NTNU
Norwegian University of
Science and Technology

Monitoring and Detecting Failures in Wide Area IoT Networks

Hans Henrik Grønsleth

Master of Science in Communication Technology
Submission date: December 2016
Supervisor: Frank Alexander Krämer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Title	Monitoring and Detecting Failures in Wide Area IoT Networks
Student	Hans Henrik Grønsleth
Responsible professor & supervisor	Frank Alexander Kraemer, ITEM

Background Low Power Wide Area Networks (LPWANs) offer inexpensive infrastructure and long range communication, making it easier than before to develop systems consisting of connected devices spread over large areas. The Carbon Track and Trace project (CTT)[1] develops an Internet of Things (IoT) system that monitors greenhouse gas (GHG) levels in cities. When data from a newly deployed sensor stops arriving, you know that something is wrong, but where do you start looking? Has the sensor run out of battery? Has the gateway lost its connection? Or is perhaps the third party provider of the core network down? Knowing what is wrong and where to look has the potential to add value to wide area IoT applications by easing the development process and making maintenance more responsive through quick failure detecting.

Problem Area How can failures in IoT applications be detected?

Objectives The thesis will investigate how data and meta data from an IoT application can be used to discover and identify failures in the application without or with minimal extra instrumentation. Based on the findings, a system that uses the data (or the lack of data) to increase the overall dependability of the application will be developed. For the system to be a reliable network monitor, it must in itself be dependable. The implementation will therefore be done using the Akka toolkit[2] to create a robust design (through the Actor model) with the possibility of scaling up (to handle multiple and larger applications) and scaling out (to avoid single point of failure) without re-writing the code.

Criteria for Goal Achievements A working prototype of a system that discovers, identifies and alerts failures in the applications it monitors has been developed. The dependability and scalability of the system are assessed. Examples of how the system handles different failures in the monitored applications are described clearly.

[1] www.carbontrackandtrace.com
[2] www.akka.io

Solutions nearly always come
from the direction you least expect,
which means there's no point trying to look in that direction
because it won't be coming from there.

— DOUGLAS ADAMS

Abstract

Low Power Wide Area Network technologies provide an infrastructure that facilitates development of wide area IoT applications. This brings with it a need for a better understanding of how components in such applications should be monitored.

Through a design science research process, an IT artifact has been developed. The artifact was released into the environment at an early stage to start giving value to the application from the beginning, allow potential improvements to be discovered more easily and increase the probability of discovering unforeseen failures produces by the environment. The thesis shows how monitoring can be done without adding extra instrumentation at the monitored components, but by using the data (or lack of such) produced by the components to detect failures. The system also enables easy comparison of the collected data with weather forecast data, to investigate possible impacts weather might have on the application.

The developed system has successfully detected, identified and notified supervisors about failures in the monitored network—and other external resources that it interacts with—for 2 and a half months, and has proved itself to be dependable even during unexpected failures.

Sammendrag

LPWAN-teknologier tilbyr en infrastruktur som gjør det lettere å utvikle IoT-applikasjoner i større områder enn tidligere mulig. Dette bringer med seg et behov for å bedre forstå hvordan komponenter i slike applikasjoner bør monitoreres.

Gjennom en design science forskningsprosess har et IT-artefakt blitt utviklet. Artefaktet ble tidlig introdusert i miljøet for å gi verdi til applikasjonen fra begynnelsen, for å lettere kunne identifisere mulige forbedringer og for å øke sannsynligheten for å oppdage uventede feil skapt av miljøet. Avhandlingen viser hvordan monitorering kan utføres uten å ekstra instrumentering på de overvåkede komponentene, men ved å bruke data (eller mangel på data) produsert av komponentene, til å oppdage feil. Systemet gjør det også mulig å enkelt sammenligne den innsamlede dataen med værprognoser, for å kunne undersøke mulige påvirkninger vær kan ha på applikasjonen

Det utviklede systemet har oppdaget, identifisert og varslet mennesker om feil i det monitorerte nettverket—og andre eksterne kilder som det interagerer med—i 2 og en halv måned, og har vist seg som et pålitelig system selv under uventede feil.

Preface

This thesis is submitted for the degree of Master of Science in Communication Technology at the Norwegian University of Science and Technology. The research presented was conducted during the fall of 2016 by Hans Henrik Grønsleth.

I would like to thank my supervisor, Frank Alexander Kraemer, for his guidance, motivation and rapid feedback whenever questions needed answers.

Contents

1 Introduction

1.1 Motivation

CTT aims at giving municipalities rapid feedback on how well climate policies work by developing an automated system for monitoring GHG levels in cities. In order for this system for this system to work, they need to know what is wrong when it stops working.

By understanding how monitoring of IoT applications in wide area network can be done without adding extra instrumentation on already deployed devices, we can help the continued development of such application.

In the bigger picture, contributing to a better understanding of greenhouse gas emissions feels meaningful from an ethical perspective. New technology can be used for good or for bad. I would argue this use of IoT definitely falls under the former.

1.2 Limitations

There are many characteristics that might be of interest when designing a system. Even though the system in the future might deal with time and privacy critical data like eHealth, *delay* and *security* are not addressed in this thesis. The use case looked at in this thesis is data from IoT devices reporting environmental characteristics like CO2, temperature, humidity and Particulate matter (PM).

Other dependability characteristics that are normally discussed together with availability and reliability are *setup time* and *data loss*. The setup time is not of

interest, since the current use case is not time critical. As we are using a framework, the data loss. The data loss prevention techniques used in the framework will be briefly introduced, but custom data loss prevention techniques will not be implemented.

1.3 Thesis Outline

The remainder of the thesis is structured as follows:

- Chapter 2 gives background information in the domain of IoT, dependability and the Carbon Track and Trace project that the work is tightly coupled with.
- Chapter 3 describes a watchdog system that can monitor a system through the use of timeouts.
- Chapter 4 extends the system described in the previous chapter to include forecast data and visualisations of data trends.
- Chapter 5 gives an overview of the implementation of the features described in Chapters 3 and 4
- Chapter 6 documents conducted failure experiments and the system's handling of these failures
- Chapter 7 evaluated the developed system

2 Background

2.1 The Carbon Track and Trace project

Trondheim municipality has a goal of reducing the total GHG emissions by 25% by 2020 and 70–90% by 2050, compared to 1991 levels [Kom10]. Traditionally, Norwegian municipalities have relied on Statistisk Sentralbyrå [Statistics Norway] (SSB) to provide data regarding GHG emissions. This data was available as yearly reports, both nationally and divided by municipality. However, due to quality concerns regarding the data, the reports for municipalities were discontinued in 2012 [Mun12]. The lack of data makes it harder for policy makers in the municipalities to justify their actions towards the ambitious GHG emission reduction goals. CTT aims to fill this knowledge gap by developing an IoT system that collects GHG data in cities and makes this data available for policy makers in the municipalities [ADK+16].

2.1.1 The Things Network

CTT use The Things Network (TTN) as the backbone of the application, meaning all data flows through this network. When a sensor sends a measurement, this is received at one or more gateways. These gateways have been configured to forward the data to TTN, which in turn makes it available for retrieval. The goal of TTN is to make it easier to create IoT applications. Their hypothesis is that if the infrastructure is already present, innovation will thrive, just like it has done on the Internet [Net16].

Figure 2.1: LoRaWAN Gateways Connected to The Things Network. Figure from [Net16]

> The Internet was created by people that connected their networks to allow traffic from, to and over their servers and cables to pass for free. As a result, there was abundant data communication and exponential innovation. The Things Network is doing the same for the Internet of Things by creating abundant data connectivity. So applications and businesses can flourish.

This is their vision. And being only approximately 1.5 years old, it is remarkable how fast the network and community around TTN has grown. Figure 2.1 shows an overview of gateways connected to TTN worldwide.

2.1.2 LoRaWAN

The protocol used for transmission between sensors and gateways is LoRaWAN. This is an alternative to 2G/3G/4G and WiFi, suitable for battery driven devices that needs long range, low bandwidth communication. In my project report, written during the spring of 2015, I investigated the suitability of the LoRaWAN protocol for the purpose of measuring GHG in urban areas. The results of the research gave insights into the limitations of frequency given the size of the data with respect to national and EU regulations, but also with respect to TTNs *fair access policy*. For the packet size used by the sensors at the time[1], sensors were restricted to sending at most approximately every 7 minutes. The work was cited in a paper co-authored by CTT and my supervisor [ADK+16].

[1]The implementation has later changed from using pure text to using binary representation, which has greatly reduced the packet size, allowing even more frequent transmissions

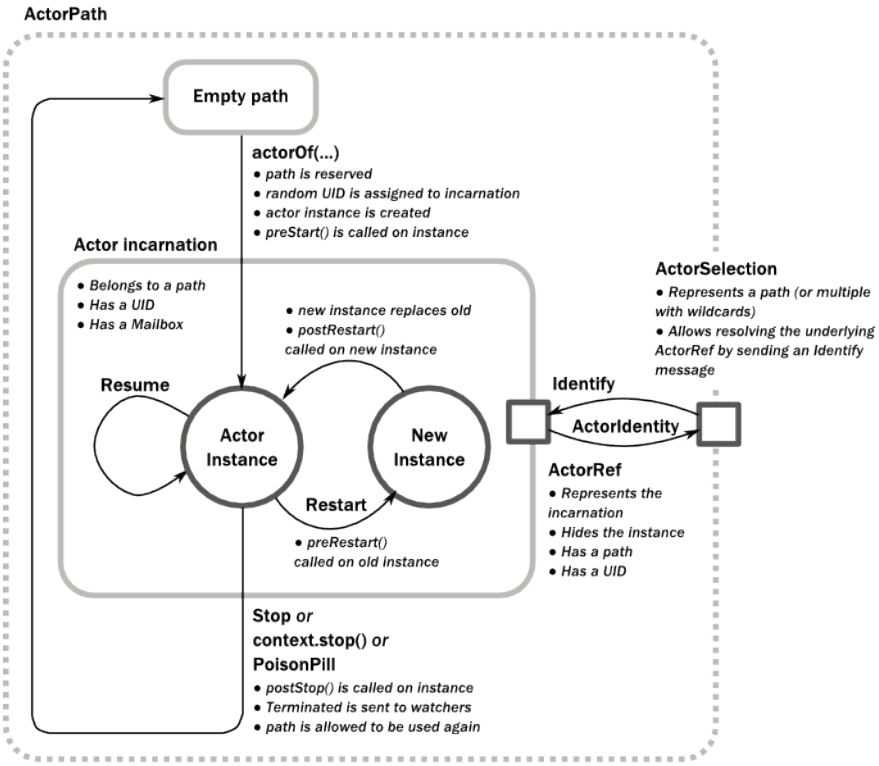


Figure 2.2: Actor Lifecycle. Figure from [Lig16].

2.2 The Actor Model

The actor model was introduced by Carl Hewitt in 1973 [HBS73]. Figure 2.2 shows the lifecycle of an actor as it is defined in Akka.

We will use the actor model through the use of Akka as a toolkit in the implementation of the system. The advantage of constructing a system with actors, is that the transition from one machine to multiple machines is greatly simplified, compared to system with a lot of shared state. Actors have their local state, and don't need to know much about the other actors, other than how to contact them. This makes systems created with Akka scalable by default.

2.3 What is Dependability?

In the introduction we say the goal of this thesis is to create a dependable system. What does it mean for a system to be dependable, exactly? In everyday language, the terms *dependable* and *reliable* are used interchangeably. Whether you say something has an *error* or a *failure*, doesn't make much difference—the people around you understand that the thing isn't working. In technical terms, there are however differences. In this section, key terms regarding dependability will be defined to clarify their use throughout this thesis, and how they are generally used throughout the literature on the field. The definitions used here are based on those given in [ALRL04] and [EHHP13].

From [EHHP13, p. 24] we have the following definition of *dependability*:

Dependability	The trustworthiness of a system such that reliance can be placed on the service it delivers.

Reliability is defined as:

Reliability	The ability of a system to provide uninterrupted service.

And *availability* is defined as:

Availability	The ability of a system to provide a service at a given instant of time or at any instant within a given time frame.

Availability and reliability are *dependability attributes*. This means, the dependability of a system is determined by the availability and reliability of the system, among other things. Availability has to do with whether or not the system is available when we need it: Is my supervisor in his office when I visit it? Reliability has to do with whether the system is we expected to keep working during operation: If I am talking to my supervisor, can I expect him to finish our conversation, or will he suddenly interrupt the conversation and leave the room?

The relationship between the terms define above can be best expressed through the use of a tree. As we can see from Figure 2.3, the dependability of a system has attributes (e.g. availability and reliability), it has threats and means to deal with these threats. We will revisit this dependability tree in the evaluation of the developed system to identify which means are emphasised in different parts of the system.

There are many other characteristics of systems, such as performance, capacity, throughput and delay, to mention some. Even though these aspects are undoubtedly

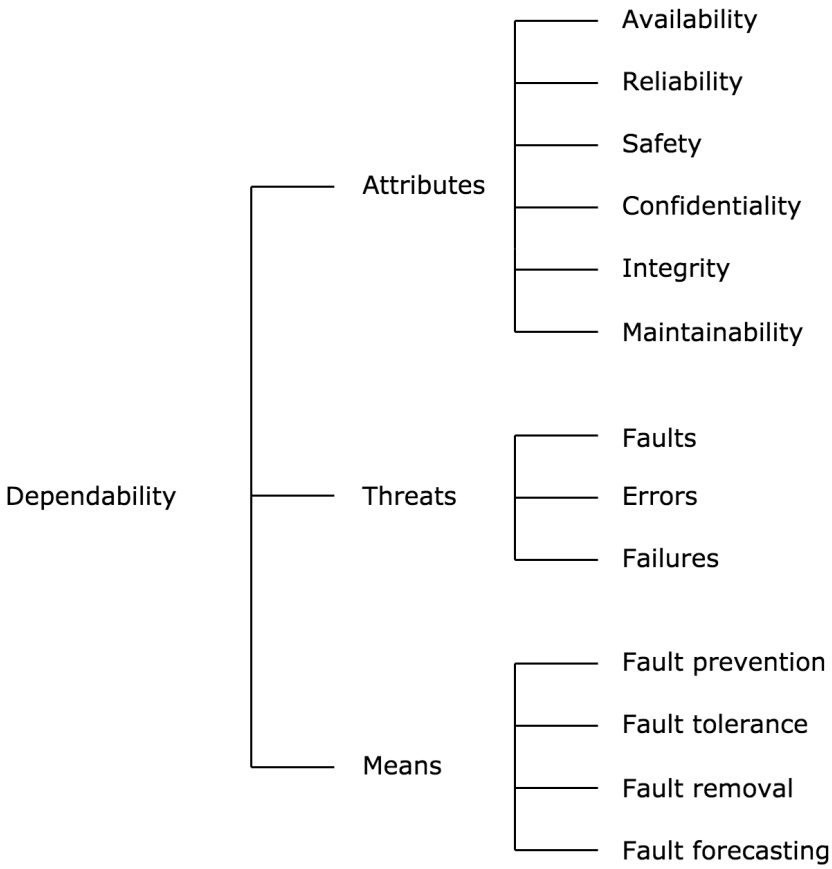


Figure 2.3: The Dependability Tree. Adapted from Figure 1.11 in [EHHP13, p. 25].

important to take into consideration when trying to create systems that are dependable and pleasant to interact with, I have chosen not to look specifically at these in this thesis. [EHHP13] gives a good introduction to the concepts that are not covered here.

