Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Eric Törn

IoT Software for Smart Houses

Managing data collection and connectivity between IoT devices for an advanced heating control algorithm

Master's thesis in Industial Cybernetics
Supervisor: Sebastien Gros
June 2021

Eric Törn

IoT Software for Smart Houses

Managing data collection and connectivity between IoT devices for an advanced heating control algorithm

Master’s thesis in Industial Cybernetics
Supervisor: Sebastien Gros
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

This project aims to create a software for an Internet of Things (IoT) based advanced heating control system, for the purpose of reducing energy consumption and power peak load in smart buildings.

The system is based on using System Identification (SYSID) to generate a dynamic temperature model of the house, and using Model Predictive Control (MPC) to control the temperature. While using MPC for managing energy in buildings is not new, previous research has been based on using expensive equipment and complicated setups that are not readily available for the average person and home. With the emergence of IoT devices, there now exists cheap and commercially available equipment that is easy to set up for already existing homes. In this project we wanted to investigate the viability of using such devices for an MPC based heating system.

The software presented in this thesis uses a multithreaded data collection system together with a web-based Application Programming Interface (API) to service the MPC algorithm with data. The results showed that this kind of system and software is viable and has potential.

This work serves as laying the ground work for further research in this area. Further research should focus on improving the software and, as well as investigate different temperature models and control algorithms for better performance.

Sammendrag

Dette prosjektet har som mål å lage en programvare for et Tingenes Internet (IoT) basert avansert oppvarmingsstyringssystem, med det formål å redusere energiforbruket og toppeffektbelastning i smarte bygninger.

Systemet er basert på å bruke Systemidentification (SYSID) til å generere en dynamisk temperaturmodell av huset, og bruke Modell Prediktiv Kontroll (MPC) til å kontrollere temperaturen. Selv om bruk av MPC for styring av energi i bygninger ikke er nytt, har tidligere forskning vært basert på bruk av dyrt utstyr og kompliserte oppsett som ikke er tilgjengelig for den gjennomsnittlige personen og hjemmet. Med fremveksten av IoT enheter eksisterer det nå billig og kommersielt tilgjengelig utstyr som er enkelt å sette opp for allerede eksisterende hjem. I dette prosjektet var det ønsket å undersøke levedyktigheten av å bruke slike enheter til et MPC-basert varmesystem.

Programvaren som presenteres i denne oppgaven bruker et flertrådet datainnsamlingssystem sammen med et webbasert Applikasjonsprogrammeringsgrensesnitt (API) for å betjene MPC-algoritmen med data. Resultatene viste at denne typen system og programvare er levedyktig og har potensial.

Dette arbeidet fungerer som grunnlag for videre forskning på dette området. Videre forskning bør fokusere på å forbedre programvaren, samt undersøke forskjellige temperaturmodeller og kontrollalgoritmer for bedre ytelse.

Preface

This master's thesis was written in the final spring of my master's degree in Industrial Cybernetics at Norwegian University of Science and Technology (NTNU).

The thesis was related to a larger project (POWIOT) by Sebastien Gros, which is looking to investigate the usage of low-cost Internet of Things (IoT) devices together with smart control algorithms to alleviate the growing problem of overloaded power distribution infrastructures.

It is my hope that the work in this thesis can serve as laying the groundwork for the further research within the POWIOT project. Throughout this project I learned a lot about writing software, Internet of Things (IoT) devices, and about the specific problem of controlling a house's heating system with an Model Predictive Control (MPC). Many hours have been spent on bug fixes and software iterations, but in the end it was all worth it. It has been a gratifying project.

I would like to thank Sebastien Gros for supervising this thesis. Sebastien also had the software from this thesis running in his house and was actively testing the software functionality whenever I deployed new updates.

I would also like to thank Kristoffer Nyborg Gregertsen for providing feedback on the report, and Vanja Skålnes Haugsnes for providing an inspiring work environment.

Acronyms

AC Air Conditioning.

AI Artificial Intelligence.

AMS Advanced Metering System.

API Application Programming Interface.

COP Coefficient Of Power.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

IoT Internet of Things.

Ipopt Interior Point Optimizer.

IR Infrared.

LAN Local Area Network.

MET Norwegian Meteorological Institute.

MHE	Moving Horizon Estimation.
ML	Machine Learning.
MPC	Model Predictive Control.
NTNU	Norwegian University of Science and Technology.
PI	Proportional-Integral.
PID	Proportional-Integral-Derivative.
SSH	Secure Shell.
SYSID	System Identification.
TRNSYS	Transient System Simulation Tool.
WCET	Worst Case Execution Time.
WSGI	Web Server Gateway Interface.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Description

In the context of smart buildings, this project aims to investigate the viability of using cheap equipment to interact with energy-related devices (such as heat pumps) and introducing high-level decisions in the system (e.g., setting temperature references on heat pumps) based on a high-level decision-making algorithm. The connectivity between sensors, algorithm and actuators will be based on commercially available Internet of Things (IoT) devices and Application Programming Interface (API) services, while the decision-making algorithm will be based on creating a model of the building rooms with System Identification (SYSID), then generating control inputs through Model Predictive Control (MPC).

More specifically, this thesis will focus on software implementation of this system, and the viability of implementing this kind of system in an existing residential home.

1.2 Background and motivation

According to the EU directive on energy performance from 2010, buildings are using 40% of the total energy consumption in the EU [14]. Buildings are therefore one of the largest energy consumers, with residential buildings taking up a large portion of that consumption. In addition, it is likely that the electricity consumption for residential homes will increase in the coming decades due to a shift to electric vehicles, and more electification in general. Another problem is the peak power load. Electric consumption in residential homes generally have peak power loads during the morning and evening when people are home and active. On a larger scale, this demands an over-sized electricity infrastructure compared to what is needed on average.

So there are two important problems worth investigating here: How can residential homes use less energy and how can they even out the consumption load throughout the day and night to lower peak power demands?

1.2.1 Smart control

The most common heating systems today are based on thermostats. These systems are easy and cheap to implement and work fine for the purpose of simply regulating temperature, but they often waste energy by non-optimal regulation. Thermostats do not consider variables like outside temperature, weather forecasts, and electricity prices. These additional variables could give an indication in how the temperature dynamics in the building might change in the coming day, or what the expected net load on the electricity system will be in the coming day. This extra information could help in making an informed decision on how to use the heating system in a smarter way, helping to reduce energy consumption and spread out the consumption over hours that are less pressured for power.

Such a system could be implemented with the help of SYSID and MPC. With SYSID we can use data to generate an approximate dynamic model of the temperature in the building. This model can then be used in a MPC algorithm, which simulates the temperature trajectory of the house and optimizes for low energy consumption. This algorithm might consider several variables like weather and price forecasts, and gives us an optimized setting for the heat pumps. Such an algorithm might grow to include additional energy-devices like heating panels, charging stations, and more. Traditionally these kind of systems have required expensive equipment and advanced system setups, and have not been viable for an average home-owner without specialized knowledge and a big investment. However, this might change with the emergence of IoT devices.

1.2.2 Internet of Things (IoT) solutions

Over the last decade we have seen an increasing amount of commercially available IoT devices and services being developed for monitoring and controlling regular houses. Regarding energy consumption there now exists services like Tibber [22], where you can access your power consumption in real time, and Sensibo [19], which provides a way to measure the temperature in the house and control your heat pumps. Installing these devices can be done by anyone, and you have easy access to sensor readings through an app or by API requests through scripts. There exists many more energy-related IoT devices with similar easy of access, some of which will be discussed in section 2.1.2. And for processing power there are the Raspberry Pi [17], a small and cheap computer often used as a dedicated home automation hub (among other things).

With this increasing availability of cheap IoT devices, homes are becoming more connected and have more and more data available. With the help of these new capabilities, an advanced control system might become practically and economically viable for already existing average homes.

1.2.3 Home automation software and algorithms

Although there already exists software for home automation, these are simple and general in nature, and not optimal for an advanced control algorithm. One example is the Tibber app, where you can add pre-approved IoT devices and use their automation features to lower energy. This algorithm is, by our understanding, very basic, and you can't customize the automation yourself. Another example is the Home Assistant open-source software, in which you can integrate over one thousand IoT devices and other services to make your own automation. You can even code your own plugins to this software. However, this software is made for simple automations, not advanced processing algorithms like SYSID or MPC, which require high amount of customization and control.

Therefore, there is a need to create our own software in order to explore the system that is envisioned (a software that uses IoT devices to collect data day and night, while running advanced control algorithms to control the house). Apart from creating the software system in this thesis, the hope is that this work can lay the groundwork for the software within the POWIOT project led by Sebastien Gros, which aims to investigate neighborhood electricity negotiation with the help of IoT devices.

1.3 Scope and limitations

The decision-making algorithm will use SYSID to generate a model of the building. This model will be used in a MPC algorithm to generate control instructions (in the form of temperature reference for the heat pumps) based on a cost function. This algorithm will take into consideration the current temperature inside and outside, the spot prices for the next 24 hours, and the weather forecast. It will then optimize for the best control input to the system.

This algorithm relies heavily on data. First it needs historical data to generate the model, then it needs current measurements to know the state of the system, and lastly it needs forecasts to try and optimize against a predicted future. To get accurate models and predictions we need reliable data. Due to this reliance, there will be a large focus on creating a robust data collection software which can run day and night on a Raspberry Pi.

But while data is important, and the quality of the algorithm will depend on the quality of the data, the main focus of this project is on the overall software system design rather than gaining the best quality measurements. This means that this project will not spend time comparing different IoT devices or services or finding the optimal number of sensors.

This project will also not focus on the SYSID and MPC algorithms, but rather focus on the software and data collection that supports these algorithms.

The IoT devices used in this project are from Tibber and Sensibo. They are commercially available devices that can be accessed through an API interface over Wi-Fi. These devices are cheap and easily available for anyone in Norway.

The collected data will include temperature inside and outside, real-time power consumption from a Smart Meter, and heat pump settings (reference temperature and heating modes). In addition, spot prices will be used within the cost function of the MPC algorithm.

The project includes the following tasks:

- Implement a robust software system for collecting data, servicing data requests, and sending control instructions to heat pumps
- Collect data
- Test a high-level decision-making algorithm on this system, using a dynamic temperature model that was generated by a System identification operation from the data.

1.4 Structure of this report

The report begins with background information in chapter 2, which encompasses state of the art in building energy controllers, home automation devices and software, and explanations of relevant theory for the project.

The requirements for the system and software is defined in chapter 3. These serves as a guide for designing the software.

Design and Implementation of the system is presented in chapter 4. Here the system solution is presented together with the software tools, libraries, and the functionality of the code. The source code is appended to a zip file.

A how-to guide for employing the system and software is presented chapter 5. This chapter includes all the information you would need to implement the system on a normal home.

A case study is presented in chapter 6, where the system and software is implemented in a house in Trondheim.

Tests and results from this case study is presented in chapter 7.

The results are discussed in chapter 8, and the report finishes with a conclusion and suggestions for future work in chapter 9.

Chapter 2

Background

This chapter presents useful background information for the thesis and is divided into the following sections:

- Section 2.1 presents the state of the art in automatic control of heating systems in buildings, IoT devices, and home automation software and services.
- Section 2.2 presents theory that is relevant to this project.

2.1 State of the art in energy management of smart houses

Research in heating automation has been going on for a relatively long time. The goal of most of this research has been two-fold: to reduce energy consumption while increasing perceived comfort of people in the building. The control methods themselves have ranged from classical control methods like the proportional-integral-derivative (PID) controller, which were popular in research around 1980-2000, to more advanced methods like MPC and Artificial Intelligence (AI) methods, which has become more common in recent decades [20].

The equipment have changed over the years. The equipment and specialized technology used to be expensive and require a comprehensive setup. Today we have access to cheaper and more accessible technology such as cheap and small computers, IoT devices, and API services. The development in equipment in the last decade is changing the game in how these kind of systems can be setup.

The following sections will go into more detail about different control methods, IoT devices and services, and home automation software.

2.1.1 Control systems and algorithms

The most common controller for heating systems is the thermostat, and it is the one most people see in their homes. A basic thermostat compares temperature measurements to the temperature reference set by the user and regulate the temperature turning a heating element on or off. The thermostat has long been the go-to controller due to its low cost and simplicity in setup and installation. However, the energy efficiency of the thermostat is not that good due to oscillating and overshooting temperatures [20].

Next, we have the classical controllers: Proportional-Integral (PI) and Proportional-Integral-Derivative (PID) controllers. These controllers have been used in both research and industrial buildings for several decades and were among the first controllers that were used in research on energy efficiency in buildings [15]. The PI and PID controllers provide a temperature regulation which has a better perceived comfort and is less energy-consuming than the thermostat due to a more precise control, giving a less oscillation and over-shooting temperature regulation. These systems are more complex than thermostats because the actuators need to have the ability for more precise control, and you need to tune the PI/PID systems parameters to fit well with the building. And while they have shown to reduce energy consumption and increase comfort, there are several limitations to these controllers: They can have a hard time handling multiple input and outputs [9], they cannot consider variables like outside temperature forecast without adding another system on top (like a feedforward system), and unexpected disturbances might cause the system to become unstable [15].

In recent decades there has been a bigger focus on more advanced control methods involving MPC, adaptive control, and AI. The MPC scheme was originally formulated in the 1970s, and is extensively used for process control applications today. MPC takes another route compared to the classical controllers: Instead of continuous feedback, the MPC uses a dynamic temperature model to try and predict the temperature trajectory and makes an optimization based on a cost function. This cost function is defined freely, and might include decision variables like weather forecasts or spot prices. Several benefits have been shown for controlling a buildings energy system with MPC: Energy savings, robustness against disturbances, multivariable control, and future control actions prediction [15]. The downside of MPC is that you need to make sure that you have a good dynamic temperature model of the building, and this is not a trivial procedure. There are a range of different methods for generating such a model, from purely mechanical based models to black-box models which is derived from AI methods. Finding purely mechanically based models are time-consuming and complex, and modeling softwares (e.g. Transient System Simulation Tool (TRNSYS)) are often used for this purpose. On the other hand, black-box methods rely on having good data, which can be time-consuming and hard to collect. A common method for generating a model is to do something in between, e.g. defining a model structure and using data to identify the parameters that is specific to the building. Whatever method you choose, the better the model, the better the MPC will perform.

In regards to MPC, an interesting and relevant study was made in Québec by École Polytechnique Montréal [9]. In this study they used an MPC scheme to control electric heating in a residential house. They used a quite complex system with a total of 17 temperature sensors, 13 power sensors, additional sensors for ambient and solar radiation, and 12 baseboard heaters. In addition, the occupant wrote a detailed logbook for events and thermostat settings. They created a detailed TRNSYS model with 17 different zones of the house, then validated the model parameters in each zone with the help of the sensors. The MPC optimized against using power in the peak power demanding periods between 5:30-9:30 and 16:30-20:30, while providing enough comfort by reaching a set temperature reference. The results showed that they were able to reduce power in peak power periods by $\frac{2}{3}$ and a total reduction of 16%. This shows both the power of MPC, and the complex model that might be needed for good results.

AI methods has also been a topic for research in energy management [15]. These systems rely on a large quantity of good quality data to be able to make good models. Since buildings are quite complex, many sensors and scenarios are needed in order for the system to learn anything about specific parameters of the buildings temperature dynamics. An example of these scenarios is to turn off all actuators except one, and then let that run for a while in order to gather data that can help the system generate parameters for this specific actuator. According to the research, the efficiency of these purely AI-based systems is uncertain.

To summarize, thermostats and classical control methods are still the most popular choice in building temperature regulation because of their low cost and simplicity of implementation. However, due to their low energy efficiency and their limitation on decision variables, more attention is turning to advanced controllers such as the MPC, where it is easier to add extra decision variables like weather, price, and peak power period forecasts. While these can be an alternative to the conventional methods, they do require a good quality dynamic model and are more complex to implement. The research using MPC has so far used complex and expensive systems which are hard to implement on the average existing house in a simple manner.

2.1.2 Internet of Things (IoT) devices for energy management

The last decade has seen a large development in the IoT industry. IoT devices now makes it easier than ever to connect devices to the internet. In the case of energy management of houses, this means connecting temperature sensors, heat pumps, charging stations, and so on. IoT devices are now relatively cheap and easy to use due to the vast commercialization in this market. Many big home-appliance companies are jumping on the wave of the connected devices as well, offering new smart versions of traditional products like washing machines, fridges, and air conditioners. Since the devices are connected to the internet, the management of the devices can be outsourced to any computer which has an internet connection. The usual management solution for these commercial IoT services are mobile apps and Hypertext Transfer Protocol (HTTP) APIs.

In this section we will discuss the new Smart Meters in houses, and some of the IoT devices used for energy management in smart houses today. The list is not exhaustive, but gives a glimpse into the current trends of home automation. Photos of these IoT devices are shown in fig. 2.1.

- **Advanced Smart Metering** — By 1 January 2019, all electricity consumers in Norway have received smart electricity meters (Advanced Metering System (AMS)) [13]. Among other things, these smart meters gives access to hourly electricity consumption and provides a HAN-port (Home Area Network) where the user can connect to gain information about their electricity consumption. This information includes real-time electricity consumption, consumption the last hour, and surplus power (e.g. from solar panels).

- **Tibber Pulse** — Tibber is a digital electricity company currently available in Norway, Sweden, and Germany. Instead of charging people extra money for electricity, like traditional electricity companies, Tibber profits by selling their IoT devices and services. These IoT devices and services takes advantage of the new capabilities of the smart meters (AMS) to give users more knowledge and control of their energy consumption. Tibber Pulse is an IoT device which connects to the HAN-port of the smart meter. It reads real-time electricity consumption and sends this information to Tibber every two seconds (over Wi-Fi). Tibber makes this information available through the their app and API.

- **Sensibo Sky and Sensibo Air** — Sensibo Sky and its newer version Sensibo Air enables Wi-Fi based control of most air-based heat pumps and air conditioners. The device communicates with the heat pump through Infrared (IR) signals (the same signal channel as the remote control) to gain access to heat pump settings and control. And through the sensibo app and API you can request data or send control instructions.

- **Easee Home** — Easee Home is is a smart home charging station for the electric car. It will automatically detect what kind of electricity your car needs and has automatic queueing between vehicles. With the Easee app you can give manual control instructions or create schedules for when to charge.

- **Mitsubishi Electric MELCloud Wi-Fi Adapter** — This adapter can be connected to Mitsubishi heat pumps to give you access to basic data and control through the MELCloud app.

- **Tado°** — Tado° makes smart thermostats and water-based radiator knobs. They also offer an IoT device for controlling the Air Conditioning (AC) pumps using IR signals, similar to the Sensibo Sky. With the app from Tado° you can control the heating system and create specific control schedules. They have an automation feature which, if turned on, uses outdoor weather forecasts to automatically reduce heating if the sun is warming up the home.

- **Ngenic Tune** — Ngenic Tune is an IoT system for controlling water-based central heating (such as ground source and electric water heaters). It uses a sensor to measure temperature inside the house and gains outside temperature data through the heating system itself. The system uses indoor and outdoor temperature data to control the heating by manipulating what the heating system thinks the outdoor temperature is. Through the app you gain information about the current measurements, historic data, and the ability to set temperatures and schedules. You can also activate the *spot-price adaptation* feature, where the system automatically tries to use less energy when the prices are high.

- **NIBE Uplink S-series** — NIBE makes water heaters and central heating systems. They have now launched an S-series, which offers connected versions of their ground source heat pumps and and air-water heat pumps. These are connected to the Wi-Fi and you can control these devices through the accompanying app. The app includes a smart control automation feature which attempts to use less energy when the prices are high.

- **Mill Wi-Fi heaters** — Mill creates various heaters available for smart control through Wi-Fi with the Mill app. You can have many heaters and homes registered on the app, making it easy to turn on and off the heating in homes when you are not there. You can also create your own control schedules.

2.1.3 Home automation software and services

Most commercial IoT devices comes with their own apps and services. However, there are a few generalized software's which were created for the purpose of combining the data and control of IoT devices from different companies into one place. These softwares are often used in home automation systems for combining and synchronizing IoT devices for various automations. This chapter looks at some of these softwares. Again, this list is not exhaustive.

(a) Tibber Pulse, connected to an AMS.

(b) Easee Home.

(c) Mitsubishi Electric MELCloud Wi-Fi Adapter.

(d) Tado° smart thermostat.

(e) Tado° smart heater knob.

(f) Ngenic Tune.

(g) NIBE Uplink S-series.

(h) Mill Wi-Fi heaters.

(i) Sensibo Sky.

Figure 2.1: Photos of the different IoT devices discussed in section section 2.1.2.

- **Home Assistant** — Home Assistant is an open-source software for home automation. It provides a simple interface for integrating IoT devices and creating your own automation schemes. What makes Home Assistant popular is that you can integrate an incredibly large amount of IoT devices.

  Home Assistant needs to be deployed on some type of microcontroller or computer, and normally Raspberry Pi's are used for this purpose. Home Assistant automatically creates a web server on the unit which is accessible through the local Wi-Fi. You can then access the Home Assistant dashboard through either a web browser or the Home Assistant app.

  Compared to Tibber app, Apple HomeKit, and similar, HomeAssistant gives you more control over your IoT devices and how you automate things. Home Assistant gives you the ability to customize your own dashboards for showing specific data, and to create your own automation schemes based on different integrations. For example, you can combine an integration for outside weather together with an integration for inner heating and create your own automation scheme that controls heating based on the weather.

  Similar software's include OpenHAB and Gladys Assistant, which both are also open-source software for home automation, running on units like the Raspberry Pi.

- **Tibber app** — While not as generalized as Home Assistant, the Tibber app integrates with many IoT devices by other companies and provides interfaces for these in their app. For many of these devices, Tibber also have automation features which tries to save you money and energy. For example, by connecting the Easee Home charging system to the Tibber app, the app gives an option to automate charging when the spot prices are low. It has similar automations for other IoT devices. Several of the IoT devices mentioned in section 2.1.2 can be integrated to the Tibber app.

- **Apple HomeKit and Google Home** — Apple and Google have their

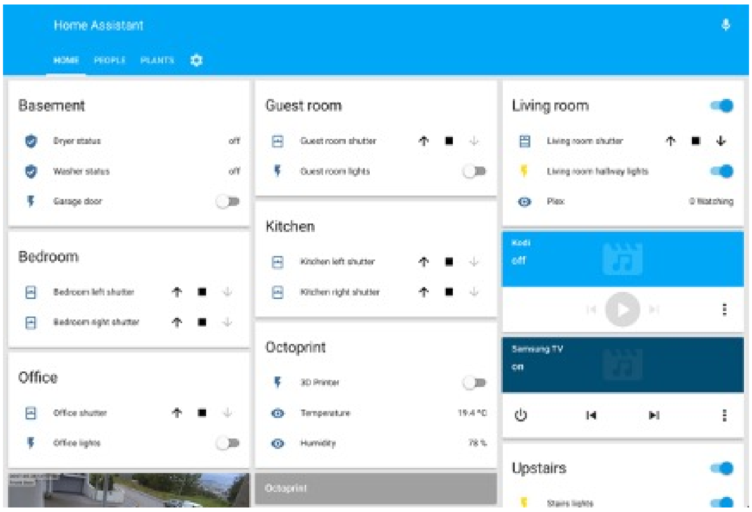


Figure 2.2: Home Assistant dashboard.

own app for home automations as well, with the ability to integrate many different IoT devices into their app. With these apps you don't need an extra computer unit, you just need the app. But you do need to make sure that the IoT devices you want to use are compatible with the respective apps. Compared to Home Assistant, these apps provide fewer compatible integrations and less opportunity for your own customizations and automations.

