. From here the MQTTpublisher seen in figure 8 acts a the clientside of a publisher, meaning this is where data is sent to the servermodel which is represented in figure 10a and figure 10b. Finally figure 9 illustrates the clients side of a subscriber.

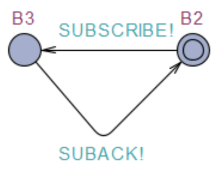


Fig. 5: Subscriber model.

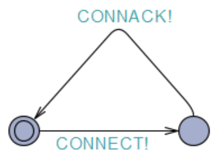


Fig. 6: Connection model.

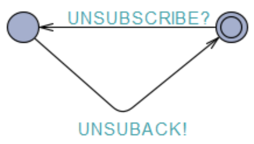


Fig. 7: Unsubscriber model.

Figure 5, figure 6 and figure 7 all represents different ways of interacting with the MQTT protocol. It is possible for a client to connect to the system, and subscribe to a specific topic, from which it then receives updates, and finally the client can decide to unsubscribe and no longer receive updates from this topic. It is important to note that each message must be associated with a topic, such that the user can subscribe to the specific topic and only receive messages that are related to the topic. This allows for creation of many topics, but users may only need to subscribe to the data they need, rather than receiving everything.

Looking at figure 8 and figure 9 it is the client side of the subscriber and the client side of the publisher. Both are made with the intent to show different paths, based on the quality assurance level. These assurance levels are MQTT specific and are used to describe how different messages should be treated.

A. *quality assurance levels*

The MQTT procol contains 3 different assurance levels when handling messages. The first level is where quality of service(QoS) is set to 0. This means that a message should be delivered at most once, and the publisher must send a packet containing the information that QoS = 0. This is the lowest possible QoS level, and there is no guarentee of delivery. The receiver does not acknowledge reception of the message. This is also the assurance level with the least overhead. The second level is where QoS is set to 1. This means that a message should be delivered at least once and it guarantees that a message is delivered at least one time to receiver. The sending unit will store the message until it gets a PUBACK packet from the receiving unit with acknowledgement of receiving the message. This also means that a message can be delivered or sent multiple times. This level has a bit more overhead compared to the first level. The third level is where QoS is set to 2. This means that a message should be delivered exactly once and thus carries the most overhead, as it requires a four-part handshake between the sender and receiver. This means that the sender will continue to send the message, but with and extra value DUP, to signal that it is a duplicate. Once the receiver sends back a PUCREC package, the sender will stop sending message, and the final handshake of acknowledgement is done. The sender sends a PUBREL package, and the receiver returns with a PUPCOMP package to finish the exchange for this message.

All three of these QoS levels has been implemented as a part of the system, but with some abstractions. First of all, before running the automatas, a user should manually set the QoS level. A user can also not set it, and the system will auto increment the value to simulate a QoS level being set. Finally the automata will follow the path based on the given level that has been chosen. These abstractions was made to more easily simulate how it interacts in our final build. Comparing our solution to the one in [3] it is clear that while the MQTT model is somewhat the same, as it is the same procedure, it focuses only on the protocol itself and therefore includes less abstractions regarding the QoS levels, but also adds some timed states and edges to check for ”hanging connections”.

In figure 8, 9, 10a and 10b it is also possible to see the inclusion of a PINGREQ and PINGRESP. These states represents the possibility for the clients on either side to send a ping request to the server in other to check for connectivity and keeping the connection alive. The server will then respond with a ping response. This is also represented in our system to some degree, but designing a more complex solution would probably give a more correct representation. This of course was not our main focus, and has therefore been implemented loosely.

IV. Formal Verification

To verify that the MQTT protocol fulfills the requirements, we use the query language provided by UPPAAL. The queries specify what properties the model should exhibit. UPPAAL then models these in order to provide counter examples in which the property does not hold. If that is the case, the requirement is not satisfies. If however it can not provide a counter example, the property is assumed to be verified.

To verify the requirements mentioned in Table I using UPPAALs query language, we make use of Computation-Tree Logic. Verifying the requirement FR1 will be done using the following query: *E<> PropPub.publish* , meaning that there exists a path where the serverside of the publisher gets to the ”publish” state. Likewise, the query for FR2 is verified using: *E<> PropSub.subscribed* . Verifying if a subscriber can unsubscribe, and therefore reset its connection can be tested using the formula: *E<> PropSub.reset*

For FR4 in Table I we want to make sure that every ping request from the client side is eventually responded to from the server side. This can be modelled by the query *PropPup.PingReq –> MQTTpublish.PingReq* and *PropSub.PingReq –> MQTTsubscribe.PingReq* respectively. These queries state, that whenever either of the clients sends a ping request, the server side will eventually end up responding to said request.