Assume we have a system S that is made up of a set of subsystems (system components) S_1, S_2 and so on. Formally, $S = \{S_1, S_2, ..., S_n\}$. Let us simplify the state of a system to be one of two possible states: i) the system is working and ii) the system has failed. Since the subsystems are also systems, they share this characteristic—they either work or they don't. This is what we call a *boolean* value. True or false, 1 or 0. Let us call the state of a system $X \in \{\text{Working, Failed}\} = \{W, F\}$. Since there might be multiple working and failing states, instead of simply using W and F to represent the state of a system, we use Ω_W and Ω_F to denote the

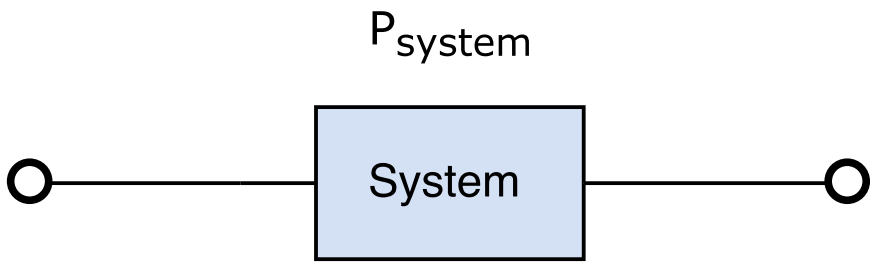


Figure 2.4: Reliability Block Diagram for General Systems

set of all working and failing states, respectively.

How the system is designed, impacts the availability and reliability of the system as a whole (even though the availabilities and reliabilities of the subsystems are the same).[EHHP13, p. 208-215]

The structure function

$$\Phi(\underline{X}) = \begin{cases} \text{True}, & \text{if } \underline{X} \in \Omega_W \\ \text{False}, & \text{if } \underline{X} \in \Omega_F \end{cases} \tag{2.1}$$

In the same way a driver of a car does not care whether it is the X or the X that causes the car not to start, the user of a system in general should not need to care for the "why" when it comes to the dependability of a system. The user should simply care whether the system (i) works or (ii) doesn't work. See Figure 2.4

Series Systems The structure function of a series system is given as

$$\Phi_{series}(\underline{X}) = X_1 \cdot X_2 \cdot ... \cdot X_n = \prod_{i=1}^{n} X_i \tag{2.2}$$

The asymptotic availability of a series system is given as

$$A_{series} = \prod_{i=1}^{n} A_i \tag{2.3}$$

The reliability of a series system is given as

$$R_{series}(t) = \prod_{i=1}^{n} R_i(t) \tag{2.4}$$

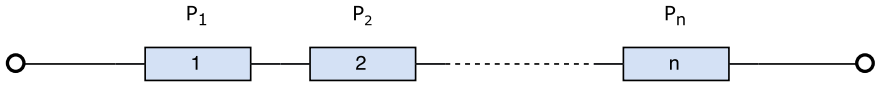


Figure 2.5: Reliability Block Diagram for Series Systems

Parallel Systems The structure function of a parallel system is given as

$$\Phi_{parallel}(\underline{X}) = X_1 + X_2 + ... + X_n = \sum_{i=1}^{n} X_i \tag{2.5}$$

The asymptotic availability of a parallel system is given as

$$A_{parallel} = 1 - \prod_{i=1}^{n}(1 - A_i) \tag{2.6}$$

The reliability of a parallel system is given as

$$R_{series}(t) = 1 - \prod_{i=1}^{n}\Big(1 - R_i(t)\Big) \tag{2.7}$$

From Equations 2.4 and 2.7 and Figures 2.5 and 2.6, we can see that the probability of a system working is higher if we have components in parallell, than if they are placed in series.

The Akka framework, the Actor model and dependability are all nice things, but why is it interesting to apply them to a system that ports data from IoT devices? In order to understand this, let us look at how the Dataport is composed today.

2.3.1 System Times

The most relevant characteristic of dependable system for this thesis is system times.

The goal when dealing with failures is of course to reduce the down time of the system as much as possible. In Figure 2.7 I have tried to map the events in a failure process to the more general domain of medical treatment in order to shed some light on what we want to accomplish when handling failures in systems.

Let us look at the case of falling down a staircase. In the examples, the person represents a component in the system, and the hospital represents the supervisor of the system, responsible for keeping the system in a working state. If the person was lucky, he might just get some bruises, but nothing more. In this case he won't

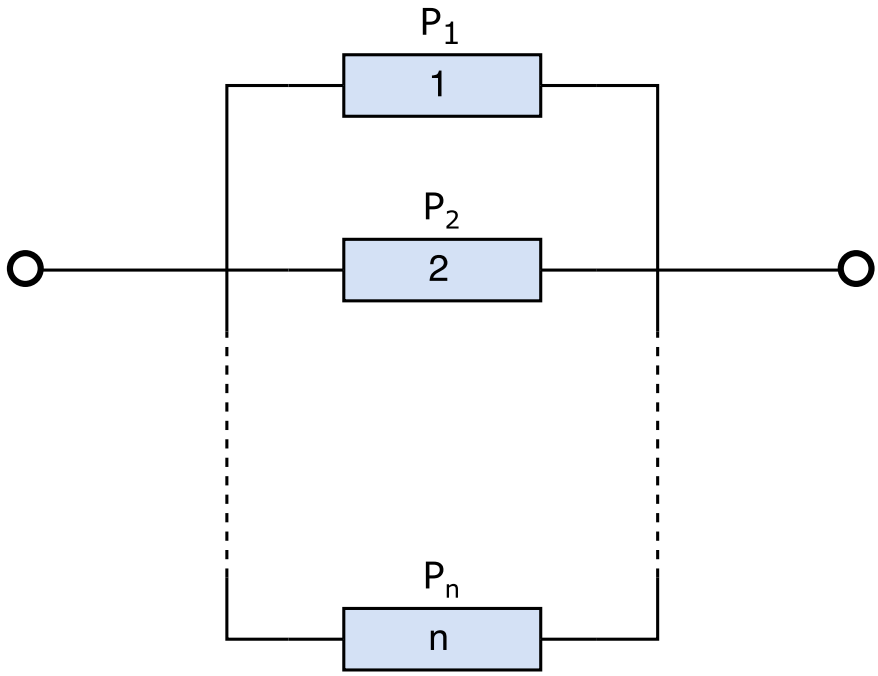


Figure 2.6: Reliability Block Diagram for Parallel Systems

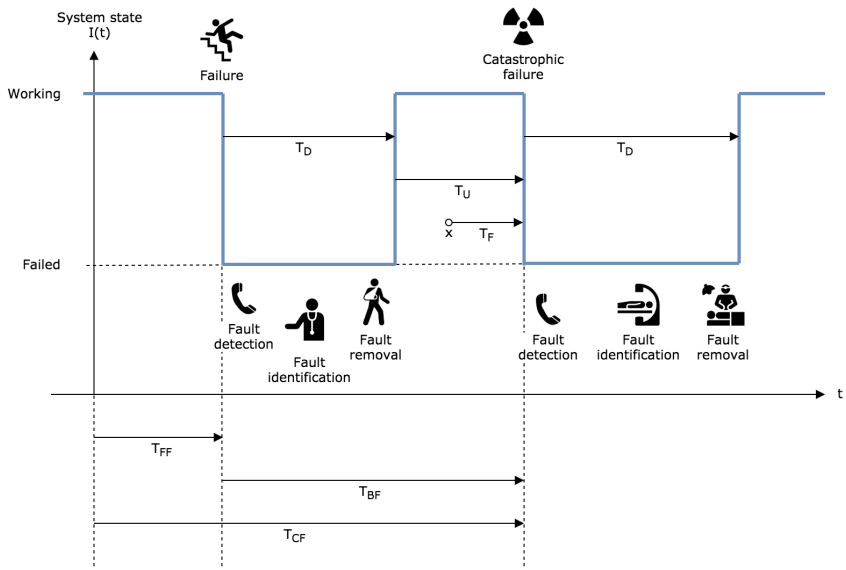


Figure 2.7: Illustration of System Times. Adapted from Figure 1.13 in [EHHP13, p. 28].

notify anyone and keep functioning as normal. This would represent self-healing in a system—the fault is removed without users of the system being aware it was present.

If the person was a bit more unlucky, he might have broken his arm. Unless he was knocked out, he would be able to feel the pain in his arm and call the ambulance. The person is now unable to function properly, since his arm is broken. Then a doctor would identify the cause of the failure to be a broken arm—this is the fault—and hopefully be able to remove the fault by plastering the arm. In this case, identification was fairly simple and quick.

If the person is extremely unlucky, he might hurt his head when he falls down the stairs. Perhaps he is knocked unconscious, and the incident is not detected before his wife comes home and calls the hospital. At the hospital the might have to do a number of tests to figure out what is wrong, and operate in order to heal the patient. This would represent a catastrophic failure in a system, where it is hard to identify the fault causing the failure. In Section 6.8 we look at some unplanned failures that happened. Fixing some of these involved reading through long logs on the server and having to recreating the failure locally.

Notice that the person actively notifies about the failure. As stated in the introduction, we are looking at how we can detect failures *without extra instrumentation*. The equivalent of the person having a phone in our system would be to give all components in the network some piece software that could issue specific commands to let the supervisor know something was wrong.

2.4 Design Science

The process used to develop the system in this thesis has followed a research paradigm called design science, conceptualized by Herbert Simon in [Sim96] and further developed by Hevner et. al. in [HMPR04], [Hev07] and [HC10], among others. The goal of design science is to improve the connection between developed IT artifacts that lives in the application domain and the knowledge base, that consists of e.g. scientific theories. Instead of just solving the problem, design science wants the problem solver to also justify *why* the solution worked, and how this can be used to help solving similar problems in the future. Or as Hevner puts it in [Hev07, p. 91]:

However, practical utility alone does not define good design science research. It is the synergy between relevance and rigor and the contributions along both the relevance cycle and the rigor cycle that define good design science research

This brings us to the three cycles involved in design science research. Figure 2.8 shows how these relates to the environment, the process of conducting the research and the knowledge base. Requirements are taken from the real world and used

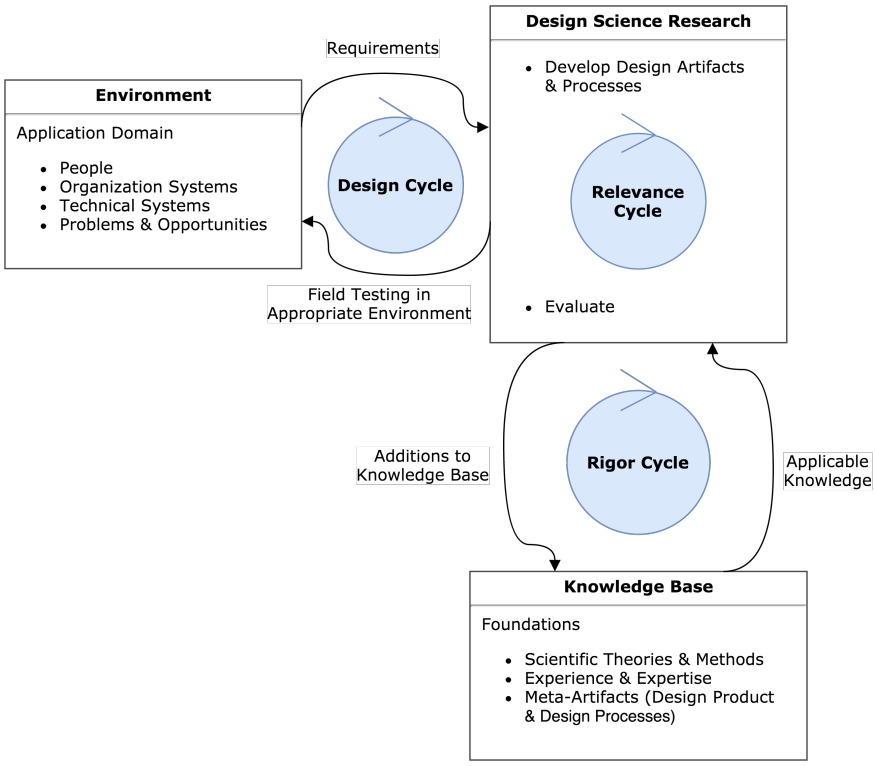


Figure 2.8: Design Science Research Cycles. Adapted from Figure 1 in [Hev07, p. 88] and Figure 2 in [HMPR04, p. 80]

as basis for what should be developed in the design process. The relevance cycle evaluates the developed artifact as it is being improved through an iterative design process. The knowledge base is used to make design decisions during the development of the artifact. In the end, the artifact and the process is documented and added to the knowledge base.

In [HMPR04]—revisited in [HC10, p. 12]—Hevner gives a detailed outline of how design science should be conducted through 7 guidelines. These guidelines are presented in Table 2.1.

[HC10, p. 20] also proposes a checklist for design science research:

1. What is the research question (design requirements)?
2. What is the artifact? How is the artifact represented?
3. What design processes (search heuristics) will be used to build the artifact?

1	**Design as an Artifact**	Design science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation
2	**Problem relevance**	The objective of design science research is to develop technology-based solutions to important and relevant business problems
3	**Design evaluation**	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods
4	**Research contributions**	Effective design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies
5	**Research rigor**	Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact
6	**Design as a search process**	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment
7	**Communication of research**	Design science research must be presented effectively to both technology-oriented and management-oriented audiences

Table 2.1: Design Science Research Guidelines [HC10, p. 12]

4. How are the artifact and the design processes grounded by the knowledge base? What, if any, theories support the artifact design and the design process?

5. What evaluations are performed during the internal design cycles? What design improvements are identified during each design cycle?

6. How is the artifact introduced into the application environment and how is it field tested? What metrics are used to demonstrate artifact utility and improvement over previous artifacts?

7. What new knowledge is added to the knowledge base and in what form (e.g., peer-reviewed literature, meta-artifacts, new theory, new method)?

8. Has the research question been satisfactorily addressed?

Each question in the 8-point list is mapped to one of the three research cycles presented in Figure 2.9. See Figure 2.9. We will revisit this list in the final Chapter of the thesis.

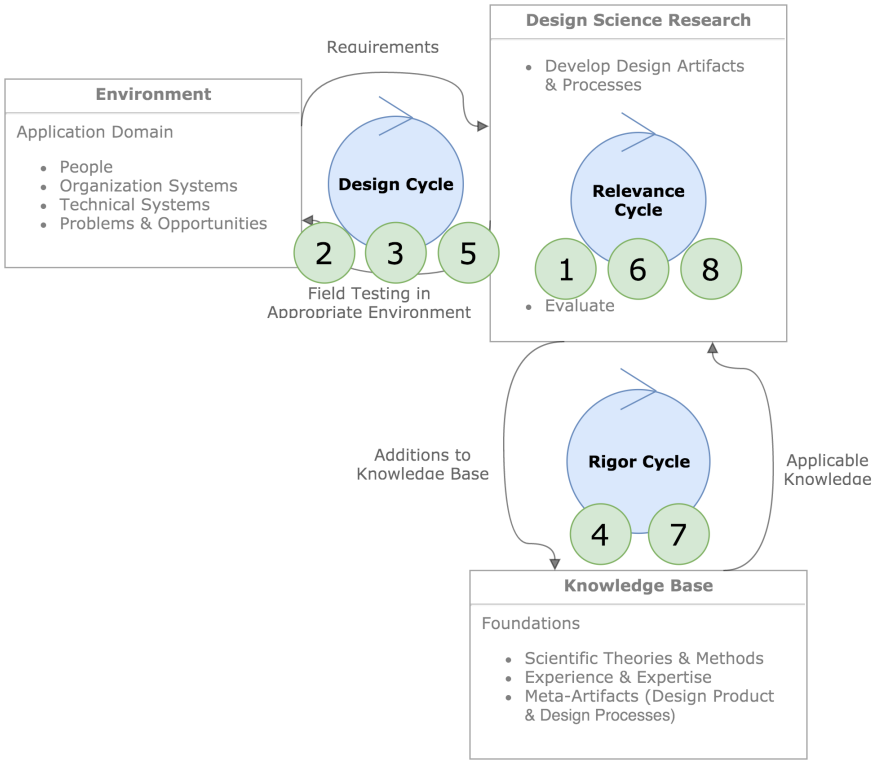


Figure 2.9: Mapping of Design Science Checklist to Research Cycles. Adapted from Figure 2.3 in [HC10, p. 20]

2.5 Related Work

In [AHS07], Aberer et. al. focus on the challenge of connecting heterogeneous sensor networks. It proposes a middleware called GSN (Global Sensor Networks) that will ease integration with existing sensor networks and adding new ones. This is achieved through the an abstraction for all components that produce data—a "virtual sensor". This virtual sensor can have any number of input data streams and it can process this input data locally. The key thing is that it will produce exactly one output stream. The format of this stream, along with e.g. identification of the sensor, is specified in the virtual sensor definition, allowing other sources to query the sensor for data. The focus is not on dependability of the system or monitoring of the sensor networks, but the concept of a virtual sensor is very similar to our digital twin.