- **Creating your own software** — Most of the IoT devices provides an API where you can access data and control units. These APIs can easily be used in programming scripts, and many hobbyists create their own automations and simple softwares with these APIs. This is the ideal method for people who want full control over the devices and their automations.

2.1.4 Summary, current challenges, and gaps in the research

While the most popular heating controller still is the thermostat due to its simplicity and low cost, research on reducing energy using other controllers have been done since the 1980's, and possibly earlier. It started with classical PI and PID controllers, with more advanced controllers like MPC and AI-based methods in recent decades. Traditionally these systems have required expensive equipment and complicated setups. Getting the model of a building is not trivial and either requires detailed physical modeling or the usage of good data to estimate a model with SYSID methods. As we saw in the study from Quebec [9], they got great results using an MPC, but only with an extremely detailed TRNSYS model and over 30 sensors. Creating a good model and getting good measurements might be the main concerns for an MPC scheme.

The systems in past research have generally been integrated with dedicated hardware and software in a laboratory setting. Little previous research have focused on creating a cheap system which can be implemented for existing houses for the average person.

The emergence of IoT devices and services in the latest decade might make it possible to implement these kinds of systems (like MPC) in a simpler and cheaper way. While there exist many automated services for IoT devices that claim to reduce energy consumption (e.g. Tibber claim a 9.3% reduction in energy consumption using their smart heating algorithm with Tibber and Sensibo Sky [23]), we have not yet seen much published research involving IoT devices in heating management. These services seems to automate mostly in the same way, by using the spot market prices (which are publicly available from Nord Pool API [12]) and reducing energy consumption when the price is high. Some of them also use outside temperature and weather forecasts for their automations. Since using an advanced scheme like MPC (and similar) requires a good model of the building, one can guess that these companies use simpler algorithms to achieve these automations (since these systems are basically plug-and-play for any house). Many of the IoT services are different sides of the same coin: they provide a Wi-Fi connected sensor or control unit which is controllable through an app. There seems an increased trend in adoption of smart appliances in general.

We also see home automation software's like Home Assistant, which gives the ability to combine IoT devices and creating your own automations in a generalized way. Since these IoT services often provide an API interface that you can use in your own scripts, it is also possible to create your own specialized software where you have full control.

There is a clear gap in the research regarding heating energy management: To investigate the usage of cheap IoT solutions together with advanced algorithms (like MPC) to control heating in smart homes. These kind of systems would be cheaper and more easily be implemented on already existing homes. For example, the research mentioned earlier using an MPC on a house in Quebec shows great results, but the equipment and installation are very complex. If an MPC could be implemented with easily available IoT devices, in already existing homes, that would have a lot of potential for saving energy. It's a new angle on old research, and this is something worth investigating.

2.2 Theory

In this section the central methods used in the system presented in this thesis will be explained. The focus in this section is to generate a basic understanding for people who have not been exposed to these methods before. The deep theory behind these methods will not be presented, as this is not the focus of this thesis.

The methods to be discussed are: System Identification (SYSID) in section 2.2.1, building model generation in section 2.2.2, Moving Horizon Estimation (MHE) in section 2.2.3, Model Predictive Control (MPC) in section 2.2.4, and shortly about numerical optimization in section 2.2.5.

2.2.1 System Identification

In essence, system identification is the science of constructing mathematical dynamic models from observed data with the help of statistical methods. Often you define the structure of the model yourself, which includes undecided parameter variables. For example, see eq. (2.2.1), which defines a simple model structure of the dynamic temperature inside a room, where the parameters a, b, and c are undecided, T_{in} is the temperature inside, T_{out} is the temperature outside, and $P_{heating}$ is the power given to the heating element. The SYSID algorithm estimates the undecided parameters a, b, and c with the help of the measurement data you provide (in this case the measurement data includes T_{in}, T_{out}, and $P_{heating}$). This is different from classical modeling, where you build the model from physical laws and figure out the specific parameters yourself (mass, sizes, and so on). Figure 2.3 gives a simple illustration of the principle of SYSID, where the parameters of the self-defined model is estimated by using measurements and input data.

The parameters are generally estimated by solving an optimization problem with a cost function that wants to minimize the differences between the measured data and what the model estimates. This often simplifies to a least-squares problem, as seen in eq. (2.2.2).

$$\dot{T}_{in} = aT_{in} - bT_{out} + cP_{heating} \tag{2.2.1}$$

$$\text{minimize} \sum_{i=1}^{k} \frac{1}{2}(y_{measurement} - y_{model})^2 \tag{2.2.2}$$

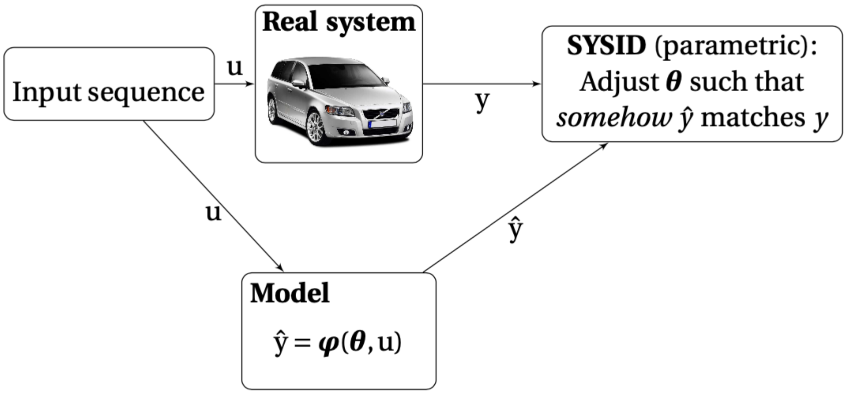


Figure 2.3: Illustration that shows that system identification tries to estimate parameters in a self-defined model to match the real-world system as close as possible.

Source: Sebastien Gros Lecture notes [7]

A general procedure:

1. Find out what you can and want to measure from the real system and start collecting data.
2. Define the structure of a model which includes symbolic representations of these measurements.

3. Formulate an optimization problem that minimizes the difference between measurements data and what the model estimates.

4. Solve this optimization problem to generate numeric parameters for the model.

There are different types of SYSID, often categorized from grey-box to black-box (with white-box being classical modeling from physical laws). In grey-box you start with a self-defined model, while in black-box the stucture of the model is estimated as well.

In this project, grey-box SYSID is used. A model structure is defined and measurement data of temperatures (inside and outside), heat pump settings, and power usage is used to generate the parameters of this model.

Considerations for this project

- Collecting good quality data is extremely important for generating a good model.

- Quantity of data is also important. The more data points that are available for the optimization procedure, the better the model accuracy will be.

- Different models should be tried to see what works best. Choosing the best model might not be obvious.

2.2.2 Generating a mathematical model for a building

This section explains the principles behind the model generation method that was used for the model in this thesis. The model used in the system of this thesis was developed by Sebastien Gros.

Generating dynamic temperature models for buildings is a complicated problem. Buildings have different structures, isolations, windows, sun exposure, heating actuators, and so on. In this project we would like to have a simple model that can be generalized to all types of residential houses. But it is important to realize that such a model will have limited validity, and to estimate parameters and validify the model accuracy of such a model, we need a lot of *consistent* and *good quality* data.

It is easier to estimate good parameter values if data are collected under specific scenarios, such as just having one heat pump running at a time. Just collecting data without having any information about what is going on in the system during that time will produce data that are less useful for SYSID.

Model structure

The following is a basic interpretation of defining a model structure for a room. These are the principles behind the model used during the tests in chapter 7.

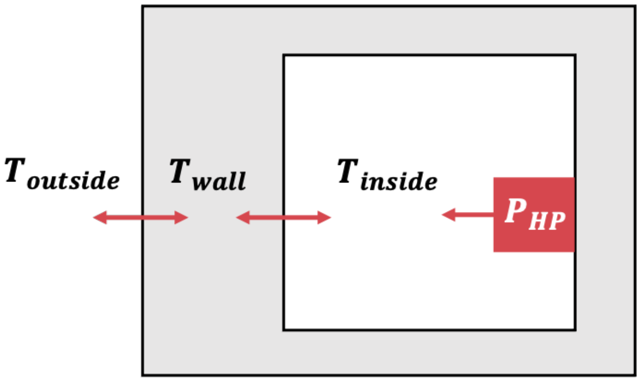


Figure 2.4: A basic illustration of the modelling problem of a building. T_{inside}, T_{wall}, and $T_{outside}$ represents temperatures and P_{HP} is the heating power from the heat pump. The arrows indicate the flow of energy.

Figure 2.4 illustrates a simplification of the modelling problem of a single room. The temperature inside T_{inside} is changing based on the heat pump power and the temperature of the wall T_{wall}. The temperature in the wall is changing based on the difference between the temperature inside T_{inside} and the temperature outside $T_{outside}$. In reality these temperatures would have a gradient over the inside room and the wall, and there would also be windows which has different properties than the wall.

In eq. (2.2.3) this is defined a mathematical model:

$$\begin{aligned}\dot{T}_{wall} &= \theta^{wall}_{gain}(T_{inside} - T_{wall}) - \theta^{wall}_{loss}(T_{wall} - T_{outside}) \\ \dot{T}_{inside} &= \theta^{inside}_{loss}(T_{inside} - T_{wall}) + \theta^{inside}_{gain}P_{HP}\end{aligned} \tag{2.2.3}$$

Where θ^{wall}_{gain}, θ^{wall}_{loss}, θ^{inside}_{loss}, and θ^{inside}_{gain} are unknown parameters that represent how big the room is, the effect of the heat pump, the thickness and material of the wall, and so on.

We don't have a sensor for the individual power consumption of the heat pumps, so these needs to be estimated. A basic model for the heat pump power can be seen in eq. (2.2.4):

$$P_{basic} = \theta^{pump}_{gain}(T_{target} - T_{inside}) + b \tag{2.2.4}$$

Where θ^{pump}_{gain} is heat pump specific parameters, and b is a heat pump specific constant. T_{target} is the temperature setting on the heat pump, and serves as the input to the system. However, this model does not take into account that the heat pump has a maximum power capacity, and that the power cannot be negative. To address this, two nested SmoothReLU [18] functions are utilized, first to remove negative values, and then to provide an upper power limit:

$$\begin{aligned}P_{no-neg.} &= \frac{ln(1 + e^{kP_{basic}}))}{k} \\ P_{HP} &= \frac{p_{max} - ln(1 + e^{k(p_{max} - P_{no-neg.})})}{k}\end{aligned} \tag{2.2.5}$$

Where k is a *sharpness* parameter, making the functions curve around zero and p_{max} smoother, and p_{max} is the maximum power the heat pump can output. P_{no-neg} represents P_{basic} with no negative values, and P_{HP} represents P_{basic} with no negative values, and with an upper power limit p_{max}. An illustration of how the nested ReLu function affects the power model is shown in fig. 2.5.

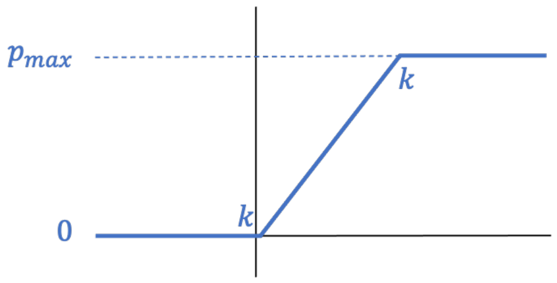


Figure 2.5: Illustration of the lower and upper limits that is created with a ReLu function, where k dictates how smooth the transition is between the limits and the function.

Estimating the parameters

To estimate the parameters θ_{gain}^{wall}, θ_{loss}^{wall}, θ_{loss}^{inside}, θ_{gain}^{inside} and θ_{gain}^{pump}, the following cost function is solved:

$$\text{minimize} \sum_{i=1}^{k} \frac{1}{2} (T_{inside,measurement} - T_{inside,model})^2 \tag{2.2.6}$$

Where k is the number of data points used. $T_{inside,model}$ is based on the model mentioned in eq. (2.2.3), which includes the model for estimating power. The input to the system is in the form of the temperature setting on the heat pump, T_{target}, and $T_{inside,measurement}$ are measured by the Sensibo Sky device.

The constraints were based on a fixed time grid of 5 minutes:

$$\begin{aligned} \dot{T}_{inside} &= T_{inside,k+1} - T_{inside,k} \\ \dot{T}_{wall} &= T_{wall,k+1} - T_{wall,k} \end{aligned} \tag{2.2.7}$$

2.2.3 Moving Horizon Estimation

Moving horizon estimation (MHE) is an optimization-based method for estimating the current state of the system based on a finite sequence of N past measurements and information from the dynamic system model. MHE uses a *sliding* *time window*, meaning that at each sampling time a new measurement is added and the oldest is removed, still maintaining N number of measurements.

The solution comes from formulating an optimization problem where:

- The calculated trajectory is constrained to conform to the dynamic model of the system.
- The initial state of the calculated state sequence is coherent with the estimate from the previous sampling interval.
- The cost function tries to minimize the difference between the computed trajectory and the measurements.

The calculated state for the latest (current) interval becomes the state estimate.

Considerations for this project

- The Moving Horizon Estimation (MHE) optimization problem is solved at each sampling instance to supply the MPC with state estimates.

2.2.4 Model Predictive Control

The MPC principle is explained well in the following quote by Mayne et al. [16]:

> Model Predictive Control (MPC) is a form of control in which the current control action is obtained by solving, at *each* sampling instant, a finite horizon open loop optimal control problem, using the current state of the plant as the initial state; the optimization yields an optimal control sequence and the first control in this sequence is applied to the plant.

The MPC uses a mathematical model to predict and optimize the state trajectory for the next given time horizon, using the latest (current) measurement as the initial state and optimizing for a cost function and given constraints. The optimization will generate an input sequence for the whole trajectory, but only the first one is implemented to the system. At the next sampling time the optimization problem is solved again. The sampling time is usually shorter than the time horizon (e.g. 5 min sampling time and 10 hour time horizon).

An example of a simple cost function is to minimize the difference between the current state and the reference state we want to go to. And a simple constraint is that the input can not be over or under a certain value.

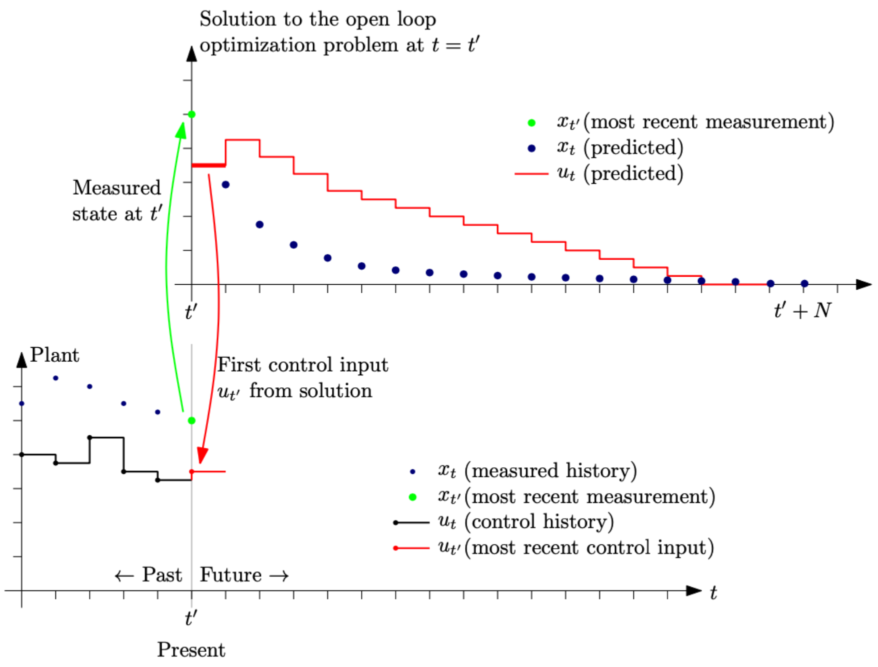
A basic MPC algorithm is given in algorithm 1 and the principle is illustrated in fig. 2.6.

Algorithm 1: State feedback MPC procedure

```
for $t$ = $0,1,2, \ldots$ do do
    Get the current state $x_t$.
    Solve a dynamic optimization problem on the prediction horizon
      from $t$ to $t+N$ with $x_t$ as the initial condition.
    Apply the first control move $u_t$ from the solution above.
end
```

Considerations for this project

- MPC schemes can be processing intensive since it has to solve an optimization problem at each iteration. Care should be taken that the processing unit can handle it.
- The model dictates how well the MPC can predict the temperature trajectory, therefore we need to make sure we have a good model in order to get good results from the MPC.



Solution to the open loop optimization problem at $t = t'$
$x_{t'}$ (most recent measurement)
x_t (predicted)
u_t (predicted)
Measured state at t'
t'
$t' + N$
Plant
First control input $u_{t'}$ from solution
x_t (measured history)
$x_{t'}$ (most recent measurement)
u_t (control history)
$u_{t'}$ (most recent control input)
← Past
Future →
t
t'
Present

Figure 2.6: Illustration of the MPC principle. The most recent measurement (green dot) can be a direct measurement or an estimation. In this project the latest states will be estimated with Moving Horizon Estimation.

Source: Bjarne Foss and Tor Aksel N. Heirung [5]

2.2.5 Numerical Optimiziation

Numerical optimiation has not been a focus of this thesis, but CasADi has been used as a tool for defining and solving optimization problems for SYSID, MHE, and MPC.

CasADi

CasADi [2] is an open-source framework for algorithmic differentiation and non-linear programming. It is used for formulating and solving optimization problems.

CasADi is interfaced with different solvers, which are different ways to reach a solution to the optimization problem. **Ipopt** is a primal-dual interior-point solver which was used in this thesis when testing the MHE/MPC scheme on the Raspberry Pi B.

2.2.6 Utilization-based schedulability test

For a processor and a given set of periodic tasks, a utilization test considers how much of the processor is utilized with these tasks. The result of the test tells you if the set of tasks can successfully run on the processor or not.

The following equation is a utilization test is for Fixed-priority scheduling with rate monotonic priority [1]:

$$U \equiv \sum_{i=1}^{N} (\frac{C_i}{T_i}) \leq N(2^{\frac{1}{N}} - 1) \tag{2.2.8}$$

Where C_i is the computation time of task i, T_i is the period of task i, and N is the total number of tasks.

This test is used in real-time systems analysis to make sure that all tasks reaches their deadlines before the next task is released. The computation time is based on the Worst Case Execution Time (WCET). It provides a *sufficient*, but not *necessary* upper-bound on the utilization. Meaning that if the test holds the tasks are guaranteed to be scheduled, but if the test fails, the tasks may still be schedulable.

The test is based on the following assumptions:

- The application is assumed to consist of N tasks
- All tasks are periodic with known periods
- The tasks are completely independentof each other
- All tasks have a deadline equal to their period
- Each task must complete before it is next released

- All tasks have a single fixed WCET
- All system overheads (context-switch times, interrupt handling and so on) are assumed absorbed in the WCETs

Considerations for this project:

- While the test is generally used for tasks with strict deadlines, the tasks in the software in this thesis does not have strict deadlines. The tasks only have to complete *at some point in time.* In other words, the system in the thesis fails if new tasks are supplied to the processor at a faster rate than the tasks are completed, but the tasks does not have to strictly complete before the next one arrives. Therefore, the assumption is that test is still *sufficient* for the case in this thesis, since it has looser rules that would be easier to schedule.
- While the test uses the tasks WCET as computation time, the thesis will use the tasks *average computation time*, as the tasks does not have strict deadlines, but just have to complete *at some point in time.* This average will be based on gathering timing data of these tasks.

Chapter 3

System and Software specification

This chapter defines the requirements of the system and software that we are implementing in this thesis. These requirements will form a structure around how we design and think about the system in later chapters. Section 3.1 describes an overview of the system and how it can be abstracted in different layers. Sections 3.2 to 3.4 identifies and presents the requirements of each of these layers.

3.1 System overview

The system we are investigating in this thesis is based around using cheap and easily available IoT and computing devices to implement an MPC algorithm that controls the heating in a house. A model is generated by SYSID, states are estimated with MHE, and the MPC uses this model together with a cost function to optimize for both comfort and reduced cost.

This system can be abstracted into three layers: Data collection, data processing,

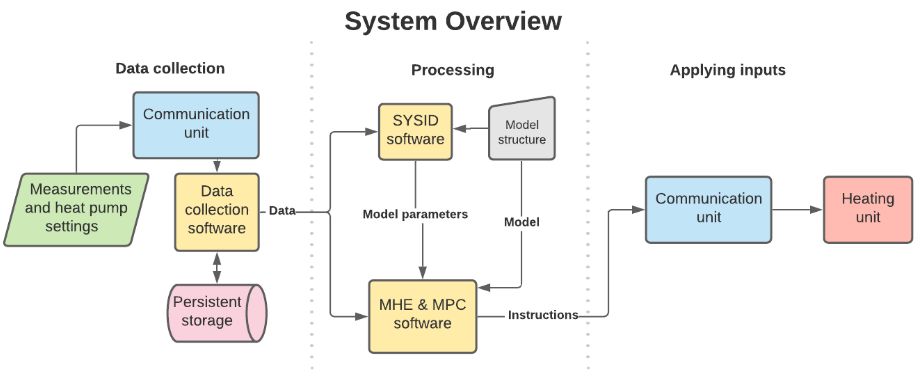


Figure 3.1: Basic illustration of the three layers in an automatic heating system.

and applying inputs from the algorithm. These layers are visualized in fig. 3.1, where the name of each layer is mentioned at the top of the figure, and the arrows indicate the flow of data. The shapes and different colors indicate different types of units or processes within each layer, and the text on the arrows indicate what type of data is sent from one component to the next. Collecting data involves having sensors in place, being able to communicate with them, and having the ability to store persistent data for later access. Processing data involves a computation from data to output through SYSID, MHE and the decision-making MPC algorithm. Applying inputs involves taking this output and implementing them on the system by sending instructions to heating actuators.

The system is dependent on data and readily available measurements. Therefore, extra care needs to be taken around the data collection layer to make sure it is robust and reliable.

3.1.1 General system requirements

Since this system is going to control the heating in a building, the first requirement is that the overall system needs to be reliable and able to continue working to some extent even in the face of unexpected errors. It is also important that this system is cheap and accessible to existing homes.