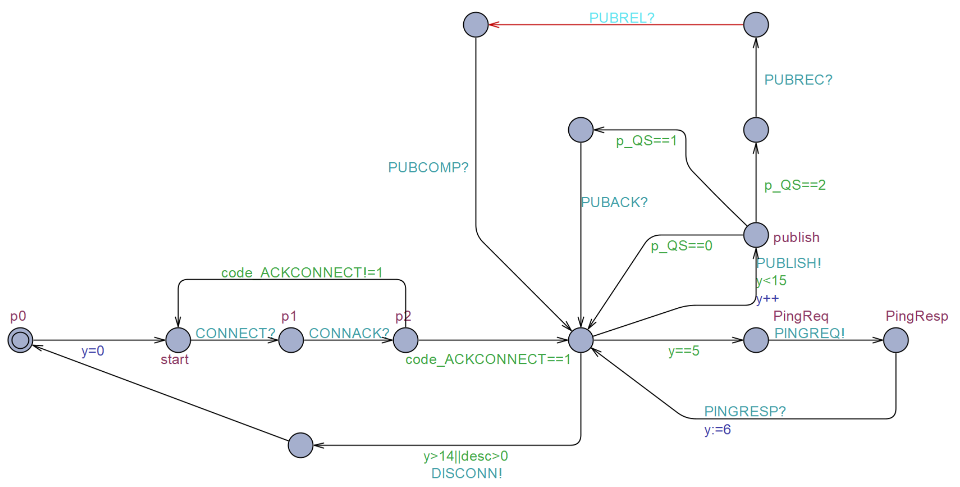


Fig. 8: Publisher client.

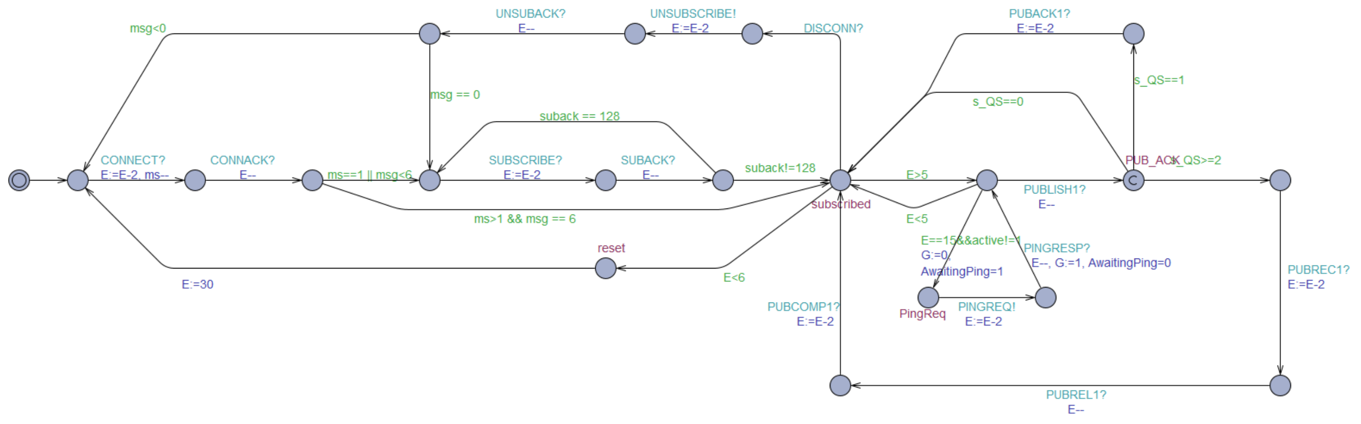


Fig. 9: Subscriber client.

Lastly we verify the safety property of the system not deadlocking with the query: *A[] not deadlock* , which states that for all paths there will never be a deadlock.

Verification results	
Query	Result
A[] not deadlock	inconclusive
E<> PropPub.publish	verified
E<> PropSub.subscribed	verified
E<> PropSub.reset	verified
PropPup.PingReq –> MQTTpublish.PingReq	inconclusive
PropSub.PingReq –> MQTTsubscribe.PingReq	inconclusive

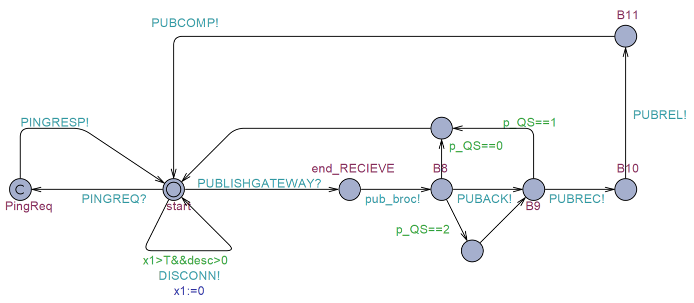
TABLE III: Functional Requirements.