Blackstock and Lea have done a lot of research on the domain of IoT hubs, i.e. creating platforms to ease integration with the “things” producing data. They have published a number of papers addressing this issue [BL12], [BL13], [BL14b], [BL14a],

[LB14]. In the latter of these, they introduce a cloud based IoT platform for Smart Cities. The focus is on the data and how access to it be made easier, not so much on monitoring of the IoT networks producing the data. Today, Lea and Blackstock run the company sensetecnic[2], providing a cloud hosted Node-RED service to ease development of IoT applications.

Most of the major companies on the technology scene offer IoT services. Even though they do not necessarily provide unbiased research, it is still valuable to examine how they solve problems. The monitoring approach in Google's Internet of Things Solutions is to explicitly send metrics from the devices to a service in the cloud [Goo16]. This is done either by installing a so-called *agent* on the device, or by sending data to their monitoring API manually. I.e., extra instrumentation is needed on the devices that needs monitoring. This is probably because the focus is on developing new systems, and one therefore has the possibility to add the instrumentation.

Without jumping into the design of the system, described in Chapter 5, it is nice to see the wide use of the same concepts we have chosen to use in order to represent physical objects digitally. In [Ama16, p. 177], Amazon use the term *thing shadow*—a document that stores information about the thing's state. Similarly, Microsoft's Azure IoT Hub use the term *device twin* for describing the software entity that represents the physical object [dom16]. The electronics company Bosch is also entering the IoT market, and they also use the same pattern for representing devices in software [Den16]. They use the term *digital twin*, which they share with GE [Gar16], who use the pattern to design and test virtual objects in order to achieve the wanted performance before beginning production of physical objects.

[2]http://sensetecnic.com/

3

The Watchdog

As a first step to get a better overview of the network, we need something that tells us when a component in our network isn't working as it should. We will call this *The Watchdog*, as it serves much of the same function as *watchdog timers* do in computer systems. A watchdog timer makes sure a process is running properly by counting down towards zero and restart the process if the counter reaches zero. To avoid being restarted, the monitored process will check in with the watchdog regularly and say "Hello, I'm working fine—no need to worry." This "heartbeat" will reset the timer and let the process live on. [SA00, p. 2]

The first iteration of design will implement a system with watchdog-like properties. For components that are expected to publish data regularly, it will use timers to know when they have not given life signs in expected time. Other components are of a different nature. Some components held a connection to an external resource. These will not use timers, but make sure the connection is restored if it was lost.

Since we rely on the watchdog system to monitor our IoT network, we will also need a third party supervising our system. Otherwise we could find ourselves in the situation where we are not notified about any failures in the IoT network, because the watchdog system is not working properly, not because there are none.

3.1 Detection

We have two main components in the IoT network: Gateways and sensors. These both send out messages with a given frequency. The messages are available through TTNs MQTT brokers. We will use the knowledge about how often devices are

supposed to send out messages to detect anomalies in the system. This let us add value to the system without having to add any extra instrumentation at the devices or modify the code that runs on the devices.

We will achieve this by representing the *physical* devices *digitally*, and adding the instrumentation there. We call this a *digital twin*. Figures 3.1 and 3.2 shows the state diagrams of the digital twins of the gateways and sensors. We have defined three possible states for these:

1. Uninitialized,
2. Unknown and,
3. OK

The digital twins will start out as Uninitialized and stay in this state until it knows it is subscribing to the topic where messages received from TTN are published. This is the task of the components handling the connections to TTNs Application MQTT Broker and Gateway Status Broker. The former publishes all messages sent from the sensors, the latter publishes periodic status messages from all gateways connected to TTN. The details on the implementation of forwarding the received messages internally in the system is left to Section 5.3

When they have successfully subscribed to the topic where their respective messages will be published, they make a transition to the state Unknown. For the sensors, once an observation is made, it will jump to the OK state. If the sensor stays in the state OK without receiving Observations, a predefined timer will eventually reach zero and the sensor will go back to Unknown until another Observation is received. When an Observation is received in the OK state, the timer is reset. In case of the gateway, the timer is also reset when status messages are received. These will also trigger a transition from state Unknown to OK.

The components holding the connections to different MQTT brokers are built a bit different. Here, we are concerned with whether or not the connection exists or not. The states are there defined to be:

1. Uninitialized,
2. Connecting and
3. Connected

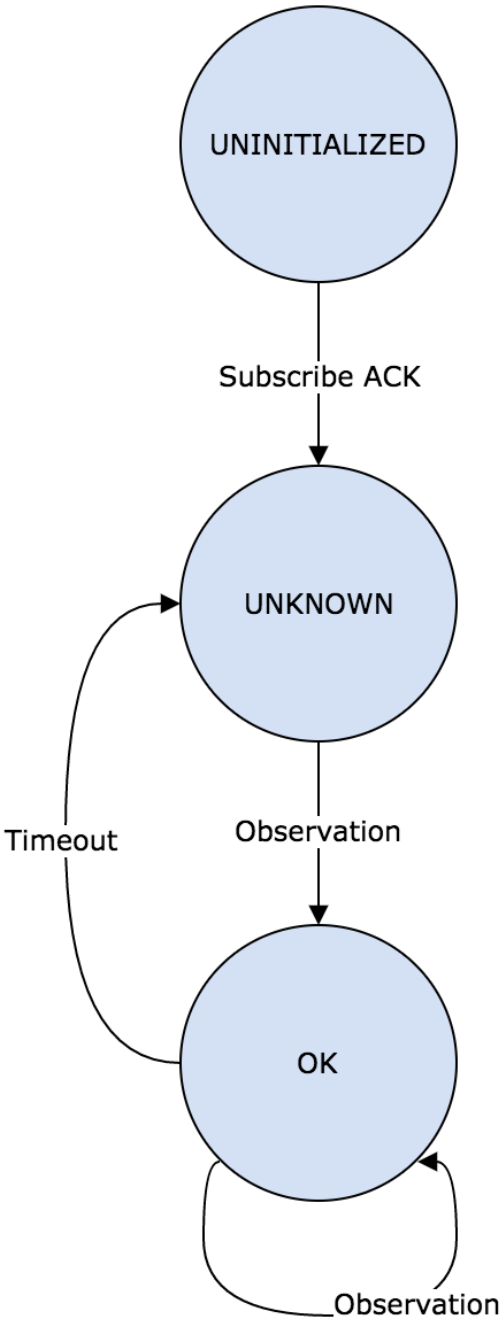


Figure 3.1: Sensor Actor State Diagram

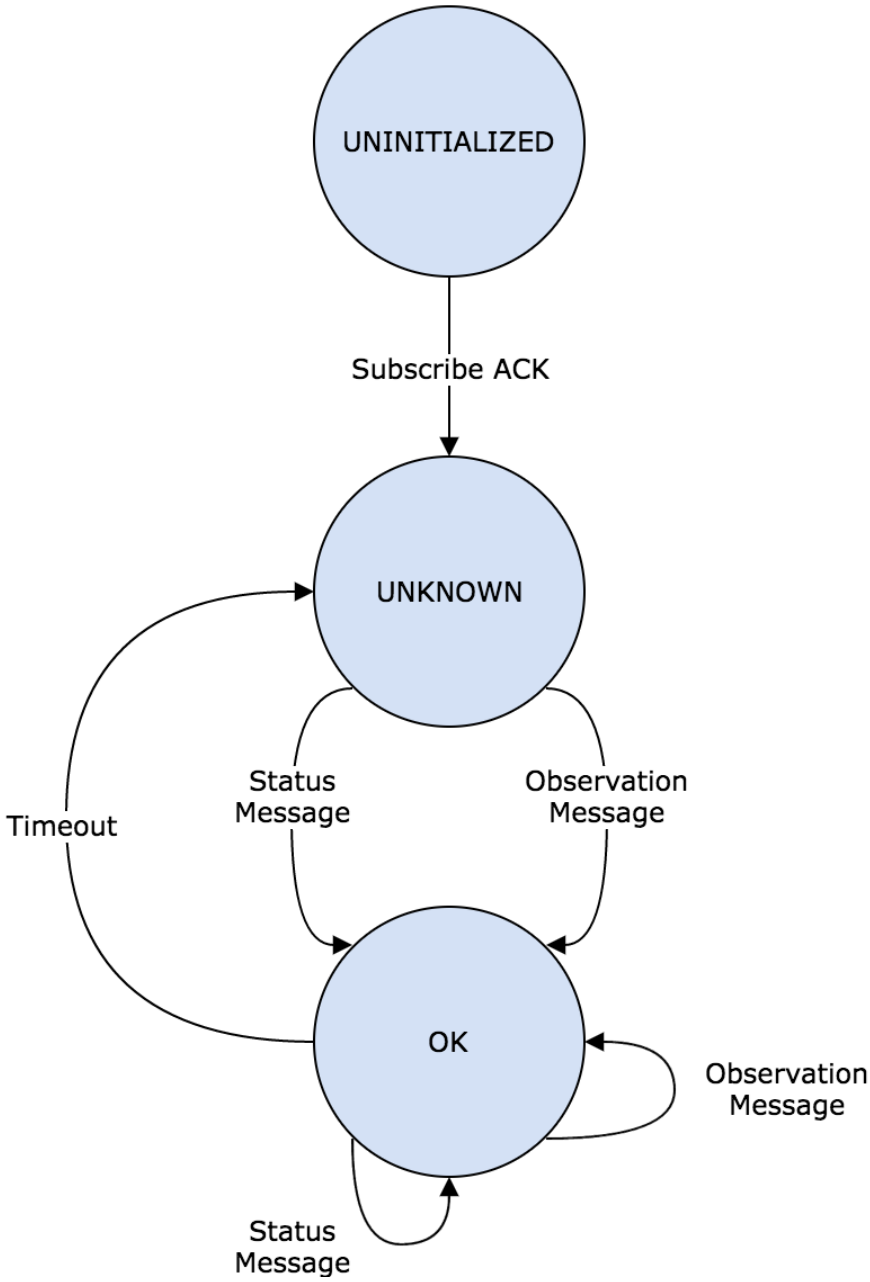


Figure 3.2: Gateway Actor State Diagram

Before the first connect is attempted, the component will stay in the state Uninitialized. When the function for connecting to the MQTT broker is called, the component will make a transition to the Connecting state. If the first connection attempt fails, the component will be restarted according to an exponential backoff algorithm. If the connection is successful, the component will go to the Connected state.

3.2 Notification

For the first design cycle, we will only notify about a few basic failures:

- Sensor timeout,
- Gateway timeout,
- Sensor has low battery,
- Lost connection to MQTT broker

The main goal of notifying about failures is to reduce the time it takes to fix the failure. Sometimes, the system can handle the failure itself, and in this case it might not be necessary to notify any humans supervisor about the failure. However, we would like to notify one time too many than one too few in the beginning, and then refine the system to not shouting "wolf, wolf" too often in future design cycles.

We have chosen to use Slack as the notification medium. This could also have been more traditional mediums, such as SMS or e-mail. Slack is very flexible, in that we create a *channel* where all notifications are posted. Workers that are on duty will receive push notifications on their phones, while worker off duty can mute the channel tot avoid receiving messages. Figure 3.4 shows the message sent for notifying about low battery level at a sensor. In Chapter 6 the result of different failures are presented, including sensor and gateway timeouts, and connection issues with MQTT brokers.

3.3 Manual Monitoring

During the spring of 2016, I developed a map [Han16][1] for The Department of Telematics (ITEM) as part of the development of the Dataport prototype. The map is meant to give the opportunity to get an instant overview of the state of the network, but also to do manual monitoring to see if messages are sent as expected. It shows the location of components in the IoT network, their status and real-time messages being sent from sensors to gateways.

[1] The map was featured in a case study of applications using sensors from Libelium [Lib16].

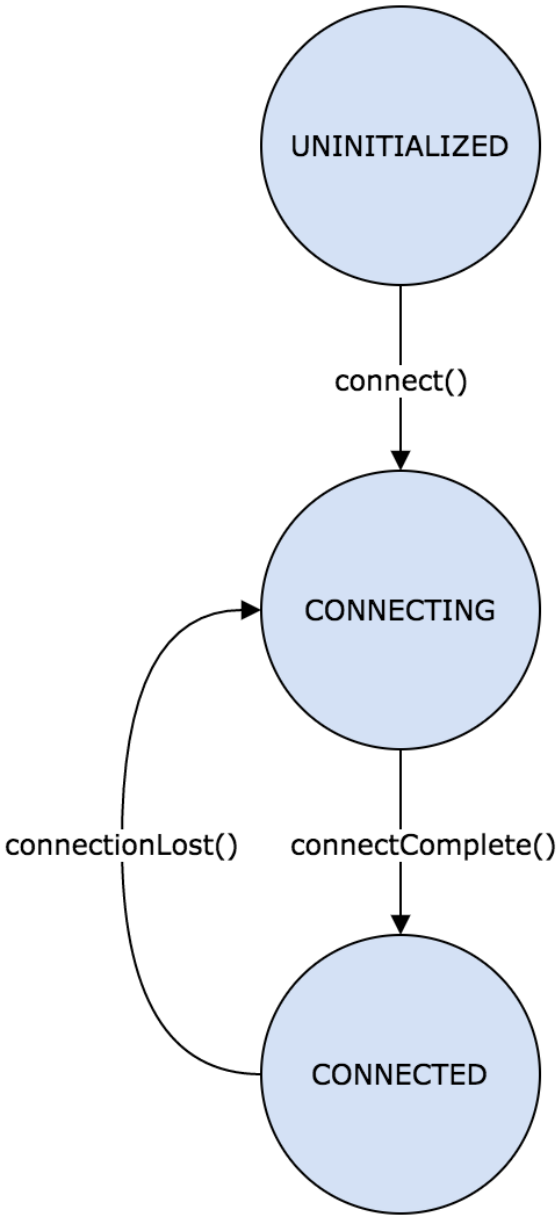


Figure 3.3: MQTT Actor State Diagram

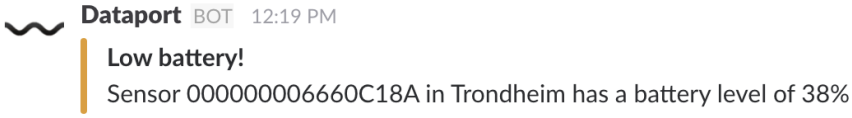
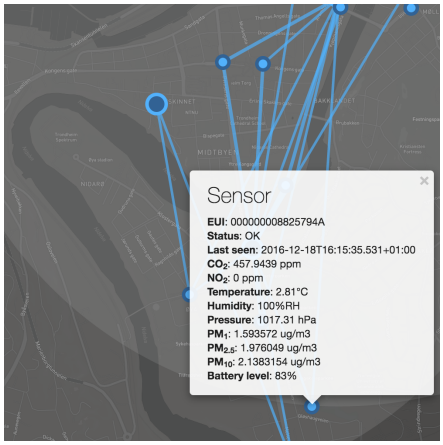


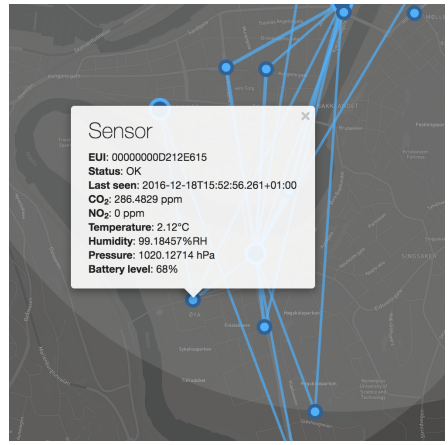
Figure 3.4: Slack Notification on Low Battery

When the Dataport prototype was replaced by the Akka implementation described in Chapter 5, some changes were needed in the map as well. These changes have been made by the author as part of this thesis. Additions include making the map handle sensors sending different measurements and showing this nicely. Figure 3.5a shows a sensors measuring PM, Figure 3.5b a sensor without these measurements and Figure 3.5c shows information about the transmission between the sensor and the gateway.