General System Requirements (GSR):

- **GSR-1:** Accessible to regular homeowners
  - **GSR-1.1:** Relatively cheap equipment
  - **GSR-1.2:** Commercially available devices
  - **GSR-1.3:** Possible to install on existing non-smart homes
- **GSR-2:** Reliable uptime and fault tolerant
  - **GSR-2.1:** Resilient against internet disconnect
  - **GSR-2.2:** Resilient against system crashes and restarts
  - **GSR-2.3:** Persistent storage of data
  - **GSR-2.4:** Tolerant against data races and data corruptions
  - **GSR-2.5:** Reliable communication
  - **GSR-2.6:** Reliable processing
- **GSR-3:** Error logging for debugging

General System Wants (GSW):

- **GSW-1:** As small and fast codebase as possible
- **GSW-2:** Minimize the dependency on third-party libraries and tools
- **GSW-3:** The code and usage-guide should be well documented so other people can continue research with this system after the thesis project is completed.

3.2 Data collection layer

The data collection layer has the job of polling the sensors for measurements and storing this data in a persistent storage, making it available for later use by the processing layer. Then for the processing layer to work well it requires accurate measurements and low sampling-times. For example, a temperature measurement is not useful if it is off by 5 degrees and you can only read the measurement once every two hours. The data collection needs to run day and night and the data needs to be available for processing at any time. This layer also requires safe handling of the data files in terms of data-races, over-writes, file corruption, and storage crashes.

For the specific measurements, the data the system needs depends on the dynamic model structure and the cost-function of the MPC. At the least, we need to know the indoor temperature, outdoor temperature, and the power usage of the heat pumps.

Data Collection Layer Requirements (DCLR):

- **DCLR-1:** Accurate measurements
- **DCLR-2:** Relatively low sampling times
- **DCLR-3:** Sensors and APIs for measurements and data used in the SYSID model and the MPC algorithm
- **DCPR-4:** Continuously collect data day and night
  - **DCLR-4.1:** Automatic restart on crashes
  - **DCLR-4.2:** Ability to handle disconnects
- **DCLR-5:** Robust against data-races, over-writes, file corruption, and storage crashes
- **DCLR-6:** Ability to easily customize how data is collected (to some extent)
  - **DCLR-6.1:** Easily change sampling times
  - **DCLR-6.2:** Turn on or off the data collection from a sensor
- **DCLR-7:** A way to access this data from another process

Data Collection Layer Wants (DCLW):

- **DCLW-1:** The data files should be easily accessible for use in various research work related to this project (such as research about model generation with SYSID).
- **DCLW-2:** Should be possible to add or remove sensors to this system relatively easily.

3.3 Processing layer requirements

This layer uses the stored data in heavy processing that involves SYSID, MHE and MPC algorithms. The output is a set of instructions to be sent to the actuators (heat pumps, in this case). The output becomes less and less useful the longer it takes to process the data, so therefore it is important that the processing is relatively fast compared to the temperature dynamics of the building. It is also important that the dynamic temperature model is fairly accurate, so the MPC has good accuracy on its temperature trajectory prediction. And lastly, the cost-function in the MPC needs to be tuned well in order for the MPC to optimize the trajectory correctly.

In addition, limitations to the system needs to be identified and added to the MPC constraints (such as maximum and minimum power output of the heat pumps).

Data Processing Layer Requirements (DPLR):

- **DPLR-1:** Fast enough processing (to finish before the next sampling interval and to not lose the usefulness of the output)
- **DPLR-2:** A model that performs well (accurately describes the temperature dynamics)
- **DPLR-3:** A well-defined optimization problem for the MPC
  - **DPLR-3.1:** Well-defined constraints of the system
  - **DPLR-3.2:** Well-tuned cost-function
- **DPLR-4:** Output format needs to conform to actuator input (heat pump settings such as temperature reference and heating mode)

3.4 Applying inputs layer requirements

This layer needs to take the output from the processing layer and apply it to the actuators. It requires constant uptime and readiness for relaying these instructions.

Applying Inputs Layer Requirements (AILR):

- **AILR-1:** Constant uptime
- **AILR-2:** Ability to relay the instructions from the processing layer to the actuators.

Chapter 4

System implementation

In this chapter we go from requirements to a specific design and implementation of the system, with specific IoT devices and a specialized software. In section 4.1 the overview of the system implementation is presented, including components and specific layer considerations. In section 4.2 the software design is presented.

4.1 System overview

The aim is to have a simple and cheap system that can be installed in a normal home. While the results may be better if there were 30 sensors, that many sensors and devices are too complicated and expensive. Instead it is desirable to have as few few devices as possible, with as simple installation as possible.

With that in mind, the system in this project is based on using existing air-based heat pumps, Sensibo Sky IoT devices for measuring temperature and interacting with the heat pumps, and Tibber IoT devices for accessing real-time power usage from the smart meter. A Raspberry Pi B computer was chosen as the *house hub*, which will collect measurement data, service data requests, and eventually run the MHE and MPC algorithm. The outside temperature is accessed from the

Norwegian Meteorological Institute (MET) API, and the spot prices is accessed from the Nord Pool API. More details about these API services are presented in section 4.1.1. All API calls are made through an internet connection and the interactions with the IoT devices requires Wi-Fi.

The processing layer (SYSID, MHE and MPC) have been running mostly on a separate processing unit (a separate laptop or computer) for the scope of this thesis. However, in the later stages of the thesis the MHE and MPC scheme was tested on the Raspberry Pi together with the main software (data collection, servicing requests, etc.) with good results. More on this in chapter 7.

With this setup you need the following:

- Heat pumps with IR-based remote control
- Tibber as your electricity provider
- One Tibber Pulse IoT device for each smart meter in the house
- One Sensibo Sky IoT device for each heat pump
- One Raspberry Pi B with SD Card (preferrably with both large RAM and storage)
- One processing unit (Separate laptop/computer or use the Raspberry Pi)
- Internet connection with Wi-Fi

This setup is relatively simple to install in a house, as we will see in the case study in chapter 6.

An overview of the system and its layers are visualized in fig. 4.1. The name of each layer is mentioned at the top of the figure. Arrows indicate the flow of information, where blue dashed lines indicate a flow over Wi-Fi and a red dotted line indicates a flow over an IR-signal. The shapes indicate different types of units or processes within each layer, and the explanation of these shapes are presented in the bottom of the figure. The layer connection between Data collection and Processing may or may not be a Wi-Fi connection, depending on if these layers are setup on the same processing unit (the Raspberry Pi) or not.

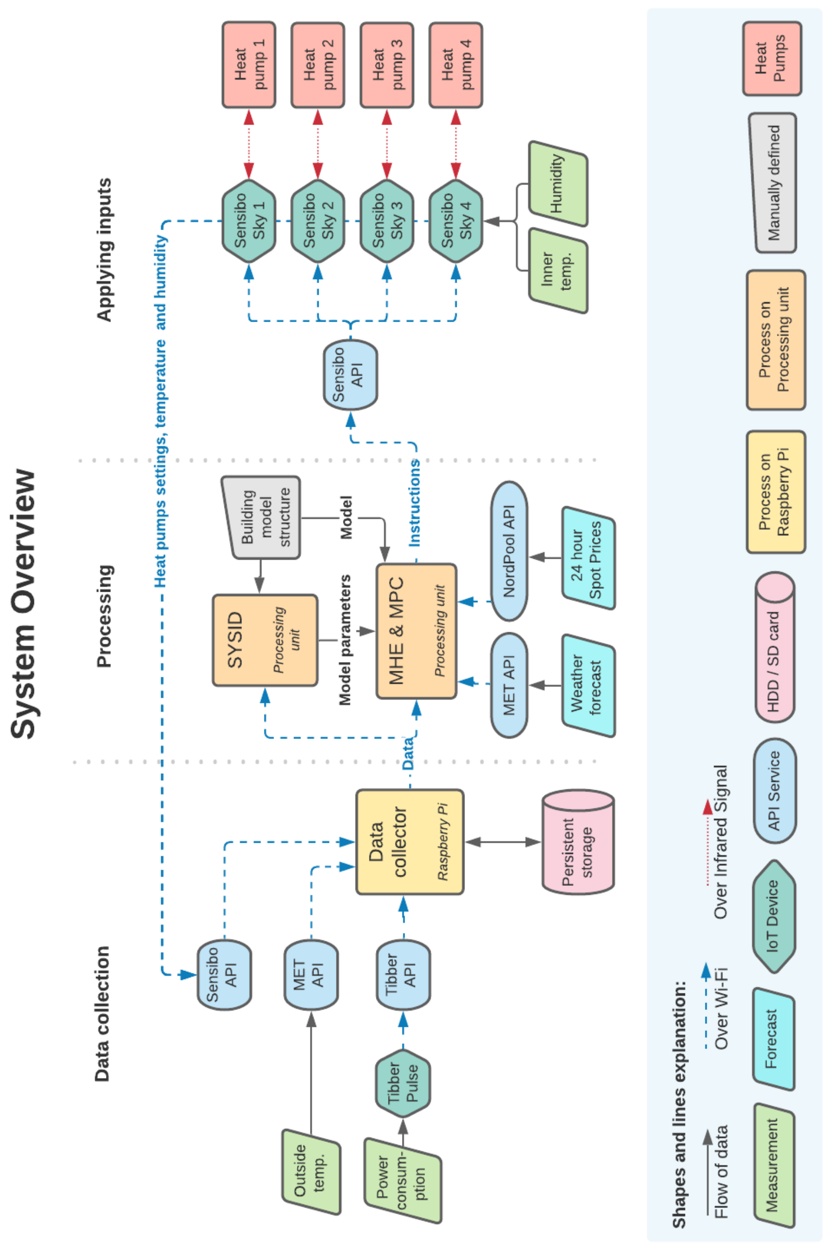


Figure 4.1: System overview, showing the abstracted layers of the system as well as all the components. Shapes and arrows explanation at bottom of figure.

4.1.1 Components

This section discusses the components used in the system. The components include heat pumps, IoT devices from Tibber and Sensibo, API services from MET and Nord Pool, the Raspberry Pi as a house hub, and the additional processing unit (laptop or other computer).

Heat pumps

Heat pumps with IR-sensor acts as the actuators. They are common in residential houses and are therefore a good choice for serving as heat actuators, since no additional actuator installations are required. With the IR-sensor it can easily be setup to communicate with an IoT device like Sensibo Sky. So, in terms of system simplicity, heat pumps are a great option.

However, the downside to heat pumps are that the power consumption is not straightforward. When a heat pump is running the power will sometimes be all over the place, often cycling from high to low power consumption. Since power consumption is used as a decision variable in the MPC, this property of heat pumps is not ideal. In addition, power usage of each heat pump can't be directly measured (only the combined power consumption of the house), so power consumption of each heat pump has to be estimated. The model used for estimating power in the case study of this thesis is presented in chapter 6.

Another complication with heat pumps is that the Coefficient Of Power (COP) depends on the outer temperature, so on colder days the heat pump will use more power to heat the inside rooms. In addition, heat pumps use *defrost mode* during freezing days in winter, which would increase the power consumption further during freezing days. Neither the COP or *defrost mode* was modeled in the power estimation model used in the case study.

Tibber and Tibber Pulse

For accessing real-time power consumption, Tibber with the *Tibber Pulse* IoT device is used. Tibber provides a simple setup process: supply tibber with your smart meter number, plug in the Tibber Pulse in your smart meter HAN-port, and setup Wi-Fi settings through the tibber app. With this done, data is collected from the smart meter and sent to Tibber's servers over a Wi-Fi internet connection. You can then access the data through their app or API. The standard Tibber service provides hourly consumption and cost data, while the addition of Tibber Pulse gives access to real-time power consumption which is updated every two seconds.

At this point in time, Tibber is the only good option in Norway for easy access to real-time power consumption data.

Sensibo and Sensibo Sky

For interacting with the heat pumps we are using the Sensibo Sky IoT device. In addition to interacting with the heat pump, this device includes a sensor for measuring temperature and humidity. In all, Sensibo Sky serves three important functions: measuring temperature, accessing the heat pumps current settings, and sending control instructions to the heat pump. The control instructions are in the format of target temperature, fan level, and heating mode. Accessing the heat pumps settings is important for identifying the dynamic temperature model of each room through SYSID.

After a simple setup through the sensibo app, the Sensibo Sky device is connected to the local Wi-Fi and can start sending data or receive heat pump instructions through the Sensibo app or API. As with Tibber, the data is stored on Sensibo's own servers and we are accessing this data through the app or API calls. Sensibo updates the current measurements every 90 seconds. New control instructions can be sent at any time, and it is possible to send scheduled settings for a later time.

Norwegian Meteorological Institute's API service

The Norwegian Metrological Institute has an open API where anyone can access the current weather, forecasts, and historical weather data. This service is used to collect data about the outdoor temperature. In Norway this is the most accurate weather service available, and the current weather data is updated every hour.

Nord Pool's API service

Nord Pool is Europe's leading power market where electricity companies can buy and sell electricity. Every day at noon, Nord Pool sets the hourly spot prices for the next 24 hours. These spot prices can be easily accessed through Nord Pool's API, and in our system it is used within the cost-function in the MPC algorithm for the purpose of minimizing cost.

House hub (The central unit of this system)

This system needs a dedicated machine which has the responsibility to collect data and handle data requests from the processing layer. For this job a Raspberry Pi Model B with 8GM RAM was chosen. This is a device that is commonly used for dedicated purposes like home automation. This device is a cheap and small computer, but with less processing power than your typical laptop or desktop computer. The Raspberry Pi can install linux-based OS's, and has a network card for connecting to the internet with Wi-Fi or ethernet cable. It also has GPIO pins, which makes it possible to connect sensors or other devices directly by wire to the Raspberry Pi. GPIO pins is not used in this thesis, but the possibility might be good for future iterations of our system.

Processing unit

For the scope of this thesis, the processing layer was outsourced to a separate processing unit (a laptop) while running a closed-loop operation. However, as mentioned earlier, a test was made in the later stages of the thesis project, where

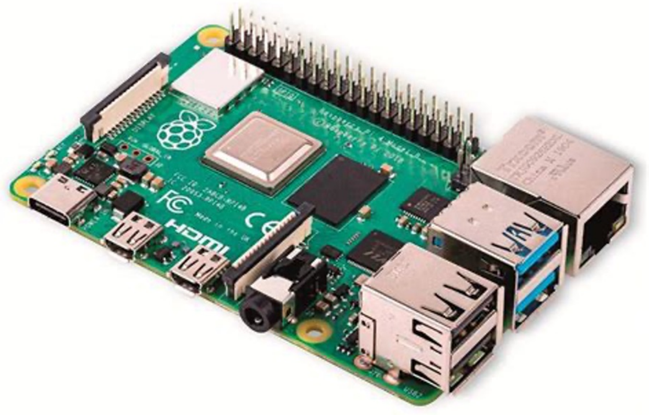


Figure 4.2: The Raspberry Pi B computer.

the MPC was installed and ran on the Raspberry Pi with good results. More on this in chapter 7.

4.1.2 Measurements and data collection

Measurements include inside temperature, outside temperature and power consumption (table 4.1). These are accessed by API calls to Sensibo, MET, and Tibber, and used for both the model generation in SYSID and as latest measurements in the /MHE and MPC algorithm. Power consumption is measured as the total consumption per smart meter, so individual pump power consumption needs to be estimated. Non-measurement data include heat pump states, weather forecast, and spot prices (table 4.2). These are accessed by API calls to Sensibo, MET, and Nord Pool. Heat pump states is used for the model generation in SYSID, while weather forecast and spot prices are used for the cost-function in the MPC.

Solar radiation (from Solcast [21]) is another relevant measurement that was initially planned to be included in the system, but due to unstable API requests it was chosen that it would not be included in the scope of this thesis. Solar radiation would give an additional term that takes into account the warmth that is generated from the radiation of the sun.

While additional sensors would give more accurate temperature models and predictions, this would also complicate the system and make the investment cost higher, which we want to avoid.

As for outside temperature, the best accuracy would of course come from a sensor device directly outside the house. Netamo [10] has such a device, but it is expensive and has varied reviews. For the scope of this thesis the measurement from MET is believed to provide a satisfactory accuracy, with the added benefit of not needing to have an additional physical sensor device in the system.

These measurements and data should be collected day and night and the latest measurements should be readily available for the MHE andMPC algorithm.

The data we collect is used for SYSID, MHE, and MPC. In future research projects, the data will also be used for Machine Learning (ML), where it will be investigated if better models can be derived using ML methods. Collecting data is a fundamental part of making the system work, and it is important that we create a good software for this.

Table 4.1: The measurements that are used in the system implementation of this thesis.

Measurement	Measured by	Accessed through	Used for
Inside temperature	Sensibo Sky	Sensibo API	SYSID, MHE/MPC
Outside temperature	MET	MET API	SYSID, MHE/MPC
Power consumption	Tibber Pulse	Tibber API	SYSID, MHE/MPC

Table 4.2: The data (non-measurements) that are used in the system implementation of this thesis.

Data	Generated by	Accessed through	Used for
Heat pump states	Heat pump	Sensibo API	SYSID
Weather forecast	MET	MET API	MPC
Spot prices	Nord Pool	Nord Pool API	MPC

Persistent storage

Persistent storage means that the data is stored on something which persists even if the power is lost. There are two important parts to persistent storage: what format the data is stored as, and on what hardware it is stored on.

There were several options that were considered when choosing the storage format: A database, python pickle files, json files, and .csv files. Python pickle files were chosen because of several reasons: you can store datetime format, the files can easily be moved around on a file system (useful for playing around with the data), and they are easy to store and load without any overhead system.

Using a database system would be a good option for a future iteration of our system iplementation, but it arguably requires too much of an overhead at this stage, when the pickle files provide good utility for playing around with data in SYSID.

When we are using pickle files for storage, we need to make sure we build a data storage system that is safe from data races, overwrites, and other potential data storage problems.

As for the hardware, the data files will be stored on the Raspberry Pi's hard drive, which is an 128GB SD card with Application Class 2. SD cards are less stable than normal hard drives, so there should be a backup system in place.

4.1.3 Processing and applying inputs to actuators

The data is used for three data processing methods, together these forms the processing layer in the system: SYSID, MHE, and MPC. The SYSID part is done "offline" while the system isn't running to generate the parameters for the dynamic temperature model structure. The MHE part is done just before each iteration of the MPC in order to estimate the current states before the MPC runs. The MPC part is running at a set sampling time, using the latest estimated states together with the dynamic model and parameters from SYSID.

The output from the MPC is an optimal setting for the heat pumps in the form of temperature reference and heating mode. These settings are relayed to the heat pumps through the Sensibo API and Sensibo Sky.

This section will not present the details about these procedures, as this thesis focuses more on the overall software than the details of the algorithms, but the section will present how these procedures affect how we design the software.

System identification

To generate accurate model parameters from SYSID, we are dependent on consistent and good quality data. There are several points that raises the quality of the data:

- It is important that the data points have time stamps, so the data points can be put in the correct place on the time grid.
- The time between each data point should be consistent, and there should not be any large gaps between clumps of data. Ideally the whole data set should have data points that are homogenously separated by the same sampling time over a long period of time.
- It is easier to estimate good parameter values with SYSID if data are collected under specific scenarios, such as just having one heat pump running

at a time. Just collecting data without having any information about what is going on in the system during that time will produce data that are less useful (the estimated parameter values will be less accurate).

We need to make sure we address these points in the software.

The model and parameters used in the case study part of this thesis is presented in chapter 6.

Model Predictive Control with Moving Horizon Estimation

In order for the MHE and MPC scheme to work well, we need to address several points:

- Just before each MPC interval, the MHE needs all measurements that has been collected since the last MPC iteration in order to estimate the current states. We therefore need a way to request this data from the data collection layer, and preferably in a way where you can request data since a specific timestamp (the last iteration).
- When running on the Raspberry Pi, the MHE and MPC scheme needs to be able to run concurrently with the data collection.
- The MHE and MPC needs to complete relatively fast compared to the interval time, in order for the output of the algorithm to not lose relevancy. However, this is a system with slow dynamics, so relatively fast in this context is somewhere below a minute or so.

The cost function and constraints used in the in the case study part of this thesis is presented in chapter 6.

4.1.4 Communication between components and layers

A big part of the system architecture is to combine all the components and enable interactions between the layers in the system. In this system all the interaction between the components will be made through an internet connection by making HTTP requests to the different APIs. While APIs already exists for the IoT devices from Tibber and Sensibo, we must create our own API for the *house hub software* on the Raspberry Pi in order to be able to request data and control the data collection from a different computer. This should also allow the possibility to change data collection settings (such as sampling times, location, etc.) by making calls from another device.

There are two main reasons for using internet and Wi-Fi for communication between components: It is easy to setup, and it enables control from outside the Local Area Network (LAN). Enabling control from outside the LAN might be useful in future research in the POWIOT project when several houses are negotiating over a collective electricity consumption.

API for the house hub (Raspberry Pi)

The main way to communicate with the Raspberry Pi's software will be through an API that we will design in section 4.2. This API will consist of several addresses, or *routings*, which anyone with an internet connections can send requests to in order to make the software do different things. So, the API is important for being able to communicate with the Raspberry Pi software without directly using the Raspberry Pi device.

The API will have several features:

- Starting, stopping and changing the data collection settings
- Request data for a specified date, or since a specified time

The final design of the API is presented in section 4.2.4.

4.2 Software design

In this section the software design is presented. Sections sections 4.2.1 to 4.2.3 presents the software tools that has been used as well as design considerations regarding performance and fault tolerance. In section 4.2.4 the code structure and design will be presented.

4.2.1 Software tools and libraries

The main software is written in Python and runs in a Docker container as a Flask application, with nginx and gunicorn as web proxy and web server, directing the web traffic to the Flask application. The Flask framework provides tools for creating a web interface for the software, making it possible to route requests from a web address to software functions. Docker serves as a supervisor, restarting the software if it crashes, or if the Raspberry Pi restarts. The result is a software which is running on a Raspberry Pi day and night, connected to the internet by a web interface which makes it possible to control the software through web requests.

- **Docker** [3] is a software for running software applications in containers. These containers run their own self-contained environment (OS, python version, packages, etc.), and the environment and installation procedure (series of commands to run each time the container is started) are defined by a *Dockerfile* within your project folder. Since the environment is self-contained within the container, it makes it easy to develop the software on a different type of machine than the one that is going to run the software, as the container will run with the same self-defined environment on both machines. Docker can also work as a supervisor, restarting containers when they crash or when the computer is restarted, which is great for a software that should be running continuously.

- **Nginx** [11] is a light weight web-proxy, which filters and redirects traffic to the *gunicorn* web server and our application.

- **Gunicorn** [8] is a Web Server Gateway Interface (WSGI) HTTP server, which makes it possible to route web addresses to python functions in an application.
- **Flask** [4] is a lightweight web application framework used for creating small web-based applications. Using Flask we can define what parts of our application specific web addresses routes to.