The results of the verification can be seen in Table III. The safety property of not having any deadlocks in the system has been returned as inconclusive, as the process took too long, due to state space explosion. In case of the requests regarding the ping request / response property, the results were inconclusive for both the subscriber and publisher model, this cause of discrepancy is due to that kind of CTL expression used in our queries, does not follow the UPPAAL grammar/logic rules.

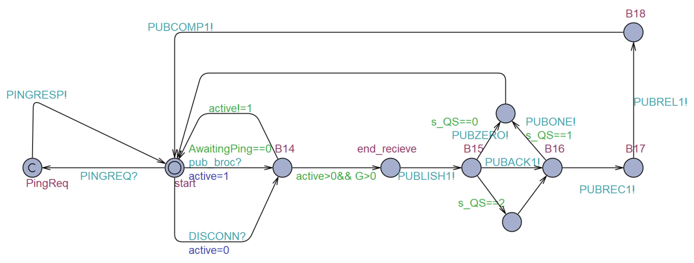
As will be apparent from Section V, there are more properties we would have liked to verify but were not able to, due to increasing complexity as well as state space explosion.

V. Discussion

When the system is build there is a need to verify the different parts of it, to ensure that the errors that exist are to be found and corrected. In a real life scenario this would also be where the necessary authorities are informed of the potential mistakes before an actual system is put into production. Looking at the performed queries, it is clear that while an attempt was made to design a model explaining our solution, the queries are not adequate compared to the size of the model. This is mostly because there was a lack of understanding for how to handle Uppaal, its query checker and a general understanding of what to check for. This leads to some of the lessons that the group has learned while building this



(a) Broker model of publisher.



(b) Broker model of subscriber

Fig. 10: Broker model

project. First and foremost was that working with the MQTT protocol was much larger than first anticipated, and required a much greater understanding of not only MQTT but also Uppaal to create a great result. For a first time project, the group should have gone with something that was more inline with the amount of time and skill that had been gained from the lessons. It is clear that Uppaal brings the possibility of conducting model checking and verification and validation on a system, but it also comes with a rather steep learning curve. Combining that with the many options for designing a system, and conducting verification and validation, it takes a longer period of time to fully understand the possibilities that Uppaal enable. Comparing our system to the model in [3] it is clear that the paper utilized many more of the provided tools, especially the utilization of checking for time constraints, which is a very relevant topic when working on this kind of system. For now it will be noted down as future work.

VI. Conclusion

The MQTT protocol is a crucial part of the overall system and should function as intended. We have therefore created a formal model of the protocol as well as abstractions of the rest of the system, to verify MQTT's behaviour in the actual system, as well as broaden our knowledge of the mechanisms that the MQTT protocol uses. A set of requirements was formalised to assure MQTT would fit with the overall requirements to the whole system. These requirements were translated into Uppaal's query language and verified using the model checker.

With an ever increasing complexity in software and hardware, as well as an increasing importance in its trustworthiness, validation and verification becomes a critical step in the development process. Assessing your systems in a predictable and reproducible environment, helps to identify problems early and verifies that the system can live up to its requirements. While Uppaal helps to make v&v techniques accessible, it still has some way to go in terms of user experience, to make it easier for developers to verify their systems.

References

[1] O. S. Incorporating., "MQTT Version 3.1.1 Plus Errata 01," accessed 2022-05-02. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/

[2] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1–2, p. 134–152, dec 1997. [Online]. Available: https://doi.org/10.1007/s100090050010

[3] M. Houimli, L. Kahloul, and S. Benaoun, "Formal specification, verification and evaluation of the mqtt protocol in the internet of things," in *2017 International Conference on Mathematics and Information Technology (ICMIT)*, 2017, pp. 214–221.

VII. Appendix

- Everyone made a fair and equal effort in creating the automatas.
- Abstract: Mathias and Patrick
- Introduction: Frederik and Mads
- Verification Validation: Mads, Patrick and Jonas
- Model: Everyone
- Formal Verification: Mathias, Patrick and Mads
- Discussion: Frederik and Jonas
- Conclussion: Mathias, Jonas and Frederik
- References: Everyone