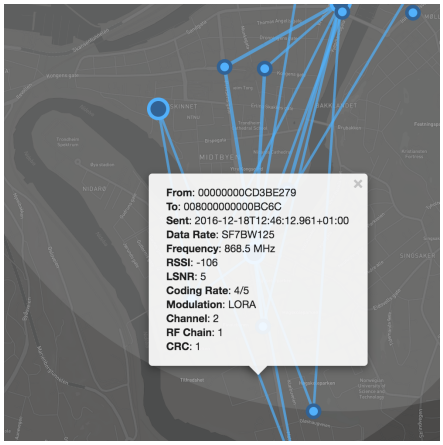
The messages are received by subscribing to the Dataport MQTT broker, described in Section 5.6.



(a) Sensor With PM Measurements



(b) Sensor Without PM Measurements



(c) Transmission Data

(d) Network State

Figure 3.5: Map Showing Network State and Data

4

The Forecaster

In the previous chapter we saw how—without any extra instrumentation in the network—the time until failures were discovered could be reduced by representing the components of the network with digital twins and use timeouts. This is all well and good: We are notified when something has gone wrong, and we can fix. But wouldn't if we didn't have to fix it? If we could somehow predict that a component is likely to fail at some point in the future, and take precautionary steps to prevent this from happening?

Since we are dealing with battery driven sensors in our network, one important concern is to make sure the sensors don't run out of battery. And given that it might take some time to replace the battery, it would be nice if we were able to discover this before the battery level is too low. In the previous chapter we saw that the system could notify us about low battery levels. In this chapter we touch on the surface of data analysis to see how the network can be made smarter without extra instrumentation. This way we can predict likely failures. Data analysis is a comprehensive field, and doing advanced analysis is beyond the scope of this thesis. A data collector is introduced to get a manual overview of trends in the measured data. In addition, data from external data sources is collected to allow trends in measured data to be compared to other types of data in the search for correlations.

4.1 Data Collection

Chapter 3 describes a system that lets data flow through itself and use e.g. timers on the frequency of the incoming data to say something about the state of the

component producing the data. This means we already have data flowing in our system, we just need to collect it somewhere. In order to collect it though, we need some new components are introduced into the system

- A database for storing the data,
- A component in the system that is responsible for pushing data to the database,
- Components for collecting data from other sources than our IoT network.

The data measured by the sensors is published by the sensor's digital twin to an internal topic, available for any component in the system to subscribe to.

Being deployed outside, the IoT network is susceptible to weather changes. Looking at how weather can affect radio transmission and performance of electrical is perhaps especially interesting in rough climates such as the Norwegian climate. Research indicate that there is an effect from clouds on the effectiveness of solar panels [Shu14] and that Received Signal Strength Indicator (RSSI) is effected by environmental conditions like temperature, fog, rain [BBH+10]. We will therefore collect data from the Norwegian Meteorological Institute through their open APIs[1]. To begin with we collect weather forecasts, UV forecasts and sunrise/sunset data.

4.2 Data Visualisation

We can visualise the data and compare it in order to predict possible future failures in the network. For example, by comparing the battery levels and the duration of sunlight, we can see a the likely connection between the two. Figure 4.1 shows the battery level of the sensors in Trondheim against the (normalized) duration of sunlight (almost linear line). If we take the average of the battery level, this becomes clearer. See Figure 4.2. Given the trends we can predict that the sensors will have a hard time being charged during the winter, unless steps are taken to deal with this.

Another way to use the forecast data, is to verify our data. The forecast data is produced with instruments many times as accurate as the sensors being used in our network. However, from Figure 4.3, we see that if we take the average of all sensors in Trondheim, the measured temperature aligns pretty well with the forecast data from the Norwegian Meteorological Institute. The blue line represents our networks measurements, the green is the forecasted temperature.

We also have the possibility to create dashboards that automatically refreshes for manual monitoring. See Figure 4.4

[1] api.met.no

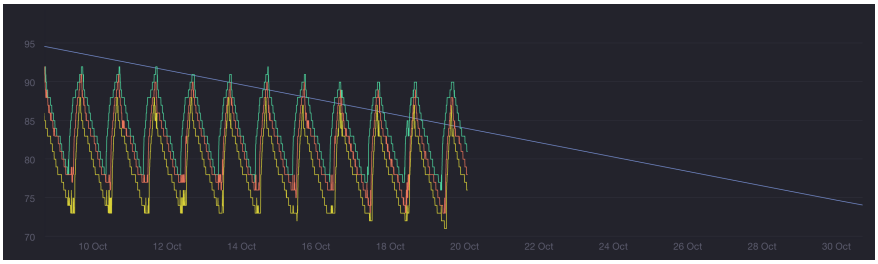


Figure 4.1: Battery Level Against Sunlight Duration

Figure 4.2: Mean of Battery Level Against Sunlight Duration

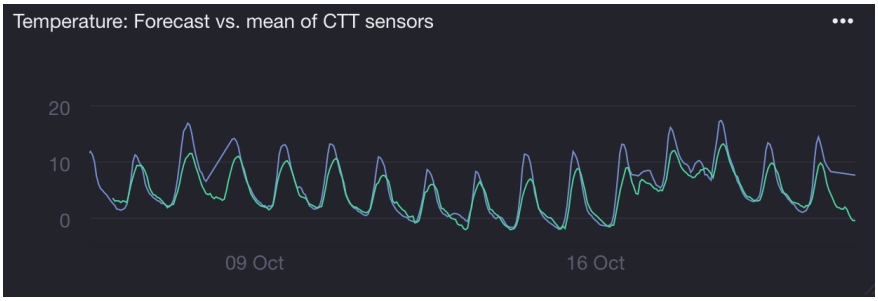


Figure 4.3: Mean of Measured Temperature Against Forecasted Temperature

CTT has deployed IoT networks in Trondheim, Norway and Vejle, Denmark. Both are being monitored by the developed system, and hence we also collect weather data for both cities.

Figure 4.4: Data Visualisation Dashboard

5

Dataport Design

Based on the features described in Chapters 3 and 4, we will in this chapter go through the design and implementation of the features and shed some light on why the implementation is robust? The project is open source and can be found at GitHub, see Figure 5.1.

Figure 5.1: Source Code at https://github.com/NTNU-ITEM/dataport-akka

5.1 Akka

Akka describes itself as a framework that is resilient through self-healing [Lig16, p. 1]. A *resilient* system is a system that can keep operating even when faults exist, i.e. it is fault-tolerant. If we consider a football team as a system, we can say that a fault arises when a player is injured. Usually, the team will keep playing, i.e. the system keeps operating even in the presence of a fault. The fault can be removed either by giving the player some medical treatment or by substituting the player for a new, fit player. This is called fault-removal, and is essential for the Akka framework.

Akka is built on the actor model, briefly described in Section 2.2. This makes applications made with Akka inherently dependable and scalable. It is used in application ranging from analysis of stock trends, telecommunication systems providing 99.99999% uptime to 3d simulation engines.

The power of Akka is that it allows you to write code that is designed to run locally, but can be deploy as distributed in the cloud without code change. From [Lig16] we have the following quote:

> The key for enabling this is to go from remote to local by way of optimization instead of trying to go from local to remote by way of generalization.

Meaning, we need to write generalized code from the beginning, which we are somewhat forced to do through the actor model.

Figure 5.2 shows the actors at the top of the hierarchy in an Akka system. At the very top we have the root guardian. This has two children: a system guardian and a user guardian. Everything we create will reside under the user guardian.

When creating our system, we want components to be as independent as possible. If something fails in one place, this should only notify the parts that needs to know that it has failed, and leave the rest of the system unaffected. If an exception in *one* actor makes the *whole* system crash, we have failed miserably in creating a robust system.

We keep this in mind when choosing a supervision strategy, i.e. how supervisors are going to handle failures in their children. Even though all actors can act as supervisors, we choose the natural hierarchical approach where parents supervise their children. The Akka framework provides two supervising strategies: (i) the One-For-One Strategy and (ii) the All-For-One Strategy. This means we can either apply an action to the child that failed, and this child only, or we can apply the

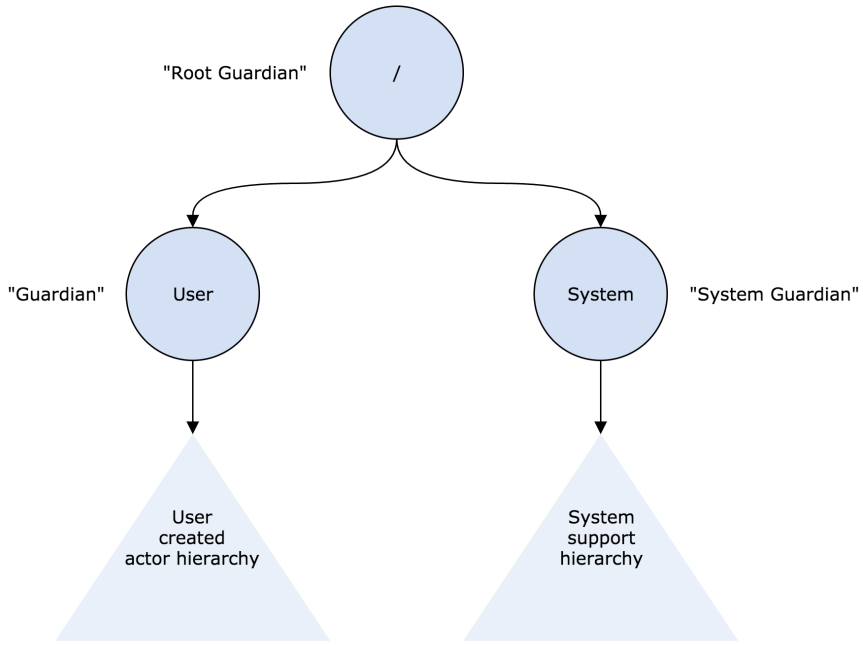


Figure 5.2: Actor System Architecture in Akka. Figure adopted from [Lig16].

same action to *all* children if one child fails. We have chosen to use a One-For-One Strategy since sensors might cause failures in their digital twins independently.

A supervisor has four choices when dealing with a failing child

1. Keep the child running, keeping the internal state,
2. Restart the child, giving it a fresh start (clean state),
3. Permanently stop the child or,
4. Escalate the failure by failing yourself

5.1.1 Design Choices When Using External Libraries

For the MqttActor we consider three design approaches, each with increasing abstraction of fault handling:

1. Let the MQTT framework used (Paho) handle all MQTT related faults
2. Let the actor implement the fault handling of MQTT related faults
3. Let the supervisor of the actor handle all faults.

Option 1 Since we are using the Paho Java client library to create and maintain the MQTT connection, we have the opportunity to use Paho's built in fault handling, e.g. one can specify that the client should automatically try to reconnect when the connection is lost using a back-off algorithm. The advantage of this approach is that we do not need to write the implementation of the fault handling ourselves. The disadvantage is that we have little information on the state of the connection. We know whether we are connected or not, but we do not know at which stage of a "trying to reconnect" session the client is. Exposure of desired information through the Paho MQTT Client API could however be implemented, since Paho is open source (https://github.com/eclipse/paho.mqtt.java). Other MQTT clients for Java exists[1], but we consider Paho to be the most comprehensive and well-maintained.

Option 2 To have more control over this, we could give the MQTT framework as little possibility to handle faults as possible, and try to handle as many errors as possible inside our actor. This would involve a number of try/catch statements to handle exceptions that might be thrown by the MQTT client and using some timer/scheduler to know when we can try to reconnect again if the connection is lost. This way, we have more information about the state of the MQTT connection that we can expose (if we want) to other parts of the system.

Option 3 The first two approaches is usually how fault handling is done in object oriented programming; either use an external library that handles everything (and don't care that we have little information about why something might not work) or implement your own handling where you use the external library. The third approach is more in line with Akka best practices. With this approach, any time an exception occurs, we simply throw it to let our supervisor know what went wrong. The supervisor will implement different strategies to handle different exceptions and will have complete knowledge of why its child might not act as it should, i.e. once an error occurs in the child, the supervisor is immediately notified, instead of the child trying to solve the problem itself first and then notifying its supervisor if it was unable to handle it on its own.

5.1.2 External Resource Failures vs. Gateway/Sensor Failures

The actors that communicate with external resources do not hold any data, they only relay data from outside the system to other actors within the system. We can therefore be "rough" when dealing with failures in these, i.e. clear out its accumulated internal state every time it fails.

For the actors representing gateways and sensors however, we will often have some data we would like to keep, e.g. what the latest measurement was, when it was

[1] https://github.com/mqtt/mqtt.github.io/wiki/libraries

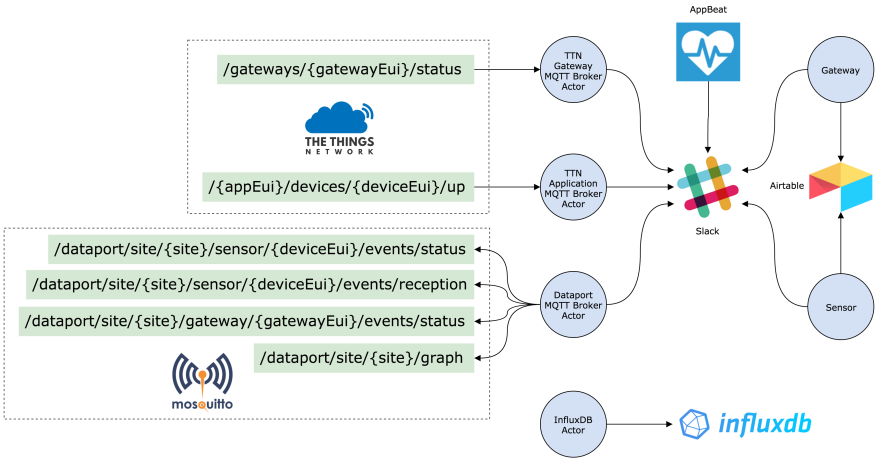


Figure 5.3: Dataport Interaction with External Services

last seen and where it was last seen (in case of mobile devices). Hence, we might want to be more careful when dealing with failures in these actors. Preferably we want to resume the failing actor and keep its internal state whenever possible.