Additional non-standard tools and packages used:

- *CasADi* (used for formulating and solving optimization problems)
- *pyTibber* (python interface for interacting with Tibber API)
- *sensiboClient* (python interface for interacting with Sensibo API)
- *nordpool* (python interface for interacting with Nord Pool API)
- *numpy* (provides additional math functionality)
- *Metno_locationforecast* (python interface for interacting with MET API)
- *requests* (used for making HTTP requests to APIs)
- *asyncio* (used for making asynchronous tasks)

4.2.2 Performance considerations

It is important to consider performance for both data collection and the MPC algorithm. The data collection software needs to be able to process and store the current data before new data becomes available. And the MHE and MPC algorithm needs to be able to solve their respective optimization problem before the next interval of the algorithm arrives. These performance considerations are tested and evaluated in chapter 7.

In the software design I chose to use threads for managing the data collection from the different API's. Threads run concurrently on the CPU, so the API calls does not have to wait synchronously for each other to finish. For example, if the data collection was running without threads, if one API requests time outs because of an error on the API, then all other API requests must wait for that one request to finish. This is obviously not good. The MPC algorithm should also run on a separate thread or process for the same reasons.

The first version of the software created here stored the same data into several pickle data files: daily, weekly, monthly and yearly data files. So, each data point was saved into each of these files. The reason for this was to provide greater utility in terms of playing around with data. However, after testing it was clear that the weekly, monthly and yearly files were becoming too large to efficiently open and store data to, and the data storage times were becoming so large that they might eventually become larger than the shortest sampling time between measurements (real-time power stores new data every two seconds). Therefore, the data is stored only in daily pickle files. There might be other better and faster storage methods, but they were not explored in this thesis.

It should also be noted that Python, CasADi and Interior Point Optimizer (Ipopt) was used in this thesis. These are not the fastest languages and tools. So, if performance is becoming a big issue, changing to faster programming language and optimization tools should be an option to consider. Of course, you can also use better hardware, but then the system is becoming more expensive, and we are trying to keep the costs down in this system design.

4.2.3 Fault tolerance, stability and availability

A big requirement for the software is that it can be running day and night while handling potential crashes and disconnects without requiring manual service to restart the software. It should also be built in a way to avoid data races (simultaneous access and writing to the data files), so the software doesn't loose data due to corrupted data files.

To make the software able to run day and night without interruption, error exception is used on small errors that does not require a software restart, simply logging the error instead of stopping the software. And Docker acts as a supervisor of the software, restarting the software whenever it crashes. Docker also starts the software if Docker itself is restarted, which is what happens when the Raspberry Pi is restarted. In this way, the software will keep running day and night, logging small errors in a log file and restarting on crashes and system restarts.

Since the software depends on API keys to the different API services, I implemented a routine for storing the API keys on an environmental variable file on the Raspberry Pi. Any time the parameters like sampling times or API keys are changed through a request, the environment file will be changed to reflect these new values. This way the software can always access the latest parameters when Docker restarts the software.

To avoid data races for the data files I implemented a producer and consumer system between the threads, where all file access is handled by one thread called the *main handler*. The main handler thread receives both data storage requests and data retrievement requests. This system is based on using a blocking FIFO queue in a loop to coordinate the different tasks for the main handler thread. More on this in section 4.2.4.

In the case of a crash in the middle of file writing, the data files might get corrupted and unusable. To avoid trying to write to a corrupted file, a session id was implemented which changes each time the docker container starts. At the startup of the software, this session id is used to create a fresh new data file. This way the software won't run into any problem of trying to store data to a corrupted file. On the other end, when requesting data, the main handler thread will go through each file in a specific time period and check if the file is corrupted before loading it.

To secure the data I added a backup feature that backups the data once a day to a zip file. This backup can be downloaded directly from the web interface of the Raspberry Pi.

4.2.4 Code structure and implementation

Low-level detail has been omitted for ease of reading. The full software code can be seen in the appended zip file to this thesis.

The main software for the Raspberry Pi is composed of three docker containers, as seen in fig. 4.3. There is one container running Ngrok, which gives access to the web interface of the Raspberry pi to requests from outside the local area network. Another container is running Nginx, which serves as a filter and redirector for web requests. Nginx will redirect certain web request to static files and other requests to the Flask application, depending on the address which is accessed. Nginx will also block illegal requests and serves as a security layer in that regard. The final container is running the Flask application software. The Flask app has the data collection package imported, and the web interface of the application will route web requests to specific functions of the data collection package, these requests can be used to start or stop the data collection, to change sampling times of the sensor threads, or to request the latest data that has been collected. The web interface of the Flask application is running with Gunicorn, which is a WSGI HTTP server, making it possible to route web addresses to python functions. The data collection module is created as a python package which can be imported into any python program.

The processing layer with SYSID, MHE and MPC is running as stand-alone scripts which can run on any computer with Python 3. The MHE and MPC is combined in a script which repeatedly requests the latest data from the Raspberry Pi by making HTTP requests to its web interface, then it processes the data through the MHE and MPC algorithm, and lastly sends the output over an

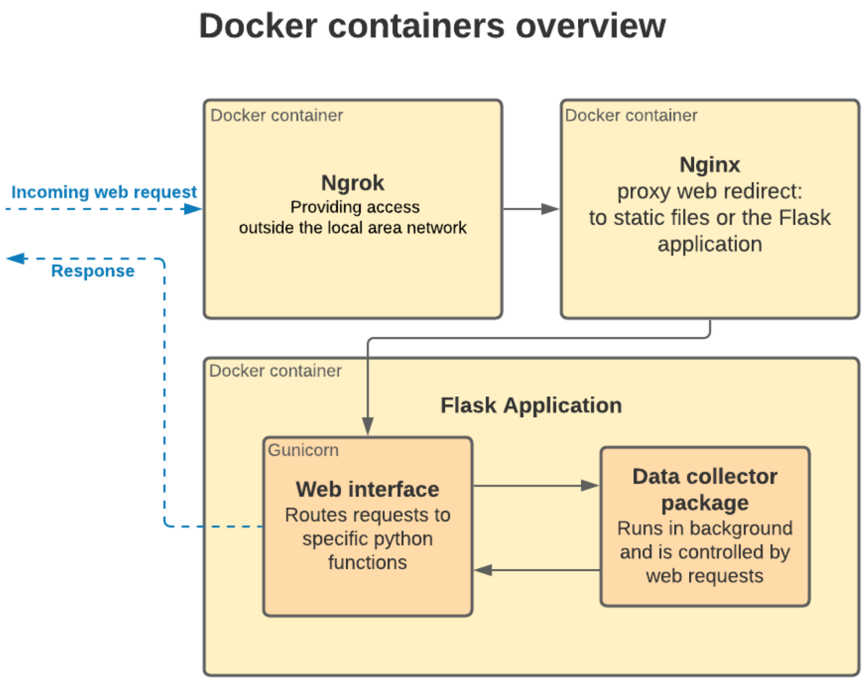

Docker containers overview
Incoming web request
Response
Docker container
Ngrok
Providing access outside the local area network
Docker container
Nginx
proxy web redirect: to static files or the Flask application
Docker container
Flask Application
Gunicorn
Web interface
Routes requests to specific python functions
Data collector package
Runs in background and is controlled by web requests

Figure 4.3: The three docker containers of the software: Ngrok, Nginx, and the Flask application

HTTP request directly to the Sensibo API. The Sensibo API relays this to the Sensobo Sky devices, which sends the instructions over IR to the heat pumps.

The Flask application and web interface

The main purpose of the Flask application is to provides the web interface for the software and linking these to functions within the data collection package.

There are five main routes:

- **POST /api/start_data_collector:** Starts the data collector with a set of parameters supplied with a payload. See table 4.3 for details.
- **POST /api/stop_data_collector:** Stops the data collector. See table 4.4 for details.
- **GET /api/get_file_list/...:** Gets a list of data files. See table 4.7 for all the possible endpoints and parameters.
- **GET /api/get_data/...:** Gets a list of data files. See table 4.6 for all the possible endpoints and parameters.
- **GET /api/get_data_since/...:** Gets a list of data files. See table 4.5 for all the possible endpoints and parameters.

These routes are appended to the web address to the Raspberry Pi software. *GET* and *POST* are different methods of sending requests over HTTP. Generally, *GET* is used to request data while *POST* (being more secure) is used to send data. In *GET* requests, the parameters can be sent as part of the url, while in *POST* requests, the parameters (or data) is sent as a payload within the request body.

The three *GET* requests mentioned above have different possible parameters within the request address, see tables 4.5 to 4.7 for a presentation of all possible endpoints for these requests.

The route /api/get_data_since/ was specifically designed for the MHE and MPC scheme. Through this route, the scheme can get the data since the last iteration of the scheme.

See section 5.9.3 in the How-To chapter for specific examples of making requests to these APIs. For an explanation of what the program flow looks like after the software receives these API requests, see fig. 4.5.

Table 4.3: Endpoint and payload for the route /start_data_collector.

POST /api/start_data_collector

Start the data collector with a given set of parameters.
See section 5.9 for a usage example with python.

Payload

Add a payload in the form of the parameter object below:

```
parameters = {
    'tibber': {
        'api_key': 'api-key', # Required
        'sampling_time': 60, # [minutes] - Optional
        'rt_sampling_time': 20 # [seconds] - Optional
    },
    'sensibo': {
        'api_key': 'api-key', # Required
        'sampling_time': 5, # [minutes] - Optional
    },
    'MET': { # Will use default values from Config if no params are supplied
        'lat': "63.4",
        'lon': "10.335",
        'altitude': 200,
        'location_name': 'ugla',
        'personal_id': 'sebastien.gros@ntnu.no ericth@stud.ntnu.no POWIOT project'
    },
    'open_weather_map': { # Optional
        'api_key': 'api-key',
        'lat': "63.4",
        'lon': "10.335",
        'sampling_time': 60, # minutes
    },
    'backup': { # Optional
        'run_backups': True, # Optional (Defaults to True)
    },
}
```

Responses

Returns a status code and message.

- 200: SUCCESS
- 400: FAILED (Bad request)
- 500: FAILED (Internal Server Error)

Table 4.4: Endpoint for the route /stop_data_collector.

POST /api/stop_data_collector

Stop the data collector. See section 5.9 for a usage example with python.

Payload

No payload.

Responses

Returns a status code and message.

- 200: SUCCESS
- 400: FAILED (Bad request)

Table 4.5: Endpoints and parameters for the route /get_data_since.

GET /api/get_data_since/{sensor}/minutes/{minutes}
GET /api/get_data_since/{sensor}/hours/{hours}
GET /api/get_data_since/{sensor}/timestamp/{timestamp}

For a given *sensor*, get data since a *timestamp*, or for the last number of *minutes* or *hours*.
See section 5.9 for a usage example with python.

Parameter	**Valid values**
{sensor}	'MET', 'sensibo', 'open_weather_map', 'tibber', 'realtime-tibber-home-pumps', 'realtime-tibber-home-up'
{minutes}	Any positive integer
{hours}	Any positive integer
{timestamp}	Timestamp on the format: 'Wed, 09 Jun 2021 11:49:09 GMT'

Responses

Returns a status code, a message, and data (if successfull).
Note: Sensibo returns data from all sensibo devices in the house.

- 200: SUCCESS
- 400: FAILED (Bad request)

Table 4.6: Endpoints and parameters for the route /get_data.

GET /api/get_data/{sensor}/daily/first	Get all data from the first daily data file
GET /api/get_data/{sensor}/daily/latest	Get all data from the latest daily data file
GET /api/get_data/{sensor}/daily/all	Get all data from all the daily data files
GET /api/get_data/{sensor}/daily/date	Get all data from a specific day

For a given *sensor*, get data from a *specific day*, or the *first data file*, or the *latest data file*, or get *all data from all data files.* See section 5.9 for a usage example with python.

Parameter	**Valid values**
{sensor}	'MET', 'sensibo', 'open_weather_map', 'tibber', 'realtime-tibber-home-pumps', 'realtime-tibber-home-up'
{date}	Date on the format: '2021-06-07'

Responses

Returns a status code, a message, and data (if successfull).
Note: Sensibo returns data from all sensibo devices in the house.

- 200: SUCCESS
- 400: FAILED (Bad request)

Table 4.7: Endpoints and parameters for the route /get_file_list.

GET /api/get_file_list/{sensor}/daily/first	Get the name of the first data file
GET /api/get_file_list/{sensor}/daily/latest	Get the name of the latest data file
GET /api/get_file_list/{sensor}/daily/all	Get list of names of all data files
GET /api/get_file_list/{sensor}/daily/date	Get list of names of data files from a specific day

For a given *sensor*, get the name of the *first data file*, the *last data file*, *all data files*, or *all data files from a specific day.* See section 5.9 for a usage example with python.

Parameter	**Valid values**
{sensor}	'MET', 'sensibo', 'open_weather_map', 'tibber', 'realtime-tibber-home-pumps', 'realtime-tibber-home-up'
{date}	Date on the format: '2021-06-07'

Responses

Returns a status code, a message, and data (if successfull).

- 200: SUCCESS
- 400: FAILED (Bad request)

The Data collection package

Great care was taken in designing the data collection package. It was important that it was not only reliable and fault tolerant, but it should also be easy to use and be well designed such that it is easy to add or remove sensors for future developers. It must handle the collection of measurements and data at different sampling intervals automatically and reliably.

As mentioned in section 4.2.3, I found that the best way to handle simultaneous API calls with different sampling times would be to use threads. When each sensor [API call] has its own thread, the calls are handled concurrently and the errors for one sensor will not affect the errors for another sensor [API]. To avoid data races for the data files I implemented a *main handler* thread with a queue system. The main handler thread handles everything that has to do with the data files: creating files, data storage and data requests. To coordinate the different incoming tasks, the main handler thread has a blocking FIFO queue in a loop, so it continually waits for a new task (which comes from the queue), then processes it completely before trying to get the next task item from the queue. Every other thread that needs to do something with the files will send a queue item for the main-handler thread to process. The queue items have several parameters, so you can specify what the main-handler thread should do. The sensor threads, which gets the data from the APIs, will send new data to the main-handler through this queue, and the main-handler will store this data on the correct files. The data requests coming from outside through the web interface will also send a queue item to this queue, which the main handler thread will handle by retrieving the data from the files and sending it back to the request. The backup thread will also send a request to the main-handler, which will make the main-handler back up all data in a zip file. In this way this system avoids the problem of data races for the data files.

This system is visualized in fig. 4.4, where the yellow boxes are different threads running simultaneously, the blue boxes are APIs that the thread makes data requests to, the green boxes are specific data that are accessed through these API's, the pink box is the persistent storage, and the orange box signifies calls made through the web interface. The green circles and lines indicates the flow of items in a FIFO blocking queue. The producers in the green queue are the different sensor threads, backup thread, and data requests from the web interface. The items in this queue is processed by the main handler thread one item at a time, and the information in each item tells the main handler what to do. In this way we achieve atomicity within the different processes that want to store or receive data from the files. The orange queue is a different FIFO blocking queue, in which the main handler sends data back to a data request. After the web interface function has sent a data request item to the green queue, it will lock the orange queue with a *mutex* (a mutex locks a resource to one task, making other tasks wait until it is released) and wait for an incoming item, which will always be the requested data from the main handler. The *mutex* is used to avoid a bug where two data requests are made simultaneously and the data comes back to the wrong request. After receiving a data item in the orange queue, the web interface function will return the data to the request coming from the web. Both these queues are blocking, which means that if the queue is empty the task will stop and wait until a new queue item arrives before continuing. The threads are running in loops, as signified by the loop icon in the yellow boxes. The sensor threads are running with different intervals (sleeping between each interval), while the main handler thread is looping after each successful request and is blocked until the next queue item arrives.

As mentioned in section 4.2.2, it is important to separate the API calls into different threads (here called sensor threads), so that the API calls don't have to wait for each other synchronously. This is also important for the accuracy of data polling, so the threads are polling data at the correct sampling rates instead of having to wait for the other sensors to finish their polling. Using threads also allows for running the data collection in the background of whatever application that is using it (in this case the Flask application).

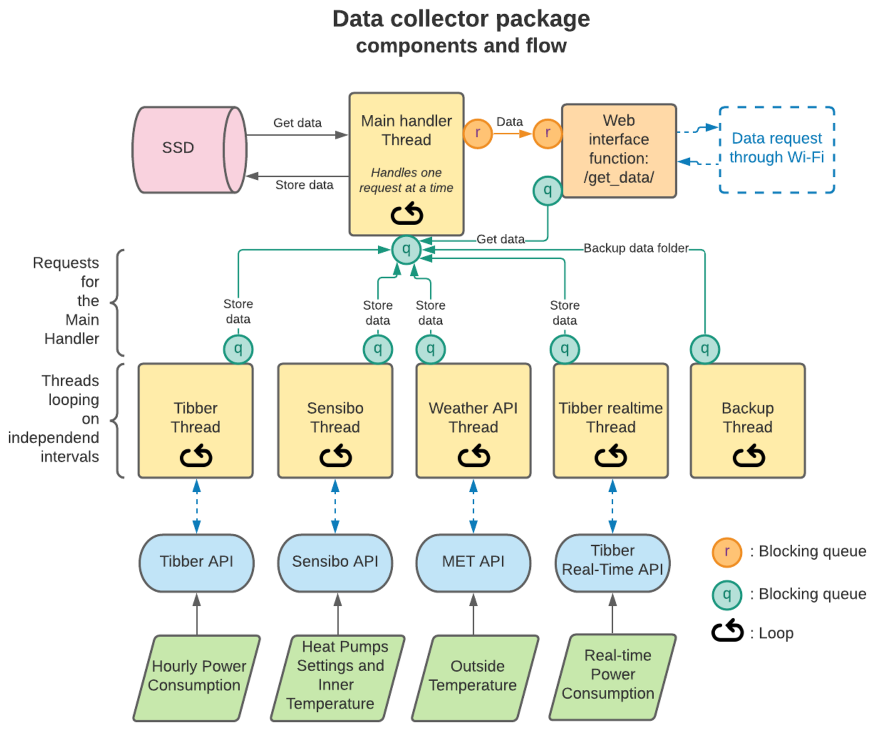
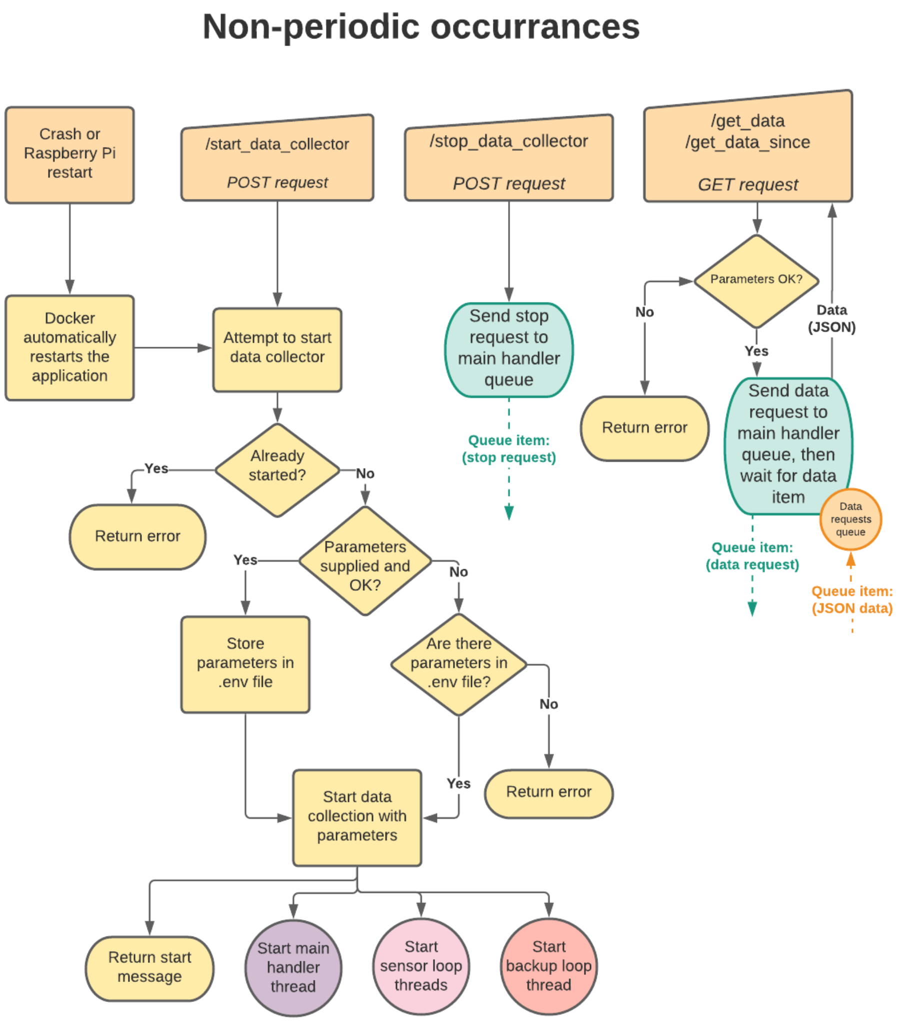


Figure 4.4: An overview of the different components within the data collection package software. Green boxes indicate data that is being collected, blue rounded boxes indicate API, yellow boxes indicate a running thread and the orange box indicate an incoming request thread. The blue dashed arrows indicate a data flow over Wi-Fi.



Non-periodic occurrances
Crash or Raspberry Pi restart
Docker automatically restarts the application
/start_data_collector POST request
Attempt to start data collector
Already started?
Yes
Return error
No
Parameters supplied and OK?
Yes
Store parameters in .env file
No
Are there parameters in .env file?
No
Return error
Yes
Start data collection with parameters
Return start message
Start main handler thread
Start sensor loop threads
Start backup loop thread
/stop_data_collector POST request
Send stop request to main handler queue
Queue item: (stop request)
/get_data /get_data_since GET request
Parameters OK?
No
Return error
Yes
Send data request to main handler queue, then wait for data item
Data (JSON)
Data requests queue
Queue item: (data request)
Queue item: (JSON data)

Figure 4.5: Non-periodic Program flow of the Flask application and data collection package.