5.2 System Overview

Figure 5.3 shows how the system interacts with external resources. Figure 5.4 shows the supervision hierarchy of the actors.

5.3 Publish-Subscribe Topic Structure

The system's publish-subscribe topic structure is presented in Figure 5.5. This also shows how different actors in the system interact with the different topics.

5.4 The Site Actor

The site actor is responsible for creating all sensors and gateways belonging to a site on start up. It retrieves a list of devices from Airtable. Figure 5.6 shows the list of devices with their respective location and timeout limits.

Figure 5.7 shows an overview of the site actor's children. Our system monitors two applications: one in Trondheim and one in Vejle.

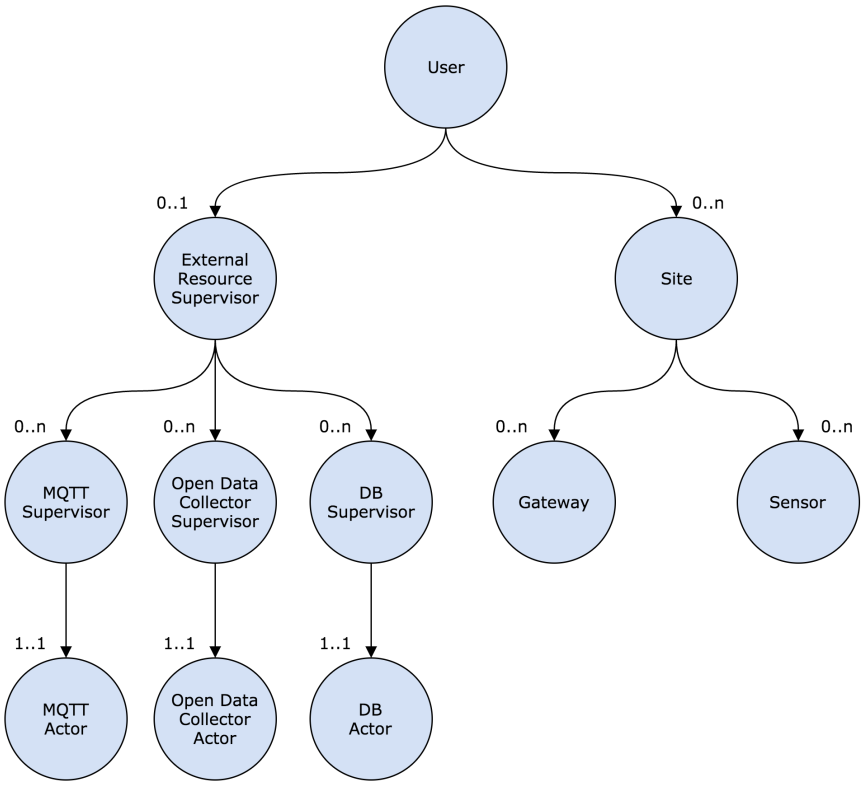


Figure 5.4: Dataport supervision overview

The site actor is also responsible for publishing a network graph message periodically to the `/dataport/site/graphs` topic. This allows other actors to know the state of the network.

See the SiteActor class for the full implementation.

5.5 The Database Actor

The Database Actor was added as part of the Forecast iteration. It maintains a connection to an InfluxDB database that runs on a remote machine. On start up it will subscribe to the `/dataport/site/graphs` topic in order to know which sensors exists. Based on this, it will subscribe to all topics for reception messages. Whenever an observation is published, the actor sends this to the database.

It will also subscribe to the forecast topics. Regarding forecast data, it will receive many points at the same time. To handle this, we use a built in batching function to avoid clogging up the connection to the database.

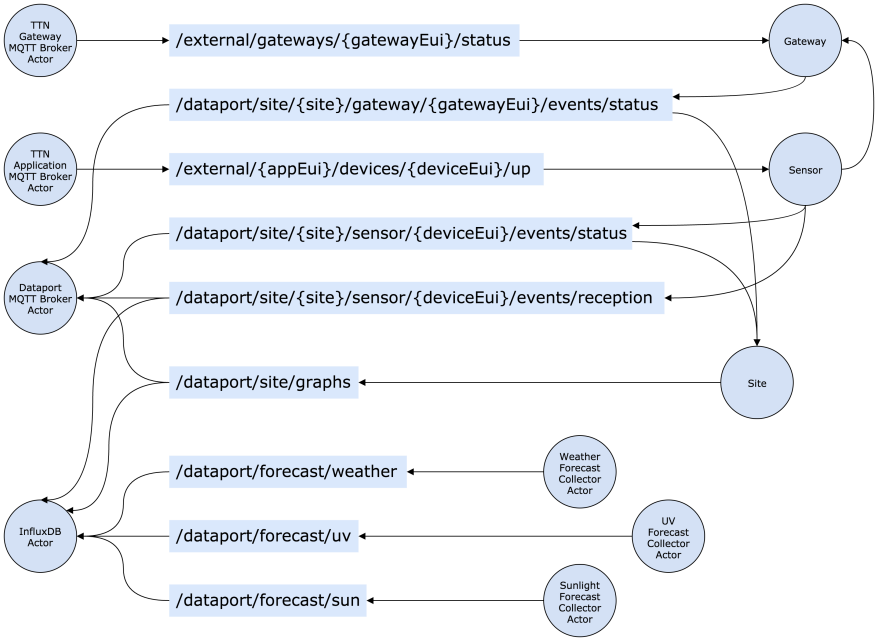


Figure 5.5: Dataport Internal Topic Structure

vejle | trondheim | +

Main View | Hide fields | Filter | Sort

	eui	type	latitude	longitude	timeout	status
1	AA555A0008060353	Gateway	63.43349	10.40365	100	OK
2	AA555A0008060252	Gateway	63.42251	10.39514	100	OK
3	008000000000BC6C	Gateway	63.42883	10.38570	100	OK
4	00000000AD6AA33E	Sensor	63.43739	10.41506	1800	
5	0000000048524DD8	Sensor	63.43587	10.40003	1800	OK
6	00000000E935A419	Sensor	63.43066	10.39223	1800	OK
7	000000007CA37D4E	Sensor	63.43061	10.39618	1800	OK
8	000000009981CAA5	Sensor	63.41553	10.40092	1800	OK
9	00000000CD3BE279	Sensor	63.41280	10.39921	1800	OK
10	00000000031F5B033	Sensor	63.40357	10.41104	1800	

Figure 5.6: Airtable Device List

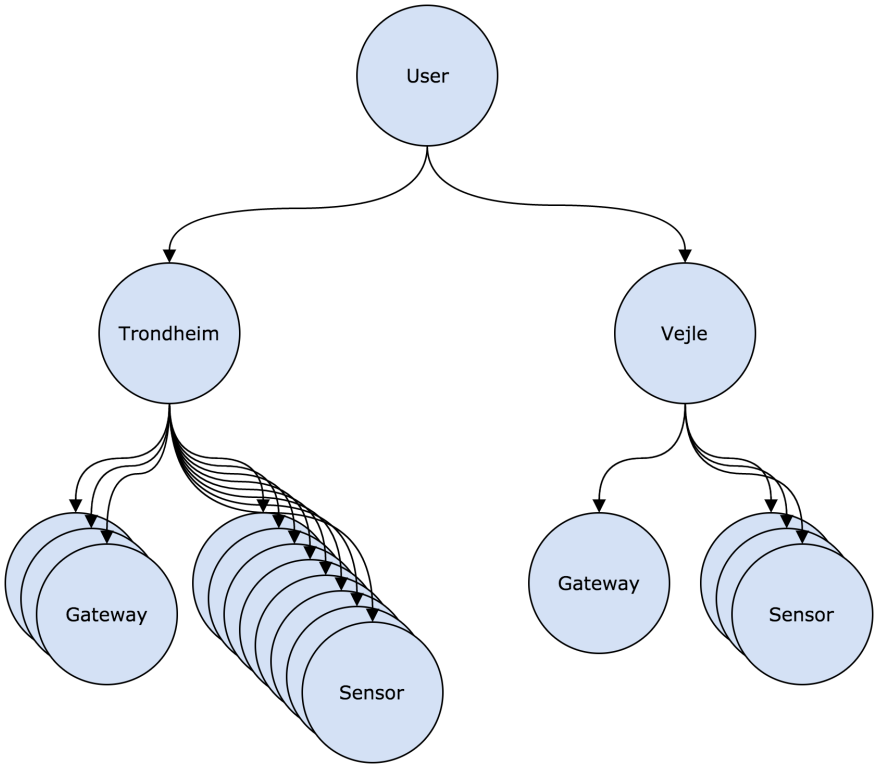


Figure 5.7: Dataport Site Overview

See the DBActor class for the full implementation.

5.6 The MQTT Actors

We have two MQTT brokers that we receive data from, and one MQTT broker that we publish data to. The actors the communicate with these brokers are very similar.

In an early design version, these actors were sent the list of devices they should subscribe to only on start up. The trouble with this was that if they were restarted because of a lost connection or some other failure, they did not know which devices they should subscribe to. This was solved by having them subscribe to the `/dataport/site/graphs` topic, that periodically publishes which devices exits in the network.

See the ApplicationMqttActor, GatewayStatusActor and PublishingMqttActor classes for the full implementation.

5.7 The Forecast Actors

We collect data for three different forecasts: weather, UV and sunlight. This is done in three separate actors to minimize the risk of an exception in one affecting others. There is always a risk of failure when parsing data, as discussed in Section 6.8.1.

Data is retrieved periodically. If the endpoint at which data is retrieved from is unavailable, the actor will simply try again later. Since data is retrieved relatively often, this will not affect the data set being stored.

See the SunForecastActor, UVForecastActor and WeatherForecastActor classes for the full implementation.

5.8 The Sensor Actor

The sensor actor is responsible representing the state of its physical twin. It will publish observations and status messages on the topics as showed in Figure 5.5. It will also interact with Slack if a timeout occurs. The code excerpt below shows the code for handling timeouts.

```
1  when(DeviceState.OK, null, // timeout duration is set in the constructor
2      matchEventEquals(StateTimeout(),
3          (event, data) -> {
4              // Update my state
5              stateData().setStatus(DeviceState.UNKNOWN);
6
7              // Send alert to Slack
8              slackAPI.call(new SlackMessage("").addAttachments(new SlackAttachment()
9                  .setFallback("Sensor timeout! Sensor " + data.getEui() +
   ↪   " in "+data.getCity() + " has been inactive for " + stateData().getTimeout())
10                 .setTitle("Sensor timeout!")
11                 .setText("Sensor " + data.getEui() + " in " + data.getCity() +
   ↪   " has been inactive for " + stateData().getTimeout())
12                 .setColor("warning")));
13
14             // Publish my status to all interested to show I timed out
15             mediator.tell(new
   ↪   DistributedPubSubMediator.Publish(internalStatusPublishTopic, stateData()),
   ↪   self());
16
17             return goTo(DeviceState.UNKNOWN).using(stateData());
18         })
19     )
20 );
```

This means if the actor is in state OK and a timeout occurs, it will send a notification to Slack.

In Figure 5.5 we can also notice that there is one connection directly between two actors, namely sensors and gateways. This is done because we want to calculate the observed maximum range of the gateways. Therefore, we make the sensor send a message containing the sensor's position directly to the gateway. If we wanted all communication to go through the distributed publish-subscribe, we could have had the Gateway actors subscribe to all `/dataport/site/{site}/sensor/{deviceEui}/up` topics for the site it resides. Unfortunately, the Akka publish-subscribe topics does not support the use of wildcards, so this would require a similar approach as for the Database actor: The Gateway actor would need to subscribe to the `/dataport/site/graphs` topic to get an overview of all sensors in the network and use this to subscribe to the topic for each sensor at its site. This approach will however become a problem if the system is run in a cluster. One of the reasons for using the publish-subscribe feature of Akka is that its

See the SensorActor class for the full implementation.

5.9 The Gateway Actor

The gateway actor subscribes to status messages from its physical twin and publish these on the internal topic.

It also calculates the distance to sensors it receives messages from by using the Haversine formula.

See the GatewayActor class for the full implementation.

5.10 AppBeat

In order to know when something is wrong with the system supervising the IoT network, we use a third party service that supervises the supervisor. AppBeat will PING the machine that the system is running on every 5 minutes from different locations in the world. If it does not get an answer in 15 secods, it will send a Slack notification. See Section 6.8.3.

5.11 Logging

To see what's been going on, we need to log. This is enabled through the use of slf4jlogger and logback. Having a well structured log can greatly reduce the time it takes to identify a failure during a debugging process. This is achieved by only logging events that are of interest. Events only needed during development should not be logged when the system is running in production. Here are a few examples.

```
1  log.debug("Got: {} from {}", message, getSender());
2
3  log.debug("MQTT Received on topic {} message: {}. Publishing on internal topic {}",
   ↪  topic, message, internalTopic);
4
5  log.info("Now subscribing to topic {}", applicationDevicesUpTopic);
6  log.error("Damn! I lost my MQTT connection. Paho's automatic reconnect with" +
   ↪  "exponential backoff kicking in because of: {}", cause.getStackTrace());
```

5.12 Scaling

By using the Distributed Publish Subscribe feature of Akka, we have already prepared the system somewhat for clustering.

```
1  akka {
2    actor {
3      provider = "akka.cluster.ClusterActorRefProvider"
4    }
5    remote {
6      netty.tcp {
7        hostname = "127.0.0.1"
8        port = 0
9      }
10   }
11   cluster {
12     seed-nodes = [
13       "akka.tcp://ClusterSystem@127.0.0.1:2551",
14       "akka.tcp://ClusterSystem@127.0.0.1:2552"
15     ]
16   }
17 }
```

6

Failure Experiments

In this chapter we test how the system described in Chapter 5 copes with various failures. For components we have control over, like the Dataport MQTT Broker and the Dataport itself, manual failures are triggered. For components we don't have control over, like TTN and the forecast API, we had to wait for failures to arise while the system was running. Several unexpected failures also occurred, where the system's handling of the failure was documented and if needed, the system was refined.

6.1 Device Timeouts

6.1.1 Gateway Status Not Received in Expected Time

Hypothesis The gateway's digital twin will timeout, go to state UNKNOWN, update the gateway's status field in the Airtable and send a Slack alert.

Method Define the timeout in the Airtable to be shorter than the sending frequency of the gateway status messages.

Figure 6.1: Slack Alert on Gateway Timeout

eui	type	latitude	longitude	timeout	status
AA555A0008060353	Gateway	63.43349	10.40365	600	OK
008000000000BC6C	Gateway	63.42883	10.38570	100	UNKNOWN
AA555A0008060252	Gateway	63.42251	10.39514	600	OK

Figure 6.2: Airtable Status Updated on Gateway Timeout

Result Figure 6.1 shows the Slack message that was generated. Figure 6.2 shows that the status field in Airtable is updated.

6.1.2 Sensor Measurement Not Received in Expected Time

Hypothesis The sensor's digital twin will timeout, go to state UNKNOWN, update the sensor's status field in the Airtable and send a Slack alert.

Method Same procedure as for gateway: Define the timeout in the Airtable to be shorter than the actual sending frequency of the sensor measurement messages in order to trigger a timeout.

Result Figure 6.3 and 6.4 show the Slack message and Airtable status update, respectively.

Figure 6.3: Slack Alert on Sensor Timeout

00000000E935A419	Sensor	63.43066	10.39223	1800	OK
000000007CA37D4E	Sensor	63.43061	10.39618	1800	UNKNOWN
000000009981CAA5	Sensor	63.41553	10.40092	1800	OK

Figure 6.4: Airtable Status Updated on Sensor Timeout

6.2 Application MQTT Broker

6.2.1 Unavailable on Startup

Hypothesis Each sensor's digital twin will stay in state UNKNOWN.

Method Define a malformed broker address.

Result The results were the sa as for the Dataport MQTT Broker

6.2.2 Becomes Unavailable

Hypothesis Paho begins its exponential backoff strategy to reconnect to the broker. A Slack notification will be sent saying the connection was lost and that it is trying to reconnect.

Method Since we have no control over TTN, we have to wait for it to happen in real life.

Result A Slack notification was sent regarding the lost connection for the broker. However, since the connection was lost for a long period of time, all sensors timed out and also sent Slack notifications. This was an unforeseen side effect. Figure 6.5 shows the Slack messages that were sent.

Since the system knows it is not connected to the source of life signs from sensors, it should also know that sensors will timeout even though they work just fine. A better implementation would be to freeze the timeout for sensors when the connection to the broker providing life signs is lost. This also allows for separation of concerns in maintenance: A person responsible for changing sensor batteries when sensors timeout shouldn't be notified that the connection to TTN is lost.

6.2.3 Becomes Available

In an early design iteration of the Dataport, the Actor responsible for the connection to TTN was only told which sensors it should listen to once: When the SiteActor for a city started. This meant that even though the supervisor of a failing MqttActor handled restarting it OK, the MqttActor was now in a state where everything seemed OK, even though that was not the case. It was connected to the external broker, but it did not subscribe to any sensor topics at the external broker, because the list of sensors it should subscribe to was only sent when the SiteActor started. To deal with this, instead of subscribing to each sensor individually, we use the wildcard `+` to subscribe to all sensors sending messages to the application. I.e., instead of subscribing to topics

```
{appEui}/devices/{deviceEui1}/up,
{appEui}/devices/{deviceEui2}/up,
[...]
{appEui}/devices/{deviceEuiN}/up}
```

we only subscribe to:

```
{appEui}/devices/+/up
```

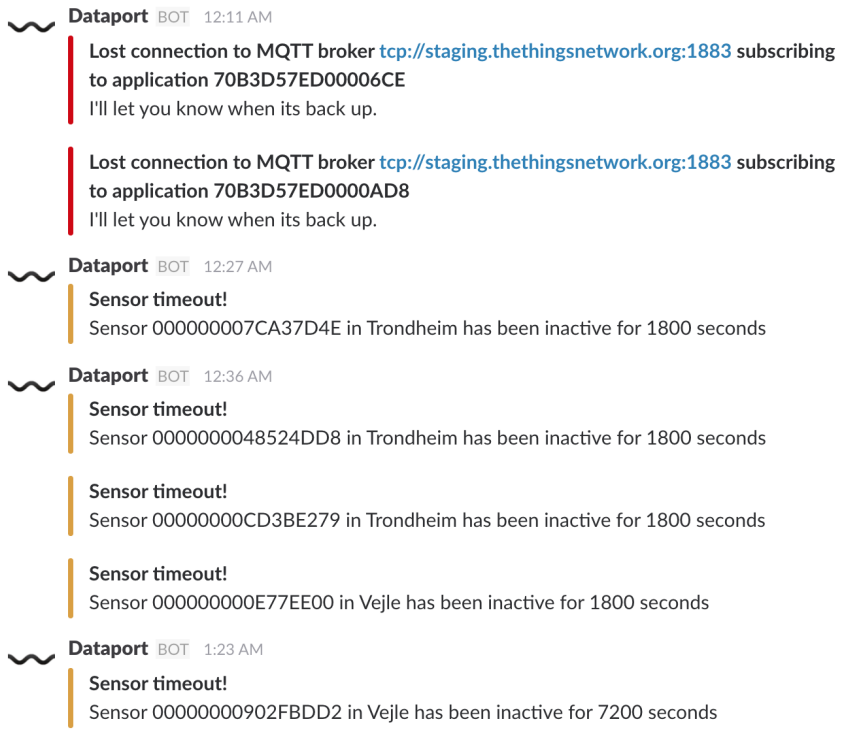


Figure 6.5: Application MQTT Broker Becomes Unavailable

Hypothesis A Slack notification is sent when the Actor is reconnected to the broker, the Actor is subscribing to the general topic and starts receiving messages from the sensors again.

Method Since we have no control over TTN, we have to wait for it to happen in real life.

Result The corresponding log to Figure 6.6 shows the order of events, the state changes of the MqttActors, a confirmation of the subscription and an example of a received message.

```
1 [INFO] [ttn-vejle-broker] Going from CONNECTED to CONNECTING
2 [INFO] [ttn-trondheim-broker] Going from CONNECTED to CONNECTING
3 [INFO] [ttn-trondheim-broker] Damn! I lost my MQTT connection. Paho automatic
  ↪  reconnect with backoff kicking in
```

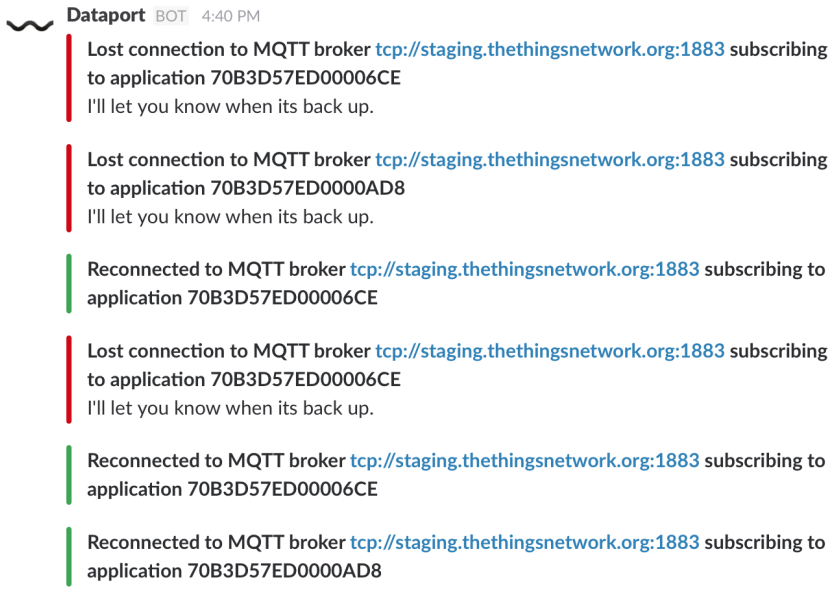


Figure 6.6: Application MQTT Broker Self Healing

```
4  [INFO] [ttn-vejle-broker] Damn! I lost my MQTT connection. Paho automatic reconnect
   ↪  with backoff kicking in
5  [INFO] [ttn-vejle-broker] Going from CONNECTING to CONNECTED
6  [INFO] [ttn-vejle-broker] Phew! I reconnected to my MQTT broker
7  [INFO] [ttn-trondheim-broker] Going from CONNECTING to CONNECTED
8  [INFO] [ttn-trondheim-broker] Phew! I reconnected to my MQTT broker
9  [INFO] [ttn-trondheim-broker] Now subscribing to topic 70B3D57ED0000AD8/devices/+/up
10 [INFO] [ttn-vejle-broker] Now subscribing to topic 70B3D57ED00006CE/devices/+/up
11 [INFO] [ttn-vejle-broker] MQTT Received on topic
   ↪  70B3D57ED00006CE/devices/000000000E77EE00/up. Publishing on internal topic
   ↪  external/70B3D57ED00006CE/devices/000000000E77EE00/up
```

6.3 Gateway Status MQTT Broker

6.3.1 Unavailable on Startup

Hypothesis Each gateway's digital twin will stay in state UNKNOWN until it receives a reception message from a sensor.

Method Define a malformed broker address.

Result All gateways stayed in state UNKNOWN until a reception message from a sensor was received. From the log excerpt we see that once gateway 0000024B080E06B3 receives a message from sensor 00000000902FBDD2 it transitions into state OK.

```
[ERROR] [2016-11-04 23:49:55] [dataportBroker] Unable to connect to server
akka.actor.ActorInitializationException: dataportBroker: exception during
[INFO] [2016-11-04 23:49:55] [Trondheim/AA555A0008060353] - Going from UNINITIALIZED
    ↪  to UNKNOWN
[INFO] [2016-11-04 23:49:55] [Trondheim/AA555A0008060252] - Going from UNINITIALIZED
    ↪  to UNKNOWN
[INFO] [2016-11-04 23:49:56] [Vejle/0000024B080E06B3] - Going from UNINITIALIZED to
    ↪  UNKNOWN
[INFO] [2016-11-04 23:49:56] [Vejle/00000000902FBDD2] - Going from UNINITIALIZED to
    ↪  UNKNOWN
[ERROR] [2016-11-04 23:50:29] [dataportBroker] Unable to connect to server
akka.actor.ActorInitializationException: dataportBroker: exception during
[INFO] [2016-11-04 23:51:28] [Vejle/00000000902FBDD2] - Going from UNKNOWN to OK
[INFO] [2016-11-04 23:51:28] [influxDBActor] - Writing to InfluxDB, observation point:
    ↪  {device_eui=00000000902FBDD2, gateway_eui=0000024B080E06B3, ...}
[INFO] [2016-11-04 23:51:28] [Vejle/0000024B080E06B3] - Going from UNKNOWN to OK
```

6.3.2 Becomes Unavailable

Hypothesis Paho begins its exponential backoff strategy to reconnect to the broker. A Slack notification will be sent saying the connection was lost and that it is trying to reconnect.

Method Since we have no control over TTN, we have to wait for it to happen in real life.

Result Figure 6.7 shows a real life example where the connection to the broker was lost for a short period of time. The Dataport handled the failure and messages from the Gateway Status Broker reappeared once the connection was reestablished.

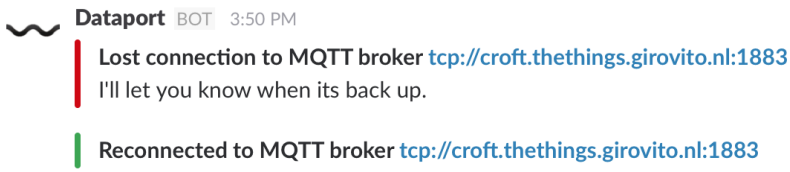


Figure 6.7: Gateway Status MQTT Broker Self Healing

6.3.3 Becomes Available

Hypothesis A Slack notification will be sent saying the broker reconnected.

Method Since we have no control over TTN, we have to wait for it to happen in real life.

Result See Figure 6.7. Gateway messages are received again.

Because we can't control TTN, we were not able to conduct an experiment where the broker was unavailable when the Dataport started and became available without becoming unavailable first. This case would *not* trigger only the last Slack notification, but we would be able to see in the logs that the Actor is being restarted according to the defined exponential backoff algorithm, in the same way as described in Section 6.4.2.

Similarly to the experiment in Section 6.2.3, an early version of the Dataport did not support recovery for the Gateway Status Actor. When restarted, it would not know which gateways it should subscribe to, as this was only given once during Dataport startup. For the Application Actor we simply used the topic wildcard in order to subscribe to all sensors connected to the application. However, the gateway status messages can only be retrieved from a global topic where *all gateways connected to TTN* send status messages. Therefore, we can't use the wildcard, as this would give us status messages from all TTN gateways. Instead, we make the Gateway MQTT Actor subscribe to the graph topic where each site publishes its network graph periodically, and subscribe to all gateways present in this.

6.4 Dataport MQTT Broker

The task of the Dataport MQTT broker is make status and reception messages from the gateways and sensors available as a stream for external sources in a nice format. This is used to update the map showing the network at dataport.item.ntnu.no in real-time. We consider three different ways a failure or recovery from failure at the broker affects the Dataport.

The experiments will be performed by manually stopping and starting the broker. This is done by issuing the following commands on the server where the broker is running:

```
1 $ sudo service mosquitto stop
2 $ sudo service mosquitto start
```

6.4.1 Unavailable on Startup

Hypothesis Monitoring of the network is unavailable through the map, but gateway and sensor status and reception messages are still logged on the server and pushed to the database. The Actor responsible for the connection to the MQTT broker will try to reconnect.

Method Stop the Dataport MQTT Broker manually before starting the Dataport.

Result As shown in the excerpt from the log below, the Actor fails to start, is attempted restarted by its supervisor and this does not affect the other parts of the system. This log excerpt only includes a few selected events.

```
[ERROR] [11/04/2016 22:28:23] [dataportBroker] Unable to connect to server
akka.actor.ActorInitializationException: dataportBroker: exception during
[ERROR] [11/04/2016 22:28:31] [dataportBroker] Unable to connect to server
akka.actor.ActorInitializationException: dataportBroker: exception during
[INFO] [11/04/2016 22:28:46] [Vejle/0000024B080E06B3] Going from UNKNOWN to OK
[ERROR] [11/04/2016 22:28:44] [dataportBroker] Unable to connect to server
[INFO] [11/04/2016 22:28:48] [Trondheim/AA555A0008060353] Going from UNKNOWN to OK
[INFO] [11/04/2016 22:29:07] [influxDBActor] Writing to InfluxDB, point {...}
```

6.4.2 Becomes Available

Hypothesis Monitoring becomes available through the map on refresh.

Method Make sure the Dataport MQTT Broker is not running before starting the Dataport and start it manually after starting the Dataport.

Result From the log excerpt below we see that the Actor responsible for the connection to the MQTT broker is failing to start because the it is unable to connect to the broker, but once the broker is started it connects. Figure 6.8 and Figure 6.9 shows the map indicating whether its connected to the broker or not. No other parts of the system was affected by this.

```
[ERROR] [11/04/2016 22:29:09] [dataportBroker] Unable to connect to server
akka.actor.ActorInitializationException: dataportBroker: exception during
[ERROR] [11/04/2016 22:30:04] [dataportBroker] Unable to connect to server
akka.actor.ActorInitializationException: dataportBroker: exception during
[INFO] [11/04/2016 22:31:59] [dataportBroker] Going from DISCONNECTED to CONNECTING
```

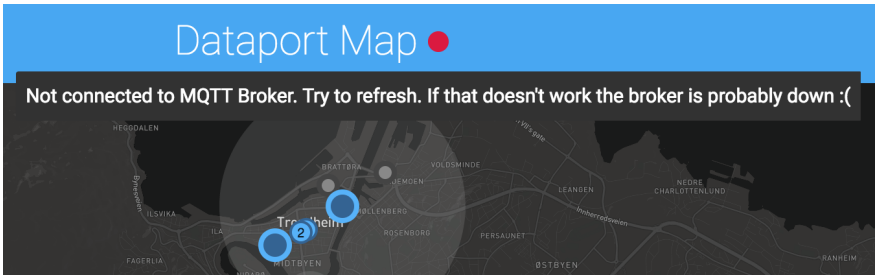


Figure 6.8: Map Disconnected

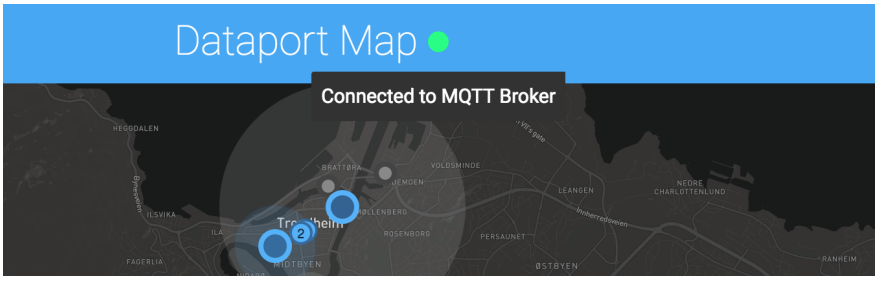


Figure 6.9: Map Connected

```
6  [INFO] [11/04/2016 22:31:59] [dataportBroker] Connecting to broker:
   ↪  tcp://dataport.item.ntnu.no:1883
7  [INFO] [11/04/2016 22:31:59] [dataportBroker] Going from CONNECTING to CONNECTED
8  [INFO] [11/04/2016 22:31:59] [dataportBroker] Yeah! I connected to my MQTT broker for
   ↪  the first time
```

6.4.3 Becomes Unavailable

Hypothesis Paho begins its exponential backoff strategy to reconnect to the broker. Monitoring becomes unavailable through the map, but gateway and sensor status and reception messages are logged on the server.