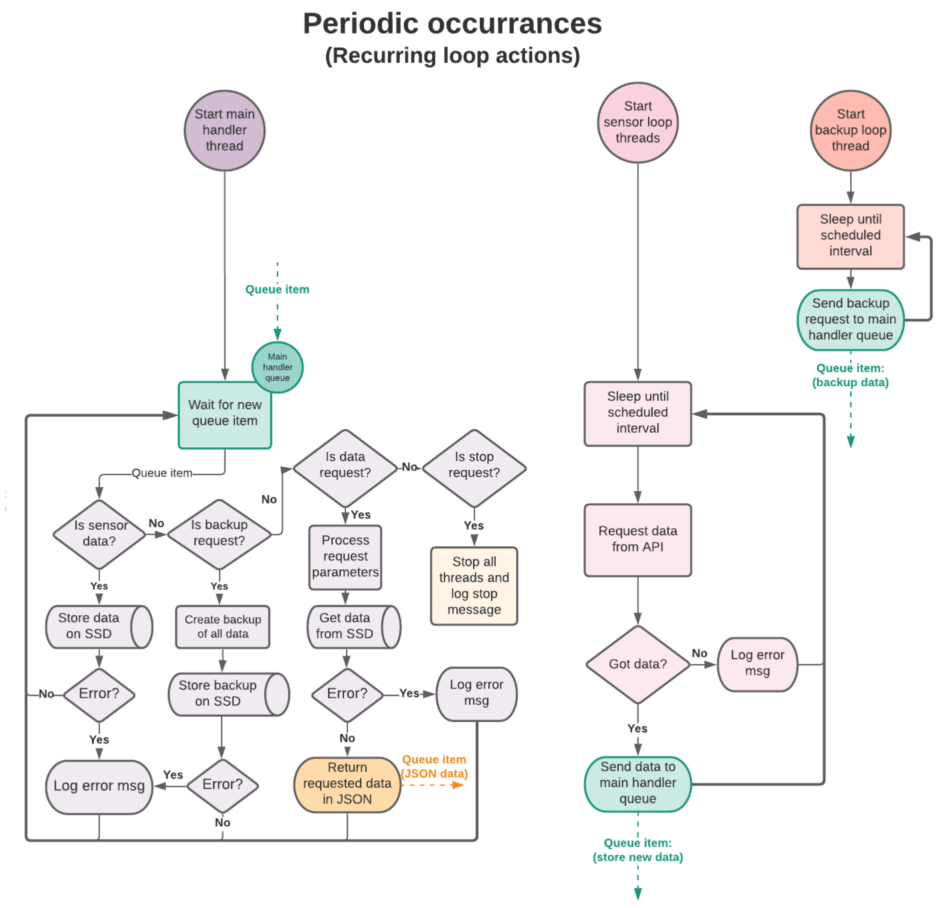


Figure 4.6: Periodic Program flow of the threads within the data collection package.

The program flow of the data collector together with the web interface is visualized in figs. 4.5 and 4.6. Figure 4.5 shows the non-periodic occurrences, such as when the data collector starts, stops or gets a data request. The green semi-round boxes with an arrow going outwards indicates that a queue item is sent to the green queue for the main handler thread to process. The orange arrow indicates a return of data to the orange queue. The three circles at the bottom indicates that threads are started and the flow continues within these threads. The flow of the threads can be seen continuing in fig. 4.6. The sensor threads, as noted in the figure, is a representation of all the different sensor threads: MET thread, Sensibo thread, and Tibber threads. The main handler thread waits for a new queue item from the other threads before continuing with the program flow, then loops back to wait again. Again, the dashed arrows going out of a box indicated that a queue item is sent to the queue, and a dashed arrow going into a box indicates that this box is processing the queue items.

File formats and data structure Each sensor has its own data files. After testing it was discovered that having files that had more than one days worth of data made the main handler thread too slow, so the file storage system now only use daily files. More on this test in chapter 7.

The folder structure can be seen in fig. 4.7. If these folders do not exist, the software will automatically create these folders when started.

The file names are structured as follows:

- {sensor name}_{date}___{session id}.pkl

Where *{sensor name}* is e.g. 'MET', 'sensibo', 'tibber', and so on. *{date}* is the current date, *{session id}* is a combination of the datetime timestamp when the file was created, and a sequence of 8 randomly generated characters. Every time the software restarts or there is a new day, it will create a new file with this structure.

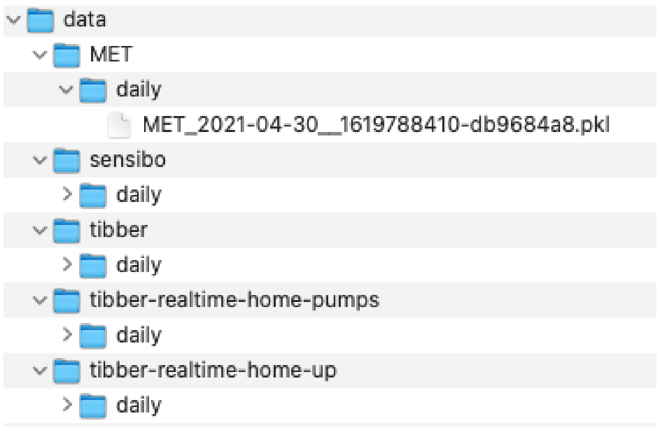


Figure 4.7: Folder structure for the pickle data files.

Backup procedure Once a day, the backup thread activates the backup routine by sending a backup request to the main handler thread. The backup routine makes a zip file of the whole data folder and stores it in the backup folder. At maximum five zip backups will be stored at a time. If five backups exists, the backup routine will remove the oldest before adding a new zip file. The backups are available for download through the web interface.

Avoiding error crashes Extensive exception handling through try/except clauses is used for avoiding crashes on non-critical errors. Instead of stopping the program, it will continue and log the error in the log file. These non-critical errors include API timeouts, connection errors, and similar. The error log is also used for debugging when unwanted problems occur.

4.2.5 Implementation on the Raspberry Pi

As mentioned earlier, the software was implemented as a Flask app running in a Docker container, with Nginx and Ngrok running in two other containers.

In the project folder we have the code for the Flask application, data collection package, Nginx configuration, Ngrok configuration, Dockerfiles, and a docker-compose file. The Dockerfiles serves as instructions on how the individual Docker containers are setup, while the docker-compose file defines the interaction between the three containers and the files they are using.

The project folder was uploaded as a repository to github.com and then downloaded directly to the Raspberry Pi by cloning the repository. After installing all the dependencies, the software is started by running the docker-compose command, which is used to simultaneously start and coordinate the containers. Exact details on how to install and run the software is presented in chapter 5.

As mentioned before, Ngrok is used to be able to communicate to the Raspberry Pi from outside the LAN. Another option to allow outside communication would be to configure the router to allow traffic on certain ports, but this is less secure than using a service like Ngrok unless you know how to set it up properly. Just opening up the router might be a good option for a future version of this software.

Chapter 5

How-To: A guide for implementing and using the system and software

The goal of this chapter is to give future developers, as well as the reader, specific instructions and tools to implement this system and software themselves. A specific focus is made for the students that will take over the software development next fall. This chapter will not be of interest to people who is not looking to implement this system themselves.

First in section 5.1 a list of the things you need will be presented. Section 5.2 presents resources for useful background information. Then sections 5.3 to 5.5 explains how to set up the system, install the software, and start the data collection. Sections 5.6 and 5.7 explains how to check the software log and restart the software. Section 5.8 explains how to update the Raspberry Pi with new software updates. Section 5.9 explains how to use the software's (house hub) API, with examples. Section 5.10 explains how to add sensors [APIs] to the software. Section 5.11 explains how to install and run CasADi on the Raspberry Pi. Lastly, sections 5.12 and 5.13 explains how to access the Raspberry Pi terminal from outside the LAN, and how to change which wifi the Raspberry Pi is connected to.

This guide is based on using github.com for deploying code to the Raspberry Pi and Secure Shell (SSH) for connecting to the Raspberry Pi. The guide also assumes a headless Raspberry Pi (no monitor, mouse or keyboard connected).

5.1 What you need

The list below lists the components and accounts you need in order to run this system:

- Raspberry Pi B (8GB RAM was used in the system for this thesis)
- An *Application Class 2* SD card for the Raspberry Pi B.
- Wi-Fi
- Tibber as your electricity provider
- Tibber Pulse (one for each smart meter)
- Sensibo Sky (one for each heat pump)
- Ngrok account
- Heat pumps compatible with Sensibo Sky [19]
- Github account
- The software (added as a zip to the thesis)

While not a requirement, it would also be good to have an understanding of Docker.

5.2 Background information

5.2.1 SSH

Secure Shell (SSH) is a remote administration protocol that allows users to control and modify their remote servers over the Internet.

Resources:

- SSH Tutorial from DigitalOcean
  - *https://www.digitalocean.com/community/tutorials/ssh-essentials-working-with-ssh-servers-clients-and-keys*
- SSH on WikiPedia
  - *https://en.wikipedia.org/wiki/Secure_Shell_Protocol*

5.2.2 Docker

Docker is a software for containerize applications, making it easy to port a piece of software to different types of machines. Docker also supervises the software. The software in this thesis is using Docker, so a knowledge of this is great if you are going to use it.

Resources:

- Katacoda Docker Course (I recommend this tutorial)
  - *https://www.katacoda.com/courses/docker/*
- Docker official website
  - *https://www.docker.com/*

5.3 Setting up the devices

- **Tibber and Tibber Pulse** — First you need to make Tibber [22] your internet provider, then buy Tibber Pulse from their website or app. Follow Tibber's instructions for setting up the Tibber Pulse. Installing Tibber Pulse through the website is recommended over using the Tibber app. This should only take a few minutes.

- **Sensibo Sky** — Follow the instructions from Sensibo Sky. This should only take a few minutes.
- **Heat pumps** — As long as you have heat pumps with IR-sensors, no further setup is required.
- **Raspberry Pi B** — This is explained in section 5.4, where you install and run the software.
- **Ngrok** — Register for a free account at Ngrok.com. Once you have that you will get an *auto token.* Then open up the *ngrok.yml* file in the code project folder (*raspberry-sw/docker-files/ngrok/ngrok.yml*) and replace the auth token on line one with yours. When the software is running you can see the Ngrok address at your profile page on Ngrok.com. The Ngrok address will change every time the software is restarted.
- **Github** — Github was used by me to deploy the code to the raspberry pi. Just get an account if you don't have it and set up a repository there with the code.

5.4 Installing and running the software

The Raspberry Pi comes completely blank. We have to install an OS, setup the Wi-Fi connection, install python, install software tools like Git and Docker, and finally we have to install our own software.

1) Setup a Raspbian OS image on the SD card

1. Download the Raspberry Pi Imager (https://www.raspberrypi.org/software/).
2. Insert the SD card into your computer and start the Raspberry Pi Imager, then select Raspbian OS (with desktop) as OS and select the SD Card as Storage.

3. Before writing, press cmd+shift+x (ctrl+shift+x on windows) to open advanced settings. Select "Enable SSH" and set a password for the "Pi" user. Select "Configure wifi" and write the Wi-Fi network name and password. Select "Set locale settings" and set the timezone.

4. Write to the SD card.

Now you have a Rasbian OS image on the SD card with SSH and Wi-Fi settings setup. To use the OS on the Raspberry Pi, simply plug in the SD card and start the Raspberry Pi.

2) Install and update software tools

1. Plug the SD card into the Raspberry Pi and start it by connecting it to the power source. Wait 10-15 seconds for it to start up.

2. Connect to it through SSH on a separate computer by entering the following command into your terminal window (enter the password you selected in the Raspberry Pi Imager):

```
$ ssh pi@raspberrypi.local
```

3. Install Raspbian OS updates:

```
$ sudo apt-get update -y
$ sudo apt-get upgrade -y
$ sudo apt install raspberrypi-kernel \
  raspberrypi-kernel-headers
```

4. Install Python 3:

```
$ sudo apt update
$ sudo apt install python3
$ sudo pip3 install --upgrade pip
```

5. Install Git, Docker, and docker-compose:

```
$ sudo apt-get install git
$ curl -sSL https://get.docker.com | sh
$ sudo pip3 install docker-compose
```

6. Enable the Docker system service to start containers on boot, and enable use of docker without admin priviles:

```
$ sudo systemctl enable docker
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

7. Check if Docker is installed:

```
$ docker
```

3) Installing and running our own software There were some permission errors when running the software as the *pi* user, therefore we start by changing the user to root, then all the following commands will be done as the user root.

1. Change to user root:

```
$ sudo su
```

2. Install and run the software (if this repository is no longer active, or you don't have permission, create your own repo on github with the raspberry-sw code folder appended to the thesis):

```
$ cd /home/pi
$ git clone \
  https://github.com/Master-thesis-Eric-Torn/raspberry-sw.git
$ cd raspberry-sw
$ docker-compose up --build -d
```

3. The last *docker-compose* command will start three Docker containers running Nginx, Ngrok, and the Flask application. Check that these three containers are running with the following command:

```
$ docker ps
```

If not all three containers are running, something went wrong, try to restart the containers (*docker-compose down* shuts down the docker containers):

```
$ docker-compose down
$ docker-compose up --build -d
```

4. Check that the web server is running by going to *http://raspberrypi.local* (NOTE: only http:// works, not https://) in your browser. You should see a page with instructions on how to use the API.

5. To see the terminal log of the web server (which logs requests and things that happen within the software), run the following command (*-f* locks your window to follow the web server terminal, *-tail=50* shows only the latest 50 logs, *raspberry-sw_flask_1* might be different for you, so double check with *docker ps* for the NAME of the Flask container):

```
$ docker logs -f --tail=50 raspberry-sw_flask_1
```

You should now have the software setup on the Raspberry Pi, and you are now ready to start the data collection.

5.5 Starting the data collection

The first time you start the data collection you will have to send a POST request with parameters that include the *API keys* for Tibber and Sensibo, the *latitude* and *longitude* position of the house for MET. You can also optionally provide an api key and position for Open Weather Map, which was used for collecting outside temperature before the MET API was added.

Open up a python file and enter the following:

```
params = {
    'tibber': {
        'api_key': 'api-key',
        'sampling_time': 60, # [minutes] - Default: 60
        'rt_sampling_time': 2 # [seconds] - Default: 20
        },
        'sensibo': {
            'api_key': 'api-key',
            'sampling_time': 5, # [minutes] - Default: 5
        },
        'MET': { # These are the values for the case study house in this thesis
            'lat': "63.4",
            'lon': "10.335",
            'altitude': 200,
            'location_name': 'ugla',
            'personal_id': 'ericth@stud.ntnu.no␣Master␣Thesis'
        },
        'open_weather_map': { # Optional
            'api_key': 'api-key',
            'lat': "63.4",
            'lon': "10.335",
            'sampling_time': 60, # [minutes] - Default: 60
        },
        'backup': { # Optional
            'run_backups': True, # Optional (Defaults to True)
        },
    }

    import requests
    import json

    response = requests.post(
        'http://raspberrypi.local/api/start_data_collector',
        json=params
    )
    # (Note that https does not work)

    print(json.loads(response.text))
```

Then run the file:

```
$ python filename.py
```

The response should tell you if the data collector was started successfully or not.

After the first time you started the data collection, the parameters you provided will be stored in a local *.env* file. So when you start up the software with *docker-compose up –build -d*, the data collector will **automatically start with these parameters**.

If you want to change the parameters you have to either stop the software and change the *.env* file before starting again, or you need to send to stop and start the data collection by sending two requests to the software's API:

```
import requests
import json
import time

# First stop the data collector
response = requests.post(
    'http://raspberrypi.local/api/stop_data_collector'
)
print(json.loads(response.text))

# then just give it a little bit of time to shut down properly
time.sleep(30)

# start the data collector with new params
params = {
    ...
}
response = requests.post(
    'http://raspberrypi.local/api/start_data_collector',
    json=params
)
print(json.loads(response.text))
```

5.6 Checking the software log

It might be informative to watch the log of the software to see when data is stored or if the software is not working properly. SSH onto the raspberry pi and connect to the docker log of the software by executing the following commands:

```
$ ssh pi@raspberrypi.local
$ docker logs -f --tail=20 raspberry-sw_flask_1
```

-f locks your window to follow the web server terminal, *–tail=20* shows only the latest 20 logs, *raspberry-sw_flask_1* might be different for you, so double check

with *docker ps* for the NAME of the Flask container

5.7 Restarting the software

Execute the following steps to restart the software:

```
$ ssh pi@raspberrypi.local
$ cd raspberry-sw
$ docker-compose down
$ sudo docker-compose up --build -d
```

5.8 Making updates to the software

Execute the following steps to make updates to the software:

1. Push new updates to the github repository containing the software.

2. Execute the following commands:

```
$ ssh pi@raspberrypi.local
$ cd raspberry-sw
$ docker-compose down
$ sudo git pull
$ sudo docker-compose up --build -d
```

Note: it can be useful to backup the .env (in the raspberry-sw folder) before pulling from github, so the latest settings you used for the data collector doesn't get overwritten by the github pull. Simply copy the .env file to a safe location before pulling and move it back after.

5.9 Using the Raspberry Pi sofware's API

This section explains how to use the the API that was developed for the software of this thesis. Section 5.9.1 explains how to access static files from the Raspberry

Pi, section 5.9.2 presents all the API endpoints and the available parameters, and section 5.9.3 shows examples of using the API endpoints.

For a quick API reference, simply go to http://raspberrypi.local to see a text page explaining all of the available API calls.

The endpoints should be appended to the address of the Raspberry Pi. This can either be its IP address, http://raspberrypi.local, or the Ngrok tunnel address (find the Ngrok address by going to http://raspberrypi.local:4551).

5.9.1 Static files

- *http://raspberrypi.local/data/* — Access and download the data files on the Raspberry Pi
- *http://raspberrypi.local/backups/* — Access and download the backups on the Raspberry Pi

5.9.2 API Reference

Please see section 4.2.4, where the API is presented in full.

5.9.3 Examples

These examples are based on using python. Try the examples by running the commands in a python shell, or by copying the code to a python file and run that file.

Starting the data collector:

Start the data collector by sending a *POST* request to the /start_data_collector endpoint, with the parameters payload as seen in table 4.3.

```
import requests

parameters = {
    # see reference
}

response = requests.post(
    'http://{ip_address}/api/start_data_collector',
    json=parameters
)
print(response.text)
```

Stopping the data collector:

```
import requests

response = requests.post('http://{ip_address}/api/stop_data_collector')
print(response.text)
```

Get data since:

See table 4.5 for reference.

```
import requests
import json

# This gets data from the last 30 minutes for sensor sensibo
response = requests.get('http://{ip_address}/api/get_data_since/sensibo/minutes/30')
# This gets data from the last 2 hours for sensor sensibo
response = requests.get('http://{ip_address}/api/get_data_since/sensibo/hours/2')
# This gets data since Wed, 09 Jun 2021 11:49:09 GMT for sensor sensibo
response = requests.get(
    'http://{ip_address}/api/get_data_since/sensibo/timestamp/Wed, 09 Jun 2021 11:49:09 GMT'
)

data = json.loads(response.text)
```

Get data:

See table 4.6 for reference.

```
import requests
import json

# This gets the data from 2020-09-02 for sensor MET
response = requests.get('http://{ip_address}/api/get_data/MET/daily/2020-09-02')
# This gets all the available data for sensor MET
response = requests.get('http://{ip_address}/api/get_data/MET/daily/all')
# This gets all the data from the last data file created for sensor MET
```

```
    response = requests.get('http://{ip_address}/api/get_data/MET/daily/latest')
    # This gets all the data from the first data file created for sensor MET
    response = requests.get('http://{ip_address}/api/get_data/MET/daily/first')

    data = json.loads(response.text)
```

Get file list:

See table 4.7 for reference.

```
import requests
import json

# Gets a list of all the data files that exist for sensor sensibo
response = requests.get('http://{ip_address}/api/get_file_list/sensibo/daily/all')

files_list = json.loads(response.text)
```

5.10 Adding sensors, API endpoints, or making other changes to the software

If you want to add sensors, API endpoints, or simply make changes to how things work, *the best way is to go through the code and making sure you understand what is going on.*

5.10.1 Change or add API endpoints

- The API endpoints are based on the Flask framework [4], and the endpoints are defined in the *run.py* file in the root folder of the project code.

- Endpoints (or routes) are defined with the decorator *@app.route('/api/start_data_collector', methods=['POST'])* and the following function. See Flask's website [4] for how to define routes.

- The *methods=['POST']* part of the decorator defines what types of requests you are allowing for the endpoint.

5.10.2 Change or add sensor [API] threads

All sensor threads are defined in the ___*init.py*___ file of the *data_collector package*, which is located in the data_collector folder within the project code root folder. See section 4.2.4 for how the data collection package works.

These instructions are made to serve as a reference after you have gone through and understood the code.

To make changes to a sensor, locate its class definition at the top of the file. The *start_mining_loop()* method is called when the software starts, and the *__get_latest_data()* method runs the loop that goes on forever, collecting data at each loop. There are some helper functions that are defined in *helpers.py* within the data collector package.

To add a sensor, follow these steps (take a look at the other sensors for reference):

1. First understand how the code works for the other sensors
2. Add sensor class for the new sensor/API with the same structure as the other sensors
   - add a *self.__sensor_name*
   - add a *self.__session_id* (see other sensors)
   - define a *self.start_mining_loop()* method (see other sensors). This method will be called when you create the thread for this sensor class in a later step.
   - define a *self.__get_latest_data()* method (see other sensors). This is the method that includes the loop that runs forever. This loop will request new data from an API and then send the data as a queue item to the main handler thread, which will store the data.
3. add corresponding file saving functions below the class definition (see other sensors)
4. Add the sensor to the main-handler loop (line 896)

    - Make sure to specify a *sensor_name* in the sensor class, and use this in the main handler thread. The main handler uses the sensor name to recognize what it should do with the data and what file saving functions it should call.

5. Add the sensor to the start function (line 956)
    - Get parameters for this sensor from the "params" argument (line 968 and onwards)
    - Make sure a sensor object is initialized (line 1081 and onwards)
    - Initialize and start a thread with the sensors mining loop (see others for example) (line 1135)
    - Add the sensor_name (must be the same) to the list in generate_all_data_folders call (line 1079)
6. Add the sensor to helpers.write_to_env_file() (line 16 in helpers.py)
7. Add default parameters to config.py

5.10.3 Adding an extra actuator (heat pump)

1. Buy and install an additional Sensibo Sky device for the heat pump (on the same Sensibo account).

2. The software should now automatically collect data from the Sensibo Sky and heat pump, and the data is accessible in the same manner as the other heat pumps.

3. To control the heat pump you need to make sure you add the new heat pump to the MPC script.

5.11 Installing CasADi on the Raspberry Pi

CasADi was used for running the MHE and MPC on the Raspberry Pi during this thesis. It is not possible to install CasADi on Raspberry Pi with *pip*, so a more convoluted way of installing CasADi was needed.

Resources:

- https://github.com/casadi/casadi/wiki/InstallationLinux

Installation steps for Raspberry Pi

1. **SSH into the Raspberry Pi**

```
$ ssh pi@raspberrypi.local
```

2. **Install CasADi dependencies**

```
$ sudo apt-get install gcc g++ gfortran git \
  cmake libclang-dev llvm-dev libblas3 \
  libblas-dev liblapack3 liblapack-dev \
  ocl-icd-opencl-dev pkg-config \
  --install-recommends
```

3. Install Python3 (If you haven't already)

```
$ sudo apt-get install python3
```

4. **OPTIONAL: Remove python2.7**. This makes a later installation step easier, but is strictly not necessary.

```
# Remove python2
$ sudo apt purge -y python2.7-minimal

# link 'python' bin to python3 bin
$ sudo ln -s /usr/bin/python3 /usr/bin/python

# Same for pip
$ sudo apt install -y python3-pip
$ sudo ln -s /usr/bin/pip3 /usr/bin/pip
```

```
# Confirm the new version of Python: 3
$ python --version
```

5. **Install python packages** (note: pretty sure I used python3 for all of these, but it is possible that I used python-dev and not python3-dev)

```
$ sudo apt-get install swig ipython python3-dev \
  python3-numpy python3-scipy python3-matplotlib \
  --install-recommends
```

6. **Install the Ipopt solver**

```
$ sudo apt-get install coinor-libipopt-dev
```

7. **Download CasADi and create a build directory**

```
# Doesn't matter where, but I did this in the
# Pi user's home dir

$ git clone https://github.com/casadi/casadi.git \
  -b master casadi
$ cd casadi
$ mkdir build
$ cd build
```

8. **Generate a make file** (The python version should be automatically detected)

```
$ cmake DWITH_PYTHON=ON DWITH_IPOPT=ON
```

9. **Check the output** of the last command to make sure that CMake was able to find your python version (It should be python, python3 or python3.x., not python2.7!)

```
# Open up the file with vi
```

```
$ vi CMakeCache.txt

# Search for the word python
$ :/python

# Press n to get to the next search result
# See which python version that is set as
# the executable/include/library.
# It should be python, python3 or python3.x.
# Not python2.7!
```

10. **IF the correct python version was *not* found** in the CMake output, or if python was not found at all, try specifying the python path in CMake arguments, do the following (**if it was found correctly, skip this step!**):

```
# First check if python3 is set to the python
# binary (Should output python version 3.x)
$ python --version

# REPLACE 'x' in the following command with the
# python version you have installed. You could
# go to the /usr/include/ folder to see what you
# have.
$ cmake -DPYTHON_EXECUTABLE=/usr/bin/python \
  -DPYTHON_INCLUDE_DIR=/usr/include/python3.x \
  -DPYTHON_LIBRARY=/usr/lib/arm-linux-gnueabihf/libpython3.x.so \
  -DNUMPY_PATH=/usr/include/python3.x/numpy \
  -DWITH_PYTHON=ON \
  -DWITH_IPOPT=ON ..

# (Don't forget the two dots..)
# Now check step 8 again
```

11. **Build CasAdi** (I am not 100% sure the 'sudo make python' is needed, but I used in in my installation procedure)

```
$ sudo make
$ sudo make install
$ sudo make python
```

12. **Test the install** (I had 8 fails, but the CasADi code still worked fine)

    ```
    $ cd ../tests/python
    $ python alltests.py
    ```

13. **Import CasADi for your programming script** by importing the casadi package like normal. ***Note that you NEED to use Ipopt as solver.***

5.12 Accessing the Raspberry Pi terminal outside the Local Area Network

During the thesis this was achieved by using the free service from Remote.it (https://remote.it/). Through their website you can generate a safe tunnel that works with SSH to access the Raspberry Pi.

5.13 Changing the wifi connection or connect the Pi with to an ethernet cable

Ethernet Cable Connection Simply connect your Raspberry Pi to the back of your network router with an ethernet cable. Once plugged in, you should observe the network LED blinking on your Raspberry Pi. In most cases, your wired internet connection will now be up and ready for use, provided that your router has DHCP enabled. If DHCP is not enabled, access your home router's management console with another computer that has already been connected. You can typically do this by entering your router's IP address into the address bar of any internet browser.

Changing WiFi connection Do the following commands from a separate computer connected to the same network as the Raspberry Pi:

```
$ ssh pi@raspberrypi.local
$ sudo raspi-config
```

Then use the arrows keys and navigate to "Network options".

Chapter 6

Case Study: House in Trondheim

In order to test and validate the software, it was implemented in a normal residential house in Trondheim in collaboration with Sebastien Gros. In addition to the four heat pumps and the Wi-Fi router, which was already installed, the house was instrumented with four Sensibo Sky devices, two Tibber Pulse devices (the house have two smart meters), and one Raspberry Pi.

6.1 Setup

The overview of the house setup can be seen in fig. 6.1. A and B denotes heat pumps on the outside of the house, while a, b, c and d denotes heat pumps on the inside of the house. Heat pumps a, c and d are connected to the outside heat pump A, while heat pump b is connected to the outside heat pump B. The green hexagons numbered 1, 2, 3 and 4 denotes the Sensibo Sky devices, and the dashed red line indicates an IR signal between the heat pump and Sensibo Sky device. The blue boxes with the symbols $T1$ and $T2$ denotes the Tibber Pulse's

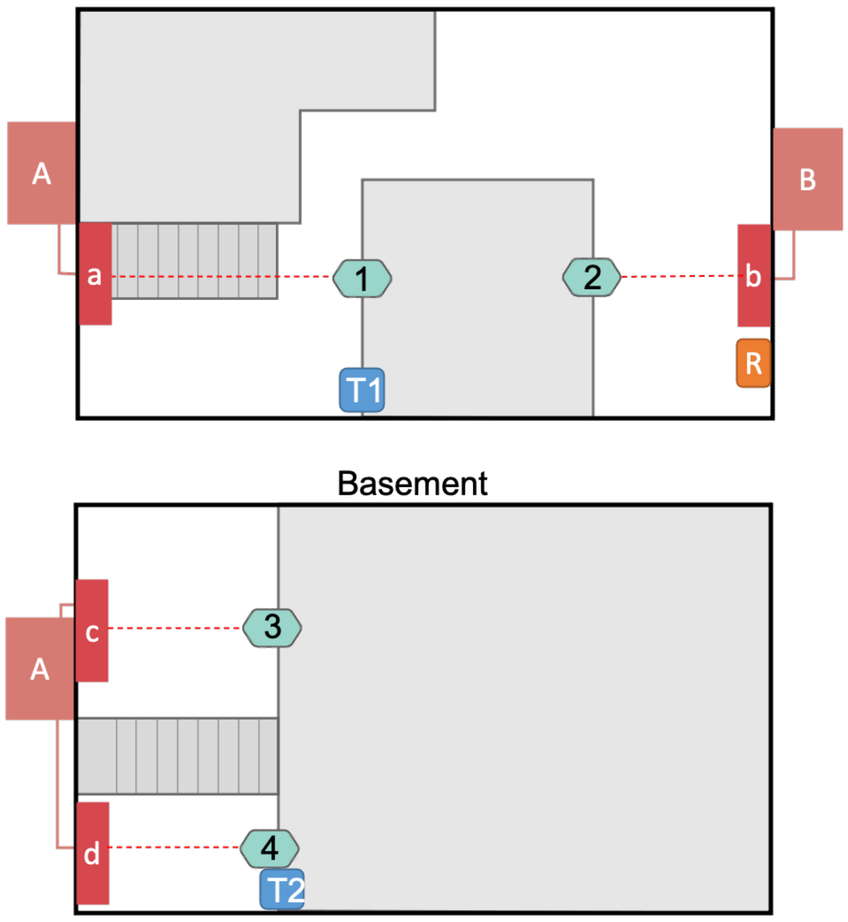
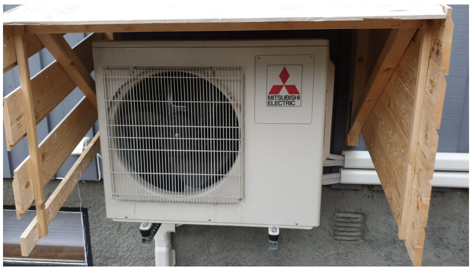


Figure 6.1: Overview of the system components in the house of the case study.

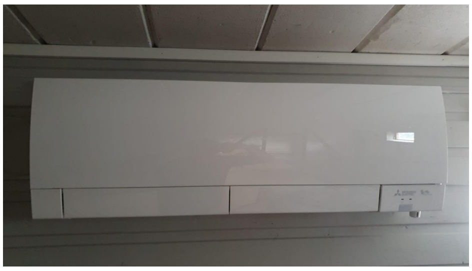
that are connected to the smart meters of the house. The orange box with an R denotes the raspberry pi device. The white color indicates areas that are heated by the heat pumps, while grey color indicates areas that are closed off. On the first floor the rooms of heat pumps a and b are connected by a corridor, while on the bottom floor the rooms are closed off to each other.

Figures 6.3 to 6.5 shows photos of the devices described above.

The sampling times was set up to collect real-time power consumption from



(a) Outside heat pump. There are two of these outside the house.



(b) Inside heat pump. There are four of these inside the house.

Figure 6.2: The heat pumps in the house.

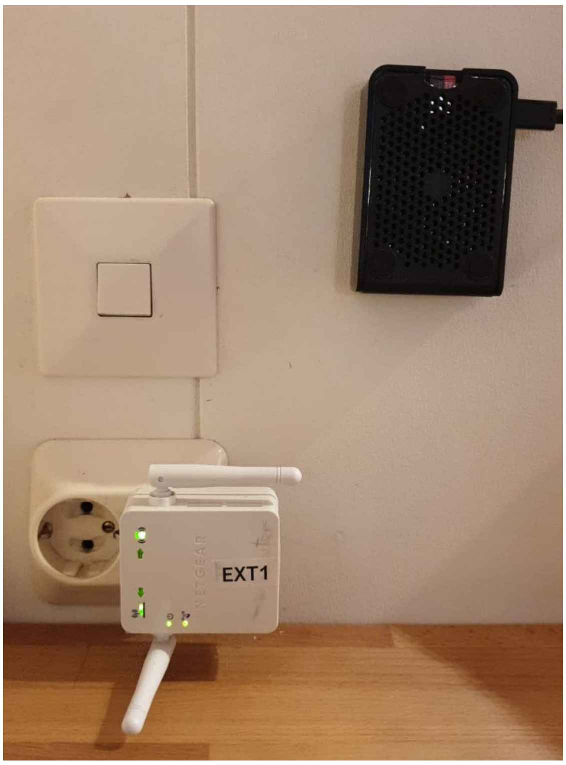


Figure 6.3: Photo of the Raspberry Pi, which is the black device hanging on the wall, and the Wi-Fi router, which is the white device.

Figure 6.4: Photo of one of the Sensibo Sky devices on the wall of the house. There are four of these in the house, one for each heat pump.

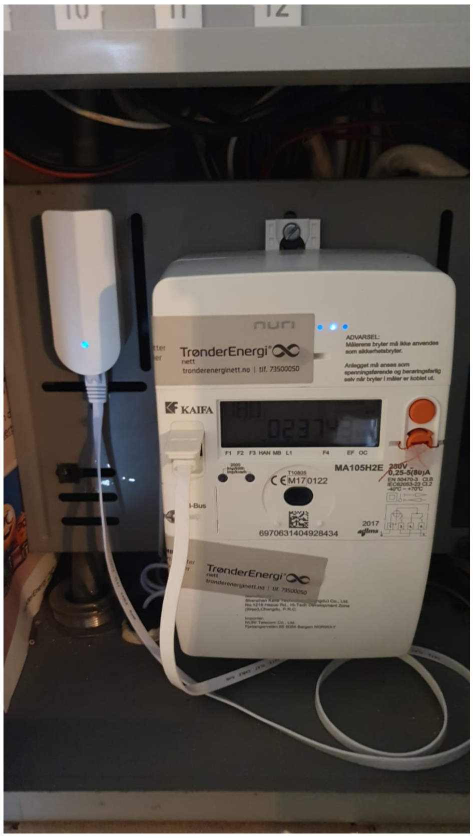


Figure 6.5: Photo of the Tibber Pulse device, which is connected to the smart meter of the house.

Tibber Pulse every two seconds, accumulated consumption from Tibber every hour, temperature and heat pump settings from Sensibo Sky every five minutes, and outside temperature from MET every hour. At the point of this writing, the software on the Raspberry Pi have been collecting and serving data to the MPC for about one month, heating the house through a closed-loop operation. The MPC have been running on a separate computer within the home.

6.2 Model and model parameters

This section presents the model and parameters that were used for the MHE and MPC algorithm when testing closed-loop control in the case study house.

The model and parameters were generated by SYSID operations designed by Sebastien Gros. This is mentioned here to give a complete picture of the setup, but more details about this will not be mentioned as it is not the focus of this thesis. For an introduction to the problem of generating a model for houses, see section 2.2.2.

The model for each room (with a total of four rooms) were:

$$\begin{aligned} \dot{T}_{wall} &= \theta^{wall}_{gain}(T_{room} - T_{wall}) - \theta^{wall}_{loss}(T_{wall} - T_{outside}) \\ \dot{T}_{room} &= \theta^{room}_{loss}(T_{room} - T_{wall}) + \theta^{room}_{gain} P_{HP} \end{aligned} \tag{6.2.1}$$

Where T is temperature, θ is a parameter that wil be estimated by SYSID, and P_{HP} is the power output from the heat pump.

Since we don't have a sensor for individual pump power consumption, this is estimated with the following equation:

$$\begin{aligned} P_{basic} &= \theta^{pump}_{gain}(T_{target} - T_{room}) + b \\ P_{no-neg.} &= \frac{ln(1 + e^{kP_{basic}}))}{k} \\ P_{HP} &= \frac{p_{max} - ln(1 + e^{k(p_{max} - P_{no-neg.})})}{k} \end{aligned} \tag{6.2.2}$$

Where P_{basic} is a basic model for estimating power, but which is not realistic since it can have negative values and has no maximum ceiling for maximum power. $P_{no-neg.}$ removes negative power for the model. Lastly, P_{HP} provides a maximum power output for the model. Providing a minimum and maximum power to the model is achieved by using a ReLu function, which is explained in section 2.2.2.

The parameters that were generated are shown in table 6.1:

Table 6.1: Model parameters

Parameter	**Value**
Main room	
b	4.348077603806764
θ_{gain}^{pump}	1.532666822799466
θ_{gain}^{room}	3.5052737757563737
θ_{loss}^{room}	5.654891096037879
θ_{gain}^{wall}	2.0
θ_{loss}^{wall}	1.205563424143914
Living room	
b	5.951606867741809
θ_{gain}^{pump}	1.382284286512041
θ_{gain}^{room}	4.933280976618673
θ_{loss}^{room}	4.832705457796633
θ_{gain}^{wall}	1.8937245769186934
θ_{loss}^{wall}	2.0
Living room down	
b	5.56003846866594
θ_{gain}^{pump}	1.5411805218431223
θ_{gain}^{room}	4.943300078549838
θ_{loss}^{room}	6.086034999780216
θ_{gain}^{wall}	0.8109454579051402
θ_{loss}^{wall}	0.5
Studio room	
b	10.939191208343102
θ_{gain}^{pump}	2.708242686866422
θ_{gain}^{room}	7.253619690564267
θ_{loss}^{room}	13.284096308111552
θ_{gain}^{wall}	2.0
θ_{loss}^{wall}	1.130949871217878

6.3 Settings for the MHE and MPC algorithm

The code and algorithm for the MHE and MPC were developed by Sebastien Gros [6]. It was then tested in collaboration, running a closed-loop operation with the software presented in this thesis. It is added here to give a complete picture of the setup.

The MHE is used to estimate room temperature, wall temperature, and heat pump power just before each iteration of the MPC. Even though we have a sensor for room temperature this measurement will always have some amount of noise, so estimating the room temperature with MHE will be more accurate than just using the measurement directly. The basics behind MHE is presented in section 2.2.3.

The states are then used by the MPC to calculate an optimized temperature trajectory based on an optimization problem with a cost function and constraints. This optimization problem will be presented here. The basics behind MPC is presented in section 2.2.4.

The time horizon for both the MHE and MPC are 24 hours with five minute sampling intervals. The cost function used in the case study is shown in the following equations:

$$\begin{aligned}
\text{minimize} \sum_{i=1}^{k} & w_1 C_{cost} P_{tot} + w_2 C_{comfort}^2 + w_3 C_{min.temp.}^2 + w_4 C_{discomfort}^2 \\
C_{cost} &= (\text{Spot market price} \times \text{VAT}) + \text{Grid costs} \\
C_{comfort} &= T_{room} - T_{reference} \\
C_{min.temp.} &= \min(T_{room} - T_{min}, 0) \\
\dot{C}_{discomfort} &= -\min(T_{room} - T_{reference}, 0) - \theta C_{discomfort}
\end{aligned}$$

Where k is the number of sampling intervals during the time horizon, w are weights given to the different cost terms, and P_{tot} is the total power used during each interval. C_{cost} is a term for the monetary cost, $C_{comfort}$ is a term for keeping the temperature close to the wanted temperature, $C_{min.temp.}$ is a term for adding cost when the temperature goes below a specific temperature, and $C_{discomfort}$ is a term for adding cost when the temperature is below the wanted temperature for a long period of time (e.g. if the prices are very high for a long time). $C_{discomfort}$ acts similar to an integrator, where the cost gets bigger the longer you are below the wanted temperature. θ is a parameter for adjusting the dynamics of $C_{discomfort}$.

The equality constraints to the optimization problem includes the constraints of the model. For the inequality constraints, the individual pump power consumption (P_{HP}) have a limit on 1.5 kW, while the total power consumption from all the pumps (P_{tot}) have a limit on 2.5 kW. These are physical limitations of the heat pump system in the case study.

The full optimization problem is described as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{k} w_1 C_{cost} P_{tot} + w_2 C_{comfort}^2 + w_3 C_{min.temp.}^2 + w_4 C_{discomfort}^2 \\
& \text{subject to:} \\
& \qquad \text{Dynamic Temperature Model} \\
& \qquad \text{Power Estimation Model} \\
& \qquad \mathbf{P_{HP}} \leq 1.5 \text{ [kW]} \\
& \qquad P_{tot} \leq 2.5 \text{ [kW]}
\end{aligned}$$

It should be noted that the model and cost function are experimental and subject to frequent change. However, in the tests in this thesis, these were the settings that were used.

Chapter 7

Tests and results

In this chapter tests and validations of the software is presented. First, the data collection results will be presented in section 7.1. Section 7.2 presents experiences of requesting data. In section 7.3, results of fault tolerance tests and evaluations will be presented. In section 7.4, results from response times analysis is presented. In section 7.5, utilization tests will be presented. In section 7.6, results from collecting data for SYSID is presented. In section 7.7, results from closing the loop with the MPC script will be presented. Lastly, evaluations of the requirements set in chapter 3 is presented in section 7.8.

7.1 Collecting data

The data collection software was implemented on the Raspberry Pi and installed in the case study house. After a period of testing the software and fixing various bugs, the Raspberry Pi have been collecting data in a stable manner for about a month. A sample week of data can be seen in fig. 7.1.

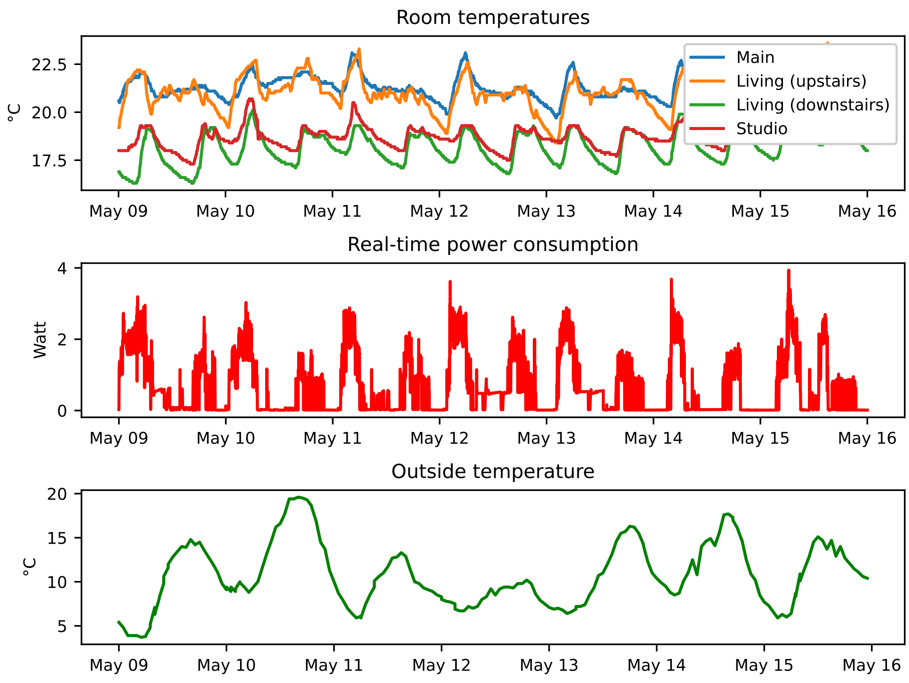


Figure 7.1: Plots of room temperatures, real-time power consumption, and outside temperature during a week in may. The power consumption is from the downstairs Smart Meter of the case study house, which the heat pumps are connected to.

7.2 Requesting data

Requesting data through the web interface have been working fine when running the MHE and MPC script on the LAN of the house, but when requesting data through the Ngrok tunnel, the requests sometimes times out, returning incomplete data.

7.3 Fault tolerance

This section will present the results of several fault tolerance tests. Section 7.3.1 presents the test results of losing internet connection, section 7.3.2 presents the test results of losing power, section 7.3.3 presents the results of running with corrupted or empty data files, and section 7.3.4 presents the results of running without supervision.

7.3.1 Losing internet connection

A test were performed where the internet was cut off while the software was running. The result was that the data collection stopped for the duration of the disconnect, but resumed shortly after. Sensibo and MET data resumed immediately, while the Tibber data collection resumed 10-30 minutes after the internet connection was back. The reason was that Tibber has a limit on the number of concurrent socket connections a user can have, so when the software tries to reconnect by creating a new socket connection, the socket connection on Tibber's end is still active and won't allow a new connection. It takes 10-30 minutes for Tibber to reset this socket connection.

Another test was performed where an ethernet cable was connected to the Raspberry Pi (and Wi-Fi turned off). There was no issues with the software during this test.

7.3.2 Losing power

A test of simply pulling the power plug was performed several times. The Raspberry Pi restarts automatically as soon as power is connected again. Each time Docker properly started the nginx and Flask containers, and the data collection was resumed. The Ngrok container sometimes did not start, and it is unclear why this was the case. The Tibber data collection took between 10-30 minutes to resume, for the same reasons mentioned in section 7.3.1

7.3.3 Corrupted or empty data files

Tests have been performed when the software is running in with the existance of corrupted and empty files. The results is that these are successfully ignored and does not affect the data collection or data requests to the software.

7.3.4 Running without supervision

The software seems to work well without supervision. At the point of this writing, it has been running for two to three weeks without requiring any manual supervision.

7.4 Response times

A big question is how the Raspberry Pi performs in terms of processing power, especially in terms of how it handles the MHE and MPC schemes, and how many sensor threads it can handle at the same time.

The following tests were made by timing the loop in the main handler thread to see how long it takes to store new data from the sensor threads and service data request from the web interface. The loop in the MHE and MPC script was also timed in order to see the response time of these on the Raspberry Pi. The MHE and MPC script was running simultaneously with the data collection and web interface.

Two data storage designs were tested. The first storage design duplicated the data into several different file types: daily, weekly, monthly and yearly. Every day a new daily file is created (to store the data from that specific day), every week a new weekly file is created (to store all the data from that specific week), and so on. More on this in section 4.2.4. The second design stored the data only in daily files. This second design was created after seeing the results of the first design.

7.4.1 Response times with design 1

Design 1 stores the data in daily, weekly, monthly and yearly pickle files.

The plot in figs. 7.2 and 7.3 shows how long it takes for the *main handler thread* (explained in section 4.2.4) to store new data that comes from the sensor threads. The x-axis displays the timetamp when the data was stored.

In fig. 7.2 we can see that the duration of the storage process for data from MET, Tibber hourly, and Sensibo over a period of 24 hours. The trend for MET and Tibber hourly does not noticably change during this pediod. The sensibo data storage process shows a sligh increase in duration until 00:00, where a new file is created and the storage duration drops. The Tibber and MET threads have a sampling rate of one hour, while the Sensibo thread have a sampling rate of five minutes.

In fig. 7.3 we see the duration of the data storing process for real-time power data from Tibber during roughly 11 hours. The sampling rate for real-time power is two seconds. Here there is a clear trend that shows an increase in how long it takes to store each data point.

The difference between these plots is because the short sampling rate of the real-time power thread compared to the other sensor threads. The number of data points in the real-time power files are 150 times more than the Sensibo data files, and 1800 times more than the MET and Tibber hourly data files.

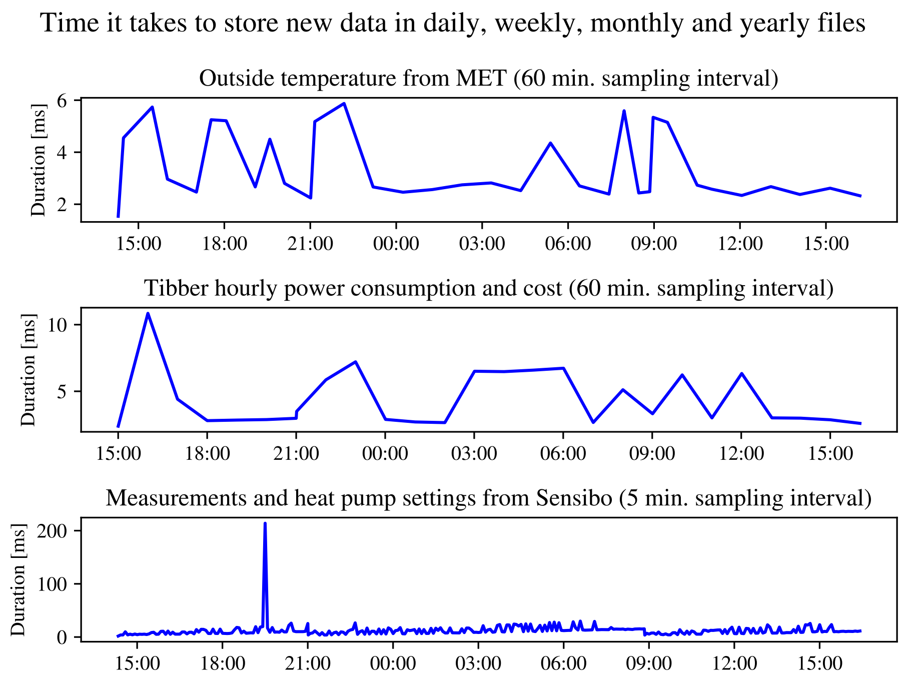


Figure 7.2: Plots of the time it takes for the main handler to store new data from MET, Tibber hourly, and Sensibo into daily, weekly, monthly and yearly pickle files.

At the timestamp 00:00 in fig. 7.3, a new daily file is created, and we can see that the main handler storage duration lowers a little bit. This is because a new daily file is created at midnight, so the daily file now takes faster to open and add data points to. However, the duration doesn't lower much because the weekly, monthly, and yearly files have not reset. We can see that after 11 hours the duration of the data storage takes about 500-550 ms. From this it can be deduced that if this design were used, the the weekly, monthly and yearly files would eventually become so large that the duration it takes to store new data would become larger than the sampling rate of new data coming in.

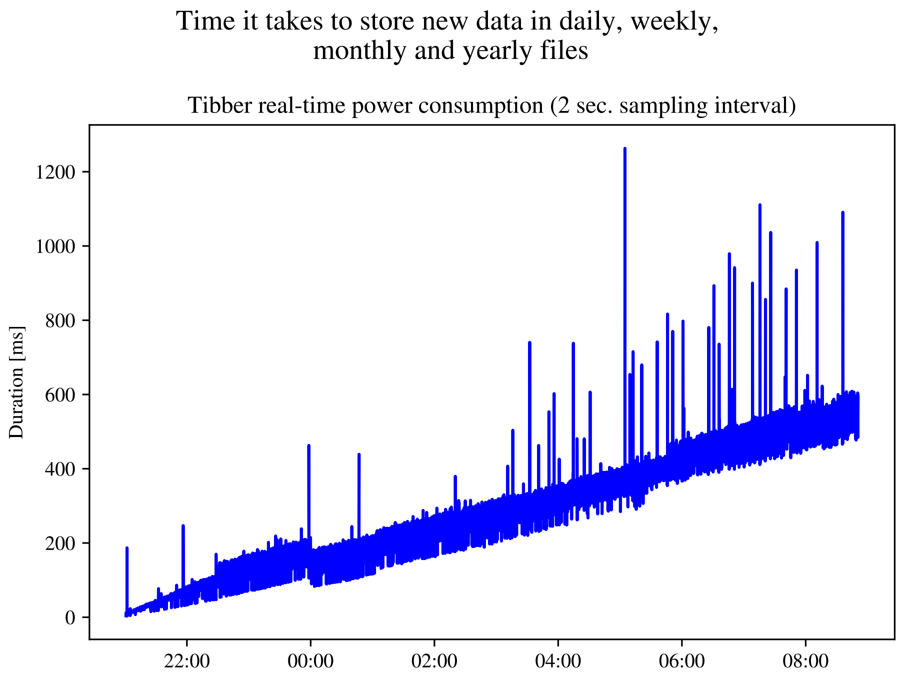


Figure 7.3: Plot of the time it takes for the main handler to store new data from Tibber real-time power into daily, weekly, monthly and yearly pickle files.

7.4.2 Response times with design 2

Design 2 stores the data only in daily pickle files.

Similarly to the plots in section 7.4.1, the plots in figs. 7.4 and 7.5 shows the duration of the data storage process. This time the data is only stored in daily files, where new files are created at midnight.

The first two plots in fig. 7.4 shows no clear increase of the duration. The last plot shows a small trend of increased duration until the timestamp 00:00, when a new file is created. This plot shows that even a five minute sampling time has noticable effects on the storage process duration as the data file gets bigger. When comparing the plots in fig. 7.4 to the plots in fig. 7.2, we can see a slight decrease in duration when using design 2.

The plot in fig. 7.5 shows the data storage time for real-time power consumption from tibber during 24 hours. The plot starts with a fresh file with no data at 00:00, and the end of the plot shows the duration of data storage just before the next daily file is created. So, with a two second sampling time for real-time consumption, the duration of the data storage process peaks at about 350 milliseconds (ignoring the occasional big jumps in duration). Comparing fig. 7.5 to fig. 7.3, we can se a large decrease in the data storage duration for design 2.

7.4.3 Response time when requesting data

This test was performed with design 2 of the data storage system

The plot in fig. 7.6 shows the time it takes to process and service a data request for a web interface call to the route /request_data_since (see section 4.2.4). Here we can clearly see an increase in the response time as the daily data files gets larger, peaking at about 500 milliseconds before a new the daily file is created. These requests were made every five minutes at the beginning of each iteration of the MHE and MPC scheme, one request for each sensor data that is used within the scheme.

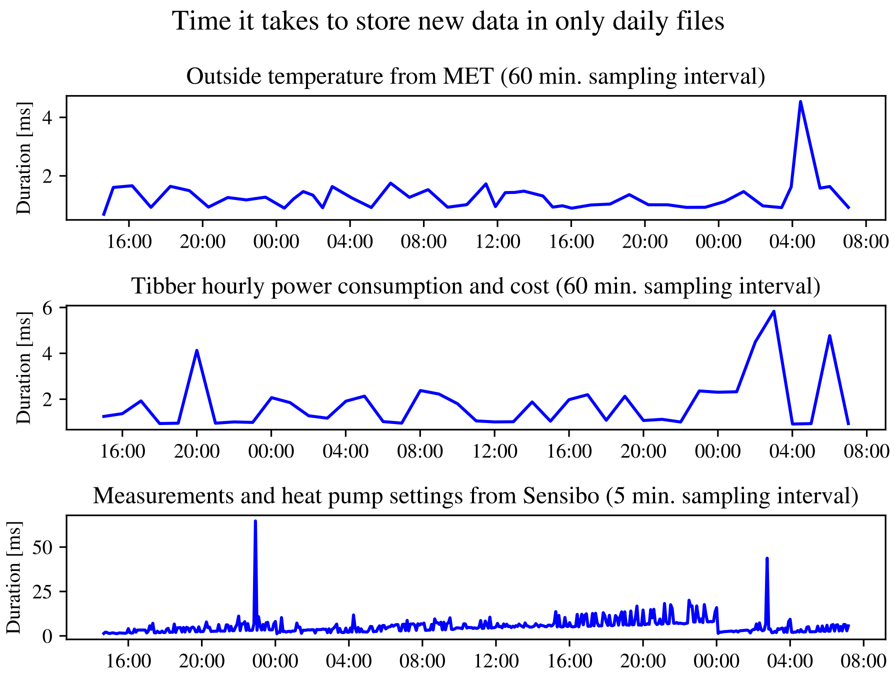


Figure 7.4: Plots of the time it takes for the main handler to store new data from MET, Tibber hourly, and Sensibo into only daily files.

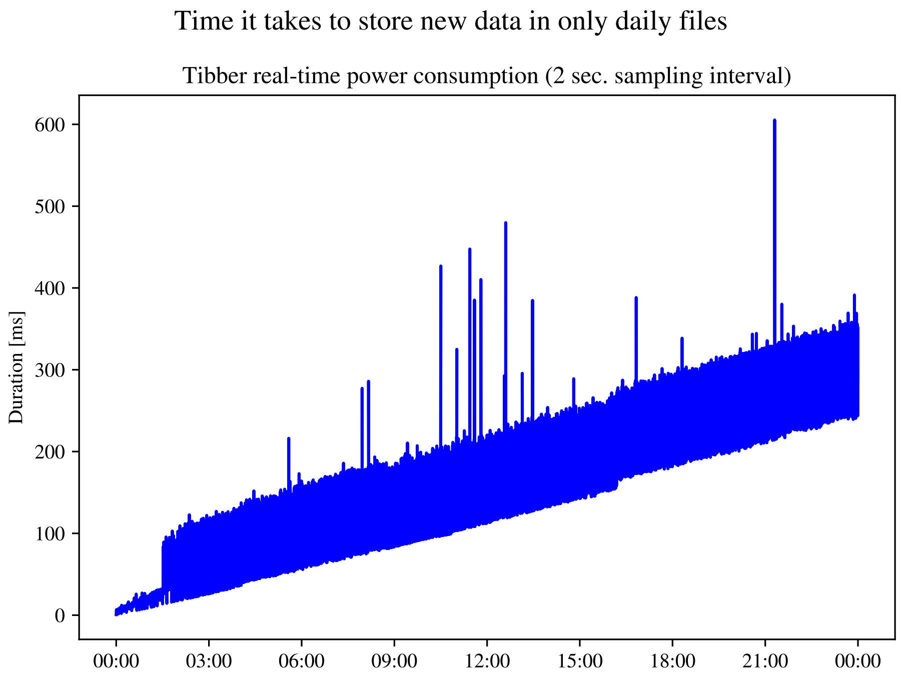


Figure 7.5: Plot of the time it takes for the main handler to store new data from Tibber real-time power into only daily files.

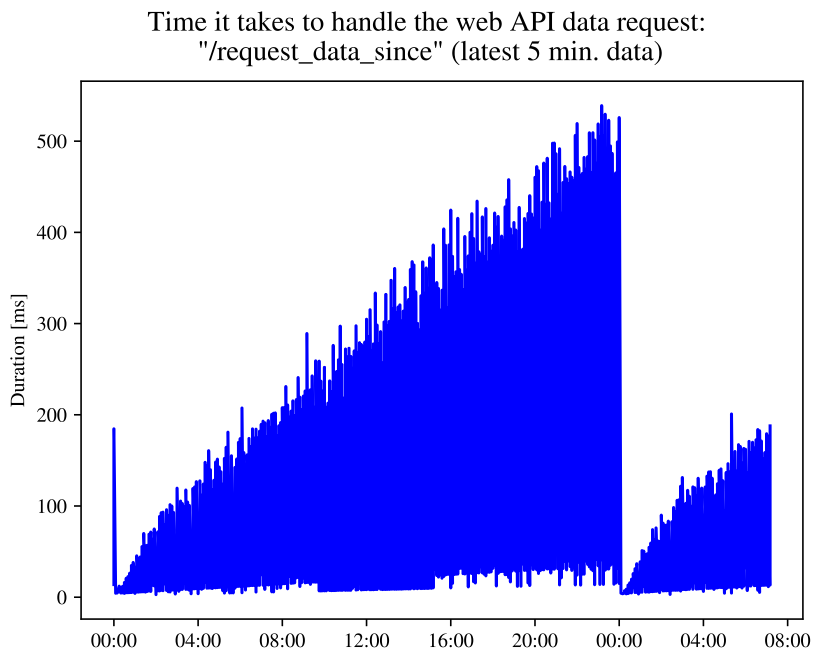


Figure 7.6: Plot of the time it takes for the main handler to service a call to the web interface route /request_data_since, requesting data from the last five minutes.

7.4.4 Running Moving Horizon Estimation and Model Predictive Control on the Raspberry Pi B

The plot in fig. 7.7 shows the processing duration of each iteration of the the MHE and MPC scheme. This scheme was running with a sampling interval of five minutes. We can see that the first iteration has a longer processing duration at 32 seconds. This is because at the first iteration we have no idea what the solution to the optimization problem is, so we start with an initial bad guess which makes it work longer to find a solution. The following iterations has a better first guess (the last solution), so it needs less time. These following iterations take between 19-24 seconds to process.

CasADi with the Ipopt solver was used for running the MHE/MPC algorithm.

This test was done simultaneously with the test in section 7.4.2, where the data storage process duration was measured. There are no noticable correlation between the MHE and MPC scheme running, and an increase in the data storage time when comparing the plots.

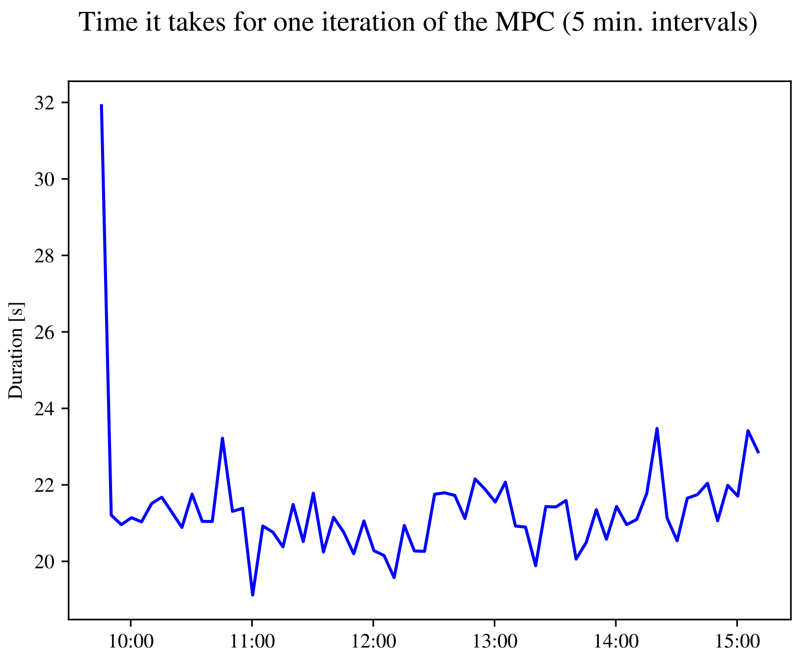


Figure 7.7: Plot of the time it takes for the MPC loop to process while the data collection threads are running. Data was only stored in daily pickle files during this test.

7.5 Utilization

To analyze how many sensors the software could handle, a utilization test was done.

7.5.1 Processing utilization of the setup in the case study

A list of the tasks that was used in the utilization test can be seen in table 7.1. Computation time is the time it takes for the task to complete, and Period is how often this task is activated. The data request comes from the web interface route /get_data_since (see section 4.2.4).

Using the utilization test from section 2.2.6 together with the tasks in table 7.1, we get the following calculation:

$$\begin{aligned} U &\equiv 2 \times \frac{0.35}{2} + \frac{0.025 + 3 \times 0.5}{300 + 22} + \frac{0.002 + 0.004}{3600} \\ &= 0.4284 \\ &\leq 9(2^{\frac{1}{9}} - 1) \\ &= 0.721 \end{aligned} \tag{7.5.1}$$

The calculation shows that the current setup is fine. However, it should be noted that this is a simplified test based on the assumptions mentioned in section 2.2.6.

7.5.2 How many sensors can the Raspberry Pi B handle with the current software?

In the plots in section 7.4.2, we see how the processing time of storing data strongly correlates to the sampling rate of the measurements and data. Using the utilization formula section 2.2.6 we can calculate how many sensor loops that the current setup could handle based on different sampling rates. The following calculations should not be taken for granted, but should be used to show the general trend of different sampling rates. Apart from the idealized calculation of the utilization test, there were some additional assumptions and simplifications

Table 7.1: Computation time and period of tasks running on the software. Computation time is the time it takes for the task to complete, and Period is how often this task is activated. The data request comes from the web interface route /get_data_since.

Task	**Computation time (C) [s]**	**Period (T) [s]**
Store new real-time power data (downstairs)	0.35	2
Store new real-time power data (upstairs)	0.35	2
Store new Sensibo data	0.025	300
Store new Tibber hourly data	0.004	3600
Store new MET weather data	0.002	3600
Service data request	3×0.5	300
Run the MHE/MPC scheme	22	300

made during this calculation:

- The computation time of the MHE and MPC scheme (table 7.1) stays the same in all the calculations. In reality, more or less data would change the computation time of this scheme.
- The time it take to complete the data storage of new data is assumed to be linear with the number of datapoints that are in each file. The plots in section 7.4.2 are suggestive of this, but it is not conclusive evidence.
- We are only considering how many sensor threads we can have of the same sampling rate.

Using the highest computation time for storing the real-time power data from fig. 7.5 (just before a new data file is generated for the next day), we get the following equation:

$$U \equiv x\frac{0.35\frac{2}{r}}{r} + \frac{C_{mpc}}{T_{mpc}} \leq x(2^{\frac{1}{x}} - 1) \tag{7.5.2}$$

Where x is the number of tasks that is possible with this computation time and period, and r is the sampling rate (period) of the sensor thread. The computation time is calculated by the term $0.35\frac{2}{s}$, using the assumption that the computation time is linear with the number of data points in the data file, and noting that the computation time of 0.35 comes from a 2 second sampling time. C_{mpc} and T_{mpc} is fixed to 22 seconds and 300 seconds, using the simplification that the computation time for the MHE and MPC scheme stays the same regardless of the sampling rate of the sensor threads.

Equation (7.5.2) was used to generate the plot in fig. 7.8, which shows the correlation between sampling rates and the maximum number of sensor threads that the software on the Raspberry Pi can handle. Using this calculation we can see a big difference when just increasing the sampling time from 2 seconds to 4 seconds, where the maximum amount of sensor threads changes from 3 to 14. However, it should again be noted that this calculation is a simplification and an extrapolation of the data from the response time tests. This should only be used as a guideline.

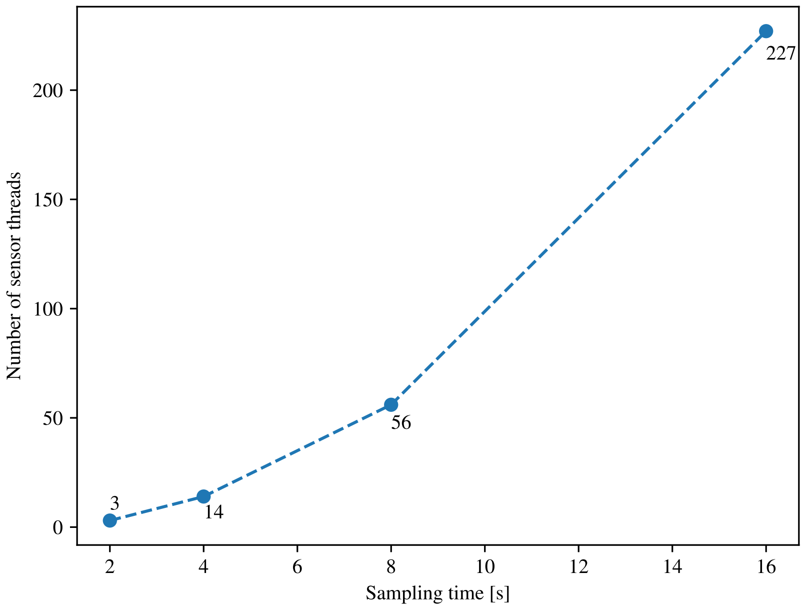


Figure 7.8: Plot showing the correlation between data sampling rates and the maximum number of sensor threads that the software on the Raspberry Pi can handle.

7.5.3 Harddrive space

After having run for the most part of two months, the status of the hard drive space is as follows:

- The *data* folder, which holds all the collected data, has a size of 581 MB.
- The *backups* folder, holding the five most recent backups, have a size of 2.8 GB.

This suggests that with a SD Card of 128 GB, the current setup in the case study can collect data for over **two years** before there should be any problems.

7.6 Data for System Identification

While the data for inside temperature, outside temperature, heat pump settings, and power consumption was successfully collected, the solar radiation measurements was not.

Solar radiation from Solcast's API was meant to be part of the model structure, but due to instabilities when requesting data from Solcast it was decided that solar radiation would not be collected during this thesis project.

Solar radiation measurements would make the model more accurate by taking into account the heating effect of direct sunlight.

7.7 Closing the loop with MHE and MPC

This section is not an evaluation of the algorithm itself, but of the system and softwares ability to support continous MHE and MPC operations. The MHE and MPC scheme was running on a separate computer during this time.

In fig. 7.9 we can see plots of room temperature, target temperature, power consumption and spot prices. At each interval of the MHE/MPC scheme, the algorithm requests temperature and power data from the Raspberry Pi, and spot prices from the Nord Pool API, to calculate a target temperature for the heat pumps. The target temperature is then relayed to the heat pumps through the Sensibo devices, which is where the target temperature plot comes from. While the plot only shows data for 24 hours, the MPC have actually been running for longer periods of times without issues.

While the MHE and MPC was running on a separate computer in the time period of fig. 7.9, the plot from fig. 7.7 shows the time each iteration of the algorithm takes when running on the Raspberry Pi. We can see that each iteration of the algorithm takes an average of 22 seconds. When we consider that the sampling interval of the algorithm is set to five minutes, and that the temperature dynamics of the system is very slow, a latency of around 22-25 seconds (taking into account the latency of API requests) is not a problem. The algorithm will successfully be able to run on the Raspberry Pi.