Method Make sure the Dataport MQTT Broker is running before starting the Dataport and stop it manually after starting the Dataport.

Result Figure 6.10 shows how the notification appears on the desktop. Figure 6.11 shows the Slack notifications with some added plausible comments from the person receiving the notifications.

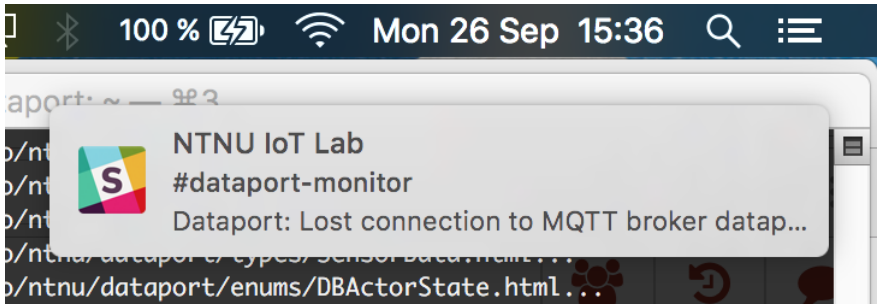


Figure 6.10: Slack Desktop Notification

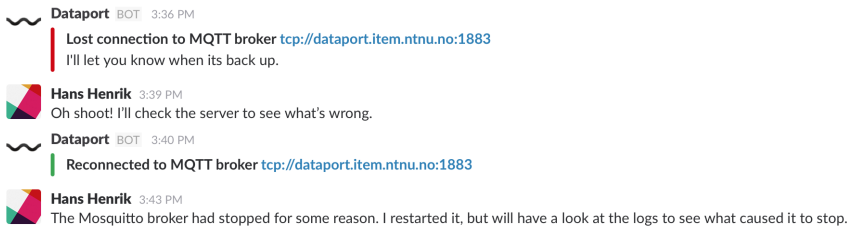


Figure 6.11: Slack Notification Leads to Manual Fix

6.5 Device List Source

As mentioned in Section 5.4, the Dataport retrieves the list of devices it will supervise from an Airtable on startup. This source can fail independently from the other components of the system.

6.5.1 Unavailable on Startup

Hypothesis The Dataport does not know which gateways and sensors exist. Hence, it does not know which gateways and sensors to subscribe to from TTN. No digital twins are created, the Dataport will throw an exception and not start.

Method Define a malformed Airtable address `INVALID.AIRTABLE.ADDRESS`.

Result All SiteActors threw an exception on Dataport startup. This because it is crucial for the application to have this information in order to function. An excerpt from the log shows the result:

```
1 [ERROR] [Trondheim] FATAL! Could not get list of devices. Sending shut down signal!
2 UnirestException: java.net.UnknownHostException: INVALID.AIRTABLE.ADDRESS
```

```
[INFO] Shutting down remote daemon.
[INFO] Remote daemon shut down; proceeding with flushing remote transports.
[INFO] Remoting shut down
[INFO] Cluster Node - Shutting down...
[INFO] Cluster Node - Successfully shut down
Process finished with exit code 0
```

The decision to terminate the program on this kind of error—instead of keeping it running and trying to connect to the devices source—was made because this is considered a severe error and it should be reflected in a non-running program, as the program has no real value if it does not know which devices exists.

6.5.2 Becomes Unavailable

In this case, since the Airtable was available on startup, the Dataport has the initial network graph (all gateways and sensors). The Airtable becoming unavailable should therefore only lead to the Airtable not being updated.

```
try {
    // Try to update the Airtable row
    Unirest.patch(airtableRecordURL)
            .header("Authorization", "Bearer " + airtableAPIKey)
            .header("Content-Type", "application/json")
            .header("accept", "application/json")
            .body(new JsonNode("{fields: {status: " + to + "}}"))
            .asJson();
} catch (UnirestException ue) {
    log().warning("Unable to update status in Airtable because of: " +
    ↪  ue.getMessage());
}
```

Hypothesis The Airtable will not be updated on gateway and sensor state changes, but the rest of the system will run as normal.

Method This experiment was not done.

Result No result.

6.5.3 Device is Lacking Position

Hypothesis No digital twin will be created for the device. It's state will therefore not be monitored and no measured data will be saved.

eui	type	latitude	longitude	timeout	status
0000024B080E06B3	Gateway	55.70777	9.53343	600	OK
00000008DEC044C	Sensor	55.70776	9.53293	1800	UNKNOWN
000000000E77EE00	Sensor	55.70752	9.53586	1800	UNKNOWN
0000000902FBDD2	Sensor	55.70540	9.52122	7200	OK
0000000031F5B033	Sensor			1800	

Figure 6.12: Device Missing Position in Airtable

Method Create a device without position in Airtable. This is shown in Figure 6.12.

Result Excerpt from the server log shows that the Dataport handles it as expected:

```
[INFO] [Vejle/000000008DEC044C] - Going from UNINITIALIZED to UNKNOWN
[ERROR] [Vejle] - Device 0000000031F5B033 did not have position in Aritable, it will
↪  not be created!
[INFO] [Trondheim/0000000048524DD8] - Going from UNINITIALIZED to UNKNOWN
```

The reason for conducting this experiment is that missing fields often lead to exceptions. If the Actor responsible for creating the digital twins of devices doesn't handle this exception correctly, it might crash. The choice of simply ignoring the device if its lacking position, is because at the time of the decision we were not storing the measurements. The first iteration of handling this was to ignore it. The next could be to give it a random position and start storing data. However, this could be confusing for people using the map, so we would also need to hide it from the map. It will also muddle the stored data with a non-real position, which could potentially influence any real-time analysis done. The best solution would probably be to allow position to be stored as `null`, hide any devices without position from the map, but still create a digital twin the monitors the device. This way the we can gather the data and add position at a later point in time.

6.6 The Dataport Machine Stops

Hypothesis AppBeat health check will fail and a Slack message will be sent, saying it is unable to reach dataport.item.ntnu.no.

Method Turn off the machine running the Dataport and leave it off for more than the specified timeout in AppBeat.

Result A Slack notification saying the PING operation failed was sent approximately 3 minutes after the machine was turned off. Another notification saying the

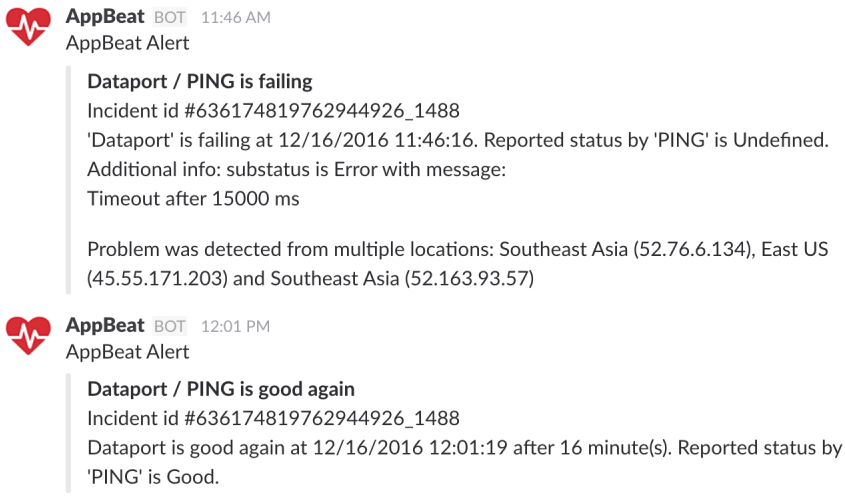


Figure 6.13: Slack Notification When Host Machine is Off

PING operation worked again was sent approximately 5 minutes after it was turned on again. The notifications are shown in Figure 6.13.

It is important to point at that AppBeat only does a PING operation against the host machine. When the PING operation fails, a number of things can be wrong, e.g

- The machine is off (this case),
- The machine is offline (discussed in Section 6.8.3),
- The process responsible for handling TCP requests isn't working properly

So we don't really know exactly what is wrong, but we know we need to check the machine.

6.7 The Forecast API is Unavailable

Hypothesis An error is logged, the data will not be stored in the database and the Actor will try again at the next scheduled time.

Method Use a malformed URL, increase the frequency at which forecasts should be retrieved and verify that it retries after a failed attempt.

Result Error was logged, as shown below. A Slack notification was not sent (this is expected, it should only occur if the API version is outdated, discussed in Section 6.8.1).

```
[ERROR] [12/14/2016 21:58:55] [weatherForecast] ikkenoeher.no
    java.net.UnknownHostException: ikkenoeher.no
[ERROR] [12/14/2016 21:59:05] [weatherForecast] ikkenoeher.no
    java.net.UnknownHostException: ikkenoeher.no
```

6.8 Unplanned Real-Life Failures

The Dataport has been running more or less continuously since October 4th. In addition to the manual failure experiments documented in this chapter, a few unforeseen failures also occurred.

6.8.1 The Forecast API Version is Outdated

When checking an unrelated error in the log, an exception from the actor responsible for retrieving sunlight forecasts was discovered. The endpoint it was trying to get data from stated that

> The specified version number is end-of-lifed for this product

No mechanism for notifying about this type of failure had been implemented at the time. However, the system is resilient to this kind of failures, which means the actor was terminated and restarted without affecting any other parts of the system. The result of the failure was some lost forecast data for approximately one month into the future. This data was re-collected when the error was detected. Forecasts with a shorter time frame into the future would have caused more severe loss of data. As a countermeasure, a Slack notification is now sent if the forecast APIs does not answer in the expected way.

Hypothesis A Slack notification is sent when the forecast API gives unexpected response.

Method Specify an outdated version of the forecast API.

Result As seen in Figure 6.14, the Slack notification works as intended.

Sunlight forecast failure
Failed to get sunrise forecast for Trondheim from http://api.met.no/weatherapi/sunrise/1.0/?lat=63.430515;lon=-10.395053;from=2016-11-26;to=2016-12-26. Check if version 1.0 has been outdated.

Sunlight forecast failure
Failed to get sunrise forecast for Vejle from http://api.met.no/weatherapi/sunrise/1.0/?lat=55.711311;lon=-9.536354;from=2016-11-26;to=2016-12-26. Check if version 1.0 has been outdated.

Figure 6.14: Slack Notification Forecast API Version is Outdated

6.8.2 Sensor Sends Malformed Point to Database Actor

On October 21st an old sensor only measuring CO2 (not NO2, temperature, humidity and pressure) as included in the monitored application. When the Actor responsible for the database connection tried to send this measurement, something went wrong. Unfortunately, nothing got written to the log. So when trying to figure out why suddenly no data got written to the database, it was hard figuring out that this was the cause. When debugging locally, it turns out an exception gets thrown to the standard output (stdout):

```
1 org.influxdb.impl.BatchProcessor write
2 SEVERE: Batch could not be sent. Data will be lost
3 java.lang.RuntimeException: {error:unable to parse ... : invalid field format}
```

This is probably because the point attempted inserted into the database was sending some null fields, since the sensor did not measure them. However, since this caused the database client to fail without the actor failing, everything looked OK from the outside. The actor was receiving measurements, but was not able to write them to the database. This error actually led to a loss of 5 days worth of data from the sensors, shown in Figure 6.15 where there are no data points in the period October 21st to October 26th.

The solution to this was to have the sensor measuring fewer things write to a separate time series in the database:

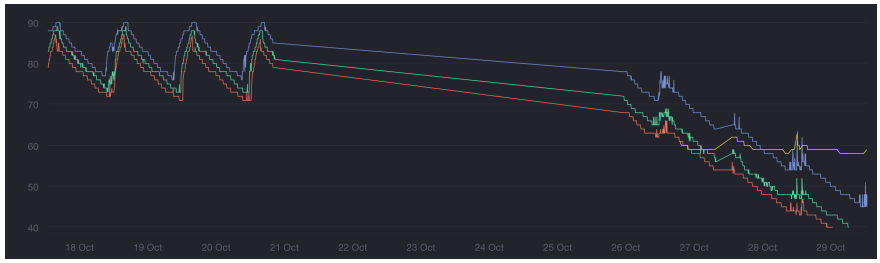


Figure 6.15: Lost Data as Result of Lost Internet Connection at Server Site

```
[INFO] [Trondheim/000000001B1A8C66] - Going from UNKNOWN to OK
[WARN] [influxDBActor] - Getting point from sensor sending too few measurements,
↪  store in separate time series
[INFO] [Trondheim/00000000CD3BE279] - Going from UNKNOWN to OK
[INFO] [influxDBActor] - Writing to InfluxDB, observation point: {...}
```

6.8.3 Internet Connection Lost at Server Site

On October 27th, the Internet connection was lost for approximately 2 hours at the Institute of Telematics at NTNU, where the server the Dataport runs at. This gave a few interesting results.

Hypothesis The application will keep running, but loose all its connections. It will not be able to notify about these lost connections as there are no Internet connection. AppBeat will trigger an alert as it will not get a response from dataport.item.ntnu.no. It will also send an alert when it is able to reach the server again. When the application is able to connect to the Internet again, everything should work as normal.

Result The AppBeat component worked as planned: A Slack alert was sent as shown in Figure 6.16. Figure 6.17 shows the notification meeting us when logging into the AppBeat application. It says everything is OK now, but something was not OK earlier, and reminds us to check that everything is *actually* OK.