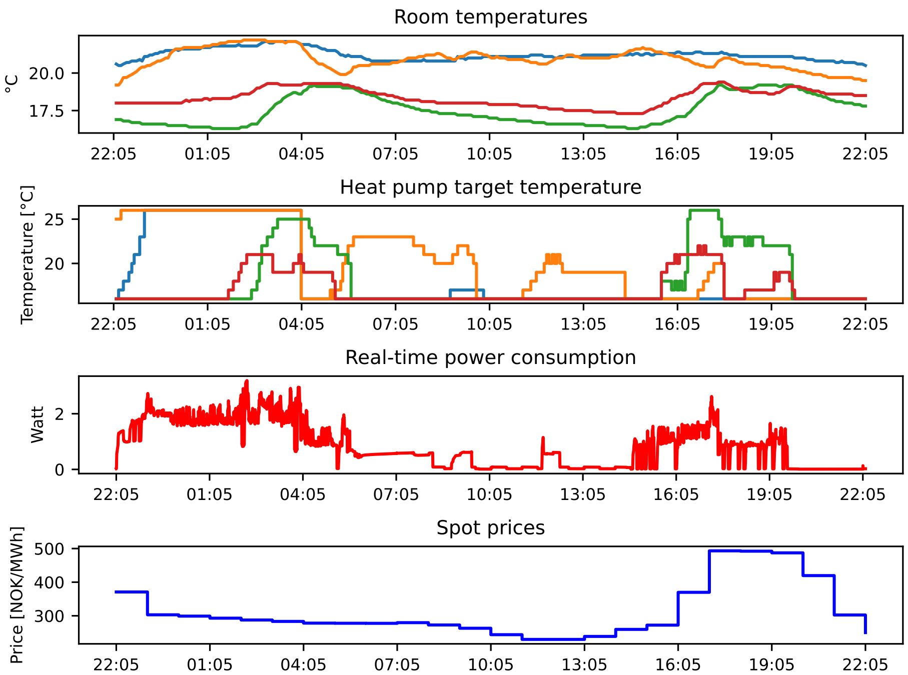


Figure 7.9: Plots showing room temperatures, target temperatures, power consumption and spot prices while the MPC scheme is running. At each five minute interval, the MPC scheme sends instructions to the heat pumps in the form of target temperature.

7.8 Requirements evaluation

In this section the software is evaluated against the requirements that were set in chapter 3. Tables tables 7.2 to 7.5 shows a simple overview of the requirements and a yes/no/partly answer to the question if they were met or not. The latter part of this section evaluates the requirements in more detail with comments.

The aim of this section is to evaluate the software in terms of what was believed to be important. This evaluation can serve as a guide to see what parts of the software that is doing well and what parts that needs improvement.

Table 7.2: Evaluation of *General System* requirements and wants.

	Requirement/Want met?		
General System Requirements (GSR) and Wants (GSW)	**Yes**	**Partly**	**No**
GSR-1: Accessible to regular homeowners	X		
GSR-2: Reliable uptime and fault tolerant	X		
GSR-3: Error logging for debugging	X		
GSW-1: As small and fast codebase as possible		X	
GSW-2: Minimize the dependency on third-party libraries and tools		X	
GSW-3: The code and usage-guide should be well documented so other people can continue research with this system after the thesis project is completed.	X		

Table 7.3: Evaluation of Data Collection Layer requirements and wants.

	Requirement met?		
Data Collection Layer Requirements and Wants	**Yes**	**Partly**	**No**
DCLR-1: Accurate measurements			
DCLR-2: Relatively low sampling times		X	
DCLR-3: Sensors and APIs for the data used in the SYSID model and the MPC algorithm	X		
DCPR-4: Continuously collect data day and night	X		
DCPR-5: Robust against data-races, over-writes, file corruption, and storage crashes	X		
DCPR-6: Ability to easily customize how data is collected (to some extent)	X		
DCLR-7: A way to access this data from another process or computer	X		
DCLW-1: The data files should be easily accessible for use in various research work related to this project (such as research about model generation with SYSID).	X		
DCLW-2: Should be possible to add or remove sensors to this system relatively easily.	X		

Table 7.4: Evaluation of *Data Processing Layer* requirements.

	Requirement met?		
Data Processing Layer Requirements and Wants	**Yes**	**Partly**	**No**
DPLR-1: Fast enough processing (to finish before the next sampling interval and to not lose the usefulness of the output)	X		
DPLR-2: A model that performs well (accurately describes the temperature dynamics)	*(Not focus of thesis)*		
DPLR-3: A well-defined optimization problem for the MPC	*(Not focus of thesis)*		
DPLR-4: Output format needs to conform to actuator input (heat pump settings such as temperature reference and heating mode)	X		

Table 7.5: Evaluation of *Applying Inputs Layer* requirements.

	Requirement met?		
Applying Inputs Layer Requirements and Wants	**Yes**	**Partly**	**No**
AILR-1: Constant uptime	X		
AILR-2: Ability to relay the instructions from the processing layer to the actuators.	X		

7.8.1 Commented evaluation:

General System Requirements and Wants:

- **GSR-1:** Accessible to regular homeowners
  - *GSR-1.1:* Relatively cheap equipment
  - *GSR-1.2:* Commercially available devices
  - *GSR-1.3:* Possible to install on existing non-smart homes

  - **Result:** Yes, the system is accessible to regular homeowners. With heat pumps already installed, the system components can be purchased for under 10 000 NOK (with four heat pumps). It is not, however, easily managed by a regular homeowner. At the moment, specialized knowledge is needed to install and run.

- **GSR-2:** Reliable uptime and fault tolerant
  - *GSR-2.1:* Resilient against internet disconnect
  - *GSR-2.2:* Resilient against system crashes and restarts
  - *GSR-2.3:* Persistent storage of data
  - *GSR-2.4:* Tolerant against data races and data corruptions
  - *GSR-2.5:* Reliable communication
  - *GSR-2.6:* Reliable processing

  - **Result:** Yes, when considering the tests and experience of running the software, the the system is fault tolerant. Of course, there might be edge cases that we have not tested for.

- **GSR-3:** Error logging for debugging
  - **Result:** Yes, most errors are logged to a file.

- **GSW-1:** As small and fast codebase as possible
  - **Result:** While a small codebase has been a focus, it is probably not as fast or small as it could be. Especially when considering that the language is Python, which is slow compared to languages like C or

C++.

- **GSW-2:** Minimize the dependency on third-party libraries and tools
  - **Result:** Partly. The software mainly uses third-party libraries for making it easier to make API requests. E.g. *metno_locationforecast*, *pyTibber*, and more.

- **GSW-3:** The code and usage-guide should be well documented so other people can continue research with this system after the thesis project is completed.
  - **Result:** Yes, it is well documented.

Data Collection Layer Requirements and Wants:

- **DCLR-1:** Accurate measurements
  - **Result:** How accurate the measurements are have not been examined.
- **DCLR-2:** Relatively low sampling times
  - **Result:** Real-time power and room temperature have low sampling times at two seconds and five minutes, respectively. Outdoor temperature does not have low sampling time at one hour.
- **DCLR-3:** Sensors and APIs for measurements and data used in the SYSID model and the MPC algorithm
  - **Result:** Yes, all the data that SYSID and MPC needs for the model are available through APIs.
- **DCPR-4:** Continuously collect data day and night
  - *DCLR-4.1:* Automatic restart on crashes
  - *DCLR-4.2:* Ability to handle disconnects

  - **Result:** Yes, the software collects data continually.

- **DCLR-5:** Robust against data-races, over-writes, file corruption, and storage crashes
  - **Result:** Yes, as far as tests and experience goes, the software is robust

against these problems.

- **DCLR-6:** Ability to easily customize how data is collected (to some extent)
  - *DCLR-6.1:* Easily change sampling times
  - *DCLR-6.2:* Turn on or off the data collection from a sensor

  - **Result:** Yes, you can easily send API requests to the software to change sampling times or to turn off the data collection.

- **DCLR-7:** A way to access this data from another process
  - **Result:** Yes, the data is accessible through the software's API.
- **DCLW-1:** The data files should be easily accessible for use in various research work related to this project (such as research about model generation with SYSID).
  - **Result:** Yes, you can either request data through the software's API, or download data files directly from the web address of the software.
- **DCLW-2:** Should be possible to add or remove sensors to this system relatively easily.
  - **Result:** Yes, this is relatively easy.

Data Processing Layer Requirements:

- **DPLR-1:** Fast enough processing (to finish before the next sampling interval and to not lose the usefulness of the output)
  - **Result:** Yes, the average processing time for each MHE/MPC interval is around 22 seconds, while the sampling interval is at five minutes.
- **DPLR-2:** A model that performs well (accurately describes the temperature dynamics)
  - **Result:** This has not been part of this thesis.
- **DPLR-3:** A well-defined optimization problem for the MPC
  - *DPLR-3.1:* Well-defined constraints of the system
  - *DPLR-3.2:* Well-tuned cost-function

  - **Result:** This has not been part of this thesis.

- **DPLR-4:** Output format needs to conform to actuator input (heat pump settings such as temperature reference and heating mode)
  - **Result:** Yes, this is very straightforward when using the Sensibo API.

Applying Inputs Layer Requirements:

- **AILR-1:** Constant uptime
  - **Result:** Yes.
- **AILR-2:** Ability to relay the instructions from the processing layer to the actuators.
  - **Result:** Yes.

Chapter 8

Discussion

This chapter presents a discussion about the results around fault tolerance, response times, utilization, and the system and software implementation in general. The results will also be discussed in terms of the aims of the thesis which were presented in chapter 1.

8.1 Tests and results

The main findings of the tests are that the software and system setup is working well for collecting data while simultaneously running a real-time MHE and MPC scheme in the case study house. The software reliably collects data at set intervals and can supply this for the MHE and MPC scheme with low response times. However, while the software is working well, it is hard to say how this software design would compare to other designs.

Fault tolerance The results of the fault tolerance tests show that the software is able to handle disconnects and power losses without long interruptions in the data collection. The main interruptions of data collection comes from Tibber's limit on socket connections, which causes a 10-30 minute delay until a reconnection can occur after a disconnect. This is not a big concern, as the disconnects does not happen often, and power losses almost never. The software can run without supervision for long periods of time.

A free version of the Ngrok tunnel was used for the purpose of accessing the Raspberry Pi from outside the local area network. This has been a simple solution which was easy to implement, but it has not been the most stable. The ngrok tunnel vary in quickness, and sometimes it seems like the tunnel has trouble with servicing data requests. The free account is not made for running continually on a software like this. It may work better with a paid account, but this is not certain. Another solution should be investigated.

In all, the software seems to have good fault tolerance based of the test results and personal experience when running the software. However, the evaluation of fault tolerance is limited by the design of the tests themselves, and there might be some edge case bugs in the code which were not detected. While some specific tests were made, a big part of the *testing* has come from having the software run at the case study house for a long period of time. The impression from this is that the most critical bugs have been fixed and the software can now run unsupervised.

Response times The results from the response tests section 7.4.2 clearly shows the effects large pickle data files have on the response time for storing new data and making data requests. When having a small sampling rate, as with the real-time power consumption in the case study (2 seconds), the response time from storing new data to the file grows so large that the response times are becoming unacceptable for anything but daily data files. And even with only daily files, the utilization calculation in section 7.5 shows that the current software (and Raspberry Pi hardware) can only support three sensor [API] threads with a two second sampling interval. It should be noted that for the system in this thesis, such a short sampling time should usually only be needed for one sensor thread (one for each smart meter in the house), with most sensors needing only a sampling rate that is in terms of minutes.

With the *current* software design there are two ways to improve response time: the data file sizes could be reduced further (for example to only store data for one hour at a time) or the sampling times could be increased for real-time power data storage, for example from two seconds to four seconds.

The MHE and MPC scheme worked well on the Raspberry Pi. With an average response time of around 22 seconds (fig. 7.7), the scheme has no problem completing before the next interval. The latency between measurement and actuation is also not a big concern, since the temperature dynamics of a house are so slow. It should be noted that CasADi and Ipopt were used, which are not optimized for running in real-time on devices with limited processing capabilities. It is therefore possible to reduce the response time of the MHE and MPC with the use of a faster optimization tool (and potentially another programming language).

The response time analysis shows that the software and hardware can support the current system. However, there are room for improvement. The results showed that daily pickle files works fine, but it also suggests that there is a need to investigate other types of storage systems as well.

Utilization The results from the utilization tests section 7.5 shows that while the current setup in the case study is fine in terms of utilization, it is not far from exceeding the limit of the Raspberry Pi's processing power. This is mainly because of the two sensor threads with a two seconds sampling rate. However, most houses only have one smart meter and would only have one such sensor thread, meaning that the utilization in those houses would be lower. If utilization becomes a problem, it is also possible to increase the sampling rate from two seconds to three or four seconds (or even average it out over five minutes to make the sampling rate coincide with the MPC sampling rate). The effects this would have on the accuracy of the system would have to be investigated.

It is important to note that the utilization calculation is based on a list of assumptions (see section 2.2.6), and it is unclear how much these assumptions affect the results of the utilization tests.

In terms of hard drive space, the results show that the setup in the case study could run for over two years without interruption. While this is a long time, it could still be a concern in a future plug-and-play system which may run for years without supervision. In this case, a solution could be to delete data that is older than two years, after having sent it to another server for storage.

The biggest concern in utilization are due to the sampling times of the sensor threads. It is possible that a different design of the data storage system would be more efficient and thus reducing the concerns around sampling times utilization. This is an important subject to investigate for a future iteration of the software.

8.2 Evaluation of system and software implementation

What's good:

- The whole system set up is inexpensive and affordable to regular people.
- System devices works well. They are reliable and easy to setup and use.
- Overall performance in terms of both response time, utilization, and fault tolerance is satisfactory.
- The software's own API works well and reliably services the MHE/MPC's data requests.
- The system will have no problem running MHE and MPC on the Raspberry Pi.
- Seems to have been a good choice for software design in terms of sensor threads, a queue system, and a main handler thread. The software handles

both data collection and data requests well.

- Docker has worked well for porting the software to the Raspberry Pi, and is also working well as a supervisor for the software (restarting it on crashes and disconnects).
- The code structure can quite easily be modified to change the API's or devices used in the system.
- The software is well documented for future development.

What could be improved:

- While Ngrok worked okay for accessing the Raspberry Pi from outside the local area network, it does have some stability issues. An alternative should be found. Perhaps it is enough to buy a premium Ngrok account, or perhaps it is better to find another solution entirely.
- The software is relatively specialized towards the case study in terms of the real-time tibber sensors and naming conventions of variables. There is therefore a need to change the code for the real-time sensor threads if this software is to be used on a different house. The software could be improved by making the code completely generalized for any home.
- I think there are room for improvement in regards to data storage. While pickle files are convenient in terms of being able to just copy the files and play around with the data, the performance might be better if using another storage format. If response times for data storage and data requests ever becomes a limiting factor, this is one area that should be investigated.
- Solar radiation from Solcast's API was discarded due to instability of requests. An alternative should be found. Having solar radiation data available would make a more accurate model, as well as be important for future systems that includes solar panels.

- The API structure of the software could be improved. The naming right now is slightly confusing for some of the API calls. For example, /get_data/sensibo/daily/all returns all data that has been collected from Sensibo by looking through the daily files, but the naming of the route suggests that perhaps it is all data from the current day.

Suggestion for the future developer of this software:

- This software design definitely works, but please use this software design as an inspiration, and not as a definite example on how to do these things. It might even be best to start from scratch and only import the parts of this software design that you want to use.
- If some part seems hard to understand and overly complex, it probably is, and there should be an opportunity for improvement there.

8.3 Aims and tasks of the thesis

This thesis set out to create a software for combining cheap IoT devices to control the heating in a house, and to investigate the viability of such a system.

The first task was to implement a robust software system for collecting data, servicing data requests, and sending control instructions to heat pumps. In regard to collecting data and servicing data requests, the software implementation holds up well. In terms of sending control instructions to the heat pumps, this was shown to just require a simple call to the Sensibo API, and was therefore included directly in the MHE and MPC script during this thesis.

The second task was to collect data. The early version of the software in this thesis started to collect data for about two months ago. Since then, there have been many updates to the software, but it has continued to collect data during this time. At the time of this writing, the software has collected data more or less uninterrupted for almost a month.

The third task was to test a high-level decision-making algorithm, using a model that was generated through SYSID. The MHE and MPC algorithm developed by Sebastien Gros was tested on the system with good results. The software is able to reliable provide data for the algorithm, and the algorithm itself can run well on the Raspberry Pi.

Summing up, this thesis have answered some important questions regarding response times and utilization in a system based on cheap IoT devices and a Raspberry Pi B computer. The viability of the system shows promise, and it will be exciting to see what the future of this research will hold.

8.3.1 Accessibility and Plug-and-play'ability

One of the goals of the POWIOT project is to see if a plug-and-play system can be developed for normal houses. I will therefore discuss here how the software and system holds up against a future plug-and-play vision.

At the current state of the system and software, while the devices are easy to setup, the software for the Raspberry Pi requires programming knowledge and a precise installation guide. The software needs to be slightly customized to a new home in terms of the number of smart meters it has (which dictates the number of real-time power sensor threads the software is running). The software also needs to collect data for a while before the SYSID operation can generate a good model for the house (which is currently done manually), and for the MHE and MPC script to operate well.

Although it is clear that the system is far from plug and play in its current state, I believe a plug-and-play system is a realistic vision. The following is a list of ideas for making this vision a reality:

- *Supply pre-installed software* by having the future user of the system supply a specification of the house: The number of heat pumps, number of smart meters, and so on. The software could be pre-installed on the Raspberry

Pi so that it just starts running when the user plugs it in.

- *Create a Graphical User Interface (GUI)* where the user can add or remove IoT devices, select weights to the MPC, or set temperature references. This GUI can be created with Flask and accessed by a web address, which is what is already being done with the softwares own API. In regard to SYSID, this GUI could have a status indicator, enabling a *start* button when the system have gathered enough data for generating a model. This *start* button would then start by generating a model with SYSID, then start the MHE/MPC scheme. The GUI could also show information about measurements and temperature trajectories, giving the user an indication of how well the system is working.

Chapter 9

Conclusion

This thesis set out to investigate the viability of using cheap equipment to interact with energy-related devices (such as heat pumps) and introducing high-level decisions in the system (e.g., setting temperature references on heat pumps) based on a high-level decision-making algorithm. Specifically, the thesis investigated the problem of designing a software that combined cheap commercially available IoT devices and API services together with a MHE and MPC scheme to control heating in a residential house.

The software implementation and the following test results has shown that this kind of system is viable, and the specific components used in the case study of the thesis are good options for this kind of system. The Raspberry Pi is a cheap and small computer that shows a good promise in handling this kind of system while being small enough to not be overly expensive or invasive in a home.

The findings in this thesis serves as laying the ground work for further software development and investigations for the system presented in this thesis. The software will continue to be used for collecting data and servicing data requests for experimental MHE and MPC schemes in the case study house. Next fall, a new student will continue working on this software.

The limitations of the current findings are mainly that it has not been compared to other software solutions. Especially in terms of the data storage method (pickle files). Other solutions, like using a database, might show to be a better alternative for a future version of the software. Nothwithstanding these limitations, the study suggests that the current software does work quite well in terms of performance during the case study.

9.1 Further work

The following lists presents the recommendations I have for further work in regards to the software and system presented in this thesis.

Software and system improvements

- Find a reliable provider of solar radiation forecasts. Having solar radiation in the model would improve the accuracy of temperature predictions.
- The method for accessing the Raspberri Pi from outside the local area network can be improved, since Ngrok had indications of instability issues.
- The API structure can be improved.
- The software can be made more generalized for being used in different houses.

Investigations

- Investigate other data storage systems. Especially high-speed database systems.

Vision and ideas

- In terms of the ultimate vision, which is a plug and play system, the software needs to be more generalized and potentially provide a graphical user interface for the home owner to make various changes (like setting weights for how cost and comfort is prioritized) or view the temperature measurements. This could be done as an app or as a webpage.

Bibliography

[1] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages (Fourth Edition)*. 2005.

[2] *CasADi*. 2021. URL: `https://web.casadi.org/`.

[3] *Docker*. 2021. URL: `https://www.docker.com/`.

[4] *Flask micro web-framework*. 2021. URL: `https://flask.palletsprojects.com/en/2.0.x/`.

[5] Bjarne Foss and Tor Aksel N. Heirung. *Merging Optimization and Control*. 2016.

[6] Sebastien Gros. 2021. URL: `https://www.ntnu.edu/employees/sebastien.gros`.

[7] Sebastien Gros. *Modelling And Simulation - Additional Lecture notes for the NTNU course TTK4130*. Jan. 2020.

[8] *Gunicorn*. 2021. URL: `https://gunicorn.org/`.

[9] Kummert M., Marie-Andrée Leduc, and Alain Moreau. *Using MPC to reduce the peak demand associated with electric heating*. Conference paper uploaded to ResearchGate. 2011. URL: `https://www.researchgate.net/publication/267429983_Using_MPC_to_reduce_the_peak_demand_associated_with_electric_heating_Contents`.

[10] *Netamo Weather Station*. 2021. URL: https://www.netatmo.com/no-no/weather.

[11] *Nginx*. 2021. URL: https://nginx.org/.

[12] *Nord Pool API*. 2021. URL: https://www.nordpoolgroup.com/trading/api/.

[13] NVE. *Smarte strømmålere (AMS)*. webpage. 2021. URL: https://www.nve.no/stromkunde/smarte-strommalere-ams/.

[14] European Parliament. *On the energy performance of buildings*. 2010, pp. 13–14. URL: https://www.buildup.eu/sites/default/files/content/EPBD2010_31_EN.pdf.

[15] D.W.U. Perera, C. F. Pfeiffer, and N.-O Skeie. "Control of temperature and energy consumption in buildings - A review". In: *International Journal of Energy and Environment* 5.4 (2014), pp. 471–484.

[16] Mayne D. Q. et al. "Constrained model predictive control: Stability and optimality". In: *Automat- ica* 36.6 (2000).

[17] *Raspberry Pi*. 2021. URL: https://www.raspberrypi.org.

[18] *Rectifier (neural networks)*. 2021. URL: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#Parametric_ReLU.

[19] *Sensibo*. 2021. URL: http://www.sensibo.com.

[20] Pervez Hameed Shaikh et al. "A review on optimized control systems for building energy and comfort management of smart sustainable buildings". In: *Renewable and Sustainable Energy Reviews* 34 (2014), pp. 409–429.

[21] *Solcast API*. 2021. URL: https://solcast.com/.

[22] *Tibber*. 2021. URL: http://www.tibber.com.

[23] Tibber. *Senk strømforbruket ved smart varmestyring*. Website. May 2021. URL: https://tibber.com/no/store/varmestyring.

Eric Törn

IoT Software for Smart Houses

NTNU
Norwegian University of
Science and Technology