The choice of using Paho's internal handling of lost connections affected the results of this failure. In the log excerpt below we can see that the Actor holding the connection to the Gateway Status Mqtt Broker (ttnGatewayStatusBroker) gets a timeout exception when trying to reconnect to the broker. This causes the Actor to fail and the supervisor strategy of restarting the Actor to kick in. However, the two Actors holding the connections to the Application MQTT Broker (ttn-Vejle-broker and ttn-Trondheim-broker) *did not* get a timeout exception. The log does not

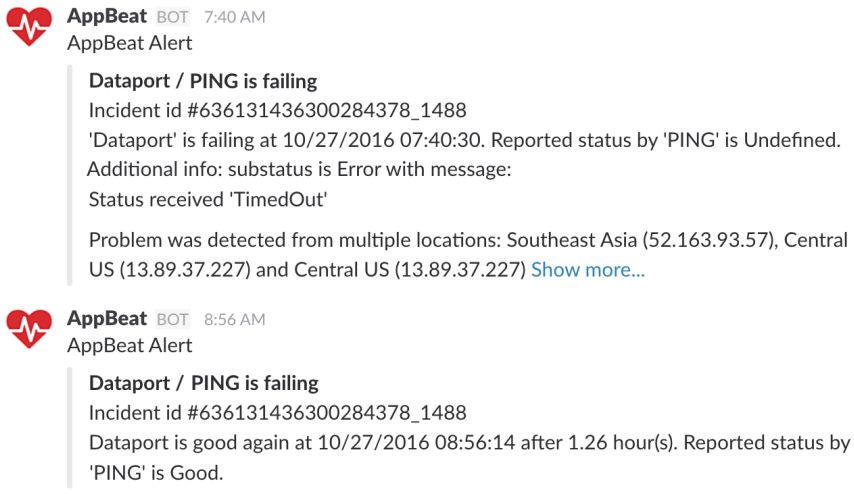


Figure 6.16: Slack Notification from AppBeat

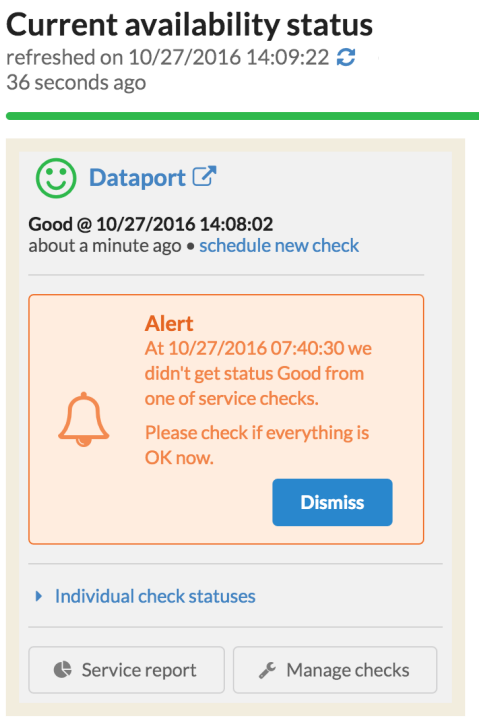


Figure 6.17: Status Alert in AppBeat Dashboard

give any information as to what went wrong. This is what needs to be considered when choosing whether to use an external library's implementation over doing the implementation yourself: How much to you trust the external library to have done a proper implementation? How much detailed information do you need if something goes wrong? It seems something went wrong with Paho's built-in reconnecting without it throwing an exception.

The fact that the Application MQTT Broker Actors did not fail properly led to some loss of data from the sensors. Figure 6.15 shows a small gap of data points on October 27th.

So to conclude: Parts of the Dataport handled the failure, other parts didn't.

```
2016-10-27 07:39:48 INFO  [ttn-Vejle-broker] - Going from CONNECTED to CONNECTING
2016-10-27 07:39:48 ERROR [ttn-Vejle-broker] - Damn! I lost my MQTT connection. Paho
↪  automatic reconnect with backoff kicking in
2016-10-27 07:40:38 INFO  [ttn-Trondheim-broker] - Going from CONNECTED to CONNECTING
2016-10-27 07:40:38 ERROR [ttn-Trondheim-broker] - Damn! I lost my MQTT connection.
↪  Paho automatic reconnect with backoff kicking in
2016-10-27 07:40:46 INFO  [ttnGatewayStatusBroker] - Going from CONNECTED to
↪  CONNECTING
2016-10-27 07:40:46 ERROR [ttnGatewayStatusBroker] - Damn! I lost my MQTT connection.
↪  Paho automatic reconnect with backoff kicking in
2016-10-27 07:40:46 ERROR [ttnGatewayStatusBroker] - MqttException: Timed out waiting
↪  for a response from the server
[...]
2016-10-27 07:41:45 ERROR [ttnGatewayStatusBroker] - MqttException
akka.actor.ActorInitializationException: ttnGatewayStatusBroker: exception during
↪  creation
Caused by: org.eclipse.paho.client.mqttv3.MqttException: MqttException
Caused by: java.net.UnknownHostException: croft.thethings.girovito.nl
[...]
2016-10-27 08:42:45 ERROR [Vejle/00000000902FBDD2] - java.net.UnknownHostException:
↪  hooks.slack.com
```

7

Discussion

7.1 Evaluation of the Dataport

Two incidents occurred where external Java client code did not work as expected.

InfluxDB driver One malformed point made the InfluxDB driver fail without throwing exception. This was solved by adding a special case handling the sensor sending too few fields.

Paho client Only 1/3 Actors got a timeout exception from Paho after its backoff algorithm kicked in. This resulted in that only the one that failed correctly reconnected when things were back OK (Internet connection was lost for 2 hours on site where server is located). The two others did not throw exception, hence the actors didn't die and get restarted, and were therefore in a locked state without a working MqttConnection and without trying to reconnect.

Sends out too many notifications, can bloat the channel and reduce the importance of the notification. These are not false positives, but we could perhaps have a timeout if a resource is reconnected immediately. However, this means we introduce a delay before notifying. The optimal duration of this delay is hard to determine.

The feedback from CTT is that the system har proven itself very valuable, especially in a deployment process which there has been several of.

Have been running for 2 and a half months, collecting data and notifying about real failures. It has proved itself useful to CTT, especially during the latest deployment of new sensors mid-December 2016.

Since some of the data is overlapping with the data measured by the network, we can use this to compare data measured by the network with high precision forecast data. How many sensors is needed for the average to match the forecast?

Hevner states that "Good design science research often begins by identifying and representing opportunities and problems in an actual application environment" [Hev07, p. 89]. The developed artifact does solve an actual problem. It has proven to be resilient to failures by running more or less continuously for 2 and a half months. The design is resilient through the use of actors.

We will not go into detail on how Akka handles clustering. For this, the reader is referred to Chapter 6 *Networking* in [Lig16, p. 320]. Due to the nature of the actor pattern used in the implementation, we saw in Section 5.12 that only a small addition to a configuration file would make the same source code run on a cluster.

It is also worth noting the difference between *scaling up* and *scaling out*. Moving an Akka application from a singel machine into a cluster is to scale out. The application becomes more dependable as if one machine fails, another can take over. Scaling up means giving the application more computational power. If the IoT application we monitor is of a different nature than the one we are monitoring now, i.e. each sensor sends many measurements per second or the number of sensors is multiplied many times, we might need to scale up.

When it comes to the possibility of data analysis, this can either be outsourced to a single powerful computer, or distributed to many workers. We already have a distributed architecture, in that each sensor has its own digital twin. For applications where sensors only send data every few minutes, the digital twin is a good candidate for doing data analysis regarding itself. If one wants to look at data from the whole network, this has to be orchestrated somehow.

7.2 Design Science Checklist

In Section 2.4 I introduced Hevner's "checklist" for design science research projects. Answering these questions gives a nice summary of the process.

What is the research question?

How can data and meta data from an IoT application can be used to discover and identify failures in the application without or with minimal extra instrumentation?

What is the artifact? How is the artifact represented?

The artifact is a piece of software that gathers data and meta data sent from the network and uses this to build a virtual representation of the network. The digital twins representing the physical objects in the network are given the extra instrumentation we need to monitor that they perform as expected. The artifact also gathers data from other data sources, allowing this to be compared with the data measured by the network.

What design processes will be used to build the artifact?

The artifact was be built through an iterative process, where a Minimal Viable Product (MVP) was quickly deployed into the field and refined based on its performance, discovered weaknesses and new requirements from the application environment.

How are the artifact and the design processes grounded by the knowledge base? What, if any, theories support the artifact design and the design process?

The artifact is built using the well tested actor model. The choice of quickly introducing the artifact into the environment and iteratively make improvements aligns with the guidelines of design science research.

What evaluations are performed during the internal design cycles? What design improvements are identified during each design cycle?

Features were added to handle more failures and extend the functionality of the Dataport in every design cycle.

How is the artifact introduced into the application environment and how is it field tested? What metrics are used to demonstrate artifact utility and improvement over previous artifacts?

The artifact was connected to the application from the very beginning of the development. It was fed real data throughout the process and had to handle new data formats while supporting legacy formats. The continued notification of timeouts and self-healing, combined with the duration of which the artifact keeps running without failure, demonstrate the usefulness of it.

What new knowledge is added to the knowledge base and in what form?

How IoT applications can be monitored without adding instrumentation to the possibly already deployed devices.

Has the research question been satisfactorily addressed?

Initially, scalability was a big concern. However, due to the size and nature of the application being monitored (only 19 devices in total sending data at a highest frequency of every 5 minutes), this has not been a pressing issue. Because of this, a thorough assessment of the systems scalability has not been prioritized. This does not mean the system is not scalable. Section 5.12 discuss the potential of scaling with Akka.

7.3 Future Work

7.3.1 Next Design Cycles

Due to the iterative development approach, new functionality demands was discovered throughout the whole design process. Not all of these were implemented, due to time needed to also evaluate the developed system. In this subsection I'll present a few thoughts on some of these, which will be used by me—and possibly other contributors—as inspiration for the continued development of the Dataport.

Dynamic creation of digital twins

Today, the Dataport reads a list of sensor IDs, positions and expected sending frequency on startup. If a new sensor is added to the application, the Dataport needs a restart to be aware that this sensor exists. It is possible to use wildcards when subscribing to topics from the TTN MQTT broker. By doing this, the Dataport can become aware of sensors that are added to the application without needing a restart. The Site Actor will be responsible for creating new Sensor Actors when a new sensor is deployed. However, we will still need to know (i) the position of the sensors—as this is not provided by TTN yet—and (ii) how often we should expect the sensors to send data in order to know when we should notify someone about lacking data. For existing sensors, the Airtable used today can be used to look this up. If the sensor is missing from the table, a default expected sending frequency can be given and a Slack notification can request the Airtable to be updated with this information.

Adaptable timeout for sensors with variable sending frequency

Early December, CTT made an update to their sensors. This update made the sensors change how often they made measurements based on their battery level. In

order to still be able to know when a sensor has actually timed out—not just changed its sending frequency—we would need to have the "sending frequency rules" and change the timeout when a battery threshold is crossed.

Send instructions to the nodes based on central analysis

TTN supports sending messages in the other direction as well. This opens the possibility of making individual sensor nodes adapt not only based on local knowledge (e.g. its battery level and measurements), but also on *global* knowledge from the whole network, analysed centrally together with data from other sources, e.g. weather forecast data. The LoRaWAN protocol requires devices to listen for responses for a short time after doing a transmission, but how to react to the possibly received message must of course be implemented.

Analyse forecast data and measured data

The institute has already planned to use the Dataport in order to learn more about the sensors and try to make components in the network more autonomous.

References

[ADK+16] Dirk Ahlers, Patric Arthur Driscoll, Frank Alexander Kraemer, Fredrik Valde Anthonisen, and John Krogstie. A Measurement-Driven Approach to Understand Urban Greenhouse Gas Emissions in Nordic Cities. https://brage.bibsys.no/xmlui/handle/11250/2423962, December 2016. Accessed: 2016-12-04.

[AHS07] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *2007 International Conference on Mobile Data Management*, pages 198–205, May 2007.

[ALRL04] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[Ama16] Amazon Web Services. AWS IoT Developer Guide. http://docs.aws.amazon.com/iot/latest/developerguide/iot-dg.pdf, 2016. Accessed: 2016-12-11.

[BBH+10] Carlo Alberto Boano, James Brown, Zhitao He, Utz Roedig, and Thiemo Voigt. *Low-Power Radio Communication in Industrial Outdoor Deployments: The Impact of Weather Conditions and ATEX-Compliance*, pages 159–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[BL12] M. Blackstock and R. Lea. Iot mashups with the wotkit. In *2012 3rd IEEE International Conference on the Internet of Things*, pages 159–166, Oct 2012.

[BL13] Michael Blackstock and Rodger Lea. Toward interoperability in a web of things. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 1565–1574, New York, NY, USA, 2013. ACM.

[BL14a] M. Blackstock and R. Lea. IoT interoperability: A hub-based approach. In *International Conference on the Internet of Things (IOT)*, pages 79–84, Oct 2014.

[BL14b] Michael Blackstock and Rodger Lea. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*, WoT '14, pages 34–39, New York, NY, USA, 2014. ACM.

[Den16] Volkmar Denner. Why a Bosch IoT Cloud? http://blog.bosch-si.com/categories/internetofthings/2016/03/why-a-bosch-iot-cloud/, March 2016. Accessed: 2016-12-11.

[dom16] ellenfosborne v-anpasi kiwhit v-cmans tysonn dominicbetts, bzurcher. Overview of device management with IoT Hub. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-device-management-overview, October 2016. Accessed: 2016-12-11.

[EHHP13] Peder J. Emstad, Poul E. Heegaard, Bjarne E. Helvik, and Laurent Paquereau. *Dependability and Performance with Discrete Event Simulation*. Kompendieforlaget, June 2013. This edition is a draft.

[Gar16] Ginger Gardiner. Digital twin, digital thread and composites. http://www.compositesworld.com/blog/post/digital-twin-digital-thread-and-composites, April 2016. Accessed: 2016-12-11.

[Goo16] Google. Overview of Internet of Things. https://cloud.google.com/solutions/iot-overview#operations, October 2016. Accessed: 2016-12-11.

[Han16] Hans Henrik Grønsleth. Dataport Map. https://github.com/NTNU-ITEM/ctt-dataport-web, http://dataport.item.ntnu.no, 2016. Accessed: 2016-12-18.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[HC10] Alan Hevner and Samir Chatterjee. *Design Science Research in Information Systems*, pages 9–22. Springer US, Boston, MA, 2010.

[Hev07] Alan R. Hevner. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, 19(2):87–92, 2007.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Q.*, 28(1):75–105, March 2004.

[Kom10] Trondheim Kommune. Energi- og klimahandlingsplan for Trondheim kommune. Mål og tiltak for perioden 2010-2020. https://www.trondheim.kommune.no/klimahandlingsplan/, 2010. Accessed: 2016-12-18.

[LB14] R. Lea and M. Blackstock. City hub: A cloud-based iot platform for smart cities. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 799–804, Dec 2014.

[Lib16] Libelium. Enhancing environmental control and reducing emissions in Nordic Smart Cities. http://www.libelium.com/enhancing-environmental-control-and-reducing-emissions-in-nordic-smart-cities/, November 2016. Accessed: 2016-12-04.

[Lig16] Lightbend Inc. Akka Java Documentation. http://doc.akka.io/docs/akka/2.4.8/AkkaJava.pdf, July 2016. Accessed: 2016-07-20.

[Mun12] Trondheim Municipality. Kommunal energi- og utslippsstatistikk oppdateres ikke. www.ssb.no/natur-og-miljo/artikler-og-publikasjoner/kommunal-energi-og-utslippsstatistikk-oppdateres-ikke, February 2012. Accessed: 2016-02-21.

[Net16] The Things Network. The Things Network. https://www.thethingsnetwork.org/, December 2016. Accessed: 2016-12-12.

[SA00] Frank Stajano and Ross Anderson. The grenade timer: Fortifying the watchdog timer against malicious mobile code. In *in Proceedings of 7th International Workshop on Mobile Multimedia Communications (MoMuC 2000), Waseda*, 2000.

[Shu14] Shubham Khandelwal and Hari Singh and P. Chaurasia. Experimental Study on the Effect of Cloud on Solar Photovoltaic Panel in Jaipur (Rajasthan). https://www.ijsr.net/archive/v3i10/T0NUMTQ1NA==.pdf, October 2014. Accessed: 2016-12-18.

[Sim96] Herbert A. Simon. *The Sciences of the Artificial (3rd Ed.)*. MIT Press, Cambridge, MA, USA, 1996.

List of Acronyms

List of Acronyms

CTT The Carbon Track and Trace project.

GHG greenhouse gas.

IoT Internet of Things.

ITEM The Department of Telematics.

LPWAN Low Power Wide Area Network.

MVP Minimal Viable Product.

NTNU Norwegian University of Science and Technology.

PM Particulate matter.

QA Quality assurance.

RSSI Received Signal Strength Indicator.

SSB Statistisk Sentralbyrå [Statistics Norway].

TTN The Things Network.

List of Figures

List of Tables